Investigations in intersection types: Confluence, and semantics of expansion in the λ-calculus, and a type error slicing method

Vincent Rahli

Submitted for the degree of Doctor of Philosophy

Heriot-Watt University

School of Mathematical and Computer Sciences

January 10, 2011

Abstract

Type systems were invented in the early 1900s to provide foundations for Mathematics where types were used to avoid paradoxes. Type systems have then been developed and extended throughout the years to serve different purposes such as efficiency or expressiveness. The λ-calculus is used in programming languages, logic, mathematics, and linguistics. Intersection types are a kind of types used for building semantic models of the λ-calculus and for static analysis of computer programs.

The confluence property was used to prove the λ-calculus' consistency and the uniqueness of normal forms. Confluence is useful to show that logics are sensibly designed, and to make equality decision procedures for use in theorem provers. Some proofs of the λ-calculus' confluence are based on syntactic concepts (reduction relations and λ-term sets) and some on semantic concepts (type interpretations). Part I of this thesis presents an original syntactic proof that is a simplification of a semantic proof based on a sound type interpretation w.r.t. an intersection type system. Our proof can be seen as bridging some semantic and syntactic proofs.

Expansion is an operation on typings (pairs of type environments and result types) in type systems for the λ-calculus. It was introduced to prove that the principal typing property (i.e., that every typable term has a strongest typing) holds in intersection type systems. Expansion variables were introduced to simplify the expansion mechanism. Part II of this thesis presents a complete realisability semantics w.r.t. an intersection type system with infinitely many expansion variables. This represents the first study on semantics of expansion. Providing sound (and complete) realisability semantics allows one to study the algorithmic behaviour of typed λ-terms through their types w.r.t. a type system. We believe such semantics will cast some light on the not yet well understood expansion operation.

Intersection types were used in a type error slicer for the SML programming language. Existing compilers for many languages have confusing type error messages. Type error slicing (TES) helps the programmer by isolating the part of a program contributing to a type error (a slice). TES was initially done for a tiny toy language (the λ-calculus with polymorphic let-expressions). Extending TES to a full language is extremely challenging, and for SML we needed a number of innovations. Some issues would be faced for any language, and some are SML-specific but representative of the complexity of language-specific issues likely to be faced for other languages. Part III of this thesis solves both kinds of issues and presents an original, simple, and general constraint system for providing type error slices for ill-typed programs. We believe TES helps demystify language features known to confuse users.

Acknowledgments

I would like to thank both Professor Fairouz Kamareddine and Doctor Joe Wells for supervising my PhD studies within the ULTRA group. I would like to thank them for their patience, comments and guidance, and more generally for all they taught me throughout the years of my studies. Also, I would like to thank them for their support inside as well as outside university.

I would like to thank Doctor Karim Nour for his collaboration on the semantics of expansion project. I would like to thank Doctor Virgile Mogbil for supervising my master dissertation and for his help in searching for a PhD position. I would like to thank Professor Mariangiola Dezani-Ciancaglini for accepting to be my external examiner and Doctor Greg Michaelson for accepting to be my internal examiner.

I would like to thank Laëtitia for her support, her understanding, for sharing my life and making each day of my life a bliss. I would like to thank my parents and my sister for their continuous support and encouragements, for always being there for me at any time. I would like to thank Catherine, Karine, Sophie, and Arnaud for their support throughout the years of my PhD studies and for their infallible friendship. I would like to thank all the members of the ULTRA group, Daniel, Jan, Robert, Manuel, Krzysztof, Christoph, Sergueï, and Sébastien for all that we shared inside and outside university. I would like to thank all the members of the type error slicing projects, and especially John, Mark, and Scott for making of our shared office a great place to work. I would also like to thank them for all our chess and go games. I would like to thank my hockey teammates and especially Mike and Ham. I would not have survived without our weekly trainings. Finally, I would like to thank any other people I may have forgotten in these acknowledgements.

Contents

List of Figures

Chapter 1

Mathematical definitions and notations

Natural numbers
Let i, j, m, n, p, q be metavariables ranging over $\mathbb{N}$, the set of natural numbers.

Metavariables
If a metavariable v ranges over a class C, then the metavariables v_x (where x can be anything) and the metavariables v', v'', etc., also range over C.

Sets
Let s range over sets. If v ranges over s, then let $\overline{v}$ range over $\mathbb{P}(s)$, the power set of s.

Disjunction
Let $\mathsf{dj}(s_1, \ldots, s_n)$ ("disjoint") hold iff for all $i, j \in \{1, \ldots, n\}$, if $i \neq j$ then $s_i \cap s_j = \varnothing$. Let $s_1 \uplus s_2$ be $s_1 \cup s_2$ if $\mathsf{dj}(s_1, s_2)$ and undefined otherwise.

Relations
Let $\llparenthesis x, y \rrparenthesis$ be the pair of x and y. If rel is a binary relation (a pair set), let $(x\ \mathit{rel}\ y)$ iff $\llparenthesis x, y \rrparenthesis \in \mathit{rel}$, let the inverse of rel be rel^{-1} defined as $\{\llparenthesis x, y \rrparenthesis \mid \llparenthesis y, x \rrparenthesis \in \mathit{rel}\}$, let $\mathsf{dom}(\mathit{rel}) = \{x \mid \llparenthesis x, y \rrparenthesis \in \mathit{rel}\}$, let $\mathsf{ran}(\mathit{rel}) = \{y \mid \llparenthesis x, y \rrparenthesis \in \mathit{rel}\}$, let $s \lhd \mathit{rel} = \{\llparenthesis x, y \rrparenthesis \in \mathit{rel} \mid x \in s\}$, and let $s \mathbin{\lhd\!\!\!-} \mathit{rel} = \{\llparenthesis x, y \rrparenthesis \in \mathit{rel} \mid x \notin s\}$.

Functions
Let f range over functions (a special case of binary relations), let $s \rightarrow s' = \{f \mid \mathsf{dom}(f) \subseteq s \wedge \mathsf{ran}(f) \subseteq s'\}$, and let $x \mapsto y$ be an alternative notation for $\llparenthesis x, y \rrparenthesis$ used when writing some functions. Let $f_1 + f_2 = f_2 \cup (\mathsf{dom}(f_2) \mathbin{\lhd\!\!\!-} f_1)$. Let $f_1 \boxplus f_2$ be $f_1 \cup f_2$ if $f_1 \cup f_2$ is a function and undefined otherwise. If $f_1, f_2 \in s_1 \rightarrow \mathbb{P}(s_2)$ then let $f_1 \Cup f_2 = \{x \mapsto f_1 \cup f_2 \mid x \in \mathsf{dom}(f_1) \cap \mathsf{dom}(f_2)\} \cup \mathsf{dom}(f_2) \mathbin{\lhd\!\!\!-} f_1 \cup \mathsf{dom}(f_1) \mathbin{\lhd\!\!\!-} f_2$.

Tuples
A tuple t is a function such that $\mathsf{dom}(t) \subset \mathbb{N}$ and if $1 \leq j \in \mathsf{dom}(t)$ then $j - 1 \in \mathsf{dom}(t)$. Let t range over tuples. If v ranges over s then let $\overrightarrow{v}$ range over

$\mathsf{tuple}(s) = \{t \mid \mathsf{ran}(t) \subseteq s\}$. We write the tuple $\{0 \mapsto x_0, \ldots, n \mapsto x_n\}$ as $\langle x_0, \ldots, x_n\rangle$. Let @ append tuples: $\langle x_1, \ldots, x_i\rangle @ \langle y_1, \ldots, y_j\rangle = \langle x_1, \ldots, x_i, y_1, \ldots, y_j\rangle$. Given n sets $s_1, \ldots, s_n$, let $s_1 \times \cdots \times s_n$ be $\{\langle x_1, \ldots, x_n\rangle \mid \forall i \in \{1, \ldots, n\}.\ x_i \in s_i\}$. Note that $s_1 \times \cdots \times s_n \subseteq \mathsf{tuple}(s_1 \cup \cdots \cup s_n)$.

Inference rules

An inference rule is a pair premises/conclusion which states that if the premises are true then the conclusion must be true as well. In the literature, an inference rule is often written as follows:

$$\frac{y_1 \quad \cdots \quad y_n}{x}\ (\mathsf{r})$$

which means that if y_i for all $i \in \{1, \ldots, n\}$ are true then x is true. This rule is named (r). Such a rule is sometimes written as follows:

$$(r)\, y_1 \wedge \cdots \wedge y_n \Rightarrow x$$

In this document we also sometimes write such a rule as follows:

$$(r)\, x \Leftarrow y_1 \wedge \cdots \wedge y_n$$

The rule name is sometimes omitted in such rules.

Chapter 2

Introduction

2.1 History of the λ-calculus

In the nineteenth century, due to the lack of precision of natural languages and the discovery of some controversial results in analysis [79], mathematicians and logicians became interested in a more precise formalisation of Mathematics. Frege [138, 79] was the first to set solid logic foundations. He, among other things, presented a formalisation of the concept of a function. The development of formal systems by Frege and his contemporaries led to the discovery of some paradoxes. The paradox in Frege's work, found by Russell [121], was due to the problem of self-reference. This problem is inherent to the fact that any function can be applied to any function (in particular to itself). In order to solve this problem, Russell [121] defined a theory of types where types are used to restrict the application of functions.

One of the great achievements in the movement led by Frege, Russell, Curry, etc., aiming at the formalisation of Mathematics has been the design of the λ-calculus[1] by Church [21]. In 1932, Church [21] introduced a system for "the foundation of formal logic", which was a formal system for logic and functions. The set of terms of this system was defined as a superset of the set of terms of the λI-calculus. In addition, Church introduced two sets of postulates. The first one called "rules of procedure" allowed, among other things, dealing with conversion of λ-terms (these rules are presented in Sec. 3.2). The second set contained the "formal postulates" which were logical axioms. However, this system and some of its subsystems turned out to be inconsistent as shown by Kleene and Rosser [91]. Nevertheless, the subsystem dealing only with functions turned out to be a "successful model for computable functions" [5]: the actual λ-calculus is a generalisation of this earlier system.

This earlier system led to the actual λ-calculus. Church defined the computable functions as the λ-definable ones. Also, it turned out that the set of computable functions defined by Turing via his machines is equivalent to the set of λ-definable

[1] Barendregt [5], Rosser [120], and Cardone and Hindley [18] provide extensive introductions to the λ-calculus.

functions [136] and also to Gödel's recursive functions [51]. These proposals are nowadays often referred as Church-Turing's thesis or as Church's thesis. As explained by Kleene [90], it is called a thesis and not a theorem because "it proposes to identify a somewhat vague intuitive concept with a concept phrased in exact mathematical terms, and thus is not susceptible of proof".

As Barendregt stresses in the introduction of his book [5], this theory presents functions as rules, and not as sets of pairs, in order to deal with their computational aspects. As explained by Kamareddine, Laan, Nederpelt [79], the λ-calculus turned out to be a generalisation of the definition of functions given, e.g., by Russell [144] ("propositional functions"). The λ-calculus is nowadays used in programming languages, logic, mathematics, and linguistics.

The λ-calculus allows one to compute thanks to rules often referred to as reduction or conversion rules. These rules were extensively studied and one of the main result was the proof of the confluence of β-reduction [24] which is the main computation rule of the λ-calculus. Confluence is the property that was originally used to prove, among other things, the consistency the λ-calculus (the theory built upon β-reduction and α-conversion) because it allows one to prove that there exists at least two closed different λ-terms. Confluence is sometimes referred to as the Church-Rosser property. It was also originally used to prove the uniqueness of normal forms [24].

In the early 1940s, Church added simple types, which are the types built upon ground types and the arrow type constructor, to the λ-calculus in a system with logical axioms to deal with logic and functions [23]. Church's approach was to directly annotate λ-terms: type-free λ-terms are replaced by typed λ-terms. Curry followed another approach. He considered the combinatory logic [31] which is a type-free calculus that can be regarded as a variant of the λ-calculus. His type system associates types with type-free terms via a typing relation [30, 31]. As explained by Barendregt [6], these two "approaches to typed lambda calculus correspond to two paradigms in programming". In a system à la Curry, given a type-free λ-term, if a type can be associated with the term w.r.t. the typing relation of the system then a type inference algorithm can infer a type for the term. It is also the case for ML-like programming languages such as SML [106, 107] or for Haskell-like programming languages [77].

Since the introduction of these systems by Church and Curry, various type systems for the λ-calculus have been developed and extended to serve different purposes such as efficiency or expressiveness. For example, the type systems of the λ-cube [6] allow one to express concepts such as polymorphism (which means that terms can have more than one type), type constructors (e.g., SML datatypes), dependent types (which means that types are depending on terms). There are several advantages of having a notion of types in a programming language. For example, they allow:

checking static correctness, e.g., find type inconsistencies; efficient implementations by generating information used for optimisations at compilation, e.g., "the type of a data determines its memory size and layout" [100]; modularity, e.g., thanks to signatures in SML or interfaces in Java.

Let us mention that there is a strong connection between type theory and proof theory known as the Curry-Howard isomorphism [76, 123]. This isomorphism allows one to consider, e.g., simple types as propositions. As a matter of fact, there is a correspondence between the minimal propositional logic and the simply typed λ-calculus (other such correspondences exit). The Curry-Howard isomorphism is often referred to as the proofs-as-programs, formulae-as-types correspondence.

2.2 Structure of this Chapter

The rest of this introduction is structured as follows. Sec. 2.3 introduces the untyped λ-calculus and some of its variants: the λI-calculus and the $\lambda\eta$-calculus. We also introduce properties of λ-calculi such as the confluence property. Sec. 2.4 presents notable typed λ-calculi: the simply typed λ-calculus, some intersection type systems, and the Hindley-Milner type system. Sec. 2.5 presents two methods of reasoning involving λ-calculi (or similar functional systems): realisability and reducibility. Finally, Sec. 2.6, summarises the contributions of the present thesis as well as its structure.

2.3 The untyped λ-calculus and some of its variants

The λ-calculus and its variants are defined on term sets and reduction relations. First, Sec. 2.3.1 presents various term sets and Sec. 2.3.2 some reduction relations. Then, Sec. 2.3.3 introduces different λ-calculi of interest based on these terms sets and reduction relations. Finally, Sec. 2.3.4 presents properties of λ-calculi such as confluence and normalisation.

2.3.1 Sets of terms

Let x, y, z range over Var, a countable infinite set of term variables (or just variables). The set of terms of the λ-calculus is defined as follows:

$$M, N, P, Q, R \in \Lambda ::= x \mid (\lambda x.M) \mid (MN)$$

We assume the usual convention for parentheses and omit them when no confusion arises. In particular, we write $MM_0 \cdots M_n$ instead of $(\cdots((MM_0)M_1)\cdots M_{n-1})M_n$.

let *rel* be a binary relation on Λ.

$$\frac{}{M\ rel\ M}\ (\mathsf{refl}) \qquad \frac{M\ rel\ N}{N\ rel\ M}\ (\mathsf{sym}) \qquad \frac{M_1\ rel\ M_2 \quad M_2\ rel\ M_3}{M_1\ rel\ M_3}\ (\mathsf{tr})$$

$$\frac{P\ rel\ Q}{\lambda x.P\ rel\ \lambda x.Q}\ (\mathsf{abs}) \qquad \frac{Q\ rel\ Q'}{PQ\ rel\ PQ'}\ (\mathsf{app_1}) \qquad \frac{P\ rel\ P'}{PQ\ rel\ P'Q}\ (\mathsf{app_2})$$

Figure 2.1 Closure rules

We call a term of the form $(\lambda x.M)$ a λ-*abstraction* (or just abstraction) and a term of the form MN an *application*.

We write $\mathsf{fv}(M)$ for the set of the free variables occurring in M. The function fv is defined as follows:

$$\begin{aligned} \mathsf{fv}(x) &= \{x\} \\ \mathsf{fv}(\lambda x.M) &= \mathsf{fv}(M) \setminus \{x\} \\ \mathsf{fv}(MN) &= \mathsf{fv}(M) \cup \mathsf{fv}(N) \end{aligned}$$

We say that a term is *closed* if no free variable occurs in it, i.e., M is closed iff $\mathsf{fv}(M) = \varnothing$. Let $\mathsf{closed}(M)$ be true iff M is closed.

Fig. 2.1 present some closure rules in Λ: rule (refl) is the reflexive closure rule (w.r.t. Λ), rule (sym) is the symmetric closure rule, rule (tr) is the transitive closure rule, and rules (abs), $(\mathsf{app_1})$, and $(\mathsf{app_2})$ are the compatible closure rules.

The α-conversion is the symmetric, reflexive (w.r.t. Λ), transitive, and compatible closure of the following rule (for readability issues, we define substitution below):

$$\lambda x.M =_\alpha \lambda y.M[x := y], \text{ where } y \text{ does not occur in } M$$

We take terms modulo α-conversion.

The substitution of the free occurrences of a x by N in M, denoted $M[x := N]$, is defined by recursion on M as follows:

$$\begin{aligned} x[y := M] &= \begin{cases} M, \text{if } x = y \\ x, \ \text{otherwise} \end{cases} \\ (\lambda x.N)[y := M] &= \lambda z.N[x := z][y := M], \text{ if } z \notin \mathsf{fv}(\lambda x.N) \cup \mathsf{fv}(y) \cup \mathsf{fv}(M) \\ (N_1 N_2)[y := M] &= N_1[y := M]N_2[y := M] \end{aligned}$$

We let $M[x_1 := N_1, \ldots, x_n := N_n]$ be the simultaneous substitution of N_i for all free occurrences of x_i in M for $i \in \{1, \ldots, n\}$.

The term set Λ_I, which is a subset of Λ, is defined as follows: for each $x \in \mathsf{Var}$, x is in Λ_I, if $x \in \mathsf{fv}(M)$ and $M \in \Lambda_I$ then $(\lambda x.M)$ is in Λ_I and if $M, N \in \Lambda_I$ then (MN) is in Λ_I.

2.3.2 Reduction relations

The β-reduction, i.e., the binary relation $\rightarrow_\beta$, is the main evaluation process of the λ-calculus. It is defined as the compatible closure of the following rule:

$$(\beta) : (\lambda x.M)N \rightarrow_\beta M[x := N]$$

The βI-reduction, i.e., the binary relation $\rightarrow_{\beta I}$, is a restriction of the β-reduction defined as the compatible closure of the following rule:

$$(\beta I) : (\lambda x.M)N \rightarrow_{\beta I} M[x := N], \text{ where } x \in \mathsf{fv}(M)$$

The h-reduction, i.e., the binary relation $\rightarrow_h$, is also a restriction of the β-reduction defined as the least relation closed by rule (app_2) (defined in Fig. 2.1) and the following rule:

$$(h) : (\lambda x.M)N \rightarrow_h M[x := N]$$

This reduction is called the weak head reduction.

The η-reduction, i.e., the binary relation $\rightarrow_\eta$ is defined as the compatible closure of the following rule:

$$(\eta) : \lambda x.Mx \rightarrow_\eta M, \text{ where } x \notin \mathsf{fv}(M)$$

This reduction expresses the concept of extensionality in the λ-calculus (see Barendregt's book [5]). The idea behind the η-reduction is that $\lambda x.Mx$ where $x \notin \mathsf{fv}(M)$ and M are computationally equivalent in the sense that they compute the same result when applied to the same argument.

The $\beta\eta$-reduction, denoted $\rightarrow_{\beta\eta}$, is defined as the relation: $\rightarrow_\beta \cup \rightarrow_\eta$.

For $r \in \{\beta, \beta I, h, \eta\}$, the term on the left-hand-side of the rule (r) is called a *r-redex* (or just redex when no ambiguity arises) and the one on the right-hand-side is called *r-contractum* (or just contractum when no ambiguity arises). Note that βI-redexes and h-redexes are β-redexes. A $\beta\eta$-redex is either a β-redex or an η-redex (and similarly for $\beta\eta$-contractums).

Note that the relation $\rightarrow_{\beta I}$ is a subset of the relation $\rightarrow_\beta$. Let $r \in \{\beta, \beta I, h\}$. If $(\lambda x.M)N \rightarrow_r M[x := N]$ and $x \in \mathsf{fv}(M)$ then $(\lambda x.M)N$ is called a I-redex, otherwise it is called a K-redex. Therefore, βI-redexes are all I-redexes.

Let $r \in \{\beta, \beta I, h, \eta, \beta\eta\}$. We define the equivalence relation $=_r$ as the symmetric, reflexive (w.r.t. Λ) and transitive closure of the following rule:

$$M =_r N \quad \text{if} \quad M \rightarrow_r N$$

We use $\rightarrow_r^*$ to denote the reflexive (w.r.t. Λ) and transitive closure (rules (refl)

and (tr) as defined in Fig. 2.1) of $\rightarrow_r$. If $M \rightarrow_r^* N$ then we say that M reduces to N or that there is a r-reduction from M to N. Also, N is called a *reduct* of M. If the r-reduction from M to N is in k steps, i.e., if there exists $M_1, \ldots, M_k$ such that $M \rightarrow_r M_1 \rightarrow_r \cdots \rightarrow_r M_k$ and $M_k = N$, we write $M \rightarrow_r^k N$. A term $(\lambda x.M')N'$ is a *direct r-reduct* of $(\lambda x.M)N$ iff $M \rightarrow_r^* M'$ and $N \rightarrow_r^* N'$.

2.3.3 Important λ-calculi

The theory $\boldsymbol{\lambda}$ consists of the equations $M = N$ between λ-terms such that $M =_\beta N$.

The λI-calculus is defined in different ways in the literature. It is defined by Church [21] on the term set Λ and the reduction relation $\rightarrow_{\beta I}$[2]. It is defined by Barendregt [5] on the term set Λ_I and the reduction $\rightarrow_{\beta I}$[3]. We could also consider the term set Λ_I and the reduction $\rightarrow_\beta$. The three corresponding theories are equivalent, and are all called $\boldsymbol{\lambda I}$.

The $\lambda\eta$-calculus is defined on the term set Λ and the $\rightarrow_{\beta\eta}$ reduction relation. The corresponding theory is called $\boldsymbol{\lambda\eta}$. This theory is built upon the λ-terms and the equivalence relation stemming from the $\beta\eta$-reduction, i.e., the relation $=_{\beta\eta}$. When considering the $\beta\eta$-reduction without ambiguity, we sometimes write λ-calculus instead of $\lambda\eta$-calculus.

2.3.4 Residuals, developments, confluence and normalisation

A *β-residual* of a β-redex is an occurrence of the propagation of the redex through a β-reduction (it is defined, e.g, by Barendregt [5, Def. 11.2.4]). For instance the two occurrences of $(\lambda x.x)y$ in $((\lambda x.x)y)((\lambda x.x)y)$ are residuals of the redex $(\lambda x.x)((\lambda x.x)y)$ in $(\lambda x.xx)((\lambda x.x)((\lambda x.x)y))$ w.r.t. the following reduction:

$$(\lambda x.xx)((\lambda x.x)((\lambda x.x)y)) \rightarrow_\beta (\lambda x.xx)((\lambda x.x)y) \rightarrow_\beta ((\lambda x.x)y)((\lambda x.x)y)$$

Although, to the best of our knowledge the definition of β-residuals is a well established concept, it does not seem to be the case for $\beta\eta$-residuals. Different definitions can be found in the literature: the $\beta\eta$-residuals as defined by Curry and Feys [31] or the λ-residuals as defined by Klop [92].

A *development* is the reduction of an initial set of redexes in a term and of its residuals w.r.t. the reduction. A development is said to be complete if all the redexes of the initial set of redexes and their residuals have been reduced.

The *confluence* property is detailed below in Sec. 3. Let us mention here that it is a property satisfied by the λ-calculus (w.r.t. the β-reduction) which states that if

[2] Church [21] defines abstractions as follows: "if $\mathbf{x}$ is a variable and $\mathbf{M}$ is well-formed then $\lambda\mathbf{x}[\mathbf{M}]$ is well-formed".

[3] Barendregt [5] defines the theory $\boldsymbol{\lambda I}$ as follows: "The theory $\boldsymbol{\lambda I}$ ("the $\boldsymbol{\lambda I}$-calculus") consists of equations between λI-terms provable by the axioms and rules of $\boldsymbol{\lambda}$ restricted to Λ_I."

a term reduces to two different terms then these two terms can reduce to the same term, i.e., for each M_1, if $M_1 \rightarrow^*_\beta M_2$ and $M_1 \rightarrow^*_\beta M_3$ then there exists M_4 such that $M_2 \rightarrow^*_\beta M_4$ and $M_3 \rightarrow^*_\beta M_4$. Developments have often been used to prove the confluence of the λ-calculus. The confluence of the λ-calculus was first proved by Church and Rosser in 1936 [24]. Therefore, this property is often referred to as the the Church-Rosser property and will sometimes be abbreviated as CR in this thesis.

A term is a *normal form* if it cannot be reduced further. Normal forms w.r.t. the β-reduction are of the following form: $\lambda x_1. \dots \lambda x_m.yM_1 \dots M_n$ where $n, m \geq 0$ and where each M_i is a normal form. We say that a term M is *weakly normalisable* (abbreviated as WN) if there exists a reduction from M to a normal form. We say that a term M is *strongly normalisable* (abbreviated as SN) if each reduction starting from M terminates in a normal form. The strong normalisation property is sometimes referred to in the literature as the termination property. The confluence of the λ-calculus was originally used to prove the uniqueness of normal forms [24].

2.4 Some notable typed λ-calculi

To avoid introducing too many notations, in this section we reuse some metavariables to range over different sets in different subsections. For example, σ is defined in Sec. 2.4.1 to range over simple types, in Sec. 2.4.2 to range over intersection types, and in Sec. 2.4.3 to range over type schemes. In order to avoid any confusion, when reused outside these sections, we will specify from which system they are taken from.

Throughout this document we follow Carlier and Wells [20] and write type judgements as $M : \langle \Gamma \vdash U \rangle$, where Γ is a type environment and U a type, instead of $\Gamma \vdash M : U$ (meaning that the triple $\langle M, \Gamma, U \rangle$ belongs to the typing relation $\vdash$).

2.4.1 The simply typed λ-calculus

Russell [121] first introduced types to avoid paradoxes in his formal system. Russell type theory enforced a hierarchy of types that precludes the self-reference issue to occur. Types are nowadays largely used in programming languages to, e.g., ensure a certain “safety” property on programs. For example, often one wishes to forbid a function on integers to be applied to, say, a string, because among other things the application does not have a well defined meaning. Therefore, types can then be used, among other things, to restrict the application of functions. As mentioned above, type systems have several advantages, such as efficiency or modularity.

One of the notable type systems that followed Russell's idea of using types to avoid the self-reference issue was the simply typed λ-calculus. Church writes [23]: “The simple theory of types was suggested as a modification of Russell's ramified theory of types by Leon Chwistek in 1921 and 1922 and by F. P. Ramsey in 1926”.

Church [23] provides his own “formulation of the simple theory of types” based on the λ-calculus. This formulation is nowadays one of the two widely known formulations along with Curry’s one. Let us first focus on Church’s version of the simply typed λ-calculus.

Church [23] defines two ground types ι and o where ι is said to be the type of individuals and o the type of propositions. Moreover, if σ and τ are types then $\sigma{\rightarrow}\tau$ is a type. Church uses α and β to range over simple types, but we shall not use his notation because of the use of α and β in conversion rule names. Moreover, Church writes $(\sigma\tau)$ instead of $\sigma{\rightarrow}\tau$. Once again we do not use his notation, but instead use the more common arrow notation. Then, Church defines his typed λ-calculus by defining a well-formedness relation on typed formulae. Along with this well-formedness relation, Church defines a notion of type assignment. A subset of the well-formed formulae (Church also considers extra typed constants for negation, conjunction and universal quantification) is as follows: each typed variable x_σ is well-formed and has type σ, if M is well-formed and has type τ then $\lambda x_\sigma.M$ is well-formed and has type $\sigma{\rightarrow}\tau$, and if M is well-formed and has type $\sigma{\rightarrow}\tau$ and N is well-formed and has type σ then MN is well-formed and has type τ.

Let us now present Curry’s version of the simply typed λ-calculus but in the λ-calculus setting as presented by Barendregt [6] rather than in the combinatory logic setting. First, let us define the set $\mathsf{SimpleTy}$ of simple types and the set $\mathsf{SimpleTyEnv}$ of simple type environments as follows:

$$\begin{array}{lll} a & \in \mathsf{TyVar} & \text{(countable infinite set of type variables)} \\ \sigma,\tau & \in \mathsf{SimpleTy} & ::= a \mid \sigma{\rightarrow}\tau \\ \Gamma & \in \mathsf{SimpleTyEnv} & = \mathsf{Var} \rightarrow \mathsf{SimpleTy} \end{array}$$

The simply typed λ-calculus à la Curry can then be defined as the binary relation $\vdash_{\rightarrow}$ which is the smallest relation closed by the following rules:

$$\frac{\Gamma(x) = \sigma}{x \vdash_{\rightarrow} \langle \Gamma, \sigma \rangle} \qquad \frac{M \vdash_{\rightarrow} \langle \Gamma, \sigma{\rightarrow}\tau \rangle \quad N \vdash_{\rightarrow} \langle \Gamma, \sigma \rangle}{MN \vdash_{\rightarrow} \langle \Gamma, \tau \rangle} \qquad \frac{M \vdash_{\rightarrow} \langle \Gamma \cup \{x \mapsto \sigma\}, \tau \rangle}{\lambda x.M \vdash_{\rightarrow} \langle \Gamma, \sigma{\rightarrow}\tau \rangle}$$

The simply typed λ-calculus satisfies CR and SN [5], and is denoted $\lambda_{\rightarrow}$.

2.4.2 Intersection type systems

Coppo and Dezani [26] introduced intersection type systems to type more terms than in the simply typed λ-calculus and to characterise normalisable terms. Pottinger [117] was the first to achieve such a characterisation. The word “intersection” in “intersection type” comes from the fact that, if types are interpreted by sets (a set-theoretical semantics), usually, an intersection type is interpreted by the intersection of sets. The authors proved that each typable term in their system is normalisable (in WN) and that the normalisable terms of the λI-calculus all have a type in their system. Also, their system restricted to the λI-calculus satisfies subject reduction and expansion (βI-equivalent terms can be typed with the same type). Without this restriction their system satisfies only subject reduction (if a term is typable in their systems then all the reducts of this terms are typable with the same type). Coppo, Dezani and Venneri [28] defined another intersection type system that we shall call CDV[4] which satisfied both subject reduction and expansion w.r.t. the β-reduction. They also obtain a characterisation of the normalisable terms (in WN) in their system. Similarly, Krivine [96] characterises the strongly normalisable terms by the terms typable in his system $\mathcal{D}$ and characterises the weakly normalisable terms by a subset of the terms typable in his system $\mathcal{D}\Omega$.

Let $\sqcap$ be the intersection type constructor. Intuitively, if a term M can be assigned a type $\sigma \sqcap \tau$ then it can usually be assigned the type σ as well as the type τ. An intersection type can be seen as a list of types that can be assigned to a term. They are used to express a finitary kind of polymorphism where types (usages of terms) are listed rather than obtained via quantification. For example, a program of type $(\sigma \rightarrow \sigma) \sqcap (\tau \rightarrow \tau)$ can be a program computing a term of type σ from a term of type σ as well as a program computing a term of type τ from a term of type τ. The same code can be used for the two types $\sigma \rightarrow \sigma$ and $\tau \rightarrow \tau$. The polymorphism of an intersection type is said to be *finitary* as opposed to the *infinitary parametric* polymorphism [124, 17] supported by *for all* type schemes such as in system F [49, 50], because a program to which is assigned an intersection type works “uniformly” (the same code is used for different types) on the finite list of types given by the intersection. These kinds of polymorphism contrasts with the “ad-hoc” polymorphism which is, e.g., the polymorphism of overloading (e.g., given an overloaded operator, different functions might be used for different types on which the operator is overloaded). The universal quantifier “∀” is well known to express polymorphism as in system F designed by Girard [49, 50]. As explained by Carlier and Wells [20] there are many advantages in using intersection types over the ∀ quantifier, such as:

- Urzyczyn [137, Theorem 3.1], found a term which is not typable in the system F_ω: $(\lambda x.z(x(\lambda f.\lambda u.fu))(x(\lambda v.\lambda g.gv)))(\lambda y.yyy)$ but which turns to be typable in the rank-3[5] restriction of intersection types.
- Wells [142] proved that type inference in system F is undecidable. Kfoury and Wells [88] defined an intersection type system for which every finite-rank restriction has a decidable type inference.

[4] Coppo, Dezani and Venneri presented in the same paper [28] two different type systems, the second one being a restriction of the first one. Their second system is similar to the one of their earlier system [27]. Sometimes CDV is used to refer to their first system [4] and sometimes to refer to their second system [20]. We shall refer to CDV as their first system.

[5] The notion of rank is, e.g., explained by Carlier and Wells [20].

- Wells [143] proved that system F does not have principal typings[6] for all terms. Kfoury and Wells [88] proved that every finite-rank restriction of their intersection type system has principal typings.

Since Coppo and Dezani first intersection type system, many other intersection type systems have been designed. Barendregt, Coppo, and Dezani [8] designed the BCD intersection type system, proposed a term and type interpretations where terms are interpreted in λ-models [73], and proved the soundness and completeness of their semantics w.r.t. the BCD system. These two results allows them to obtain that the interpretation of a term is in the interpretation of a type iff the term is typable by the type in BCD. Their proof is based on the construction of a particular model of the λ-calculus called filter model where filters are type sets closed under some rules such that intersection introduction, i.e., if σ and τ are types in a filter then $\sigma \cap \tau$ has to be in the filter as well, where $\cap$ is their notation for the intersection type constructor. They prove that their filter model is a λ-model. Hindley [69] proved a similar result but using a term models which interprets terms by terms.

Some intersection type systems involve a constant type often written ω as a 0-ary version of the intersection types. This type expresses a universality in the sense that this type does not contain any information. When types are interpreted by subsets of a certain set (the domain of the model), this type is usually interpreted by the universe of discourse (the whole domain itself).

Let us present Krivine's system $\mathcal{D}$ [96]. We will use a slightly different notation. For example, Krivine uses $\wedge$ as the intersection type constructor. We use the symbol $\sqcap$ instead. The set TyVar of type variables is the same as in Sec. 2.4.1. First, let us define the set $\mathsf{InterTy}$ of intersection types and the set $\mathsf{InterTyEnv}$ of intersection type environments as follows:

$$\begin{array}{lll} \sigma, \tau \in \mathsf{InterTy} & ::= a \mid \sigma \to \tau \mid \sigma \sqcap \tau \\ \Gamma \quad \in \mathsf{InterTyEnv} & = \mathsf{Var} \to \mathsf{InterTy} \end{array}$$

The intersection type system $\mathcal{D}$ can be defined as the binary relation $\vdash_{\mathcal{D}}$ which is the smallest relation closed by the following rules:

$$\frac{\Gamma(x) = \sigma}{x \vdash_{\mathcal{D}} \langle \Gamma, \sigma \rangle} \qquad \frac{M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \to \tau \rangle \quad N \vdash_{\mathcal{D}} \langle \Gamma, \sigma \rangle}{MN \vdash_{\mathcal{D}} \langle \Gamma, \tau \rangle} \qquad \frac{M \vdash_{\mathcal{D}} \langle \Gamma \uplus \{x \mapsto \sigma\}, \tau \rangle}{\lambda x.M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \to \tau \rangle}$$

$$\frac{M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \rangle}{M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \sqcap \tau \rangle} \qquad \frac{M \vdash_{\mathcal{D}} \langle \Gamma, \tau \rangle}{M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \sqcap \tau \rangle} \qquad \frac{M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \rangle \quad M \vdash_{\mathcal{D}} \langle \Gamma, \tau \rangle}{M \vdash_{\mathcal{D}} \langle \Gamma, \sigma \sqcap \tau \rangle}$$

[6] Wells [143] explains that "a typing t is defined to be principal in some system S for program fragment M if and only if t is at least as strong as all other typings for M in S, where a typing t_1 is defined to be stronger than typing t_2 if and only if the set of terms that can be assigned t_1 in S is a subset of the set of terms that can be assigned t_2 in S".

2.4.3 ML-like programming languages

ML is a higher-order impure functional programming language[7] originally designed, as part of a proof system called LCF (Logic for Computable Functions), to perform proofs of facts within PPλ (Polymorphic Predicate λ-calculus), a formal logical system [52, 53]. ML is a typed programming language based on the λ-calculus with let-expressions which allow one to generate local bindings. Let-expressions are usually more or less of the form `let x = exp1 in exp2` where `exp1` and `exp2` are expressions. Such a let-expression binds `x` to `exp1` in `exp2`. Nowadays ML is used to refer to a collection of programming languages which share common features, such as SML or Caml. As explained by Milner et al., Standard ML (SML) [106, 107] is the result of the re-design and extension of ML. SML has formally defined static and dynamic semantics [106, 107]. Also, SML (and similar programming languages such as OCaml, Haskell, etc.) has polymorphic types allowing considerable flexibility, and almost fully automatic type inference, which frees the programmer from writing many explicit types. We say "almost fully" because some explicit types are required in SML, e.g., as part of datatype definitions, module types, and type annotations sometimes needed in special circumstances[8]. Milner's W algorithm [32] is the original type-checking algorithm of the functional language core ML, which is the λ-calculus extended with polymorphic let-expressions. Given an expression e and a type environment Γ covering the free variables of e, if e is typable then W outputs a type σ of e and a substitution sub. The type σ is the principal type of e w.r.t. the application of sub to Γ. If e is not typable, an error is reported.

Let us now present Damas and Milner's type system [32, 33], also known as the Hindley-Milner type system and therefore called HM. First we define the set of terms of core ML as follows:

$$e \in \mathsf{MLExp} ::= x \mid (\lambda x.e) \mid (e_1\, e_2) \mid (\texttt{let}\ x = e_1\ \texttt{in}\ e_2)$$

The set TyVar of type variables is the same as in Sec. 2.4.1. Let us now define the set HMTy of simple types, the set HMTyScheme of type schemes, and the set HMTyEnv of type environments as follows:

$$\begin{array}{lll}
\iota \in \mathsf{PrimitiveTy} & & \text{(countable infinite set of primitive types)} \\
\tau \in \mathsf{HMTy} & ::= & a \mid \iota \mid \tau_1 \rightarrow \tau_2 \\
\sigma \in \mathsf{HMTyScheme} & ::= & \tau \mid \forall a.\, \sigma \\
\Gamma \in \mathsf{HMTyEnv} & = & \mathsf{Var} \rightarrow \mathsf{HMTyScheme}
\end{array}$$

[7] ML has functional as well as imperative programming features: functions are first-class objects and expressions can have side effects (e.g., references, exceptions). Therefore, we say that ML is an imperative functional-like programming language, or an impure functional programming language.

[8] Explicit types are sometimes required, e.g., for "flexible" record patterns as in the function `fn {x,...} => x`, which would be used to select a field named `x` in any record that contains at least a field named `x`.

Damas and Milner write $\forall a_1 \cdots a_n.\, \tau$ for the type scheme $\forall a_1. \cdots \forall a_n.\, \tau$. They also define the relation $>$ on type schemes as follows: $\sigma > \sigma'$ iff $\sigma = \forall a_1 \cdots a_n.\, \tau$ and $\sigma' = \forall a'_1 \cdots a'_m.\, \tau'$ and $\tau' = [\tau_i/a_i]\tau$ for some types $\tau_1, \ldots, \tau_n$ and the a'_i do not occur free in σ, where $[\tau_i/a_i]\tau$ is Damas and Milner's notation for the simultaneous substitution of each occurrences of a_i by τ_i, for $i \in \{1, \ldots, n\}$, in τ. Damas and Milner call σ', a *generic instance* of σ.

The HM type system can be defined as the binary relation $\vdash_{\mathsf{HM}}$ which is the smallest relation closed by the following rules:

$$\frac{\Gamma(x) = \sigma}{x \vdash_{\mathsf{HM}} \langle \Gamma, \sigma \rangle}\ (\mathsf{TAUT}) \qquad \frac{e \vdash_{\mathsf{HM}} \langle \Gamma, \sigma \rangle \quad a \text{ does not occur free in } \Gamma}{e \vdash_{\mathsf{HM}} \langle \Gamma, \forall a.\, \sigma \rangle}\ (\mathsf{GEN})$$

$$\frac{e \vdash_{\mathsf{HM}} \langle \Gamma, \sigma_1 \rangle \quad \sigma_1 > \sigma_2}{e \vdash_{\mathsf{HM}} \langle \Gamma, \sigma_2 \rangle}\ (\mathsf{INST}) \qquad \frac{e_1 \vdash_{\mathsf{HM}} \langle \Gamma, \tau_1 \to \tau_2 \rangle \quad e_2 \vdash_{\mathsf{HM}} \langle \Gamma, \tau_1 \rangle}{e_1\, e_2 \vdash_{\mathsf{HM}} \langle \Gamma, \tau_2 \rangle}\ (\mathsf{COMB})$$

$$\frac{e \vdash_{\mathsf{HM}} \langle \Gamma + \{x \mapsto \tau_1\}, \tau_2 \rangle}{\lambda x.e \vdash_{\mathsf{HM}} \langle \Gamma, \tau_1 \to \tau_2 \rangle}\ (\mathsf{ABS}) \qquad \frac{e_1 \vdash_{\mathsf{HM}} \langle \Gamma, \sigma \rangle \quad e_2 \vdash_{\mathsf{HM}} \langle \Gamma + \{x \mapsto \sigma\}, \tau \rangle}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \vdash_{\mathsf{HM}} \langle \Gamma, \tau \rangle}\ (\mathsf{LET})$$

2.5 Some methods of reasoning involving λ-calculi

In this section we discuss two closely related methods of reasoning involving λ-calculi (or similar functional systems): realisability which is a method originally developed to provide semantics to intuitionistic systems dealing with arithmetic, and reducibility which is a semantic method based on type interpretation to prove the normalisation of functional theories.

2.5.1 Realisability

Kleene's original realisability method [89] was a "systematic method of making the constructive content of arithmetical sentences explicit" [135]. His method associates Gödel numbers of partial recursive functions with sentences of the first order intuitionistic arithmetic. This system is Heyting arithmetic, often referred to as the theory HA which is the intuitionistic predicate logic with equality, natural numbers, and the primitive recursive functions [135, Ch. 3]. Informally, there exists a Gödel number of a recursive function that realises a formula if the formula is true in HA. Such a number is called a realiser and can be seen as "a witness for the constructive truth" [74] of the realised formula. For example, Kleene [89] defines, among other things, that "If *a realizes* A, then $2^0 \cdot 3^a$ *realizes* $A \vee B$. Also, if *b realizes* B, then $2^1 \cdot 3^b$ *realizes* $A \vee B$", where A and B are closed formulae and where $\cdot$ is the multiplication function on natural numbers. A realizer of a disjunction encodes the information that for a disjunction $A \vee B$ to be true one has to either be able to provide a proof of A or a proof of B. Implications are realised as follows: "The formula $A \supset B$ is *realized* by the Gödel number e of a partial recursive function ϕ such that, whenever *a realizes* A then $\phi(a)$ *realizes* B" [89], where A and B are closed formulae and where $\supset$ is the logical implication symbol. Van Oosten [113] explains that Kleene "wished to give some precise meaning to the intuition that there should be a connection between Intuitionism and the theory of recursive functions". However, Rose [119] disproved Kleene's intuition that realisability mirrors intuitionistic reasoning. Realisability was found useful, among other things, "for proving underivability and relative consistency results of intuitionistic formal systems" [38].

Variants of Kleene's realisability, often referred to as "recursive" or "numerical" realisability, have been developed throughout the years. Kreisel's *modified* realisability [95] is such a variant. Asperti and Tassi [3] explain that modified realisability is a variant of Kleene's realisability "essentially providing interpretations of HA^ω into itself". Van Oosten [113] explains that "HA^ω is "Gödel's T with predicate logic"". Gödel's system T can be regarded as an extension of the simply typed λ-calculus with natural numbers and recursion. Asperti and Tassi add that with the modified realisability interpretations "each theorem is realized by a typed function of system T". For example, "if the type of realizers of A is σ, and the type of realizers of B is τ, the type of realizers of $A \to B$ is $(\sigma \Rightarrow \tau)$" [113], where $\Rightarrow$ is the functional type constructor.

Kreisel was not the only one interested in realisability and nowadays there exist many notions of realisability used in various areas. Van Oosten [113] writes about realisability: "Quite apart from the huge amount of literature to cover, there is the task of creating unity where there is none. For Realizability has many faces, each of them turned towards different areas of Logic, Mathematics and Computer Science". Similarly, Hofstra [75] writes: "In the area of research known as realizability, we have the interesting phenomenon that there are many different realizability definitions, but no definition of realizability. What this means is that we have many instances of realizability interpretations [..] but that there is no clear answer to the question of what constitutes a notion of realizability."[9]

Realisability in general is closely related to the Curry-Howard isomorphism. Sørensen and Urzyczyn [123] write (where "this interpretation" refers to Kleene's realisability semantics): "One can see the Curry-Howard isomorphism [..] as a syntactic reflection of this interpretation. It shows that a certain notation system for denoting certain recursive functions coincides with a system for expressing proofs."

2.5.2 Reducibility

Reducibility is a method based on realisability semantics [89], developed by Tait [130] in order to prove the normalisation of some functional theories. The idea of Tait's reducibility method is to interpret types by λ-term sets closed under some properties. Since its introduction, this method has gone through a number of improvements and

[9] We use "[..]" in quotes to show that parts of citations have been omitted.

generalisations to prove properties of the λ-calculus and to characterise properties of the λ-calculus w.r.t. type systems. For example, Girard [50] designed a similar method based on *reducibility candidates* which are sets of λ-terms satisfying some properties. Also, Krivine [96] uses reducibility to prove the strong normalisation of the terms of his intersection type system called system $\mathcal{D}$. Koletsos [93] uses reducibility to prove that the set of simply typed λ-terms satisfies CR w.r.t. β-reduction (for more details on CR see Sec. 3.1 and for more details on Koletsos' proof see Sec. 3.6). Gallier [44, 43, 45, 46] also uses reducibility to, e.g., characterise sets of λ-terms closed under some properties in terms of typability in type systems such as the intersection type system $\mathcal{D}$. Although it is well known that β-reduction satisfies CR, reducibility proofs of CR are in line with proofs of SN and hence, one can establish both SN and CR for some calculus using the same method.

2.6 Contributions and structure of this thesis

The present thesis is composed of three parts all revolving around intersection type systems and the study of some of their aspects. Part I emerged from the study of intersection type systems to prove properties of the untyped λ-calculus. Part II constitutes a study of the semantics of intersection type systems. Part III evolved from a system using intersection types as a tool for doing type error reporting and type inference. Let us now detail each of the three parts and their contributions.

Part I is based on a publication by Kamareddine and Rahli [84]. It presents two proofs of the confluence of the λ-calculus using a purely syntactic method, i.e., not based on type interpretations. These two proofs share the same proof scheme. The first proof is w.r.t. β-reduction and the second one is w.r.t. $\beta\eta$-reduction. These two syntactic proofs are derived from a semantic one based on sound type interpretation w.r.t. an intersection type system. Various simplifications to the original method led to the simplification of the considered type system and finally to its discarding. It turned out that in this case intersection types constitute a powerful tool unnecessary to prove the confluence of the λ-calculus: only a small portion of the initially considered intersection type system was necessary to prove the confluence of the λ-calculus.

Part II is based on three papers by Kamareddine, Nour, Rahli, and Wells: a workshop paper [83], a conference paper [82] and a journal paper submitted to Fundamenta Informaticae [81]. It presents a complete realisability semantics w.r.t. a type system with infinite number of expansion variables. It also describes the steps that led us to this semantics. Expansion is a powerful operations on typings in type systems for the λ-calculus. Unfortunately, to the best of our knowledge, there has been no study of semantics of intersection type systems with expansion. Our semantics provides a first step in the study of the semantics of intersection types with expansion and therefore in the study of the semantics of expansion.

Part III is based on a technical report by Rahli, Wells and Kamareddine [118]. It presents a type error slicer (TES) for the SML language. Modern programming languages such as SML, Haskell, or OCaml rely on type systems which allow (almost fully) automatic type inference, freeing programmers from explicitly writing types. Also, these type inference algorithms allow one to detect some programming errors at an early stage (at compile-time). As a matter of fact, types are used to automatically check the well-defined behaviour of pieces of code, for a certain notion of behaviour. Unfortunately, it is well known that type error reports provided by compilers for higher-order programming languages such as SML can be intricate. An issue being that programmers tend to lose their time by trying to decipher type error reports and by manually tracking down their type errors. TES helps the programmer by isolating the part of an ill-typed program contributing to a type error (a slice). The presentation of our TES is divided into two major parts. In a first part, we present a core of our TES. We present a new, original, and simple constraint language and its use in a type error slicer for a small subset of SML which contains interesting core and module features such that datatypes and `open` declarations. In a second part we present other interesting features of our TES necessary to handle more of the SML programming language, such as some signatures and functors. We also discuss issues w.r.t. the implementation of our TES. Concerning this part, we have achieved both: (1) the formalisation of a type error slicer for SML which handles many interesting features of the language; (2) and an implementation of our TES which handles most of the SML language. Note that the first version of TES developed by Haack and Wells [56, 57] for a tiny core language (the λ-calculus augmented with polymorphic let-expressions) made use of intersection types. It turned out that their system was not scalable on real size programs. To solve this issue, we have moved on to a TES that makes use of *for all* type schemes instead of intersection types. Interestingly, one of our latest innovation was to reintroduce the use of intersection types in order to handle SML's functors.

These three parts are not presented in chronological order. The first project we have carried out was the study of a semantics of expansion. We have then developed a proof method to prove the confluence of the λ-calculus. This was part of a larger project aiming at studying general methods to prove properties of the λ-calculus using reducibility. Last but not least, we have developed a type error slicer for the SML language. This last project represents the major contribution to the present document. The three parts do not rely on one another. These three parts are presented in an incremental complexity order. Part I concerns only the untyped λ-calculus. In Part II we add types to the untyped λ-calculus. We consider intersection types. Finally, in Part III we consider a more complicated polymorphic type system: a variant of a portion of SML's type system.

Part I

A new proof method of the confluence of the λ-calculus

Chapter 3

The confluence property and its main proofs

3.1 Confluence

The confluence property is a property satisfied by the λ-calculus stating that if $M_1 =_\beta M_2$ then there exists M_3 such that $M_1 \to_\beta^* M_3$ and $M_2 \to_\beta^* M_3$. It can equivalently be defined as follows: if $M_1 \to_\beta^* M_2$ and $M_1 \to_\beta^* M_3$ then there exists M_4 such that $M_2 \to_\beta^* M_4$ and $M_3 \to_\beta^* M_4$. Confluence is not restricted to the λ-calculus and can be more generally defined in the term rewriting systems setting [10]. We will however restrict ourselves to the context of the λ-calculus. The confluence of the λ-calculus (w.r.t. the β-reduction) was first proved by Church and Rosser [24], and is therefore often referred to as the Church-Rosser property. We will use the terms confluence and Church-Rosser without distinction.

Confluence is also satisfied when considering $\beta\eta$-reduction instead of β-reduction.

Given a binary relation r on terms, if whenever $M_1 \to_r^* M_2$ and $M_1 \to_r^* M_3$, there exists M_4 such that $M_2 \to_r^* M_4$ and $M_3 \to_r^* M_4$, then we say that M_1 satisfies or has the Church-Rosser property. We also sometimes write that M_1 has r-CR. We define $\mathsf{CR}^r = \{M \mid M \text{ has } r\text{-CR}\}$. Let $\mathsf{CR} = \mathsf{CR}^\beta$.

Confluence was among other things used to prove the consistency of the λ-calculus and the uniqueness of normal forms as first proved by Church [22]. This property has been extensively studied in the literature since its first proof. We describe below some of its proofs. First, we show how it allows one to prove the consistency of the λ-calculus.

3.2 Consistency

To the best of our knowledge, Church was the first one to provide a proof of the consistency of the λ-calculus in 1935 [22]. Church considers the λI-calculus augmented with a special symbol δ which is used in his paper as an equality test (a conditional). Church considers a rule for α-conversion, two rules for β-conversion and four rules related to the equality test. Church defines substitution as follows: "The expression $\mathsf{S}^x_N M$ is used to stand for the result of substituting N for x throughout M". Church's seven conversion rules are stated as follows (in these rules we use the syntax of λ-terms as presented in Sec. 2.3.1 instead of using Church's notation):

I To replace any part $\lambda x.R$ by $\lambda y.\mathsf{S}^x_y R$, where y is any variable which does not occur in R.

II To replace any part $(\lambda x.M)N$[1] of a formula by $\mathsf{S}^x_N M$, provided that the bound variables in M are distinct both from x and from the free variables in N.

III To replace any part $\mathsf{S}^x_N M$ (not immediately following λ) of a formula by $(\lambda x.M)N$, provided that the bound variables in M are distinct both from x and from the free variables in N.

IV To replace any part $\delta(M, N)$ of a formula by $\lambda f.\lambda x.f(fx)$[2], where M and N are in normal form and contain no free variables and M conv-I N[3].

V To replace any part $\delta(M, N)$ of a formula by $\lambda f.\lambda x.fx$[4], where M and N are in normal form and contain no free variables and it is not true that M conv-I N.

VI To replace any part $\lambda f.\lambda x.f(fx)$ of a formula by $\delta(M, N)$, where M and N are in normal form and contain no free variables and M conv-I N.

VII To replace any part $\lambda f.\lambda x.fx$ of a formula by $\delta(M, N)$, where M and N are in normal form and contain no free variables and it is not true that M conv-I N.

Then Church defines an encoding of the natural numbers (except 0, because Church considers a variant of the λI-calculus) into the λ-calculus. He chooses $\lambda f.\lambda x.fx$ to stand for 1, $\lambda f.\lambda x.f(fx)$ for 2, etc. As a matter of fact, the natural numbers are defined as abbreviations for the corresponding λ-terms and used as such below. Note that $\lambda f.\lambda x.x$ usually stands for 0 but this term is not a λI-term. Note also that Church uses a slightly different notation than the one defined in Sec 2.3.1. For example, we write $\lambda f.\lambda x.fx$ when Church writes $\lambda fx.f(x)$.

The first rule (rule I) corresponds to the α-conversion rule. The second rule (rule II) corresponds to the β-reduction. The third rule (rule III) corresponds to the

[1] Church writes $(\lambda x.M)N$ as $\{\lambda x.M\}(N)$.

[2] The term $\lambda f.\lambda x.f(fx)$ is the Church numeral 2.

[3] Church defines M conv-I N as follows: "We are using the notation M conv-I N to mean that N is obtainable from M by a sequence of applications of Rule I.", which is to check whether that two expressions are α-convertible.

[4] The term $\lambda f.\lambda x.fx$ is the Church numeral 1.

β-extension which is the inverse of the β-reduction relation. The fourth and fifth rules (rule IV and V) are to check whether two terms in normal forms are equivalent modulo α-conversion. If two normal terms are equivalent modulo α-conversion then δ is used to derive 2's encoding. If they are different then δ is used to derive 1's encoding. In Church's formalism, 1 stands for false and 2 stands for true. Church stresses that this choice is arbitrary and that the "viewpoint taken is that formal logic requires nothing of the ideas of *true* and *false* except that they be distinct". The two last rules (rules VI and VII) are the inverse rules of rules IV and V.

Church encodes the logical negation by the term: $\lambda x.6 - [\delta(x, 1) + 2 \times \delta(x, 2)]$, denoted by $\sim$ and where $-$, $+$, $\times$ are the usual encodings of addition, subtraction and multiplication. He also defines an encoding of conjunction. Note that using Church's encoding of negation one obtains [22, Theorem IV]: ~ 1 reduces to 2, i.e., the negation of false reduces to true; ~ 2 reduces to 1, i.e., the negation of true reduces to false; and $\sim n$, such that $n \geq 3$, reduces to 3 (because only 1 and 2 have a logical content)[5].

Church then proves that "There is no formula P such that both P and $\sim P$ are provable" [22, Therorem VI].

This result is obtained using the Church-Rosser property and because the encodings of 1 and 2 are distinct closed λ-terms.

3.3 1936: Church and Rosser [24]

Church and Rosser aim at proving the following result [24, Theorem 1]:

$$\text{if } M =_{\beta I\alpha} N \text{ then there exists } P \text{ such that } M \rightarrow^*_{\beta I\alpha} P \text{ and } N \rightarrow^*_{\beta I\alpha} P$$

where $=_{\beta I\alpha}$ is $=_{\beta I} \cup =_\alpha$ and $M \rightarrow_{\beta I\alpha} N$ iff $M =_\alpha M'$, $M' \rightarrow_{\beta I} N'$, and $N' =_\alpha N$.

Let us now describes the main lines of Church and Rosser's proof.

Church and Rosser define residuals, developments and complete developments.

Then, they prove the developments' termination as well as the complete developments' confluence [24, Lemma 1]. These two results set the basis to prove the Church-Rosser theorem.

They use another important result [24, Lemma 2] which states among other things that if the reduction of a redex r in A_1 results in B_1, and $A_1 \rightarrow_{\beta I\alpha} A_2 \rightarrow_{\beta I\alpha} A_3 \rightarrow_{\beta I\alpha} \cdots$ (a possibly infinite reduction), and for all k, B_k is the result of a terminating sequence of contractions on the residuals of r in A_k then for all k, $B_k =_{\beta I} B_{k+1}$.

[5] Note that, e.g., the term $\lambda x.\delta(x, 1)$ would not be a suitable encoding of the logical negation because the negation of any natural number greater or equal to 3, which do not have any logical content in Church's formalism, would be convertible to 1 (i.e., false).

They can then state the confluence of the λ-calculus w.r.t. the $\beta I\alpha$-equivalence relation. Proving this theorem consists in replacing the reductions $A_1 \to_{\beta I\alpha} \cdots \to_{\beta I\alpha} A_n$ and $A_1 \to_{\beta I\alpha} B$ ("a peak with a single reduction") by the reductions $A_n \to^*_{\beta I\alpha} C$ and $B \to^*_{\beta I\alpha} C$ ("a valley"). The point being that such a C can always be found.

Based on their first theorem (the confluence theorem), Church and Rosser obtained another important result about normal forms: the uniqueness of the normal forms modulo α-conversion [24, Corollary 2].

The last paragraph of Church and Rosser's paper [24] is devoted to the untyped λ-calculus (and not only the λI-calculus). The same results are claimed to be true as well in this unrestricted setting but no proof is given.

3.4 1972: Tait and Martin-Löf [102, 5, 131]

The famous method developed by Tait and Martin-Löf is based on the *parallel reduction*. A parallel reduction is a new reduction relation based on the β-reduction, denoted $\Rightarrow_\beta$ below, and defined as follows:

- $x \Rightarrow_\beta x$
- $\lambda x.M \Rightarrow_\beta \lambda x.M'$ if $M \Rightarrow_\beta M'$
- $MN \Rightarrow_\beta M'N'$ if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$
- $(\lambda x.M)N \Rightarrow_\beta M'[x := N']$ if $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$

This parallel reduction also provides a definition of developments: $M \Rightarrow_\beta M'$ is a development. Note that because of the two last rules, this reduction leaves the choice whether or not to reduce the occurrence of a redex.

For example, $((\lambda x.x)(\lambda x.x))(\lambda x.x) \Rightarrow_\beta ((\lambda x.x)(\lambda x.x))(\lambda x.x)$ is a parallel reduction, as well as $((\lambda x.x)(\lambda x.x))(\lambda x.x) \Rightarrow_\beta (\lambda x.x)(\lambda x.x)$. However, one cannot reduce $((\lambda x.x)(\lambda x.x))(\lambda x.x)$ to $\lambda x.x$ via a parallel reduction (because $(\lambda x.x)(\lambda x.x)$ is not an abstraction).

This reduction is called "parallel" reduction because if a redex is formed during a reduction, then the redex reduced during the reduction and the redex formed during the reduction cannot both be reduced in a parallel reduction. For example, the redex $(\lambda z.z)y$, is formed during the reduction: $(\lambda x.xy)(\lambda z.z) \to_\beta (\lambda z.z)y$. But one cannot reduce $(\lambda x.xy)(\lambda z.z)$ to y via a parallel reduction.

The Church-Rosser property is then proved to be satisfied w.r.t. this new reduction. This can be proved by an induction on terms or using the complete developments (i.e. a complete parallel reduction where the last rule of the definition of the parallel reduction is used as much as possible). Finally, by proving the equivalence between $\to^*_\beta$ and the transitive closure of $\Rightarrow_\beta$ they prove that the untyped λ-calculus satisfies the Church-Rosser property (w.r.t. the β-reduction).

3.5 1978: Hindley [68]

To the best of our knowledge Hindley was one of the first to provide a proof of the finiteness of developments w.r.t. $\beta\eta$-reduction [68, Sec. 1]. Hindley [68] first starts by giving a proof for the β-reduction (and not only for the βI-reduction as Church and Rosser did [24]). His proof tends to be more precise than the former ones.

At that time, as claimed by Hindley, "all the proofs of the Church-Rosser theorem for λ-calculi, slick or clumsy, turn out to be based on reductions of residuals, and the finiteness property is one of the two main underlying facts which make all such proofs work". Note that it is not the case anymore that the finiteness result is required to prove the Church-Rosser property [48, 94, 84].

In his introduction, Hindley claims that his proof of the finiteness of developments uses the confluence of the developments when others need the finiteness property to prove confluence. To prove the finiteness result, Hindley provides a method to transform any development of a term into another "equivalent" one (Hindley defines a notion of equivalence between reductions) such that the length of the latter one provides a bound of the length of the former one.

Though very similar to the proof provided by Church and Rosser, Hindley's proof is much more detailed. For example, the replacement of a sequence of reductions by another one (the "equivalence" of two sequences of reductions) is left unproved by Church and Rosser.

3.6 1985: Koletsos [93]

Koletsos proved the Church-Rosser property of the terms typable in the simply typed λ-calculus using the reducibility method (see Sec. 2.5.2). Koletsos provides an interpretation of types based on a predicate called "monovaluedness". Koletsos considered typed λ-terms as Church [23] does. In this section only, we consider $\rightarrow$ and CR to be the relation $\rightarrow_\beta$ and the set of (simply typed) terms satisfying the Church-Rosser property.

Let $\mathsf{0}$ be a ground constant type. Following similar definitions [6], Koletsos defines the set of simple types as follows: $\sigma, \tau, \rho \in \mathsf{Ty} ::= \mathsf{0} \mid \sigma \rightarrow \tau$ (Koletsos' definition differs from other definition by the fact that he considers only one ground type because only one is needed in his proof).

First, let us mention that Koletsos writes $M(N)$ for the application of M to N when we write (MN). We will use (MN) (or MN using the convention for parentheses defined in Sec.2.3.1) instead of $M(N)$ in this section.

We will now present a variant of Koletsos' syntax of simply typed terms. We will slightly depart from Koletsos' definition because of some ambiguity in his language. For example, Koletsos allows $\lambda x.x^{\mathsf{0}\rightarrow\mathsf{0}}x^{\mathsf{0}}$ to be a valid term. The issue is that $x^{\mathsf{0}\rightarrow\mathsf{0}}$

and x^{0} are two different terms and that there is an implicit type associated with the abstracted x which is not explicitly stated. The above term is then ambiguous because the abstracted x can only bind one of these: $x^{0\to 0}$, x^{0}, or x^{σ} where $\sigma \notin \{0 \to 0, 0\}$. When defining his abstractions, Koletsos explains that an abstraction $\lambda x.M$ of type $\sigma{\to}\tau$ is built from a variable x of type σ and a term M of type τ. However, x's type is not made explicit in the abstraction. Church [23] enforces such abstracted variables to be annotated by their type. We will therefore add type annotations to abstracted (untyped) term variables. Instead of the above term we would then write $\lambda x^{0\to 0}.x^{0\to 0}(x^{0})$ to bind the first occurrence of x in the application.

The set Var of term variables is the one defined in Sec. 2.3.1. Our variant of Koletsos' definition of the simply typed λ-terms is as follows (a and b are defined to range over simply typed λ-terms): let x^{σ} be a term of type σ; if a is a term of type τ then let $(\lambda x^{\sigma}.a)$ be a term of type $\sigma \to \tau$; and if a is a term of type $\sigma \to \tau$ and b is a term of type σ then let (ab) be a term of type τ. Note that if $\sigma \neq \tau$ then x^{σ} and x^{τ} are two different terms.

For each type ρ and term a of type ρ, the monovaluedness predicate is defined by induction on ρ as follows:

$$\begin{array}{ll}\mathsf{MON}^{0}(a) & \text{iff } a \in \mathsf{CR} \\ \mathsf{MON}^{\sigma\to\tau}(a) & \text{iff } a \in \mathsf{CR} \text{ and for every term } b \text{ of type } \sigma,\ \mathsf{MON}^{\sigma}(b) \Rightarrow \mathsf{MON}^{\tau}(ab)\end{array}$$

Koletsos' method is equivalent to the one consisting in defining a type interpretation as a function which associates with each type σ a term set $[\![\sigma]\!]$, such that $\mathsf{MON}^{\sigma}(a)$ iff $a \in [\![\sigma]\!]$, as is done in many other works following Koletsos' [94, 84].

We now define a variant of Koletsos' definition of substitution used, e.g., by his first axiom (β-reduction) to generate his reduction relation: let $a_{x^{\tau}}[b]$ be defined as the replacing of all the free occurrences of x^{τ} in a by b (Koletsos' definition does not involve the type annotation τ). Note that because b does not have to be of type τ then $a_{x^{\tau}}[b]$ is not always a simply typed λ-term. For example, $(x^{0\to 0}y^{0})_{x^{0\to 0}}[y^{0}]$ is $(y^{0}y^{0})$ which is not a simply typed λ-term. Such a type restriction could be explicitly enforced. However, substitution is only used when the substituted variable and the term that substitutes the variable have the same type.

Then, Koletsos proves two important results:

- If $a \in \mathsf{CR}$ and (if for each $\lambda x^{\sigma}.b$ such that $a \to^{*} \lambda x^{\sigma}.b$ then $\mathsf{MON}^{\rho}(\lambda x^{\sigma}.b)$) then $\mathsf{MON}^{\rho}(a)$.
- If a is a term of type σ and for every term b, $\mathsf{MON}^{\tau}(b)$ implies $\mathsf{MON}^{\sigma}(a_{x^{\tau}}[b])$ then $\mathsf{MON}^{\tau\to\sigma}(\lambda x^{\tau}.a)$.

The first result allows one to prove among other things that for each term variable x and each type σ, $\mathsf{MON}^{\sigma}(x^{\sigma})$. The second result proves the saturation [96] of the type interpretation based on the monovaluedness predicate.

Finally, using these results, Koletsos trivially obtains the confluence of the set of simply typed λ-terms by an induction on the structure of terms.

3.7 1988: Shankar [122]

Shankar's paper [122] is a notable paper because of the formalisation and proof of the Church-Rosser property in the Boyer-Moore theorem prover[6]. Shankar's proof is similar to Tait and Martin-Löf's one. In order not to have to deal with α-conversion, the proof is carried out using the de Bruijn [34] notation for the λ-calculus (as is often the case when using a theorem prover). The proof is then carried out into the usual notation. Shankar claims that using the Boyer-Moore theorem prover some of the proofs were proved automatically ("The proofs of several of the lemmas that were proved automatically would tax most humans").

3.8 1989: Takahashi [131]

Takahashi's method is based on Tait and Martin-Löf's parallel method. She proves that the method extends easily to the $\beta\eta$-case. Even if different from the developments defined for example by Curry and Feys [31][7], Takahashi's method (as for Tait and Martin-Löf's method) consists in defining a new parallel reduction (non overlapping reductions) which is useful to develop a term without defining residuals. The usual $\beta\eta$-reduction is then trivially proved to be the transitive closure of the parallel $\beta\eta$-reduction. Then, the proof of the Church-Rosser property of the untyped λ-calculus w.r.t. the parallel $\beta\eta$-reduction leads to the proof of the Church-Rosser property of the untyped λ-calculus w.r.t. the $\beta\eta$-reduction. The Church-Rosser property of the untyped λ-calculus w.r.t. the parallel $\beta\eta$-reduction is obtained using complete developments (i.e., complete parallel $\beta\eta$-reductions which maximise the number of redexes reduced in a parallel reduction): if M reduces to N by a parallel $\beta\eta$-reduction then N reduces to P via a $\beta\eta$-parallel reduction where P is the unique term (modulo α-conversion) obtained from M by a complete parallel $\beta\eta$-reduction.

3.9 2001: Ghilezan and Kunčak [48]

Ghilezan and Kunčak's proof can be depicted by the diagram in Fig. 3.1. We present the method and the different relations and functions it uses below. This method is

[6] The Boyer-Moore theorem prover is based on a first order, quantifier free logic of recursive functions

[7] For example, if $x \notin \mathsf{fv}(\lambda y.M)$ then $\lambda x.(\lambda y.M)x$ reduces by a parallel $\beta\eta$-reduction to $\lambda y.M$ by reducing the η-redex $\lambda x.(\lambda y.M)x$. Hence, $(\lambda x.(\lambda y.M)x)N$ reduces by a parallel $\beta\eta$-reduction to $M[y := N]$. There is no corresponding development as defined by Curry and Feys, because $(\lambda y.M)N$ is not a residual of $(\lambda x.(\lambda y.M)x)N$ after reduction of the η-redex $\lambda x.(\lambda y.M)x$.

$$\begin{tikzcd}
 & & & & & M \arrow[d, "\Psi"] \arrow[ddddlllll, "I"'] \arrow[ddddrrrrr, "I"] & & & & & \\
 & & & & & \Psi(M) \arrow[dl, "o"'] \arrow[dr, "o"] & & & & & \\
 & & & & M_1 \arrow[dr, "o"] \arrow[ddll, "\beta"'] & ① & M_2 \arrow[dl, "o"'] \arrow[ddrr, "\beta"] & & & & \\
 & & & ② & & M_3 \arrow[ddll, "\beta"] \arrow[ddrr, "\beta"'] & & ③ & & & \\
P \arrow[r, "\Psi"] \arrow[ddddrrrrr, "I"'] & \Psi(P) \arrow[r, "o"] & P_1 \arrow[dr, "o"'] & & & & & & Q_1 \arrow[dl, "o"] & \Psi(Q) \arrow[l, "o"'] & Q \arrow[l, "\Psi"'] \arrow[ddddlllll, "I"] \\
 & & & P_2 \arrow[drr, "\beta"'] & & ④ & & Q_2 \arrow[dll, "\beta"] & & & \\
 & & & & & R_2 & & & & & \\
 & & & & & \Psi(R) \arrow[u, "o"'] & & & & & \\
 & & & & & R \arrow[u, "\Psi"'] & & & & &
\end{tikzcd}$$

Figure 3.1 The method of Ghilezan and Kunčak for the confluence of $\to_I$

thoroughly explained by Ghilezan and Kunčak [48] and Kamareddine and Rahli [84]. The method consists of the following steps:

- The formalisation of a development: $\to_I$ (I in Fig. 3.1). A development is defined as follows: all the redexes in a term are *frozen*[8] using two "distinguished" term variables (using the function Ψ); some of the frozen redexes are unfrozen (using the reduction relation o); some of these unfrozen redexes are β-reduced; all the redexes are unfrozen (the "distinguished" term variables are removed).

- The proof of the confluence of the developments using a simple embedding of the developments into the simply typed λ-calculus and thanks to the proof of typability of the frozen terms (where all the redexes are frozen) into the simply typed λ-calculus. The confluence of the typable terms in the simply typed λ-calculus is a well known result (see, e.g., Koletsos' proof mentioned in Sec. 3.6) and provides the confluence of the developments.

- As in many other approaches, β-reduction is proved to be the transitive closure of developments. This provides the confluence of the untyped λ-calculus.

[8] Informally, we say that a redex $(\lambda x.M)N$ is frozen when it is transformed into another similar term where the redex does not exist anymore and such that there exists a method to obtain back the original term from its frozen version.

This method provides an embedding of developments into the well known simply typed λ-calculus for which many properties have already been proved (such as confluence or strong normalisation). The defined developments can easily be proved to be equivalent to the usual ones as defined in Barendregt's book [5]. The advantages of this method over the similar method of Barendregt [5, Sec. 11.2] which uses a labelled calculus is that it does not make use of the finiteness of developments, does not introduce new symbols (Barendregt uses extra labelled λ's to define a new relation that uses the labels to distinguish between redexes to reduce or leave unreduced) and is based on an already well known background: the simply typed λ-calculus. We do not present Barendregt's proof [5, Sec. 11.2] of the confluence of his untyped λ-calculus using a labelled calculus, even though his proof is older than Ghilezan and Kunčak's proof, because the two proofs share the same steps (proof schemes). We therefore concentrate on Ghilezan and Kunčak's proof and provide below (in Sec. 5.2.2) a short comparison with one of our own method [84] (the method provided in Ch.4).

3.10 2007: Koletsos and Stavrinos [94]

Koletsos and Stavrinos' proof is similar to Ghilezan and Kunčak's proof. They share the same proof scheme. However, Koletsos and Stavrinos' result is based on the embedding of their developments into Krivine's intersection type system $\mathcal{D}$ [96] instead of the simply typed λ-calculus (as in Ghilezan and Kunčak's method [48]). Their formalisation of developments is more complicated (and sophisticated) than that of Ghilezan and Kunčak in the sense that they handle occurrences of redexes explicitly (even though not fully formalised) when Ghilezan and Kunčak handle them implicitly (without explicitly referring to instances of redexes). Also, their definition of developments is simpler than that of Ghilezan and Kunčak in the sense that the calculus on which developments are based, is simpler: Koletsos and Stavrinos use one term variable to freeze redexes when Ghilezan and Kunčak use two.

3.11 2007: Kamareddine, Rahli and Wells [85]

We have adapted, extended and formalised the work done by Koletsos and Stavrinos [94]. We adapted it to the case of the λI-calculus and extended it to the case of the $\lambda\eta$-calculus, using a formal definition of occurrences of redexes (we dealt with them formally and not intuitively as Koletsos and Stavrinos did [94]). In this work we tried to use a definition of developments based on residuals which are as close as possible to Klop's λ-residuals [92]. We failed in formalising the concept of λ-residuals as defined by Klop and came up with a new definition that we believe can be regarded as less restrictive than the "common" one as defined by Curry and Feys [31] (called $\beta\eta$-residuals) and more restrictive than Klop's one.

Let us now present the method we have used to prove the confluence of the λ-calculus w.r.t. the $\lambda\eta$-calculus. First, $\beta\eta$-redexes are explicitly defined as paths in λ-terms. Then, developments are defined as a reduction relation between pairs of a λ-term and a set of redexes in the term such that only the mentioned redexes are allowed to be reduced. A single step of a development is then a pair of pairs as follows: $\langle\langle M_1, \overline{p}_1\rangle, \langle M_2, \overline{p}_2\rangle\rangle$ where $\overline{p}_1$ is a set of paths to redexes in M_1, reducing one of these redexes leads to M_2, and $\overline{p}_2$ is the set of residuals of $\overline{p}_1$. Developments are defined via an embedding into a parametric calculus (based on the λ-calculus) where a distinguished variable (the parameter) is used to freeze some redexes. Our embedding associates a term in our parametric language with each pair of a λ-term and a set of redexes in the term. The frozen redexes are the ones that do not occur in the redex set. We proved that the terms of this parametric calculus are all typable in Krivine's system $\mathcal{D}$. We obtain that our parametric calculus is confluent by first proving that each typable term in Krivine's system $\mathcal{D}$ is in $\mathsf{CR}^{\beta\eta}$. We obtain the confluence w.r.t. the $\beta\eta$-reduction of the terms typable in Krivine's system $\mathcal{D}$ by using a reducibility method where types are interpreted by saturated sets of λ-terms (a set s is usually said to be saturated if whenever $M[x := N]M_1 \cdots M_n \in s$ then $(\lambda x.M)NM_1 \cdots M_n \in s$) and especially where type variables are interpreted by $\mathsf{CR}^{\beta\eta}$ (itself saturated). We can then prove the soundness of the type interpretation which is that if a term is typable in system $\mathcal{D}$ then it is in the interpretation of the type and because each type is interpreted by a subset of $\mathsf{CR}^{\beta\eta}$ then each typable term is in $\mathsf{CR}^{\beta\eta}$. From the confluence of our parametric calculus and using results on the embedding of our developments into our parametric calculus, we prove the confluence of our developments. Finally, we can prove that the reflexive and transitive closure of our developments is equal to the reflexive and transitive closure of the $\beta\eta$-reduction, which gives us the confluence of the λ-calculus w.r.t. the $\beta\eta$-reduction.

3.12 2008: Kamareddine and Rahli [84]

We then set out to simplify our method based on the intersection type system $\mathcal{D}$ [85] by basing our approach on the simply typed λ-calculus instead and also by handling redexes implicitly rather that explicitly. It turns our that formalising redex occurrences and reduction of redex occurrences involves heavy technicalities that are not necessary to prove the confluence of the λ-calculus. We came up with a method very similar to the method designed by Ghilezan and Kunčak [48]. Then, the observation that only a few of the types of the simply typed λ-calculus were needed in the method led us to a first simplification. We then observed that instead of introducing a type machinery, interpreting types by sets of λ-terms, and then proving the soundness of the interpretation, we could obtain a much simpler result by directly considering sets of λ-terms (the interpretations of the types and not the types themselves). We therefore completely discarded the use of a type system from our method. The side effect of the obtained method is that it is not based anymore on the well known framework of the simply typed λ-calculus and it is therefore not anymore a reducibility method (see Sec. 2.5.2). But since the power of this framework turned out not to be needed, the advantage is that we removed from the method the burden of the syntax coming along with the definition of the simply typed λ-calculus. From a semantic method based on reducibility (we say a semantic method because it involves interpreting types), we have obtained a simple syntactic method (where no interpretation is needed anymore). The obtained method shares some resemblance in its scheme with Barendregt's method [5, Sec. 11.2]. However, we believe our proof to be simpler for the same reasons that Ghilezan and Kunčak's method is simpler than Barendregt's one (see Sec. 3.9). Our method is also simpler than Barendregt, Bergstra, Klop and Volken's method [7, 5]. It is also easily generalisable into a new proof of CR for $\beta\eta$-reduction. Our simplification of a semantic proof resulted in a syntactic proof which is projectable into a semantic method (by interpreting sets of terms by types) and can therefore be used as a bridge between syntactic and semantic methods.

Our method to prove the confluence of λ-calculus w.r.t. β- and $\beta\eta$-reductions is detailed in Sec. 4.

3.13 Summary of the proof methods of the Church-Rosser property

In the literature, most of the proof methods to establish the confluence of the λ-calculus or its variants use the following scheme already detailed in the previous sections:

- Provide a definition of developments.
- Prove the confluence of the defined developments.
- Prove the confluence of the considered calculus using a correspondence between the reduction relation of the calculus and developments.

The simplest method is the syntactic method designed by Tait and Martin-Löf (see Sec. 3.4). Their proof is based on a new reduction called parallel reduction. Let us note that in their method the concept of residuals is not as clear as in our formalisation of developments [84].

The more difficult step is usually to prove the developments' confluence. Earlier works [48, 94] proved interesting embedding of developments into well known frameworks such as the simply typed λ-calculus or system $\mathcal{D}$, using known properties of these systems (such as the Church-Rosser property). It is interesting to see that some of these proofs can easily be extended to the $\beta\eta$-reduction [85, 84].

Chapter 4

From a semantic proof to a syntactic one

Many CR proofs use the notion of developments [7, 48, 94, 85]. Both Koletsos and Stavrinos [94] as well as Kamareddine and Rahli [85] use a complicated handling of developments. On the other hand, Barendregt et al. [7], Ghilezan and Kunčak [48] as well as our method presented below are based on some simpler and sufficient notions of developments. These notions of developments are technically less involved because, as in the so called method of parallel reductions [102, 131], they do not deal with residuals. Because our method presented below does not make use of a type system and does not deal with residuals, it can be regarded as a simplification of Koletsos and Stavrinos' method [94] as well as a simplification of Kamareddine, Rahli and Wells' method [85]. It can also be regarded as a simplification and a generalisation of the work done by Barendregt et al. [7] because it does not involve a new calculus and does not use the finiteness of developments, and also by Ghilezan and Kunčak [48] because is does not make use of a type system.

Let us provide a detailed description of our method. Proofs can be found in Appendix A.

4.1 Saturation, variable, abstraction properties

We consider the terms and reductions as presented in Sec. 2.3.

Def. 4.1.1 defines the three sets of terms SAT, VAR, and ABS.

Definition 4.1.1. Let the set SAT of the sets satisfying the saturation property be defined as follows: $\mathsf{SAT} = \{s \subseteq \Lambda \mid M[x := N] \in s \Rightarrow (\lambda x.M)N \in s\}$.

Let the set VAR of the sets satisfying the variable property be defined as follows: $\mathsf{VAR} = \{s \subseteq \Lambda \mid (n \geq 0 \wedge (\forall i \in \{1, \ldots, n\}.\ M_i \in s)) \Rightarrow xM_1 \cdots M_n \in s\}$.

Let the set ABS of the sets satisfying the abstraction property be defined as follows: $\mathsf{ABS} = \{s \subseteq \Lambda \mid M \in s \Rightarrow \lambda x.M \in s\}$. □

Lemma 4.1.2 presents different well known results concerning the λ-calculus (w.r.t. the β and the $\beta\eta$-reductions) as well as results concerning the sets SAT, VAR, and ABS. Lemma 4.1.2.1 is a well known result concerning the β-reduction as well as the $\beta\eta$-reduction. Lemmas 4.1.2.2 and 4.1.2.3 are well known results regarding the free variables of the terms in a reduction (β as well as $\beta\eta$). Lemmas 4.1.2.4 and 4.1.2.5 characterise some $\beta\eta$-reductions. Lemma 4.1.2.6 provides a characterisation of non-direct reduces of β-redexes. Lemma 4.1.2.7 characterise β and $\beta\eta$-reductions of β-redexes. Finally, the main result is Lemma 4.1.2.8 which states that the set of terms satisfying CR (w.r.t. β as well as $\beta\eta$) satisfies the saturation property, the variable property and the abstraction property.

Lemma 4.1.2. *Let $r \in \{\beta, \beta\eta\}$. The following hold:*

1. *If $M \rightarrow_r^* N$ and $P \rightarrow_r^* Q$ then $M[x := P] \rightarrow_r^* N[x := Q]$.*

2. *$\mathsf{fv}(M[x := N]) \subseteq \mathsf{fv}((\lambda x.M)N)$.*

3. *If $M \rightarrow_r^* N$ then $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$.*

4. *If $\lambda x.M \rightarrow_{\beta\eta}^* N$ then either N is of the form $\lambda x.M'$ such that $M \rightarrow_{\beta\eta}^* M'$ or $M \rightarrow_{\beta\eta}^* Nx$ such that $x \notin \mathsf{fv}(N)$.*

5. *If $x \notin \mathsf{fv}(M)$ and $Mx \rightarrow_{\beta\eta}^* N$ then there exists P such that $M \rightarrow_{\beta\eta}^* P$ and either N is of the form Px or P is of the form $\lambda x.N$.*

6. *If $n \geq 0$, Q is of the form $(\lambda x.M)N$, $Q \rightarrow_r^k P$ and P is not a direct r-reduct of Q then (a) $k \geq 1$, (b) if $k = 1$ then $P = M[x := N]$ and (c) there exists a direct r-reduct $(\lambda x.M')N'$ of Q such that $M'[x := N'] \rightarrow_r^* P$.*

7. *Let $n \geq 0$ and $(\lambda x.M)N \rightarrow_r^* P$. There exists P' such that $P \rightarrow_r^* P'$ and $M[x := N] \rightarrow_r^* P'$.*

8. *a)* $\mathsf{CR}^r \in \mathsf{SAT}$ *b)* $\mathsf{CR}^r \in \mathsf{VAR}$ *c)* $\mathsf{CR}^r \in \mathsf{ABS}$ □

4.2 Pseudo Development Definitions

Remark 4.2.1. Various approaches to prove the Church-Rosser property, use a function which freezes redexes in terms using new variables or constants [48, 94, 96]. We noted that this can lead to problems.

For example, Ghilezan and Kunčak [48] use two distinct term variables called f and g and introduced as "predefined constants". They then assume that "terms from Λ do not contain constants f and g". It is then not clear whether f and g are supposed to be taken as not belonging to the untyped λ-calculus or whether a new set Λ is defined to exclude terms involving f and g. The second seems to be the case. The issue is that their freezing function Ψ (similar to our function Ψ_c defined below and which is used to prevent redexes from being reduced) is proved to be a function from Λ to Λ_0 where Λ_0 is defined as follows: $\Lambda_0 = \{M \in \Lambda \mid \exists x_1, \ldots, x_n.\ \Gamma_0, x_1 : 0, \ldots, x_n : 0 \vdash M : 0\}$, which is the set of terms in Λ which are typable in simply typed λ-calculus, and where 0 is a ground type and Γ_0 is a predefined type environment assigning types to f and g. Hence, by their definition, $\Lambda_0 \subset \Lambda$. It is obvious that their function Ψ does not associate a term in Λ_0 with each term in Λ since Ψ adds some f and g to the terms (for example $\Psi(xx) = fxx$, but $fxx \notin \Lambda$, so $fxx \notin \Lambda_0$).

Moreover, typing environments (contexts) are defined as sets of type assignments of the form $x : \varphi$ where x is a term variable and φ is a simple type. Later, some contexts are built with type assignments of the form $f : \varphi$, but f is not defined as a term variable. More generally, the introduction of a new variable or a new constant implies that the considered type system has to be defined on the new calculus.

This idea behind such variables is that when freezing the redexes of a term then one wants to use a variable that does not occur in the term. However one cannot use a unique variable from the set of term variables because one can always find a term in which this variable occurs free. We solve this issue by defining parametrised sets of λ-terms as well as parametrised freezing and unfreezing relations.

□

We call *current redex* any occurrence of a redex in a given term M. For example, $(\lambda x.x)y$ is a current redex in $(\lambda x.x)yy$. We call *potential redex* an application which is not a current redex in a given term M but which is the occurrence of a redex in the term obtained after at least one reduction step from M. For example, yx is a potential redex in $(\lambda y.yx)(\lambda z.z)$. As done by Krivine [96] and many others after him [48, 94, 85], we use a term variables to freeze current or potential redexes in terms. The parametrised calculi with parameter c, a term variable in Var, presented in Def. 4.2.2 are the "frozen" calculi based on the λ-calculus where some reductions are frozen by the use of c. For example, in Λ_c^β, $(\lambda x.xy)(\lambda z.z) \rightarrow_\beta (\lambda z.z)y \rightarrow_\beta y$, but $(\lambda x.cxy)(\lambda z.z) \rightarrow_\beta c(\lambda z.z)y$ which does not reduce further. It is easy to see that for all $c \in \mathsf{Var}$, $\Lambda_c^\beta \subset \Lambda_c^{\beta\eta} \subset \Lambda$. (We define a family of term sets for each $c \in \mathsf{Var}$.)

Definition 4.2.2 (Λ_c^β, $\Lambda_c^{\beta\eta}$)**.**

$x, y \in \mathsf{Var}_c = \mathsf{Var} \setminus \{c\}$

$M, N, P, Q, R \in \Lambda_c^\beta ::= x \mid (\lambda x.M) \mid ((\lambda x.M_1)M_2) \mid ((cM_1)M_2)$

$M, N, P, Q, R \in \Lambda_c^{\beta\eta} ::= x \mid (\lambda x.M) \mid ((\lambda x.M_1)M_2) \mid ((cM_1)M_2) \mid (cM)$

In Λ_c^β and $\Lambda_c^{\beta\eta}$'s definitions (in the variable production rules), $x \in \mathsf{Var}_c$.

Because we let x, y range over Var and Var_c, when it is ambiguous, we will make explicit whether x is taken from Var or from Var_c. The same goes for M, N, P, Q, R.

□

Def. 4.2.3, introduces the freezing function which allows one to freeze the potential redexes of a term. Unlike definitions in the literature [48, 94, 96, 85], our function (the third clause below) does not freeze the current β-redexes. Furthermore, our definition does not freeze any of the current or potential η-redexes. For example, in $\Lambda_c^{\beta\eta}$, M of the form $\lambda x.(\lambda y.czx)z$ does not contain any η-redex but contains a potential η-redex, since $M \rightarrow_\beta \lambda x.czx$ and $\lambda x.czx$ is an η-redex. As we will see below, there is not need to freeze η-redexes.

Definition 4.2.3 (Ψ_c)**.** The parametric freezing Ψ_c function is defined as follows:

1. $\Psi_c(x) = x$
2. $\Psi_c(\lambda x.N) = \lambda x.\Psi_c(N)$, where $x \neq c$
3. If P is a λ-abstraction then $\Psi_c(PQ) = \Psi_c(P)\Psi_c(Q)$
4. If P is not a λ-abstraction then $\Psi_c(PQ) = c\Psi_c(P)\Psi_c(Q)$. □

Note that we do not enforce that Ψ_c only applies to terms M such that $c \notin \mathsf{fv}(M)$. For example, $\Psi_c(c) = c \notin \Lambda_c^\beta$. We will see later that given a term M we only apply function Ψ_c to M for a $c \notin \mathsf{fv}(M)$. The function Ψ is a function that takes two parameters: a term variable and a term.

Def. 4.2.4 introduces the parametric reduction relation $\rightarrow_c$ used to remove the c's from a term. This reduction can be regarded as a simplification of the reduction $\rightarrow_o$ defined by Ghilezan and Kunčak [48]. (We define a family of reduction relations for each $c \in \mathsf{Var}$.)

Definition 4.2.4 ($\rightarrow_c$)**.** Let the c-reduction relation $\rightarrow_c$ be the least compatible relation on Λ closed under the rule:

$$(c) : cM \rightarrow_c M$$

As usual $\rightarrow_c^*$ is the reflexive and transitive closure of $\rightarrow_c$. □

In Def. 4.2.5, we introduce our β-developments (the reduction relation $\rightarrow_1$) as well as our $\beta\eta$-developments (the reduction relation $\rightarrow_2$).

Definition 4.2.5 (Developments: $\rightarrow_1$, $\rightarrow_2$)**.** Let $\langle d, r\rangle \in \{\langle 1, \beta\rangle, \langle 2, \beta\eta\rangle\}$.

$$M \rightarrow_d N \Leftrightarrow \exists P.\ \Psi_c(M) \rightarrow_r^* P \wedge P \rightarrow_c^* N \wedge c \notin \mathsf{fv}(MN)$$

As usual, $\rightarrow_d^*$ is the reflexive and transitive closure of $\rightarrow_d$. (Note that $\rightarrow_d$ is reflexive, but in order not to have to introduce a new symbol for its transitive closure, we consider $\rightarrow_d^*$.) □

Developments are not parametric because a development of a term is obtained by picking a variable that does not occur free in the term, by freezing the potential redexes of the term using this free variable, by reducing the frozen term, and by finally removing all occurrences of the picked free variable.

Def. 4.2.6 defines the parametric set of terms A_c built over the parameter c using application. (We define a family of term sets for each $c \in \mathsf{Var}$.) Such terms contain only c's and no abstraction. This set of terms is especially needed to state Lemma 4.2.7.7. The particularity of such terms being that they can be completely erased by the c-reduction when applied to a term (see Lemma 4.2.7.5).

Definition 4.2.6. $d \in \mathsf{A}_c ::= c \mid dd$ □

Let us now provide some results on the reduction relation $\rightarrow_c$. Lemma 4.2.7.1 stresses the relation between the freezing function and the unfreezing relation $\rightarrow_c$: one can always undo the freezing done by the freezing function using the unfreezing relation. Using Lemmas 4.2.7.4 and 4.2.7.6, one can deduce that if one c-reduces a term in $\Lambda_c^{\beta\eta}$ then the reduct cannot be in A_c. For example, one cannot obtain c by c-reducing a term in $\Lambda_c^{\beta\eta}$. Lemma 4.2.7.7 characterises c-reductions. Lemma 4.2.7.10 is a sort of weak confluence property w.r.t. $\rightarrow_c^*$.

Lemma 4.2.7.

1. $\Psi_c(M) \rightarrow_c^* M$.
2. *If* $M \rightarrow_c^* N$ *then* $\mathsf{fv}(M) \setminus \{c\} = \mathsf{fv}(N) \setminus \{c\}$.
3. $\mathsf{fv}(M) \setminus \{c\} = \mathsf{fv}(\Psi_c(M)) \setminus \{c\}$.
4. $\Lambda_c^{\beta} \cap \mathsf{A}_c = \varnothing = \Lambda_c^{\beta\eta} \cap \mathsf{A}_c$.
5. *If* $d \in \mathsf{A}_c$ *then* $dM \rightarrow_c^* M$.
6. *If* $M \rightarrow_c^* N$ *then* $M \in \mathsf{A}_c$ *iff* $N \in \mathsf{A}_c$.
7. *Let* $M \rightarrow_c^* N$. *If* $M = x$ *then* $N = x$. *If* $M = \lambda x.M_1$ *then* $N = \lambda x.N_1$ *such that* $M_1 \rightarrow_c^* N_1$. *If* $M = M_1M_2$ *then either* $M_1 \in \mathsf{A}_c$ *and* $M_2 \rightarrow_c^* N$ *or* $N = N_1N_2$ *and* $M_1 \rightarrow_c^* N_1$ *and* $M_2 \rightarrow_c^* N_2$.
8. *If* $M \rightarrow_c^* M'$, $N \rightarrow_c^* N'$ *and* $x \neq c$ *then* $M[x := N] \rightarrow_c^* M'[x := N']$.
9. *If* $c \notin \mathsf{fv}(M)$ *and* $M \rightarrow_c^* N$ *then* $M = N$.
10. *If* $M \rightarrow_c^* N$, $M \rightarrow_c^* P$ *and* $c \notin \mathsf{fv}(N)$ *then* $P \rightarrow_c^* N$. □

Proof.

1,8,10 By induction on the structure of M.

3 Corollary of Lemma 4.2.7.1 and Lemma 4.2.7.2.

4 Let $M \in \Lambda_c^{\beta\eta}$. We prove by induction on the structure of M that $M \notin \mathsf{A}_c$.

5 By induction on the structure of d.

6 $\Rightarrow$) By induction on the length of the reduction $M \rightarrow_c^* d$.

$\Leftarrow$) By induction on the reduction $d \rightarrow_c^* N$.

7,9 By induction on the length of the reduction $M \rightarrow_c^* N$. □

4.3 A simple Church-Rosser proof for β-reduction

Koletsos and Stavrinos [94] gave a proof of the Church-Rosser property for the set of terms typable in an intersection type system called system $\mathcal{D}$ [96] w.r.t. β-reduction and showed that this can be used to establish the confluence of their β-developments without using strong normalisation. Ghilezan and Kunčak [48] gave a proof of the Church-Rosser property for the set of terms typable in the simply typed λ-calculus w.r.t. β-reduction and showed that this can be used to establish the confluence of their β-developments without using strong normalisation.

The first aim of the work presented in this section was to simplify the proof of Koletsos and Stavrinos [94]. During this simplification, we obtained a proof that bore some resemblance to the proof of Ghilezan and Kunčak [48]. A second simplification of our proof started with the observation that in both proofs of Ghilezan and Kunčak [48] and of Koletsos and Stavrinos [94] only a few types were really needed and that one can actually completely get rid of the type system. We considered two type interpretations based on the sets CR^{β} and $\mathsf{CR}^{\beta\eta}$ and interpreted the few needed types by sets of terms satisfying simple properties: saturation, variable and abstraction (see Def. 4.1.1). Since the calculus used by Koletsos and Stavrinos to prove the confluence of developments is simpler than the one used by Ghilezan and Kunčak, a third simplification which led to our actual simple proof has been to come back to the use of a calculus similar to the one used by Koletsos and Stavrinos as well as Krivine [96] before them (see Def. 4.2.2). As mentioned above, our proof is carried out in an untyped setting but one can relate the first part of the method to a reducibility proof using, e.g., the type system $\mathcal{D}$. Out proof can also be related to Barendregt, Bergstra, Klop and Volken's proof [7, 5].

The second aim of this section is to provide a framework for our main result: the extension of our proof to $\beta\eta$-reduction where we give a purely syntactic proof of Church-Rosser for $\beta\eta$-reduction (see Sec. 4.4) which is projectable into a semantic proof (based on type interpretation).

Lemma 4.3.1 states a result on Λ_c^β which we call "soundness" because it is a simplification of an earlier soundness result of a type interpretation (as part of a reducibility method) such that the needed part of our type interpretation corresponds to sets of terms satisfying the saturation, variable and abstraction properties presented in Def. 4.1.1.

Lemma 4.3.1 (Soundness)**.** *If $M \in \Lambda_c^\beta$, $\mathsf{fv}(M) \setminus \{c\} = \{x_1, \dots, x_n\}$, for all $i \in \{1, \dots, n\}$, $M_i \in s$ and $s \in \mathsf{VAR} \cap \mathsf{SAT} \cap \mathsf{ABS}$ then $M[x_1 := M_1, \dots, x_n := M_n] \in s$.* □

Proof. By induction on the structure of M. □

Using Lemma 4.3.1, we can prove that each term in Λ_c^β has β-CR.

Corollary 4.3.2. $\Lambda_c^\beta \subseteq \mathsf{CR}$. □

Proof. Let $M \in \Lambda_c^\beta$ and $\mathsf{fv}(M) \setminus \{c\} = \{x_1, \dots, x_n\}$. By Lemma 4.1.2.8, $\mathsf{CR} \in \mathsf{SAT} \cap \mathsf{VAR} \cap \mathsf{ABS}$ and $x_1, \dots, x_n \in \mathsf{CR}$. So by Lemma 4.3.1, $M \in \mathsf{CR}$. □

Lemma 4.3.3 states that the freezing function associates a term in the language Λ_c^β with each term of the untyped λ-calculus (in which c does not occur).

Lemma 4.3.3. *If $c \notin \mathsf{fv}(M)$ then $\Psi_c(M) \in \Lambda_c^\beta$.* □

Proof. By induction on the structure of M. □

Let us now prove some result concerning the calculus based on Λ_c^β and the β-reduction. Lemma 4.3.4.2 states that terms in Λ_c^β can only β-reduce to terms in Λ_c^β. Because frozen β-redexes can occur in terms in Λ_c^β (e.g., $c(\lambda x.x)y \in \Lambda_c^\beta$), Lemma 4.3.4.3 states that each term in Λ_c^β can always c-reduce to a version where only its current β-redexes are frozen. Lemma 4.3.4.4 states that our c-reduction can always remove all the c's in a term in Λ_c^β (termination of our c-reduction).

Lemma 4.3.4. *Let $M, N \in \Lambda_c^\beta$ and $x \in \mathsf{Var}_c$.*

1. *$M[x := N] \in \Lambda_c^\beta$.*
2. *If $M \rightarrow_\beta^* N$ then $N \in \Lambda_c^\beta$.*
3. *If $M \rightarrow_c^* N$ and $c \notin \mathsf{fv}(N)$ then $M \rightarrow_c^* \Psi_c(N)$.*
4. *There exists N such that $c \notin \mathsf{fv}(N)$ and $M \rightarrow_c^* N$.* □

Proof. Items 1, 3 and 4 are by induction on the structure of M. Item 2 is by induction on the length of the derivation $M \rightarrow_\beta^* N$. □

Lemma 4.3.5 states that we can simulate any β-reduction of a term in Λ_c^β from any of its (partially or totally) "unfrozen" versions.

Lemma 4.3.5.

1. *If $M_1 \in \Lambda_c^\beta$, $M_1 \rightarrow_\beta N_1$ and $M_1 \rightarrow_c^* M_2$ then there exists N_2 such that $M_2 \rightarrow_\beta N_2$ and $N_1 \rightarrow_c^* N_2$.*

2. *If $M_1 \in \Lambda_c^\beta$, $M_1 \rightarrow_\beta^* N_1$ and $M_1 \rightarrow_c^* M_2$ then there exists N_2 such that $M_2 \rightarrow_\beta^* N_2$ and $N_1 \rightarrow_c^* N_2$.* □

Proof. 1. by induction on the structure of M_1. 2. by induction on the length of the reduction $M_1 \rightarrow_\beta^* N_1$ using Lemma 4.3.5.1. □

Lemma 4.3.6 is a key lemma of simulating a reduction by developments. It states that the reflexive and transitive closure of $\rightarrow_\beta$ is equal to the reflexive and transitive closure of $\rightarrow_1$.

Lemma 4.3.6. *$M \rightarrow_\beta^* N \Leftrightarrow M \rightarrow_1^* N$.* □

Proof.

⇒) Let $M \rightarrow_\beta^* N$. We prove that $M \rightarrow_1^* N$ by induction on the size of the reduction $M \rightarrow_\beta^* N$.

⇐) Let $M \rightarrow_1^* N$. We prove that $M \rightarrow_\beta^* N$ by induction on the size of the derivation $M \rightarrow_1^* N$. □

Lemma 4.3.7 states the confluence of the β-developments.

Lemma 4.3.7.

1. *If $M \rightarrow_1 M_1$ and $M \rightarrow_1 M_2$ then there exists M_3 such that $M_1 \rightarrow_1 M_3$ and $M_2 \rightarrow_1 M_3$.*

2. *If $M \rightarrow_1^* M_1$ and $M \rightarrow_1^* M_2$ then there exists M_3 such that $M_1 \rightarrow_1^* M_3$ and $M_2 \rightarrow_1^* M_3$.* □

Proof.

1 By definition, there exist P_1, P_2 such that $\Psi_c(M) \rightarrow_\beta^* P_1$, $\Psi_c(M) \rightarrow_\beta^* P_2$, $P_1 \rightarrow_c^* M_1$, $P_2 \rightarrow_c^* M_2$ and $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(M_1) \cup \mathsf{fv}(M_2)$. By Lemma 4.3.3, $\Psi_c(M) \in \Lambda_c^\beta$. So by Corollary 4.3.2, there exists P_3 such that $P_1 \rightarrow_\beta^* P_3$ and $P_2 \rightarrow_\beta^* P_3$. By Lemma 4.3.4.2, $P_1, P_2, P_3 \in \Lambda_c^\beta$. By lemma 4.3.4.4, there exists M_3 such that $P_3 \rightarrow_c^* M_3$ and $c \notin \mathsf{fv}(M_3)$. By Lemma 4.3.4.3, $P_1 \rightarrow_c^* \Psi_c(M_1)$ and $P_2 \rightarrow_c^* \Psi_c(M_2)$. By Lemma 4.3.5.2, there exist Q_1, Q_2 such that $P_3 \rightarrow_c^* Q_1$, $P_3 \rightarrow_c^* Q_2$, $\Psi_c(M_1) \rightarrow_\beta^* Q_1$ and $\Psi_c(M_2) \rightarrow_\beta^* Q_2$. By Lemma 4.2.7.10, $Q_1 \rightarrow_c^* M_3$ and $Q_2 \rightarrow_c^* M_3$. So $M_1 \rightarrow_1 M_3$ and $M_2 \rightarrow_1 M_3$.

2 By Lemma 4.3.7.1 □

The confluence of the untyped λ-calculus w.r.t. β-reduction is now proved using the confluence of the β-developments and the equality between $\rightarrow_{\beta}^{*}$ and $\rightarrow_{1}^{*}$.

Theorem 4.3.8. $\Lambda = \mathsf{CR}$. □

Proof. $\mathsf{CR} \subseteq \Lambda$ is trivial, we only prove $\Lambda \subseteq \mathsf{CR}$. Let $M, M_1, M_2 \in \Lambda$ such that $M \rightarrow_{\beta}^{*} M_1$ and $M \rightarrow_{\beta}^{*} M_2$. By Lemma 4.3.6, $M \rightarrow_{1}^{*} M_1$ and $M \rightarrow_{1}^{*} M_2$. By Lemma 4.3.5.2, there exists M_3 such that $M_1 \rightarrow_{1}^{*} M_3$ and $M_2 \rightarrow_{1}^{*} M_3$. By Lemma 4.3.6, $M_1 \rightarrow_{\beta}^{*} M_3$ and $M_2 \rightarrow_{\beta}^{*} M_3$. □

4.4 A simple Church-Rosser proof for $\beta\eta$-reduction

Now that we have stated the principal steps of our method to prove the Church-Rosser property of the untyped λ-calculus w.r.t. β-reduction, we will generalise it to $\beta\eta$-reduction following exactly the same steps and using the $\Lambda_c^{\beta\eta}$ language. This generalisation can be regarded both as a simplification and an extension of methods by for example Ghilezan and Kunčak [48], Kamareddine and Rahli [85], Barendregt [5, Sec. 11.2], and Barendregt et al. [7].

Lemma 4.4.1 states a result on $\Lambda_c^{\beta\eta}$ which we call "soundness" for the same reason as for the similar Lemma 4.3.1.

Lemma 4.4.1 (Soundness). *If $M \in \Lambda_c^{\beta\eta}$, $\mathsf{fv}(M) \setminus \{c\} = \{x_1, \ldots, x_n\}$, for all $i \in \{1, \ldots, n\}$, $M_i \in s$ and $s \in \mathsf{SAT} \cap \mathsf{VAR} \cap \mathsf{ABS}$ then $M[x_1 := M_1, \ldots, x_n := M_n] \in s$.* □

Proof. By induction on the structure of M. □

Using lemma 4.4.1, we can now prove that each term in $\Lambda_c^{\beta\eta}$ has $\beta\eta$-CR.

Corollary 4.4.2. $\Lambda_c^{\beta\eta} \subseteq \mathsf{CR}^{\beta\eta}$. □

Proof. Let $M \in \Lambda_c^{\beta\eta}$ and $\mathsf{fv}(M) \setminus \{c\} = \{x_1, \ldots, x_n\}$. By Lemma 4.1.2.8, $\mathsf{CR}^{\beta\eta} \in \mathsf{SAT} \cap \mathsf{VAR} \cap \mathsf{ABS}$ and $x_1, \ldots, x_n \in \mathsf{CR}^{\beta\eta}$. So by Lemma 4.4.1, $M \in \mathsf{CR}^{\beta\eta}$. □

Lemma 4.4.3 states that for each term of the λ-calculus one can choose a variable c that does not occur in the term and which can be used to freeze the term to obtain a term in $\Lambda_c^{\beta\eta}$. This result is trivial because $\Lambda_c^{\beta} \subset \Lambda_c^{\beta\eta}$.

Lemma 4.4.3. *If $c \notin \mathsf{fv}(M)$ then $\Psi_c(M) \in \Lambda_c^{\beta\eta}$.* □

Proof. By Lemma 4.3.3, $\Psi_c(M) \in \Lambda_c^{\beta}$. Since $\Lambda_c^{\beta} \subset \Lambda_c^{\beta\eta}$ then $\Psi_c(M) \in \Lambda_c^{\beta\eta}$. □

Let us now prove some result concerning the calculus based on $\Lambda_c^{\beta\eta}$ and the $\beta\eta$-reduction. This lemma is similar to Lemma 4.3.4. Lemma 4.4.4.2 states that the terms in $\Lambda_c^{\beta\eta}$ can only $\beta\eta$-reduce to terms in in $\Lambda_c^{\beta\eta}$. Lemma 4.4.4.3 differs from Lemma 4.3.4.3 by the fact that terms in $\Lambda_c^{\beta\eta}$ can be of the form cM where $M \in \Lambda_c^{\beta\eta}$ while this is not possible in Λ_c^{β} (and similarly for Lemma 4.4.4.4).

Lemma 4.4.4. *Let $M, N \in \Lambda_c^{\beta\eta}$ and $x \in \mathsf{Var}_c$.*

1. *$M[x := N] \in \Lambda_c^{\beta\eta}$.*
2. *If $M \rightarrow_{\beta\eta}^{*} N$ then $N \in \Lambda_c^{\beta\eta}$.*
3. *If $M \rightarrow_c^{*} N$ and $c \notin \mathsf{fv}(N)$ then $M \rightarrow_c^{*} \Psi_c(N)$.*
4. *There exists N such that $c \notin \mathsf{fv}(N)$ and $M \rightarrow_c^{*} N$.* □

Proof. Items 1, 3 and 4 are by induction on the structure of M. Item 2 is by induction on the length of the derivation $M \rightarrow_{\beta\eta}^{*} N$. □

Lemma 4.4.5 states that we can simulate any $\beta\eta$-reduction of a term in $\Lambda_c^{\beta\eta}$ from any of its (partially or totally) "unfrozen" versions.

Lemma 4.4.5.

1. *If $M_1 \in \Lambda_c^{\beta\eta}$, $M_1 \rightarrow_{\beta\eta} N_1$ and $M_1 \rightarrow_c^{*} M_2$ then there exists N_2 such that $M_2 \rightarrow_{\beta\eta} N_2$ and $N_1 \rightarrow_c^{*} N_2$.*
2. *If $M_1 \in \Lambda_c^{\beta\eta}$ such that $M_1 \rightarrow_{\beta\eta}^{*} N_1$ and $M_1 \rightarrow_c^{*} M_2$ then there exists N_2 such that $M_2 \rightarrow_{\beta\eta}^{*} N_2$ and $N_1 \rightarrow_c^{*} N_2$.* □

Proof. 1. By induction on the structure of M_1. 2. By Lemma 4.4.5.1. □

Lemma 4.4.6 is a key lemma of the simulation method of a reduction by developments. It states that the reflexive and transitive closure of $\rightarrow_{\beta\eta}$ is equal to the reflexive and transitive closure of $\rightarrow_2$.

Lemma 4.4.6. *$M \rightarrow_{\beta\eta}^{*} N \Leftrightarrow M \rightarrow_2^{*} N$.* □

Proof.

$\Rightarrow$) Let $M \rightarrow_{\beta\eta}^{*} N$. We prove that $M \rightarrow_2^{*} N$ by induction on the size of the reduction $M \rightarrow_{\beta\eta}^{*} N$.

$\Leftarrow$) Let $M \rightarrow_2^{*} N$. We prove that $M \rightarrow_{\beta\eta}^{*} N$ by induction on the size of the derivation $M \rightarrow_2^{*} N$. □

It is then easy to deduce the confluence of the $\beta\eta$-developments.

Lemma 4.4.7.

1. *If $M \rightarrow_2 M_1$ and $M \rightarrow_2 M_2$ then there exists M_3 such that $M_1 \rightarrow_2 M_3$ and $M_2 \rightarrow_2 M_3$.*
2. *If $M \rightarrow_2^{*} M_1$ and $M \rightarrow_2^{*} M_2$ then there exists M_3 such that $M_1 \rightarrow_2^{*} M_3$ and $M_2 \rightarrow_2^{*} M_3$.* □

Proof.

1 By definition, there exist P_1, P_2 such that $\Psi_c(M) \rightarrow^*_{\beta\eta} P_1$, $\Psi_c(M) \rightarrow^*_{\beta\eta} P_2$, $P_1 \rightarrow^*_c M_1$, $P_2 \rightarrow^*_c M_2$ and $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(M_1) \cup \mathsf{fv}(M_2)$. By Lemma 4.4.3, $\Psi_c(M) \in \Lambda^{\beta\eta}_c$. So by Corollary 4.4.2, there exists P_3 such that $P_1 \rightarrow^*_{\beta\eta} P_3$ and $P_2 \rightarrow^*_{\beta\eta} P_3$. By Lemma 4.4.4.2, $P_1, P_2, P_3 \in \Lambda^{\beta\eta}_c$. By lemma 4.4.4.4, there exists M_3 such that $P_3 \rightarrow^*_c M_3$ and $c \notin \mathsf{fv}(M_3)$. By Lemma 4.4.4.3, $P_1 \rightarrow^*_c \Psi_c(M_1)$ and $P_2 \rightarrow^*_c \Psi_c(M_2)$. By Lemma 4.4.5.2, there exist Q_1, Q_2 such that $P_3 \rightarrow^*_c Q_1$, $P_3 \rightarrow^*_c Q_2$, $\Psi_c(M_1) \rightarrow^*_{\beta\eta} Q_1$ and $\Psi_c(M_2) \rightarrow^*_{\beta\eta} Q_2$. By Lemma 4.2.7.10, $Q_1 \rightarrow^*_c M_3$ and $Q_2 \rightarrow^*_c M_3$. So $M_1 \rightarrow_2 M_3$ and $M_2 \rightarrow_2 M_3$.

2 Easy by Lemma 4.4.7.1. □

The confluence of the untyped λ-calculus w.r.t. $\beta\eta$-reduction is then proved using the confluence of the $\beta\eta$-developments and the equality between $\rightarrow^*_{\beta\eta}$ and $\rightarrow^*_2$.

Theorem 4.4.8. $\Lambda = \mathsf{CR}^{\beta\eta}$. □

Proof. $\mathsf{CR}^{\beta\eta} \subseteq \Lambda$ is trivial, we only prove $\Lambda \subseteq \mathsf{CR}^{\beta\eta}$. Let $M, M_1, M_2 \in \Lambda$ such that $M \rightarrow^*_{\beta\eta} M_1$ and $M \rightarrow^*_{\beta\eta} M_2$. By Lemma 4.4.6, $M \rightarrow^*_2 M_1$ and $M \rightarrow^*_2 M_2$. By Lemma 4.4.7.2, there exists M_3 such that $M_1 \rightarrow^*_2 M_3$ and $M_2 \rightarrow^*_2 M_3$. By Lemma 4.4.6, $M_1 \rightarrow^*_{\beta\eta} M_3$ and $M_2 \rightarrow^*_{\beta\eta} M_3$. □

Chapter 5

Comparisons and conclusions

In this chapter we compare our method to two other methods (based on type systems) to prove confluence [48, 94]. We also compare our developments to those of Tait and Martin-Löf. In this section and only in this section, we consider the confluence property w.r.t. β-reduction. In Fig. 3.1 and 5.1, an arrow labelled with c, o or β stands for $\rightarrow_c^*$, $\rightarrow_o^*$ or $\rightarrow_\beta^*$ respectively. An arrow labelled with Ψ or Ψ_c stands for the application of the function with the same name to the term at the arrow's start.

5.1 Ghilezan and Kunčak's method [48]

5.1.1 Highlighting of Ghilezan and Kunčak's method

Fig. 3.1 presents Ghilezan and Kunčak's proof method [48] for the confluence of the untyped λ-calculus w.r.t. β-reduction. Their proof, based on the embedding of the developments into $\lambda_\rightarrow$, uses the confluence w.r.t. another reduction $\rightarrow_I$ (a development) whose transitive closure is equal to $\rightarrow_\beta^*$. The reduction $\rightarrow_I$ is defined as $\tau^{-1} \circ \rightarrow_\beta^* \circ \tau$ where:

- The relation τ is defined as the composition $\rightarrow_o^* \circ \Psi$.
- The relation $\rightarrow_o$ is the compatible closure of the rule $(o) : f(g(\lambda x.M))N \rightarrow_o (\lambda x.M)N$. This relation is their unfreezing relation.
- Ψ is recursively defined on the terms of the λ-calculus as follows: $\Psi(x) = x$, $\Psi(\lambda x.M) = g(\lambda x.\Psi(M))$ and $\Psi(MN) = f\Psi(M)\Psi(N)$, where f and g are two constants (see Remark 4.2.1). This function is their freezing function.

The relation τ allows one to freeze some β-redexes and the potential β-redexes (the other applications) of a term. As a matter of fact, τ does more, because Ψ does more by encapsulating the λ-abstractions using g. This technicality is needed by Ghilezan and Kunčak to prove the typability of a defined set of terms in $\lambda_\rightarrow$. The reduction τ^{-1} is similar to our own unfreezing relation $\rightarrow_c$ (see Def. 4.2.4) and to Krivine's erasure function [96], which "unfreezes" the redexes in a term.

5.1.2 Ghilezan and Kunčak's simple and sufficient notion of developments

By definition of $M \rightarrow_I P$ (a development), there exist M_1 and P_1 such that $\Psi(M) \rightarrow_o^* M_1 \rightarrow_\beta^* P_1$ and $\Psi(P) \rightarrow_o^* P_1$ (left part of Fig. 3.1). By definition of $M \rightarrow_I Q$, there exist M_2 and Q_1 such that $\Psi(M) \rightarrow_o^* M_2 \rightarrow_\beta^* Q_1$ and $\Psi(Q) \rightarrow_o^* Q_1$ (right part of Fig. 3.1). Because M_1 can be different from M_2, a confluence lemma for the unfreezing relation reduction $\rightarrow_o$ (mark ① in Fig. 3.1) and a commutation lemma for the reductions $\rightarrow_o^*$ and $\rightarrow_\beta^*$ (marks ② and ③ in Fig. 3.1) are needed. The central part of Fig. 3.1 (mark ④) corresponds to the well known result of the confluence of the terms typable in $\lambda_\rightarrow$. Koletsos [93] proved the confluence of their frozen language using a reducibility method based on a type interpretation of the types of the intersection type system $\mathcal{D}$.

The reduction $\rightarrow_I$ designed by Ghilezan and Kunčak [48] defines a development without explicitly specifying the set of redexes allowed to be reduced by the development (as done, e.g., by Barendregt et al. [7] which differs from other approaches where redexes are explicitly handled like those of Barendregt [5, Sec. 11.2] or Hindley [68]). Let us consider the reduction $M \rightarrow_I P$ (unfolded above). First, the function Ψ freezes all the redexes in M. Then, $\rightarrow_o^*$ allows one to unfreeze some of the frozen redexes in $\Psi(M)$ and therefore allows one to select a set of redexes in M which are allowed to be reduced without explicitly naming them. The reduction $M_1 \rightarrow_\beta^* P_1$ reduces some of the allowed redexes and their residuals. Finally, in $\Psi(P) \rightarrow_o^* P_1$, P is the totally unfrozen version of P_1 and the reduction $\rightarrow_o^*$ selects the set of residuals of the set of redexes in M_1 w.r.t. $M_1 \rightarrow_\beta^* P_1$ without explicitly referring to them.

This implicit way of dealing with occurrences of redexes is simple and sufficient enough to prove the confluence of the λ-calculus. Other approaches handle occurrences of redexes in a more complicated way. For example, Krivine [96] or Koletsos and Stavrinos [94] deal with occurrences of redexes explicitly but only informally. It turns out that a formalisation of their approaches is much more complicated than it seems at first [85]. Ghilezan and Kunčak [48] do not face the same issue. The reduction $\rightarrow_o^*$ allows one to unfreeze some redexes without explicitly specifying them. In Ghilezan and Kunčak's approach, as in Barendregt et al.'s approach [7], a development of a term is defined without explicit control on the set of occurrences of reduced redexes. It turns our that in Church-Rosser proofs such a control is unnecessary. One only needs to be able to freeze potential redexes and therefore allow the development of a term to reduce the current redexes of the term and their residuals.

5.1.3 Comparison of Ghilezan and Kunčak's method with other methods

Although Ghilezan and Kunčak [48] consider a simpler definition of developments than the "common" one (as defined by Barendregt [5]), their proof method scheme is exactly the one followed by Koletsos and Stavrinos [94]. Koletsos and Stavrinos consider the following "common" definition of developments: there exists a development from M to N iff $\llparenthesis M, s_1 \rrparenthesis \rightarrow_d^* \llparenthesis N, s_2 \rrparenthesis$ where $M \rightarrow_\beta^* N$, s_1 is a set of redexes in M, s_2 is the set of residuals of s_1 in N, and $\rightarrow_d^*$ is a new (complex) reduction relation based on $\rightarrow_\beta^*$. Their proof of the confluence of developments uses, among other things, the following claim: if $\llparenthesis M, s_1 \rrparenthesis \rightarrow_d^* \llparenthesis N, s_2 \rrparenthesis$ then there exists s_4 such that $\llparenthesis M, s_1 \cup s_3 \rrparenthesis \rightarrow_d^* \llparenthesis N, s_2 \cup s_4 \rrparenthesis$, where s_3 is a set of redexes of M. It is useful to prove that if $\llparenthesis M, s_1 \rrparenthesis \rightarrow_d^* \llparenthesis M_1, s'_1 \rrparenthesis$ and $\llparenthesis M, s_2 \rrparenthesis \rightarrow_d^* \llparenthesis M_2, s'_2 \rrparenthesis$ are two developments of M then there exist s''_1 and s''_2 such that $\llparenthesis M, s_1 \cup s_2 \rrparenthesis \rightarrow_d^* \llparenthesis M_1, s'_1 \cup s''_2 \rrparenthesis$ and $\llparenthesis M, s_2 \cup s_1 \rrparenthesis \rightarrow_d^* \llparenthesis M_2, s'_2 \cup s''_1 \rrparenthesis$ which allow one to develop the same redex set. This corresponds to Ghilezan and Kunčak's proof of $\rightarrow_o$'s confluence, which is useful to obtain the reductions ($\Psi(M) \rightarrow_o^* M_1 \rightarrow_o^* M_3 \rightarrow_\beta^* P_2$ and $\Psi(P) \rightarrow_o^* P_1 \rightarrow_o^* P_2$) and ($\Psi(M) \rightarrow_o^* M_2 \rightarrow_o^* M_3 \rightarrow_\beta^* Q_2$ and $\Psi(Q) \rightarrow_o^* Q_1 \rightarrow_o^* Q_2$).

Let us now present some differences between Ghilezan and Kunčak's method and that of Barendregt et al.:

- Ghilezan and Kunčak do not use the finiteness of developments when Barendregt et al. do;

- Ghilezan and Kunčak base their result on a well known result (the confluence of the simply typed λ-terms) to give a simple proof of the confluence of developments when Barendregt et al. have to prove everything;

- Ghilezan and Kunčak do not really introduce new terms when Barendregt et al. do: underlined terms are introduced to prove the confluence of developments.

Barendregt et al. also give a definition of developments without explicitly naming occurrences of redexes (no occurrence set is explicitly defined), introducing among other things, a second abstraction $\underline{\lambda}$. There exists a simple correspondence between the calculus with this second abstraction and the "frozen" calculus obtained via the freezing function introduced by Krivine and reused in the present document as well as in many other works [96, 48, 94, 85]. Informally, on can turn an underlined term as defined by Barendregt et al. into one of our frozen terms (which can be obtained using our function Ψ_c on λ-terms) by turning all the underlined λ-abstractions into non-underlined λ-abstractions and by then applying Ψ_c on the obtained term. One can turn a frozen term in Λ_c^β, obtained by applying Ψ_c to a λ-term, into an underlined term by underlining each λ such that the corresponding λ-abstraction is applied to a

$$\begin{tikzcd}
 & & & & M \arrow[dddllll, "1"'] \arrow[d, "\Psi_c"] \arrow[ddrrrr, "1"] & & & & \\
 & & \text{①} & & \Psi_c(M) \arrow[dll, "\beta"'] \arrow[drr, "\beta"] & & \text{②} & & \\
P \arrow[dr, "\Psi_c"] \arrow[ddddrrrr, "1"'] & & P_1 \arrow[ll, "c"'] \arrow[dl, "c"] \arrow[ddrr, "\beta"] & & \text{③} & & Q_1 \arrow[rr, "c"] \arrow[dr, "c"'] \arrow[ddll, "\beta"'] & & Q \arrow[dl, "\Psi_c"'] \arrow[ddddllll, "1"] \\
 & \Psi_c(P) \arrow[ddrr, "\beta"] & & & & & & \Psi_c(Q) \arrow[ddll, "\beta"'] & \\
 & & & & R_1 \arrow[dl, "c"'] \arrow[dr, "c"] \arrow[dd, "c"] & & & & \\
 & & & P_2 \arrow[dr, "c"'] & & Q_2 \arrow[dl, "c"] & & & \\
 & & & & R & & & &
\end{tikzcd}$$

Figure 5.1 Our method for the confluence of $\rightarrow_1$

term into an underlined one and by removing all occurrences of c. Their underlined β-reduction corresponds then to the β-reduction in our frozen language.

5.2 Our method

5.2.1 Highlighting of our method

Fig. 5.1 presents our method to prove the confluence of the λ-calculus. By definition of $M \rightarrow_1 P$ (Def. 4.2.5), there exists P_1 such that $\Psi_c(M) \rightarrow_\beta^* P_1$ and $P_1 \rightarrow_c^* P$, such that $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(P) \cup \mathsf{fv}(Q)$ (mark ① in Fig. 5.1). By definition of $M \rightarrow_1 Q$, there exists Q_1 such that $\Psi_c(M) \rightarrow_\beta^* Q_1$ and $Q_1 \rightarrow_c^* Q$ (mark ② in Fig. 5.1). Moreover $P_1 \rightarrow_c^* \Psi_c(P)$ and $Q_1 \rightarrow_c^* \Psi_c(Q)$ (By Lemma 4.3.3 and Lemma 4.3.4). So, because P_1 and $\Psi_c(P)$ might be different (as for Q_1 and $\Psi_c(Q)$), as Ghilezan and Kunčak [48], we need a commutation result for the reductions $\rightarrow_\beta^*$ and $\rightarrow_c^*$ (see Lemma 4.3.5). Then, the whole diagram commutes because P_2, R_1 and Q_2 all c-reduce to the same term R (by Lemma 4.2.7.10 and lemma 4.3.4.3). As in Fig. 3.1, the central part (mark ③ in Fig. 5.1) is due to the confluence of our frozen terms (typable in $\lambda_\rightarrow$ for Ghilezan and Kunčak and typable in system $\mathcal{D}$ in our case even though we do not use this fact).

5.2.2 Comparison with Ghilezan and Kunčak's developments

Our method is also based on a simple definition of developments, where first, all current β-redexes are left unfrozen and where all potential β-redexes (all the other applications) are frozen. In the present document we define two simple developments: $\to_1$ for the β case and $\to_2$ for the $\beta\eta$ case. In that way, we do not need Ghilezan and Kunčak's reduction $\to_o^*$ to unfreeze some redexes in order to perform some β-reductions. Even though we do not need this reduction relation, it does not seem possible to get rid of the work done by this reduction. Indeed, our choice implies the introduction of some other material which turns out to be similar to the reduction $\to_o^*$. Both methods need the introduction of similar material but used at different places in our methods. The reduction $\to_o^*$ is used by Ghilezan and Kunčak to unfreeze some redexes in order to allow some reductions to occur whereas we use the reduction $\to_c^*$ to, among other things, unfreeze some redexes which become frozen after some reductions.

As one can observe when comparing Fig. 3.1 and Fig. 5.1, because occurrences of redexes are not explicitly handled in our methods, a freezing function can either freeze all current redexes of terms or leave them all unfrozen. If all the redexes are frozen, a reduction such as $\to_o$ is needed before being able to perform some reductions (seeq Figure 3.1). In this case some technical results are needed such as the confluence of $\to_o$. If all current redexes are left unfrozen, because a term whose current redexes are all unfrozen does not necessarily reduce to a term whose current redexes are all unfrozen, some technical results on a reduction such as $\to_o$ (in our method, on the c-reduction) are also needed as explained above (see Figure 5.1).

5.2.3 Conclusions on our method

Finally, although our work derives from the one done by Koletsos and Stavrinos [94] and Kamareddine, Rahli and Wells [85], it turned out that it is also a simplification and generalisation of the work done by Ghilezan and Kunčak [48] and Barendregt et al. [7]. Because our method resemble Ghilezan and Kunčak's method the most, we have adopted some of of their notations and focused on comparing our method with theirs.

Thus, the two improvements of the present document can be regarded as the simplification of the work done by Ghilezan and Kunčak [48] by getting rid of the type machinery and the extension of the defined method to the $\beta\eta$-reduction.

As explained above, the main lines of our method are as follows:

- Defining simple developments;
- Proving the confluence of a simple calculus w.r.t. the considered reduction (β or $\beta\eta$) using a method based on saturated sets (e.g., reducibility in Ghilezan

and Kunčak's method);

- Proving the confluence of the defined developments;
- Proving the equality between the reflexive and transitive closure of the developments and the reflexive and transitive closure of the considered reduction;
- Proving that the untyped λ-calculus satisfies CR w.r.t. a given reduction, simulating the considered reduction using developments.

5.3 Comparison with Tait and Martin-Löf's method

Tait and Martin-Löf's proof [102, 5] (and its extensions by, e.g., example Takahashi [131]) of the confluence of the untyped λ-calculus is, to the best of our knowledge, the simplest. Our method started from the semantic framework (based on a type interpretation) when we attempted to simplify and generalise existing semantic proofs. It turned out that our simplification and generalisation of such semantic proofs led to the method presented in this document which does not require types anymore. Hence, the type interpretation and the reducibility argument are no longer used in our method. Thus, our method shifted from the semantic camp to the purely syntactic one. Nonetheless, our method can still be projected into a semantic method (something that is not obvious for methods like those of Tait and Martin-Löf, and Takahashi). We therefore consider our work to be a bridge between the syntactic and semantic methods. There is another notable difference with our method: our developments allow strictly more reductions than those of Takahashi (for both the β and $\beta\eta$ cases) as we establish in this section.

Definition 5.3.1 (Takahashi [131])**.** Let $r \in \{\beta, \beta\eta\}$. We define $\Rightarrow_r$ as follows:

- $M \Rightarrow_r M$
- $\lambda x.M \Rightarrow_r \lambda x.N$ if $M \Rightarrow_r N$
- $MN \Rightarrow_r M'N'$ if $M \Rightarrow_r M'$ and $N \Rightarrow_r N'$
- $(\lambda x.M)N \Rightarrow_r M'[x := N']$ if $M \Rightarrow_r M'$ and $N \Rightarrow_r N'$
- $\lambda x.Mx \Rightarrow_{\beta\eta} N$ if $x \notin \mathsf{fv}(M)$ and $M \Rightarrow_{\beta\eta} N$ □

Let us now prove that developments as defined by Takahashi (and Tait and Martin-Löf for the β-case) are developments w.r.t. our notion of developments.

Lemma 5.3.2.

1. *If $M \Rightarrow_\beta N$ or $M \Rightarrow_{\beta\eta} N$ then $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$.*

2. *Let M, N such that $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(N)$. If $M \Rightarrow_\beta N$ then $M \rightarrow_1 N$.*

3. *Let M, N such that $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(N)$. If $M \Rightarrow_{\beta\eta} N$ then $M \rightarrow_2 N$.* □

Proof. 2. Let $M \Rightarrow_\beta N$. The proof is by induction on the size of the derivation of $M \Rightarrow_\beta N$ and then by case on the last rule of the derivation.
3. Let $M \Rightarrow_{\beta\eta} N$. The proof is by induction on the size of the derivation of $M \Rightarrow_{\beta\eta} N$ and then by case on the last rule of the derivation. □

Remark 5.3.3.

1. We have $M = (\lambda x.xx)((\lambda z.z)y) \rightarrow_1 y((\lambda z.z)y)$ because by definition of a β-development ($\rightarrow_1$): $\Psi_c(M) = (\lambda x.cxx)((\lambda z.z)y) \rightarrow_\beta c((\lambda z.z)y))((\lambda z.z)y) \rightarrow_\beta cy((\lambda z.z)y) \rightarrow_c y((\lambda z.z)y)$, where $c \notin \{x, y, z\}$. But, we do not have $M \Rightarrow_\beta y((\lambda z.z)y)$.

2. We have $M = \lambda x.y((\lambda z.z)x) \rightarrow_2 y$ because by definition of a $\beta\eta$-development ($\rightarrow_2$): $\Psi_c(M) = \lambda x.cy((\lambda z.z)x) \rightarrow_\beta \lambda x.cyx \rightarrow_\eta cy \rightarrow_c y$, where $c \notin \{x, y, z\}$. But, we do not have $M \Rightarrow_{\beta\eta} y$. □

Part II

Complete semantics of intersection type systems with expansion variables

Chapter 6

Introduction

6.1 Expansion

6.1.1 Introduction of the expansion mechanism

Expansion was introduced at the end of the 1970s as a crucial procedure for calculating *principal typings* for λ-terms in type systems with intersection types (see Sec. 2.4.2), allowing support for compositional type inference. Coppo, Dezani, and Venneri [27] introduced the operation of *expansion* on *typings* (pairs of a type environment and a result type) for calculating the possible typings of a term when using intersection types. As a simple example, there exists an intersection type system S where the λ-term $M = (\lambda x.x(\lambda y.yz))$ can be assigned the typing $\Phi_1 = \langle \{z \mapsto a\}, (((a \to b) \to b) \to c) \to c \rangle$, which happens to be its principal typing in S. The term M can also be assigned the typing $\Phi_2 = \langle s\{z \mapsto a_1 \sqcap a_2\}, ((((a_1 \to b_1) \to b_1) \sqcap ((a_2 \to b_2) \to b_2)) \to c) \to c \rangle$, and an expansion operation can yield Φ_2 from Φ_1.

6.1.2 Expansion variables

Because the early definitions of expansion were complicated, *E-variables* were introduced in order to make the calculations easier to mechanize and reason about. For example, in System E [19], the typing Φ_1 presented above is replaced by $\Phi_3 = \langle \{z \mapsto ea\}, ((e((a \to b) \to b)) \to c) \to c \rangle$, which differs from Φ_1 by the insertion of the E-variable e at two places (in both components of the Φ_3), and Φ_2 can be obtained from Φ_3 by substituting for e the *expansion term* $E = (a := a_1, b := b_1) \sqcap (a := a_2, b := b_2)$. Carlier and Wells [20] have surveyed the history of expansion and also E-variables.

6.2 Type interpretation

6.2.1 Designing a space of meanings for expansion variables

In many kinds of semantics, a type T is interpreted by a second order function $[T]_\nu$ that takes two parameters, the type T and also a valuation ν that assigns to type variables the same kind of meanings that are assigned to types. To extend this idea to types with E-variables, we need to devise some space of possible meanings for E-variables. Given that a type eT can be turned by expansion into a new type $S_1(T) \sqcap S_2(T)$, where S_1 and S_2 are arbitrary substitutions (which can themselves introduce expansions), and that this can introduce an unbound number of new variables (both E-variables and regular type variables), the situation is complicated. Because it is unclear how to devise a space of meanings for expansions and E-variables, we instead restrict ourselves to E-variables and develop a space of meanings for types that is hierarchical in the sense that we can split it w.r.t. a certain concept of degree. Although this idea is not perfect, it seems to go quite far in giving an intuition for E-variables, namely that each E-variable occurring in a typing associated with a λ-term, acts as a capsule that isolates parts of the λ-term. As future work, we wish to come up with a higher order function that interprets types involving expansion terms by sets of λ-terms. We believe this function would help regarding the substitution mechanism introduced by expansion in terms of λ-expressions.

6.2.2 Our semantic approach

The semantic approach we use in the current document is a realisability semantics in the sense that it is derived from Kreisel's modified realisability and its variants, where "a formula "x realizes A" can be defined in a completely straightforward way: the type of the variable x is determined by the logical form of A" [113], x being the code of a function. Our semantics is strongly related to the semantic argument used in reducibility methods as used and developed by Tait [130] and many others after him [96, 93, 44, 43, 45, 46]. Atomic types (e.g., type variables) are interpreted as *saturated* sets of λ-terms, meaning that they are closed under β-expansion (the inverse of β-reduction). Arrow types are interpreted by function spaces (see the semantics provided by Scott in the open problems published in the proceedings of the Lecture Notes in Computer Science symposium held in 1975 [13]) and intersection types are interpreted by set intersections. Such a realisability semantics allows one to prove *soundness* w.r.t. a type system S, i.e., the meaning of a type T contains all closed λ-terms that can be assigned T in S. This has been shown useful for characterising the behaviour of typed λ-terms [96]. One also wants to show the converse of soundness which is called *completeness*, i.e., every closed λ-term in the meaning of T can be assigned T in S.

6.2.3 Completeness results

Hindley [70, 71, 72] was one of the first to investigate such completeness results for a simple type system and he showed that all the types of that system have the completeness property. He then generalised his completeness proof to an intersection type system [69]. Using his completeness theorem based on saturated sets of λ-terms w.r.t. $\beta\eta$-equivalence, Hindley showed that simple types were "realised"[1] by all and only the λ-terms which are typable by these types. Note that Hindley's completeness theorems were established with the sets of λ-terms saturated by $\beta\eta$-equivalence. In the present document, our completeness result depends only on the weaker requirement of β-equivalence, and we have managed to make simpler proofs that avoid needing η-reduction, confluence, or SN (although we do establish both confluence and SN for both β and $\beta\eta$).

6.2.4 Similar approaches to type interpretation

Recent work on realisability related to ours include that by Labib-Sami [97], Farkh and Nour [40], and Coquand [29], although none of this work deals with intersection types or E-variables. Similar work on realisability dealing with intersection types includes that by Kamareddine and Nour [80], which gives a sound and complete realisability semantics w.r.t. an intersection type system. This system does not deal with E-variables and is therefore different from the three hierarchical systems presented in this document.

6.3 Towards a semantics of expansion

Initially, we aimed to give a realisability semantics for a system of expansions proposed by Carlier and Wells [20]. In order to simplify our study, we considered the system with expansion variables but without the expansion rewriting rules (without the expansion mechanism). In essence, this meant that the type syntax was: $T \in \mathsf{Ty} ::= a \mid \omega \mid T_1 \to T_2 \mid T_1 \sqcap T_2 \mid eT$ where a is a type variable ranging over a countably infinite type variable set TyVar and e is an expansion variable ranging over a countably infinite expansion variable set ExpVar, and that the typing rules were as follows:

[1] We say that a λ-term M "realises" a type T if M is in T's interpretation. Hindley's semantics is not a realisability semantics but it bears some resemblance with modified realisability. One of Hindley's semantics is called "the simple semantics" and is based on the concept of model of the untyped λ-calculus [73]. Our type interpretation is also similar to Hindley's[69].

$$\frac{}{x : \langle \{x \mapsto T\} \vdash T \rangle}\ (\mathsf{var}) \qquad \frac{}{M : \langle \varnothing \vdash \omega \rangle}\ (\omega)$$

$$\frac{M : \langle \Gamma \uplus \{x \mapsto T_1\} \vdash T_2 \rangle}{\lambda x.M : \langle \Gamma \vdash T_1 \to T_2 \rangle}\ (\mathsf{abs}) \qquad \frac{M_1 : \langle \Gamma_1 \vdash T_1 \to T_2 \rangle \quad M_2 : \langle \Gamma_2 \vdash T_1 \rangle}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash T_2 \rangle}\ (\mathsf{app})$$

$$\frac{M : \langle \Gamma_1 \vdash T_1 \rangle \quad M : \langle \Gamma_2 \vdash T_2 \rangle}{M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash T_1 \sqcap T_2 \rangle}\ (\sqcap) \qquad \frac{M : \langle \Gamma \vdash T \rangle}{M : \langle e\Gamma \vdash eT \rangle}\ (\mathsf{e\text{-}app})$$

To provide a realisability semantics for this system, we needed to define the interpretation of a type to be a set of terms having this type. For our semantics to be informative on expansion variables, we needed to distinguish between the interpretation of T and eT. However, in the typing rule (e-app) presented above, the term M is unchanged and this poses difficulties. For this reason, we modified slightly the above type system by indexing the terms of the λ-calculus giving us the following syntax of terms: $M ::= x^i \mid (MN) \mid (\lambda x^i.M)$ (where M and N need to satisfy a certain condition before (MN) is allowed to be a term) and by slightly changing our type rules and in particular rule (e-app):

$$\frac{M : \langle \Gamma \vdash U \rangle}{M^+ : \langle e\Gamma \vdash eU \rangle}\ (\mathsf{e\text{-}app})$$

In this new (e-app) rule, M^+ is M where all the indices are increased by 1. Obviously these indices needed a revision regarding β-reduction and the typing rules in order to preserve the desirable properties of the type system and the realisability semantics. For this, we defined the good terms and the good types and showed that these notions go hand in hand (e.g., good types can only be assigned to good terms).

We developed a realisability semantics where each use of an E-variable in a type corresponds to an index at which evaluation occurs in the λ-terms that are assigned the type. This was an elegant solution that captured the intuition behind E-variables. However, in order for this new type system to behave well, it was necessary to consider λI-terms only (removing a subterm from M also removes important information about M as in the reduction $(\lambda x.y)M \to_\beta y$ where M is thrown away). It was also necessary to drop the universal type ω completely. This led us to the introduction of the $\lambda I^{\mathbb{N}}$-calculus and to our first type system $\vdash_1$ for which we developed a sound realisability semantics for E-variables.

However, although the first type system $\vdash_1$ is crucial to understand the intuition behind the indexing we propose, the realisability semantics we proposed was not complete w.r.t. $\vdash_1$ (subject reduction does not hold either). For this reason, we modified our system $\vdash_1$ by considering a smaller set of types (where intersections and expansions cannot occur directly to the right of an arrow), and by adding subtyping rules. This new type system $\vdash_2$ has subject reduction. Our semantics turned out to be sound w.r.t. $\vdash_2$. As for completeness, we needed to limit the list of expansion variables to a single element list. This completeness issue for $\vdash_2$ comes from the fact that the natural numbers as indexes do not allow one to differentiate between the types e_1T and e_2T if $e_1 \neq e_2$. Again, we were forced to revise our type system. We decided to restrict our λ-terms by indexing them by lists of natural numbers (where each natural number represents a difference expansion variable). We updated the type system $\vdash_2$ in consequence to obtain the type system $\vdash_3$ based among other things on the following new (e-app) rule:

$$\frac{M : \langle \Gamma \vdash U \rangle}{M^{+i} : \langle e\Gamma \vdash eU \rangle}\ (\mathsf{e\text{-}app})$$

where i is the natural number associated with the expansion variable e and where M^{+i} is M where all the lists of natural numbers are augmented with i. This new rule (e-app) allows us to distinguish the interpretations of the types e_1T and e_2T when $e_1 \neq e_2$. Furthermore, our λ-terms are constructed in such a way that K-reductions do not limit the information on the reduced terms (as in the $\lambda I^{\mathbb{N}}$-calculus, β-reduction is not always allowed, and in addition we impose further restriction on applications and abstractions). In order to obtain completeness in presence of the ω-rule, we also consider ω indexed by lists. This means that the new calculus becomes rather heavy but this seems unavoidable. It is needed to obtain a complete realisability semantics where an arbitrary (possibly infinite) number of expansion variables is allowed and where ω is present. The use of lists complicates matters and hence, needs to be understood in the context of the first semantics where indices are natural numbers rather than lists of natural numbers. In addition to the above, we have considered three saturation notions (in line with the literature) illustrating that these notions behave well in our complete realisability semantics.

6.4 Road map

Sec. 7.1 gives the syntax of the indexed calculi considered in this document: the $\lambda I^{\mathbb{N}}$-calculus, which is the λI-calculus with each variable annotated by a natural number called a *degree* or *index*, and the $\lambda^{\mathcal{L}_{\mathbb{N}}}$-calculus which is the full λ-calculus (where K-redexes are allowed) indexed with finite sequences of natural numbers. We show the confluence of β, $\beta\eta$ and weak head h-reduction on our indexed λ-calculi. Sec. 7.2 introduces the syntax and terminology for types used in both indexed calculi. Sec. 7.3 introduces our three intersection type systems with E-variables $\vdash_i$ for $i \in \{1, 2, 3\}$, where in the first one, the syntax of types is not restricted (and hence subject reduction fails) but in the other two it is restricted but then the systems are extended with a subtyping relation. In Sec. 7.4.1 and Sec. 7.4.2 we study the properties of our three type systems including subject reduction and expansion with respect to our various reduction relations $(\beta, \beta\eta, h)$. Sec. 8.1 introduces our realisability semantics and show its soundness w.r.t. each of the three considered type systems (and for each reduction relation). Sec. 8.2 establishes the challenges of showing completeness of the realisability semantics designed for the first two systems. We show that completeness does not hold for the first system and that it also does not hold for the second system if more than one expansion variable is used, but does hold for a restriction of this system to one single E-variable. This is already an important step in the study of the semantics of intersection type systems with expansion variables since a single expansion variable can be used many times and can occur nested. Sec. 8.3 establishes the completeness of a given realisability semantics w.r.t. $\vdash_3$ by introducing a special interpretation. We conclude in Sec. 9 and proofs are presented in Appendix B.

Chapter 7

The $\lambda I^{\mathbb{N}}$ and $\lambda^{\mathcal{L}_{\mathbb{N}}}$ calculi and associated type systems

7.1 The syntax of the indexed λ-calculi

Definition 7.1.1 (Indices)**.** We introduce two kinds of indices: natural numbers for our first semantics and sequences of natural numbers for our second semantics. Let $\mathcal{L}_{\mathbb{N}} = \mathsf{tuple}(\mathbb{N})$. We let I, J, range over indices. The metavariables I and J will range over $\mathbb{N}$ when considering the $\lambda I^{\mathbb{N}}$-calculus and over $\mathcal{L}_{\mathbb{N}}$ when considering the $\lambda^{\mathcal{L}_{\mathbb{N}}}$-calculus (both these calculus are defined below). Let L, K, R range over $\mathcal{L}_{\mathbb{N}}$. We sometimes write $\langle n_1, \dots, n_m\rangle$ as $(n_1, \dots, n_m)$ or as $(n_i)_{1 \leq i \leq m}$ or as $(n_i)_m$. We denote $\oslash$ the empty sequence of natural numbers ($\oslash$ stands for $\langle\rangle$). Let $::$ add an element to a sequence: $j :: (n_1, \dots, n_m) = (j, n_1, \dots, n_m)$. We sometimes write $L_1@L_2$ as $L_1 :: L_2$. We define the relation $\preceq$ and $\succeq$ on $\mathcal{L}_{\mathbb{N}}$ as follows: $L_1 \preceq L_2$ (or $L_2 \succeq L_1$) iff there exists $L_3 \in \mathcal{L}_{\mathbb{N}}$ such that $L_2 = L_1 :: L_3$. □

Lemma 7.1.2. *$\preceq$ is a partial order on $\mathcal{L}_{\mathbb{N}}$.* □

The set Var is the same λ-term variable set as defined in Sec. 2.3.1.

We define below two indexed calculi: the $\lambda I^{\mathbb{N}}$-calculus (whose set of terms is $\mathcal{M}_1$ as well as $\mathcal{M}_2$ for notational reasons) and the $\lambda^{\mathcal{L}_{\mathbb{N}}}$-calculus (whose set of terms is $\mathcal{M}_3$). As obvious, indices in $\lambda I^{\mathbb{N}}$ are simple but only allow the I-part of the calculus.

We let M, N, P, Q, R range over any of $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$ (we make explicit when a term is taken from either one of these sets). We use $=$ for syntactic equality. We assume the usual definition of subterms (see Barendregt [5] and Krivine [96]) and the usual convention for parentheses and their omission (see Sec. 2.3.1). We also consider in this part an extension of the function fv that gathers the indexed λ-term variables occurring free in terms (redefined below).

The joinability $M \diamond N$ of terms M and N ensures that in any term in which M and N occur, each variable has a unique index (note that it is more accurate to include this as part of the simultaneous inductions in Def. 7.1.4 and 7.1.7 defining $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$, but for clarity, we define it separately here).

Definition 7.1.3 (Joinability $\diamond$)**.** Let $i \in \{1, 2, 3\}$.

- Let M, N be terms of $\lambda I^{\mathbb{N}}$ (resp. $\lambda^{\mathcal{L}_{\mathbb{N}}}$) and let $\mathsf{fv}(M)$ and $\mathsf{fv}(N)$ be the corresponding free variables. We say that M and N are joinable and write $M \diamond N$ iff for all $x \in \mathsf{Var}$, if $x^{L_1} \in \mathsf{fv}(M)$ and $x^{L_2} \in \mathsf{fv}(N)$ (where $L_1, L_2 \in \mathbb{N}$ (resp. $\in \mathcal{L}_{\mathbb{N}}$)) then $L_1 = L_2$.
- If $\overline{M} \subseteq \mathcal{M}_i$ such that $\forall M, N \in \overline{M}.\ M \diamond N$, we write $\diamond \overline{M}$.
- If $\overline{M} \subseteq \mathcal{M}_i$ and $M \in \mathcal{M}_i$ such that $\forall N \in \overline{M}.\ M \diamond N$, we write $M \diamond \overline{M}$. □

Now we give the syntax of $\lambda I^{\mathbb{N}}$, an indexed version of the λI-calculus where indices (which range over $\mathbb{N}$) help categorise the *good terms* where the degree of a function is never larger than that of its argument. This amounts to having the full λI-calculus at each index and creating new λI-terms through a mixing recipe. Note that one could also define $\lambda I^{\mathbb{N}}$ by dividing Var into an countably infinite number of sets and by defining a bijective function that associates a unique index with each of these sets. We did not choose to do so because we believe explicitly writing down indexes to be clearer.

Definition 7.1.4 (The set of terms $\mathcal{M}_1$ (also called $\mathcal{M}_2$))**.** The set of terms $\mathcal{M}_1$, $\mathcal{M}_2$ (where $\mathcal{M}_1 = \mathcal{M}_2$), the set of free variables $\mathsf{fv}(M)$ of $M \in \mathcal{M}_2$ and the degree $\mathsf{deg}(M)$ of a term M, are defined by simultaneous induction:

- If $x \in \mathsf{Var}$ and $n \in \mathbb{N}$ then $x^n \in \mathcal{M}_2$, $\mathsf{fv}(x^n) = \{x^n\}$, and $\mathsf{deg}(x^n) = n$.
- If $M, N \in \mathcal{M}_2$ such that $M \diamond N$ (see Def. 7.1.3) then $MN \in \mathcal{M}_2$, $\mathsf{fv}(MN) = \mathsf{fv}(M) \cup \mathsf{fv}(N)$ and $\mathsf{deg}(MN) = \mathsf{min}(\mathsf{deg}(M), \mathsf{deg}(N))$ (where min returns the smallest of its arguments).
- If $M \in \mathcal{M}_2$ and $x^n \in \mathsf{fv}(M)$ then $\lambda x^n.M \in \mathcal{M}_2$, $\mathsf{fv}(\lambda x^n.M) = \mathsf{fv}(M) \setminus \{x^n\}$, and $\mathsf{deg}(\lambda x^n.M_1) = \mathsf{deg}(M_1)$.

Let $ix \in \mathsf{IVar}_2 ::= x^n$ and $\mathsf{IVar}_1 = \mathsf{IVar}_2$. For each $n \in \mathbb{N}$, let $\mathcal{M}_2^n = \{M \in \mathcal{M}_2 \mid \mathsf{deg}(M) = n\}$. Note that a subterm of $M \in \mathcal{M}_2$ is also in $\mathcal{M}_2$. Closed terms are defined as in Sec. 2.3.1. Let $\mathsf{closed}(M)$ be true iff M is closed, i.e., iff $\mathsf{fv}(M) = \varnothing$. □

Here is now the syntax of good terms in the $\lambda I^{\mathbb{N}}$-calculus.

Definition 7.1.5 (The set of good terms $\mathbb{M} \subset \mathcal{M}_2$)**.**

1. The set of good terms $\mathbb{M} \subset \mathcal{M}_2$ is defined by:
   - If $x \in \mathsf{Var}$ and $n \in \mathbb{N}$ then $x^n \in \mathbb{M}$.

- If $M, N \in \mathbb{M}$, $M \diamond N$, and $\mathsf{deg}(M) \leq \mathsf{deg}(N)$ then $MN \in \mathbb{M}$.
- If $M \in \mathbb{M}$ and $x^n \in \mathsf{fv}(M)$ then $\lambda x^n.M \in \mathbb{M}$.

Note that a subterm of $M \in \mathbb{M}$ is also in $\mathbb{M}$.

2. For each $n \in \mathbb{N}$, we let $\mathbb{M}^n = \mathbb{M} \cap \mathcal{M}_2^n$ □

Lemma 7.1.6.

1. *($M \in \mathbb{M}$ and $x^n \in \mathsf{fv}(M)$) iff $\lambda x^n.M \in \mathbb{M}$.*
2. *($M_1, M_2 \in \mathbb{M}$, $M_1 \diamond M_2$ and $\mathsf{deg}(M_1) \leq \mathsf{deg}(M_2)$) iff $M_1M_2 \in \mathbb{M}$.* □

Now, we give the syntax of $\lambda^{\mathcal{L}_{\mathbb{N}}}$. Note that in $\mathcal{M}_3$, an application MN is only allowed when $\mathsf{deg}(M) \preceq \mathsf{deg}(N)$. This restriction did not exist in $\lambda I^{\mathbb{N}}$ (in $\mathcal{M}_2$'s definition). Furthermore, we only allow abstractions of the form $\lambda x^L.M$ in $\lambda^{\mathcal{L}_{\mathbb{N}}}$ when $L \succeq \mathsf{deg}(M)$ (a similar restriction holds in $\lambda I^{\mathbb{N}}$ since it is a variant of the λI-calculus). The elegance of $\lambda I^{\mathbb{N}}$ is the ability to give the syntax of good terms, which is not obvious in $\lambda^{\mathcal{L}_{\mathbb{N}}}$.

Definition 7.1.7 (The set of terms $\mathcal{M}_3$)**.** The set of terms $\mathcal{M}_3$, the set of free variables $\mathsf{fv}(M)$ and degree $\mathsf{deg}(M)$ of $M \in \mathcal{M}_3$ are defined by simultaneous induction:

- If $x \in \mathsf{Var}$ and $L \in \mathcal{L}_{\mathbb{N}}$ then $x^L \in \mathcal{M}_3$, $\mathsf{fv}(x^L) = \{x^L\}$, and $\mathsf{deg}(x^L) = L$.
- If $M, N \in \mathcal{M}_3$, $\mathsf{deg}(M) \preceq \mathsf{deg}(N)$, and $M \diamond N$ (see Def. 7.1.3) then $MN \in \mathcal{M}_3$, $\mathsf{fv}(MN) = \mathsf{fv}(M) \cup \mathsf{fv}(N)$ and $\mathsf{deg}(MN) = \mathsf{deg}(M)$.
- If $x \in \mathsf{Var}$, $M \in \mathcal{M}_3$, and $L \succeq \mathsf{deg}(M)$ then $\lambda x^L.M \in \mathcal{M}_3$, $\mathsf{fv}(\lambda x^L.M) = \mathsf{fv}(M) \setminus \{x^L\}$ and $\mathsf{deg}(\lambda x^L.M) = \mathsf{deg}(M)$.

Let $ix \in \mathsf{IVar}_3 ::= x^L$. Note that each subterm of $M \in \mathcal{M}_3$ is also in $\mathcal{M}_3$. Closed terms are defined as in Sec. 2.3.1. Let $\mathsf{closed}(M)$ be true iff M is closed, i.e., iff $\mathsf{fv}(M) = \varnothing$. □

In our systems, expansions change the degree of a term. Therefore we define functions to increase and decrease indexes in terms. The next definitions turn terms of degree n into terms of higher degrees and also, if $n > 0$, they can be turned into terms of lower degrees. Note that both the increasing and the decreasing functions are well behaved operations with respect to all that matters (free variables, reduction, joinability, substitution, etc.).

Definition 7.1.8.

1. For each $n \in \mathbb{N}$, let $\mathcal{M}_2^{\geq n} = \{M \in \mathcal{M}_2 \mid \mathsf{deg}(M) \geq n\}$ and $\mathcal{M}_2^{> n} = \mathcal{M}_2^{\geq n+1}$.
2. We define $^+$ ($\in \mathcal{M}_2 \to \mathcal{M}_2$) and $^-$ ($\in \mathcal{M}_2^{>0} \to \mathcal{M}_2$) as follows:

$$(x^n)^+ = x^{n+1} \qquad (M_1M_2)^+ = M_1{}^+M_2{}^+ \qquad (\lambda x^n.M)^+ = \lambda x^{n+1}.M^+$$
$$(x^n)^- = x^{n-1} \qquad (M_1M_2)^- = M_1{}^-M_2{}^- \qquad (\lambda x^n.M)^- = \lambda x^{n-1}.M^-$$

3. Let $\overline{M} \subseteq \mathcal{M}_2$. If $\forall M \in \overline{M}$. $\mathsf{deg}(M) > 0$, we write $\mathsf{deg}(\overline{M}) > 0$. Also:

$$(\overline{M})^+ = \{M^+ \mid M \in \overline{M}\} \qquad \text{If } \mathsf{deg}(\overline{M}) > 0,\ (\overline{M})^- = \{M^- \mid M \in \overline{M}\}$$

4. We define M^{-n} by induction on $\mathsf{deg}(M) \geq n > 0$. If $n = 0$ then $M^{-n} = M$ and if $n \geq 0$ then $M^{-(n+1)} = (M^{-n})^-$. □

Definition 7.1.9. Let $i \in \mathbb{N}$ and $M \in \mathcal{M}_3$.

1. For each $L \in \mathcal{L}_{\mathbb{N}}$, let:

$$\mathcal{M}_3^L = \{M \in \mathcal{M}_3 \mid \mathsf{deg}(M) = L\} \qquad \mathcal{M}_3^{\succeq L} = \{M \in \mathcal{M}_3 \mid \mathsf{deg}(M) \succeq L\}$$

2. We define M^{+i} as follows:

$$(x^L)^{+i} = x^{i::L} \qquad (M_1M_2)^{+i} = M_1^{+i}M_2^{+i} \qquad (\lambda x^L.M)^{+i} = \lambda x^{i::L}.M^{+i}$$

3. If $\mathsf{deg}(M) = i :: L$, we define M^{-i} as follows:

$$(x^{i::L})^{-i} = x^L \qquad (M_1M_2)^{-i} = M_1^{-i}M_2^{-i} \qquad (\lambda x^{i::L'}.M)^{-i} = \lambda x^{L'}.M^{-i}$$

4. Let $\overline{M} \subseteq \mathcal{M}_3$. Let $(\overline{M})^{+i} = \{M^{+i} \mid M \in \overline{M}\}$.

   Note that $(\overline{M}_1 \cap \overline{M}_2)^{+i} = (\overline{M}_1)^{+i} \cap (\overline{M}_2)^{+i}$. □

Definition 7.1.10 (Substitution, alpha conversion, compatibility, reduction)**.**

- Let $M, N_1, \ldots, N_n$ be terms of $\lambda I^{\mathbb{N}}$ (resp. $\lambda^{\mathcal{L}_{\mathbb{N}}}$) and $I_1, \ldots, I_n \in \mathbb{N}$ (resp. $\mathcal{L}_{\mathbb{N}}$). The simultaneous substitution $M[x_1^{I_1} := N_1, \ldots, x_n^{I_n} := N_n]$ of N_i for all free occurrences of $x_i^{I_i}$ in M, where $i \in \{1, \ldots, n\}$, is defined as a partial substitution satisfying these conditions:

  - $\diamond\overline{M}$ where $\overline{M} = \{M\} \cup \{N_i \mid i \in \{1, \ldots, n\}\}$.
  - $\forall i \in \{1, \ldots, n\}$. $\mathsf{deg}(N_i) = I_i$[1].

  We sometimes write $M[x_1^{I_1} := N_1, \ldots, x_n^{I_n} := N_n]$ as $M[(x_i^{I_i} := N_i)_{1 \leq i \leq n}]$ (or simply $M[(x_i^{I_i} := N_i)_n]$).

- In $\lambda I^{\mathbb{N}}$ (resp. $\lambda^{\mathcal{L}_{\mathbb{N}}}$), we take terms modulo *α-conversion* given by: $\lambda x^I.M = \lambda y^I.(M[x^I := y^I])$ where $\forall I'$. $y^{I'} \notin \mathsf{fv}(M)$ (where $I, I' \in \mathbb{N}$ (resp. $\mathcal{L}_{\mathbb{N}}$)).

[1] We can prove the following lemma: if $\overline{M} = \{M\} \cup \{N_j \mid j \in \{1, \ldots, n\}\}$ then we have ($\diamond\overline{M}$ and $\forall j \in \{1, \ldots, n\}$. $\mathsf{deg}(N_j) = I_j$) iff $M[x_1^{I_1} := N_1, \ldots, x_n^{I_n} := N_n] \in \mathcal{M}_i$ where $i \in \{1, 2, 3\}$.

- Let $i \in \{1, 2, 3\}$. We say that a relation on $\mathcal{M}_i$ is *compatible* iff for all $M, N, P \in \mathcal{M}_i$:
  - (iabs): If $M \ rel \ N$ and $\lambda x^I.M, \lambda x^I.N \in \mathcal{M}_i$ then $(\lambda x^I.M) \ rel \ (\lambda x^I.N)$.
  - (iapp$_1$): If $M \ rel \ N$ and $MP, NP \in \mathcal{M}_i$ then $MP \ rel \ NP$.
  - (iapp$_2$): If $M \ rel \ N$, and $PM, PN \in \mathcal{M}_i$ then $PM \ rel \ PN$.
- Let $i \in \{1, 2, 3\}$. The reduction relation $\rightarrow_\beta$ on $\mathcal{M}_i$ is defined as the least compatible relation closed under the rule: $(\lambda x^I.M)N \rightarrow_\beta M[x^I := N]$ if $\mathsf{deg}(N) = I$.
- Let $i \in \{1, 2, 3\}$. The reduction relation $\rightarrow_\eta$ on $\mathcal{M}_i$ is defined as the least compatible relation closed under the rule: $\lambda x^I.Mx^I \rightarrow_\eta M$ if $x^I \notin \mathsf{fv}(M)$.
- Let $i \in \{1, 2, 3\}$. The weak head reduction $\rightarrow_h$ on $\mathcal{M}_i$ is defined as the least relation closed by rule (iapp$_2$) presented above and also by the following rule: $(\lambda x^I.M)N \rightarrow_h M[x^I := N]$ if $\mathsf{deg}(N) = I$.
- Let $\rightarrow_{\beta\eta} = \rightarrow_\beta \cup \rightarrow_\eta$.
- For a reduction relation $\rightarrow_r$, we denote by $\rightarrow_r^*$ the reflexive (w.r.t. $\mathcal{M}_i$) and transitive closure of $\rightarrow_r$. We denote by $\simeq_r$ the equivalence relation induced by $\rightarrow_r^*$ (symmetric closure). □

The next theorem states that reductions do not introduce new free variables and preserve the degree of a term.

Theorem 7.1.11. *Let $i \in \{1, 2, 3\}$, $M \in \mathcal{M}_i$, and $r \in \{\beta, \beta\eta, h\}$.*

1. *If $M \rightarrow_\eta^* N$ then $\mathsf{fv}(N) = \mathsf{fv}(M)$ and $\mathsf{deg}(M) = \mathsf{deg}(N)$.*
2. *If $i = 3$ and $M \rightarrow_r^* N$ then $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$ and $\mathsf{deg}(M) = \mathsf{deg}(N)$.*
3. *If $i \neq 3$ and $M \rightarrow_\beta^* N$ then $\mathsf{fv}(M) = \mathsf{fv}(N)$, $\mathsf{deg}(M) = \mathsf{deg}(N)$, and $M \in \mathbb{M}$ iff $N \in \mathbb{M}$.* □

Proof. 1. By induction on $M \rightarrow_\eta^* N$. 2. Case $r = \beta$, by induction on $M \rightarrow_\beta^* N$. Case $r = \beta\eta$, by the β and η cases. Case $r = h$, by the β case. 3. By induction on $M \rightarrow_\beta^* N$. □

Normal forms are defined as usual.

Definition 7.1.12 (Normal forms)**.** Let $i \in \{1, 2, 3\}$ and $r \in \{\beta, \beta\eta, h\}$.

- $M \in \mathcal{M}_i$ is in r-normal form if there is no $N \in \mathcal{M}_i$ such that $M \rightarrow_r N$.

- $M \in \mathcal{M}_i$ is r-normalising if there is an $N \in \mathcal{M}_i$ such that $M \rightarrow\!\!\!\triangleright^*_r N$ and N is in r-normal. □

Finally, the indexed lambda calculi are confluent w.r.t. β-, $\beta\eta$- and h-reductions:

Theorem 7.1.13 (Confluence). *Let $i \in \{1,2,3\}$, $M, M_1, M_2 \in \mathcal{M}_i$, and $r \in \{\beta, \beta\eta, h\}$.*

1. *If $M \rightarrow\!\!\!\triangleright^*_r M_1$ and $M \rightarrow\!\!\!\triangleright^*_r M_2$ then there is $M' \in \mathcal{M}_i$ such that $M_1 \rightarrow\!\!\!\triangleright^*_r M'$ and $M_2 \rightarrow\!\!\!\triangleright^*_r M'$.*

2. *$M_1 \simeq_r M_2$ iff there is a term $M \in \mathcal{M}_i$ such that $M_1 \rightarrow\!\!\!\triangleright^*_r M$ and $M_2 \rightarrow\!\!\!\triangleright^*_r M$.* □

Proof. We establish the confluence using the parallel reduction method. Full details can be found Appendix B. □

7.2 The types of the indexed calculi

Let us start by defining type variables and expansion variables.

Definition 7.2.1 (Type variables and expansion variables). We assume that a, b range over a countably infinite set of type variables TyVar, and that e ranges over a countably infinite set of expansion variables $\mathsf{ExpVar} = \{\mathsf{e}_0, \mathsf{e}_1, \dots\}$. □

With each expansion variable we associate a unique natural number which is the subscript of the expansion variable. Instead of explicitly naming the elements in ExpVar, we could also have considered a bijective function from expansion variables to natural numbers in order to associate a unique natural number with each expansion variable. We have decided not to do so for clarity reason. Our solution avoids defining an extra function.

For $\lambda I^{\mathbb{N}}$, we study two type systems (none of which has the ω-type). In the first, there are no restrictions on where intersection types and expansion variables occur (see set ITy_1 defined below). In the second, intersections and expansions cannot occur directly to the right of an arrow (see set ITy_2 defined below).

Definition 7.2.2 (Types, good types and degree of a type for $\lambda I^{\mathbb{N}}$).

- The type set ITy_1 is defined as follows:

$$T, U, V, W \in \mathsf{ITy}_1 ::= a \mid U_1 \rightarrow U_2 \mid U_1 \sqcap U_2 \mid eU$$

The type sets Ty_2 and ITy_2 are defined as follows (note that $\mathsf{Ty}_2 \subseteq \mathsf{ITy}_2 \subseteq \mathsf{ITy}_1$):

$$\begin{array}{ll} T & \in \mathsf{Ty}_2 ::= a \mid U \rightarrow T \\ U, V, W & \in \mathsf{ITy}_2 ::= U_1 \sqcap U_2 \mid eU \mid T \end{array}$$

- We define a function deg ($\in \mathsf{ITy}_1 \to \mathbb{N}$) by (hence deg is also defined on ITy_2):

$$\begin{array}{llll} \mathsf{deg}(a) & = 0 & \mathsf{deg}(U \to T) & = \mathsf{min}(\mathsf{deg}(U), \mathsf{deg}(T)) \\ \mathsf{deg}(eU) & = \mathsf{deg}(U) + 1 & \mathsf{deg}(U \sqcap V) & = \mathsf{min}(\mathsf{deg}(U), \mathsf{deg}(V)) \end{array}$$

- We define the set GITy which is the set of good ITy_1 types as follow (this also defines the set of good ITy_2 types: $\mathsf{GITy} \cap \mathsf{ITy}_2$):

$$\begin{array}{ll} a \in \mathsf{TyVar} & \Rightarrow a \in \mathsf{GITy} \\ U \in \mathsf{GITy} \quad \wedge e \in \mathsf{ExpVar} & \Rightarrow eU \in \mathsf{GITy} \\ U, T \in \mathsf{GITy} \wedge \mathsf{deg}(U) \geq \mathsf{deg}(T) & \Rightarrow U \to T \in \mathsf{GITy} \\ U, V \in \mathsf{GITy} \wedge \mathsf{deg}(U) = \mathsf{deg}(V) & \Rightarrow U \sqcap V \in \mathsf{GITy} \end{array}$$

When $U \in \mathsf{GITy}$, we sometimes say that U is good. □

Let $n \leq m$. Let $\vec{\mathsf{e}}_{i(n:m)}U$ or $\vec{\mathsf{e}}_L U$ where $L = (i_n, \ldots, i_m)$ denote $\mathsf{e}_{i_n} \ldots \mathsf{e}_{i_n} U$. Also, let $\vec{e}_{i(n:m),j}U$ denote $e_{\langle n,j\rangle} \ldots e_{\langle m,j\rangle} U$. We consider the application of an expansion variable to a type (eU) to have higher precedence than $\sqcap$ which itself has higher precedence than $\to$. In all our type systems, we quotient types by taking $\sqcap$ to be commutative (i.e., $U_1 \sqcap U_2 = U_2 \sqcap U_1$), associative (i.e., $U_1 \sqcap (U_2 \sqcap U_3) = (U_1 \sqcap U_2) \sqcap U_3$) and idempotent (i.e., $U \sqcap U = U$), by assuming the distributivity of expansion variables over $\sqcap$ (i.e., $e(U_1 \sqcap U_2) = eU_1 \sqcap eU_2$). We denote $U_n \sqcap \ldots \sqcap U_m$ by $\sqcap_{i=n}^{m} U$ (when $n \leq m$).

The next lemma states when arrow, intersection and applications of expansion variables to types are good.

Lemma 7.2.3.

1. *On ITy_1 (hence on ITy_2), we have the following:*

   (a) *($U, T \in \mathsf{GITy}$ and $\mathsf{deg}(U) \geq \mathsf{deg}(T)$) iff $U \to T \in \mathsf{GITy}$.*

   (b) *($U, V \in \mathsf{GITy}$ and $\mathsf{deg}(U) = \mathsf{deg}(V)$) iff $U \sqcap V \in \mathsf{GITy}$.*

   (c) *$U \in \mathsf{GITy}$ iff $eU \in \mathsf{GITy}$.*

2. *On ITy_2, we have in addition the following:*

   (a) *If $T \in \mathsf{Ty}_2$ then $\mathsf{deg}(T) = 0$.*

   (b) *If $\mathsf{deg}(U) = n$ then U is of the form $\sqcap_{i=1}^{m} \vec{e}_{j(1:n),i} V_i$ such that $m \geq 1$ and $\exists i \in \{1, \ldots, m\}.\ V_i \in \mathsf{Ty}_2$.*

   (c) *If $U \in \mathsf{GITy}$ and $\mathsf{deg}(U) = n$ then U is of the form $\sqcap_{i=1}^{m} \vec{e}_{j(1:n),i} T_i$ such that $m \geq 1$ and $\forall i \in \{1, \ldots, m\}.\ T_i \in \mathsf{Ty}_2 \cap \mathsf{GITy}$.*

   (d) *$U, T \in \mathsf{GITy}$ iff $U \to T \in \mathsf{GITy}$ (in ITy_2 and ITy_3).*

□

For $\lambda^{\mathcal{L}_{\mathbb{N}}}$, we study a type system (with the universal type ω). In this type system, in order to get subject reduction and hence completeness, intersections and expansions cannot occur directly to the right of an arrow (see ITy_3 below). Note that the type sets ITy_3 and Ty_3 defined below are far more restricted than the type sets considered for the $\lambda I^{\mathbb{N}}$-calculus and that we do not have the luxury of giving a separate syntax for good types. Note also that the definitions of degrees and types are simultaneous (unlike for ITy_2 and Ty_2 where types were defined without any reference to degrees).

Definition 7.2.4 (Types and degrees of types for $\lambda^{\mathcal{L}_{\mathbb{N}}}$)**.**

- We define the two sets of types Ty_3 and ITy_3 such that $\mathsf{Ty}_3 \subseteq \mathsf{ITy}_3$, and a function deg ($\in \mathsf{ITy}_3 \to \mathcal{L}_{\mathbb{N}}$) by simultaneous induction as follows:
  - If $a \in \mathsf{TyVar}$ then $a \in \mathsf{Ty}_3$ and $\mathsf{deg}(a) = \oslash$.
  - If $U \in \mathsf{ITy}_3$ and $T \in \mathsf{Ty}_3$ then $U \to T \in \mathsf{Ty}_3$ and $\mathsf{deg}(U \to T) = \oslash$.
  - If $L \in \mathcal{L}_{\mathbb{N}}$ then $\omega^L \in \mathsf{ITy}_3$ and $\mathsf{deg}(\omega^L) = L$.
  - If $U_1, U_2 \in \mathsf{ITy}_3$ and $\mathsf{deg}(U_1) = \mathsf{deg}(U_2)$ then $U_1 \sqcap U_2 \in \mathsf{ITy}_3$ and $\mathsf{deg}(U_1 \sqcap U_2) = \mathsf{deg}(U_1) = \mathsf{deg}(U_2)$.
  - $U \in \mathsf{ITy}_3$ and $\mathsf{e}_i \in \mathsf{ExpVar}$ then $\mathsf{e}_i U \in \mathsf{ITy}_3$ and $\mathsf{deg}(\mathsf{e}_i U) = i :: \mathsf{deg}(U)$.

  Note that deg uses the subscript of expansion variables in order to keep track of the expansion variables contributing to the degree of a type.

- We let T range over Ty_3, and U, V, W range over ITy_3. We quotient types further by having ω^L as a neutral (i.e., $\omega^L \sqcap U = U$). We also assume that for all $i \geq 0$ and $L \in \mathcal{L}_{\mathbb{N}}$, $\mathsf{e}_i \omega^L = \omega^{i::L}$. □

All our type systems use the following definition (of course within the corresponding calculus, with the corresponding indices and types):

Definition 7.2.5 (Environments and typings)**.**

- Let $k \in \{1, 2, 3\}$. We define the three sets of type environments TyEnv_1, TyEnv_2, and TyEnv_3 as follows: $\Gamma, \Delta \in \mathsf{TyEnv}_k = \mathsf{Var}_k \to \mathsf{ITy}_k$. When writing environments, we sometimes write $x : y$ instead of $x \mapsto y$. We sometimes write $\{x_1^{I_1} \mapsto U_1, \dots, x_n^{I_n} \mapsto U_n\}$ as $x_1^{I_1} : U_1, \dots, x_n^{I_n} : U_n$ or as $(x_i^{I_i} : U_i)_n$. We sometimes write () for the empty environment $\varnothing$. If $\mathsf{dj}(\mathsf{dom}(\Gamma_1), \mathsf{dom}(\Gamma_2))$, we write Γ_1, Γ_2 for $\Gamma_1 \cup \Gamma_2$.

- We say that Γ_1 and Γ_2 are joinable and write $\Gamma_1 \diamond \Gamma_2$ iff $(\forall x^{I_1} \in \mathsf{dom}(\Gamma_1).\ x^{I_2} \in \mathsf{dom}(\Gamma_2) \Rightarrow I_1 = I_2)$.

- We say that Γ is OK and write $\mathsf{ok}(\Gamma)$ iff $\forall x^I \in \mathsf{dom}(\Gamma).\ \mathsf{deg}(\Gamma(x^I)) = I$.

- Let $\Gamma_1 = \Gamma_1' \uplus \Gamma_1''$ and $\Gamma_2 = \Gamma_2' \uplus \Gamma_2''$ such that $\mathsf{dj}(\mathsf{dom}(\Gamma_1''), \mathsf{dom}(\Gamma_2''))$, $\mathsf{dom}(\Gamma_1') = \mathsf{dom}(\Gamma_2')$, and $\forall x^I \in \mathsf{dom}(\Gamma_1').\ \mathsf{deg}(\Gamma_1'(x^I)) = \mathsf{deg}(\Gamma_2'(x^I))$. We denote $\Gamma_1 \sqcap \Gamma_2$ the type environment $\{x^I \mapsto \Gamma_1'(x^I) \sqcap \Gamma_2'(x^I) \mid x^I \in \mathsf{dom}(\Gamma_1')\} \cup \Gamma_1'' \cup \Gamma_2''$. Note that $\mathsf{dom}(\Gamma_1 \sqcap \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$ and that, on environments, $\sqcap$ is commutative, associative and idempotent.

- In $\lambda I^{\mathbb{N}}$ (i.e., on TyEnv_1 and TyEnv_2), we define the set of good type environments as follows: $\mathsf{GTyEnv} = \{\Gamma \mid \forall x^I \in \mathsf{dom}(\Gamma).\ \Gamma(x^I) \in \mathsf{GITy}\}$. If $\Gamma = (x_i^{n_i} : U_i)_m$ then let $\mathsf{deg}(\Gamma) = \mathsf{min}(n_1, \ldots, n_m, \mathsf{deg}(U_1), \ldots, \mathsf{deg}(U_m))$. Let $e\Gamma = \{x^{n+1} \mapsto e\Gamma(x^n) \mid x^n \in \mathsf{dom}(\Gamma)\}$. So $e(\Gamma_1 \sqcap \Gamma_2) = e\Gamma_1 \sqcap e\Gamma_2$.

- In $\lambda^{\mathcal{L}_{\mathbb{N}}}$ (i.e., on TyEnv_3), if $M \in \mathcal{M}_3$ and $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ then let $\mathsf{env}_M^{\emptyset}$ be the type environment $(x_i^{L_i} : \omega^{L_i})_n$. For all $\mathsf{e}_j \in \mathsf{ExpVar}$, let $\mathsf{e}_j\Gamma = \{x^{j::L} \mapsto \mathsf{e}_j\Gamma(x^L) \mid x^L \in \mathsf{dom}(\Gamma)\}$. Note that $e(\Gamma_1 \sqcap \Gamma_2) = e\Gamma_1 \sqcap e\Gamma_2$. If $\Gamma = (x_i^{L_i} : U_i)_n$ and $s = \{L \mid \forall i \in \{1, \ldots, n\}.\ L \preceq L_i \wedge L \preceq \mathsf{deg}(U_i)\}$ then $\mathsf{deg}(\Gamma) = L$ such that $L \in s$ and $\forall L' \in s.\ L' \preceq L$. □

As we did for terms, we decrease the indexes of types and environments.

Definition 7.2.6 (Degree decreasing in $\lambda I^{\mathbb{N}}$).

- If $\mathsf{deg}(U) > 0$ then we inductively define the type U^- as follows:

$$(U_1 \sqcap U_2)^- = U_1{}^- \sqcap U_2{}^- \qquad (eU)^- = U$$

  If $\mathsf{deg}(U) \geq n$ then we inductively define the type U^{-n} as follows:

$$U^{-0} = U \qquad U^{-(n+1)} = (U^{-n})^-$$

- If $\mathsf{deg}(\Gamma) > 0$ then let $\Gamma^- = \{x^{n-1} \mapsto \Gamma(x^n)^- \mid x^n \in \mathsf{dom}(\Gamma)\}$.

  If $\mathsf{deg}(\Gamma) \geq n$ then we inductively define the type Γ^{-n} as follows:

$$\Gamma^{-0} = \Gamma \qquad \Gamma^{-(n+1)} = (\Gamma^{-n})^-.$$

□

Definition 7.2.7 (Degree decreasing in $\lambda^{\mathcal{L}_{\mathbb{N}}}$).

1. If $\mathsf{deg}(U) \succeq L$ then U^{-L} is inductively defined as follows:

$$U^{-\oslash} = U \qquad (U_1 \sqcap U_2)^{-i::L'} = U_1^{-i::L'} \sqcap U_2^{-i::L'} \qquad (\mathsf{e}_i U)^{-i::L'} = U^{-L'}$$

   We write U^{-i} instead of $U^{-(i)}$.

2. If $\Gamma = (x_i^{L_i} : U_i)_m$ and $\mathsf{deg}(\Gamma) \succeq L$ then by definition $\forall i \in \{1, \ldots, m\}.\ L_i = L :: L_i' \wedge L \preceq \mathsf{deg}(U_i)$, and we define $\Gamma^{-L} = (x^{L_i'} : U_i^{-L})_m$. We write Γ^{-i} instead of $\Gamma^{-(i)}$. □

Let $i \in \{1,2\}$. In $\vdash_1$, U and T range over ITy_1. In $\vdash_2$, U ranges over ITy_2 and T ranges only over Ty_2.

$$\frac{T \in \mathsf{GITy} \quad \mathsf{deg}(T) = n}{x^n : \langle (x^n : T) \vdash_1 T \rangle}\ (\mathsf{ax}) \qquad \frac{T \in \mathsf{GITy}}{x^0 : \langle (x^0 : T) \vdash_2 T \rangle}\ (\mathsf{ax}) \qquad \frac{M : \langle \Gamma, (x^n : U) \vdash_i T \rangle}{\lambda x^n.M : \langle \Gamma \vdash_i U \to T \rangle}\ (\to_{\mathsf{I}})$$

$$\frac{M_1 : \langle \Gamma_1 \vdash_i U \to T \rangle \quad M_2 : \langle \Gamma_2 \vdash_i U \rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i T \rangle}\ (\to_{\mathsf{E}}) \qquad \frac{M : \langle \Gamma \vdash_i U \rangle}{M^+ : \langle e\Gamma \vdash_i eU \rangle}\ (\mathsf{exp})$$

$$\frac{M : \langle \Gamma_1 \vdash_i U_1 \rangle \quad M : \langle \Gamma_2 \vdash_i U_2 \rangle}{M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i U_1 \sqcap U_2 \rangle}\ (\sqcap_{\mathsf{I}}) \qquad \frac{M : \langle \Gamma \vdash_2 U \rangle \quad \Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'}{M : \langle \Gamma' \vdash_2 U' \rangle}\ (\sqsubseteq)$$

The following relation $\sqsubseteq$ is defined on ITy_2, TyEnv_2, and Typing_2:

$$\frac{}{\Psi \sqsubseteq \Psi}\ (\mathsf{ref}) \qquad \frac{\Psi_1 \sqsubseteq \Psi_2 \quad \Psi_2 \sqsubseteq \Psi_3}{\Psi_1 \sqsubseteq \Psi_3}\ (\mathsf{tr}) \qquad \frac{U_2 \in \mathsf{GITy} \quad \mathsf{deg}(U_1) = \mathsf{deg}(U_2)}{U_1 \sqcap U_2 \sqsubseteq U_1}\ (\sqcap_{\mathsf{E}})$$

$$\frac{U_1 \sqsubseteq V_1 \quad U_2 \sqsubseteq V_2}{U_1 \sqcap U_2 \sqsubseteq V_1 \sqcap V_2}\ (\sqcap) \qquad \frac{U_2 \sqsubseteq U_1 \quad T_1 \sqsubseteq T_2}{U_1 \to T_1 \sqsubseteq U_2 \to T_2}\ (\to) \qquad \frac{U_1 \sqsubseteq U_2}{eU_1 \sqsubseteq eU_2}\ (\sqsubseteq_{\mathsf{exp}})$$

$$\frac{U_1 \sqsubseteq U_2 \quad y^n \notin \mathsf{dom}(\Gamma)}{\Gamma, (y^n : U_1) \sqsubseteq \Gamma, (y^n : U_2)}\ (\sqsubseteq_{\mathsf{c}}) \qquad \frac{U_1 \sqsubseteq U_2 \quad \Gamma_2 \sqsubseteq \Gamma_1}{\Gamma_1 \vdash_2 U_1 \sqsubseteq \Gamma_2 \vdash_2 U_2}\ (\sqsubseteq_{\langle\rangle})$$

Figure 7.1 Typing rules / Subtyping rules for $\vdash_1$ and $\vdash_2$

7.3 The type systems $\vdash_1$ and $\vdash_2$ for $\lambda I^{\mathbb{N}}$ and $\vdash_3$ for $\lambda^{\mathcal{L}_{\mathbb{N}}}$

In this section we introduce our three type systems $\vdash_i$ for $i \in \{1,2,3\}$, our intersection type systems with expansion variables. The system $\vdash_1$ uses the ITy_1 types and the TyEnv_1 type environments, and is for $\lambda I^{\mathbb{N}}$. The system $\vdash_2$ uses the ITy_2 types and the TyEnv_2 type environments, and is for $\lambda I^{\mathbb{N}}$. The system $\vdash_3$ uses the ITy_3 types and the TyEnv_3 type environments, and is for $\lambda^{\mathcal{L}_{\mathbb{N}}}$. In $\vdash_1$, types are not restricted and subject reduction (SR) fails. In $\vdash_2$, the syntax of types is restricted (see ITy_2's definition), and in order to guarantee SR for this type system (and hence completeness later on), we introduce a subtyping relation which allows intersection type elimination (which does not hold in the first type system). In $\vdash_3$, the syntax of types is restricted further (see ITy_3's definition) so that completeness holds with an arbitrary number of expansion variables.

Definition 7.3.1 (The type systems)**.** Let $i \in \{1,2,3\}$. The type system $\vdash_i$ uses the set ITy_i of Def. 7.2.2 (for $i \in \{1,2\}$) and 7.2.4 (for $i = 3$). The typing rules of $\vdash_1$ and $\vdash_2$ are given on the left of Fig. 7.1[2]. In $\vdash_1$, U and T range over ITy_1, and Γ range

[2] The type system $\vdash_1$ is the smallest relation closed by the rules presented on the left of Fig. 7.1 (and similarly for $\vdash_2$).

U ranges over ITy_3 and T Ty_3.

$$\frac{}{x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T \rangle}\ (\mathsf{ax}) \qquad \frac{}{M : \langle \mathsf{env}^{\varnothing}_{M} \vdash_3 \omega^{\mathsf{deg}(M)} \rangle}\ (\omega)$$

$$\frac{M : \langle \Gamma, (x^L : U) \vdash_3 T \rangle}{\lambda x^L.M : \langle \Gamma \vdash_3 U \to T \rangle}\ (\to_{\mathsf{I}}) \qquad \frac{M : \langle \Gamma \vdash_3 T \rangle \quad x^L \not\in \mathsf{dom}(\Gamma)}{\lambda x^L.M : \langle \Gamma \vdash_3 \omega^L \to T \rangle}\ (\to_{\mathsf{I}}')$$

$$\frac{M_1 : \langle \Gamma_1 \vdash_3 U \to T \rangle \quad M_2 : \langle \Gamma_2 \vdash_3 U \rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle}\ (\to_{\mathsf{E}}) \qquad \frac{M : \langle \Gamma \vdash_3 U \rangle}{M^{+j} : \langle \mathsf{e}_j \Gamma \vdash_3 \mathsf{e}_j U \rangle}\ (\mathsf{exp})$$

$$\frac{M : \langle \Gamma \vdash_3 U_1 \rangle \quad M : \langle \Gamma \vdash_3 U_2 \rangle}{M : \langle \Gamma \vdash_3 U_1 \sqcap U_2 \rangle}\ (\sqcap_{\mathsf{I}}) \qquad \frac{M : \langle \Gamma \vdash_3 U \rangle \quad \Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'}{M : \langle \Gamma' \vdash_3 U' \rangle}\ (\sqsubseteq)$$

The following relation $\sqsubseteq$ is defined on ITy_3, TyEnv_3, and Typing_3.

$$\frac{}{\Psi \sqsubseteq \Psi}\ (\mathsf{ref}) \qquad \frac{\Psi_1 \sqsubseteq \Psi_2 \quad \Psi_2 \sqsubseteq \Psi_3}{\Psi_1 \sqsubseteq \Psi_3}\ (\mathsf{tr}) \qquad \frac{\mathsf{deg}(U_1) = \mathsf{deg}(U_2)}{U_1 \sqcap U_2 \sqsubseteq U_1}\ (\sqcap_{\mathsf{E}})$$

$$\frac{U_1 \sqsubseteq V_1 \quad U_2 \sqsubseteq V_2 \quad \mathsf{deg}(U_1) = \mathsf{deg}(U_2)}{U_1 \sqcap U_2 \sqsubseteq V_1 \sqcap V_2}\ (\sqcap) \qquad \frac{U_2 \sqsubseteq U_1 \quad T_1 \sqsubseteq T_2}{U_1 \to T_1 \sqsubseteq U_2 \to T_2}\ (\to)$$

$$\frac{U_1 \sqsubseteq U_2}{eU_1 \sqsubseteq eU_2}\ (\sqsubseteq_{\mathsf{exp}}) \qquad \frac{U_1 \sqsubseteq U_2 \quad y^L \not\in \mathsf{dom}(\Gamma)}{\Gamma, y^L : U_1 \sqsubseteq \Gamma, y^L : U_2}\ (\sqsubseteq_{\mathsf{c}}) \qquad \frac{U_1 \sqsubseteq U_2 \quad \Gamma_2 \sqsubseteq \Gamma_1}{\Gamma_1 \vdash_3 U_1 \sqsubseteq \Gamma_2 \vdash_3 U_2}\ (\sqsubseteq_{\langle\rangle})$$

Figure 7.2 Typing rules / Subtyping rules for $\vdash_3$

over TyEnv_1. In $\vdash_2$, U range over ITy_2, T range over Ty_2, and Γ range over TyEnv_1. The typing rules of $\vdash_3$ are given on the left of Fig. 7.2. In both figures, the last clause makes use of a subtyping relation $\sqsubseteq$ which is defined on ITy_2 in Fig. 7.1 and on ITy_3 in Fig. 7.2. These subtyping relations are extended to type environments and typings (defined below).

We define the three typing sets Typing_1, Typing_2, and Typing_3 as follows: $\Phi \in \mathsf{Typing}_i ::= \Gamma \vdash_i U$, where $\Gamma \in \mathsf{TyEnv}_i$ and $U \in \mathsf{ITy}_i$.

Let $\mathsf{Sorts} = \cup_{i=1}^{3}\{\mathsf{Typing}_i, \mathsf{TyEnv}_i, \mathsf{ITy}_i\}$ and let Ψ range over $\cup_{s \in \mathsf{Sorts}} s$.

We say that Γ is $\vdash_i$-legal if there exist M, U such that $M : \langle \Gamma \vdash_i U \rangle$.

Let $j \in \{1, 2\}$. Let $\mathsf{GTyping} = \{\Gamma \vdash_j U \mid \Gamma \in \mathsf{GTyEnv} \wedge U \in \mathsf{GITy}\}$. If $\Phi \in \mathsf{GTyping}$ then we say that Φ is good. Let $\mathsf{deg}(\Gamma \vdash_j U) = \mathsf{min}(\mathsf{deg}(\Gamma), \mathsf{deg}(U))$.

If $s = \{L \mid L \preceq \mathsf{deg}(\Gamma) \wedge L \preceq \mathsf{deg}(U)\}$ then $\mathsf{deg}(\Gamma \vdash_3 U) = L$ such that $L \in s$ and $\forall L' \in s.\ L' \preceq L$. □

To illustrate how our indexed type system works, we give an example:

Example 7.3.2. Let $L_1 = (3) \preceq L_2 = (3, 2) \preceq L_3 = (3, 2, 1) \preceq L_4 = (3, 2, 1, 0)$ and let $a, b, c, d \in \mathsf{TyVar}$. Consider M, M', U as follows:

$M = \lambda x^{L_2}.\lambda y^{L_1}.(y^{L_1}(x^{L_2}\lambda u^{L_3}.\lambda v^{L_4}.(u^{L_3}(v^{L_4}v^{L_4})))) \in \mathcal{M}_3$
$M' = \lambda x^2.\lambda y^1.(y^1(x^2\lambda u^3.\lambda v^4.(u^3(v^4v^4)))) \in \mathcal{M}_2$
$U = \mathsf{e}_3(\mathsf{e}_2(\mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d)\to(((\mathsf{e}_2 d\to a) \sqcap b)\to a)) \in \mathsf{ITy}_2 \cap \mathsf{ITy}_3$

One can check that $M : \langle () \vdash_3 U\rangle$ and $M' : \langle () \vdash_2 U\rangle$. We simply give some steps in the derivation of $M : \langle () \vdash_3 U\rangle$ (note that the derivation of $M' : \langle () \vdash_2 U\rangle$ only differs from the derivation of $M : \langle () \vdash_3 U\rangle$ by replacing everywhere $\vdash_3$ by $\vdash_2$ and any list $(n_1, \dots, n_k)$ by k for any $k \geq 0$):

- $v^{\oslash}v^{\oslash} : \langle v^{\oslash} : a \sqcap (a\to b) \vdash_3 b\rangle$
- $v^{(0)}v^{(0)} : \langle v^{(0)} : \mathsf{e}_0(a \sqcap (a\to b)) \vdash_3 \mathsf{e}_0 b\rangle$
- $u^{\oslash} : \langle u^{\oslash} : \mathsf{e}_0 b\to c \vdash_3 \mathsf{e}_0 b\to c\rangle$
- $u^{\oslash}(v^{(0)}v^{(0)}) : \langle u^{\oslash} : \mathsf{e}_0 b\to c, v^{(0)} : \mathsf{e}_0(a \sqcap (a\to b)) \vdash_3 c\rangle$
- $\lambda v^{(0)}.u^{\oslash}(v^{(0)}v^{(0)}) : \langle u^{\oslash} : \mathsf{e}_0 b\to c \vdash_3 \mathsf{e}_0(a \sqcap (a\to b))\to c\rangle$
- $\lambda u^{\oslash}.\lambda v^{(0)}.u^{\oslash}(v^{(0)}v^{(0)}) : \langle () \vdash_3 (\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c)\rangle$
- $\lambda u^{(1)}.\lambda v^{(1,0)}.u^{(1)}(v^{(1,0)}v^{(1,0)}) : \langle () \vdash_3 \mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\rangle$
- $x^{\oslash} : \langle x^{\oslash} : \mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a\sqcap(a\to b))\to c))\to d \vdash_3 \mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a\sqcap(a\to b))\to c))\to d\rangle$
- $x^{\oslash}(\lambda u^{(1)}.\lambda v^{(1,0)}.u^{(1)}(v^{(1,0)}v^{(1,0)})) : \langle x^{\oslash} : \mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d \vdash_3 d\rangle$
- $x^{(2)}(\lambda u^{(2,1)}.\lambda v^{(2,1,0)}.u^{(2,1)}(v^{(2,1,0)}v^{(2,1,0)}))$
  $: \langle x^{(2)} : \mathsf{e}_2(\mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d) \vdash_3 \mathsf{e}_2 d\rangle$
- $y^{\oslash}(x^{(2)}(\lambda u^{(2,1)}.\lambda v^{(2,1,0)}.u^{(2,1)}(v^{(2,1,0)}v^{(2,1,0)})))$
  $: \langle x^{(2)} : \mathsf{e}_2(\mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d), y^{\oslash} : (\mathsf{e}_2 d\to a) \sqcap b \vdash_3 a\rangle$
- $\lambda y^{\oslash}.(y^{\oslash}(x^{(2)}(\lambda u^{(2,1)}.\lambda v^{(2,1,0)}.u^{(2,1)}(v^{(2,1,0)}v^{(2,1,0)}))))$
  $: \langle x^{(2)} : \mathsf{e}_2(\mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d) \vdash_3 ((\mathsf{e}_2 d\to a) \sqcap b)\to a\rangle$
- $\lambda x^{(2)}.\lambda y^{\oslash}.(y^{\oslash}(x^{(2)}(\lambda u^{(2,1)}.\lambda v^{(2,1,0)}.u^{(2,1)}(v^{(2,1,0)}v^{(2,1,0)}))))$
  $: \langle () \vdash_3 \mathsf{e}_2(\mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d)\to(((\mathsf{e}_2 d\to a) \sqcap b)\to a)\rangle$
- $\lambda x^{L_2}.\lambda y^{L_1}.(y^{L_1}(x^{L_2}(\lambda u^{L_3}.\lambda v^{L_4}.u^{L_3}(v^{L_4}v^{L_4}))))$ □
  $: \langle () \vdash_3 \mathsf{e}_3(\mathsf{e}_2(\mathsf{e}_1((\mathsf{e}_0 b\to c)\to(\mathsf{e}_0(a \sqcap (a\to b))\to c))\to d)\to(((\mathsf{e}_2 d\to a) \sqcap b)\to a))\rangle$

Let us now define our decreasing functions on the Typing_2.

Definition 7.3.3.

1. If $U \in \mathsf{ITy}_2$ and $\Gamma \in \mathsf{TyEnv}_2$ such that $\mathsf{deg}(\Gamma) > 0$ and $\mathsf{deg}(U) > 0$ then we let $(\Gamma \vdash_2 U)^- = \Gamma^- \vdash_2 U^-$.
2. If $U \in \mathsf{ITy}_3$ and $\Gamma \in \mathsf{TyEnv}_3$ such that $\mathsf{deg}(\Gamma) \succeq L$ and $\mathsf{deg}(U) \succeq L$ then we let $(\Gamma \vdash_3 U)^{-L} = \Gamma^{-L} \vdash_3 U^{-L}$. □

Next we show how ordering propagates to environments and relates degrees:

Lemma 7.3.4.

1. *If* $\Gamma \sqsubseteq \Gamma'$*,* $U \sqsubseteq U'$*, and* $x^I \notin \mathsf{dom}(\Gamma)$ *then* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ *and* $\Gamma, (x^I : U) \sqsubseteq \Gamma', (x^I : U')$.
2. $\Gamma \sqsubseteq \Gamma'$ *iff* $\Gamma = (x_i^{I_i} : U_i)_n$*,* $\Gamma' = (x_i^{I_i} : U'_i)_n$ *and* $\forall i \in \{1, \ldots, n\}.\ U_i \sqsubseteq U'_i$.
3. *Let* $j \in \{2, 3\}$*.* $\Gamma \vdash_j U \sqsubseteq \Gamma' \vdash_j U'$ *iff* $\Gamma' \sqsubseteq \Gamma$ *and* $U \sqsubseteq U'$.
4. *If* $U_1 \sqsubseteq U_2$ *then* $\mathsf{deg}(U_1) = \mathsf{deg}(U_2)$ *and* $U_1 \in \mathsf{GITy} \Leftrightarrow U_2 \in \mathsf{GITy}$.
5. *If* $\Gamma_1 \sqsubseteq \Gamma_2$ *then* $\mathsf{deg}(\Gamma_1) = \mathsf{deg}(\Gamma_2)$.
6. *Let* $j \in \{2, 3\}$*. The relation* $\sqsubseteq$ *is well defined on* $\mathsf{ITy}_j \times \mathsf{ITy}_j$*, on* $\mathsf{TyEnv}_j \times \mathsf{TyEnv}_j$*, and on* $\mathsf{Typing}_j \times \mathsf{Typing}_j$.
7. *If* $\Gamma_1, \Gamma_2 \in \mathsf{TyEnv}_2$ *and* $\Gamma_1 \sqsubseteq \Gamma_2$ *then* $\Gamma_1 \in \mathsf{GTyEnv} \Leftrightarrow \Gamma_2 \in \mathsf{GTyEnv}$ □

Proof. We prove 1. and 2. by induction on the derivation $\Gamma \sqsubseteq \Gamma'$. We prove 3. by induction on the derivation $\Gamma \vdash_j U \sqsubseteq \Gamma' \vdash_j U'$. We prove 4. by induction on the derivation $U_1 \sqsubseteq U_2$. We prove 5. by induction on the derivation $\Gamma_1 \sqsubseteq \Gamma_2$. We prove 6. by induction on a subtyping derivation. We prove 7. by induction on the derivation of $\Gamma_1 \sqsubseteq \Gamma_2$. □

The next theorem states that typings are well defined, that within a typing, degrees are well behaved and that we do not allow weakening.

Theorem 7.3.5. *Let* $j \in \{1, 2, 3\}$*. We have:*

1. $\vdash_j$ *is well defined on* $\mathcal{M}_j \times \mathsf{TyEnv}_j \times \mathsf{ITy}_j$.
2. *Let* $M : \langle \Gamma \vdash_j U \rangle$.
   - *(a)* $\mathsf{deg}(M) = \mathsf{deg}(U)$*,* $\mathsf{ok}(\Gamma)$*, and* $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$.
   - *(b) If* $j \neq 3$ *then* $U \in \mathsf{GITy}$*,* $M \in \mathbb{M}$*,* $\Gamma \in \mathsf{GTyEnv}$*, and* $\mathsf{deg}(\Gamma) \geq \mathsf{deg}(M)$.
   - *(c) If* $j = 3$ *then* $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(U)$.
   - *(d) If* $j = 2$ *and* $\mathsf{deg}(U) \geq k$ *then* $M^{-k} : \langle \Gamma^{-k} \vdash_2 U^{-k} \rangle$.
   - *(e) If* $j = 3$ *and* $\mathsf{deg}(U) \succeq K$ *then* $M^{-K} : \langle \Gamma^{-K} \vdash_3 U^{-K} \rangle$.

□

Proof. We prove 1. and 2. by induction on the derivation $M : \langle \Gamma \vdash_j U \rangle$. □

Let us now present admissible typing (and subtyping) rules.

Remark 7.3.6.

1. The rule $\dfrac{M : \langle \Gamma_1 \vdash_3 U_1 \rangle \quad M : \langle \Gamma_2 \vdash_3 U_2 \rangle}{M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 U_1 \sqcap U_2 \rangle}\ (\sqcap_{\mathsf{I}}')$ is admissible

2. The rule $\dfrac{U \in \mathsf{GITy} \quad \mathsf{deg}(U) = n}{x^n : \langle (x^n : U) \vdash_2 U \rangle}\ (\mathsf{ax}')$ is admissible

3. The rule $\dfrac{}{x^{\mathsf{deg}(U)} : \langle (x^{\mathsf{deg}(U)} : U) \vdash_3 U \rangle}\ (\mathsf{ax}'')$ is admissible

4. The rule $\dfrac{}{U \sqsubseteq \omega^{\mathsf{deg}(U)}}\ (\omega')$ is admissible □

Let us now present some results concerning the ω type and joinability.

Lemma 7.3.7.

1. *If* $M : \langle \Gamma \vdash_3 U \rangle$ *then* $\Gamma \sqsubseteq \mathsf{env}^{\emptyset}_{M}$

2. *If* $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$ *and* $\mathsf{ok}(\Gamma)$ *then* $M : \langle \Gamma \vdash_3 \omega^{\mathsf{deg}(M)} \rangle$.

3. *If* $i \in \{1, 2, 3\}$, $M_1 : \langle \Gamma_1 \vdash_i U_1 \rangle$ *and* $M_2 : \langle \Gamma_2 \vdash_i U_2 \rangle$ *then* $\Gamma_1 \diamond \Gamma_2 \Leftrightarrow M_1 \diamond M_2$. □

Proof.

1. Let $\Gamma = (x_i^{L_i} : U_i)_n$ where $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ by Theorem 7.3.5.2a. By Remark 7.3.6.4, $\forall i \in \{1, \ldots, n\}.\ U_i \sqsubseteq \omega^{\mathsf{deg}(U_i)}$. By Theorem 7.3.5.2a, $\mathsf{ok}(\Gamma)$ and therefore $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) = L_i$. Finally, by Lemma 7.3.4.2, $\Gamma \sqsubseteq \mathsf{env}^{\emptyset}_{M}$.

2. Let $\Gamma = (x_i^{L_i} : U_i)_n$. Then by hypotheses $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ and $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) = L_i$. By Remark 7.3.6.4, $\forall i \in \{1, \ldots, n\}.\ U_i \sqsubseteq \omega^{L_i}$. By Lemma 7.3.4.2, $\Gamma \sqsubseteq \mathsf{env}^{\emptyset}_{M} = (x^{L_i} : \omega^{L_i})_n$. Since by rule (ω), $M : \langle \mathsf{env}^{\emptyset}_{M} \vdash_3 \omega^{\mathsf{deg}(M)} \rangle$, we have by rules $(\sqsubseteq)$ and $(\sqsubseteq_{\langle\rangle})$, $M : \langle \Gamma \vdash_3 \omega^{\mathsf{deg}(M)} \rangle$.

3. $\Leftarrow$) Let $x^{I_1} \in \mathsf{dom}(\Gamma_1)$ and $x^{I_2} \in \mathsf{dom}(\Gamma_2)$ then by Theorem 7.3.5.2a, $x^{I_1} \in \mathsf{fv}(M_1)$ and $x^{I_2} \in \mathsf{fv}(M_2)$. Because $M_1 \diamond M_2$, then $I_1 = I_2$ and therefore $\Gamma_1 \diamond \Gamma_2$. $\Rightarrow$) Let $x^{I_1} \in \mathsf{fv}(M_1)$ and $x^{I_2} \in \mathsf{fv}(M_2)$ then by Theorem 7.3.5.2a, $x^{I_1} \in \mathsf{dom}(\Gamma_1)$ and $x^{I_2} \in \mathsf{dom}(\Gamma_2)$. Because $\Gamma_1 \diamond \Gamma_2$, then $I_1 = I_2$ and therefore $M_1 \diamond M_2$. □

7.4 Subject reduction and expansion properties of our type systems

7.4.1 Subject reduction and expansion properties for $\vdash_1$ and $\vdash_2$

Now we list the generation lemmas for $\vdash_1$ and $\vdash_2$ (for proofs see Appendix B).

Lemma 7.4.1 (Generation for $\vdash_1$)**.**

1. *If* $x^n : \langle \Gamma \vdash_1 T \rangle$ *then* $\Gamma = (x^n : T)$.

2. *If* $\lambda x^n.M : \langle \Gamma \vdash_1 T_1 \to T_2 \rangle$ *then* $M : \langle \Gamma, x^n : T_1 \vdash_1 T_2 \rangle$.

3. *If* $MN : \langle \Gamma \vdash_1 T \rangle$ *and* $\mathsf{deg}(T) = m$ *then* $\Gamma = \Gamma_1 \sqcap \Gamma_2$, $T = \sqcap_{i=1}^{n} \vec{e}_{j(1:m),i} T_i$, $n \geq 1$, $M : \langle \Gamma_1 \vdash_1 \sqcap_{i=1}^{n} \vec{e}_{j(1:m),i} (T'_i \to T_i) \rangle$ *and* $N : \langle \Gamma_2 \vdash_1 \sqcap_{i=1}^{n} \vec{e}_{j(1:m),i} T'_i \rangle$. □

Lemma 7.4.2 (Generation for $\vdash_2$)**.**

1. *If* $x^n : \langle \Gamma \vdash_2 U \rangle$ *then* $\Gamma = (x^n : V)$ *where* $V \sqsubseteq U$.

2. *If* $\lambda x^n.M : \langle \Gamma \vdash_2 U \rangle$ *and* $\mathsf{deg}(U) = m$ *then* $U = \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} (V_i \to T_i)$ *where* $k \geq 1$ *and* $\forall i \in \{1, \ldots, k\}$. $M : \langle \Gamma, x^n : \vec{e}_{j(1:m),i} V_i \vdash_2 \vec{e}_{j(1:m),i} T_i \rangle$.

3. *If* $MN : \langle \Gamma \vdash_2 U \rangle$ *and* $\mathsf{deg}(U) = m$ *then* $U = \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} T_i$ *where* $k \geq 1$, $\Gamma = \Gamma_1 \sqcap \Gamma_2$, $M : \langle \Gamma_1 \vdash_2 \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} (U_i \to T_i) \rangle$, *and* $N : \langle \Gamma_2 \vdash_2 \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} U_i \rangle$. □

We also show that no β-redexes are blocked in a typable term.

Remark 7.4.3 (No β-redexes are blocked in typable terms). Let $i \in \{1, 2\}$ and $M : \langle \Gamma \vdash_i U \rangle$. If $(\lambda x^n.M_1)M_2$ is a subterm of M then $\mathsf{deg}(M_2) = n$ and hence $(\lambda x^n.M_1)M_2 \rhd_\beta M_1[x^n := M_2]$. □

Lemma 7.4.4 (Substitution for $\vdash_2$)**.** *If* $M : \langle \Gamma, x^I : U \vdash_2 V \rangle$, $N : \langle \Delta \vdash_2 U \rangle$ *and* $M \diamond N$ *then* $M[x^I := N] : \langle \Gamma \sqcap \Delta \vdash_2 V \rangle$. □

Proof. By induction on the derivation $M : \langle \Gamma, x^I : U \vdash_2 V \rangle$. □

Lemma 7.4.5 (Substitution and Subject β-reduction fails for $\vdash_1$)**.** *Let* a, b, c *be different type variables. We have:*

1. $(\lambda x^0.x^0 x^0)(y^0 z^0) \rhd_\beta (y^0 z^0)(y^0 z^0)$.

2. $x^0 x^0 : \langle x^0 : (a \to c) \sqcap a \vdash_1 c \rangle$.

3. $(\lambda x^0.x^0 x^0)(y^0 z^0) : \langle y^0 : b \to ((a \to c) \sqcap a), z^0 : b \vdash_1 c \rangle$.

4. *It is not possible that* $(y^0 z^0)(y^0 z^0) : \langle y^0 : b \to ((a \to c) \sqcap a), z^0 : b \vdash_1 c \rangle$.

Hence, the substitution and subject β-reduction lemmas fail for $\vdash_1$. □

Proof. 1., 2., and 3. are easy.

For 4., assume $(y^0 z^0)(y^0 z^0) : \langle y^0 : b \to ((a \to c) \sqcap a), z^0 : b \vdash_1 c \rangle$. By Lemma 7.4.1.3 twice, Theorem 7.3.5 and Lemma 7.4.1.1:

- $y^0 z^0 : \langle y^0 : b \to ((a \to c) \sqcap a), z^0 : b \vdash_1 \sqcap_{i=1}^{n} (T_i \to c) \rangle$ and $n \geq 1$.

- $y^0 : \langle y^0 : b{\to}((a{\to}c) \sqcap a) \vdash_1 \sqcap_{i=1}^{n} T'_i{\to}T_i{\to}c\rangle$.
- $\sqcap_{i=1}^{n} T'_i{\to}T_i{\to}c = b{\to}((a{\to}c) \sqcap a)$.

Hence, for some $i \in \{1, \ldots, n\}$, $b = T'_i$ and $T_i{\to}c = (a{\to}c) \sqcap a$ which is absurd. □

Nevertheless, we show that β subject reduction and expansion hold in $\vdash_2$. This will be used in the proof of completeness (more specifically in Lemma 8.2.8 which is the basis of the completeness Theorem 8.2.9).

Lemma 7.4.6 (Subject reduction and expansion for $\vdash_2$ w.r.t. β)**.**

1. *If $M : \langle \Gamma \vdash_2 U\rangle$ and $M \rightarrow\!\!\!\triangleright_\beta^* N$ then $N : \langle \Gamma \vdash_2 U\rangle$.*
2. *If $N : \langle \Gamma \vdash_2 U\rangle$ and $M \rightarrow\!\!\!\triangleright_\beta^* N$ then $M : \langle \Gamma \vdash_2 U\rangle$.* □

7.4.2 Subject reduction and expansion properties for $\vdash_3$

Now we list the generation lemmas for $\vdash_3$ (for proofs see Appendix B).

Lemma 7.4.7 (Generation for $\vdash_3$)**.**

1. *If $x^L : \langle \Gamma \vdash_3 U\rangle$ then $\Gamma = (x^L : V)$ and $V \sqsubseteq U$.*
2. *If $\lambda x^L.M : \langle \Gamma \vdash_3 U\rangle$, $x^L \in \mathsf{fv}(M)$ and $\mathsf{deg}(U) = K$ then $U = \omega^K$ or $U = \sqcap_{i=1}^{p} \vec{\mathsf{e}}_K(V_i{\to}T_i)$ where $p \geq 1$ and $\forall i \in \{1, \ldots, p\}$. $M : \langle \Gamma, x^L : \vec{\mathsf{e}}_K V_i \vdash_3 \vec{\mathsf{e}}_K T_i\rangle$.*
3. *If $\lambda x^L.M : \langle \Gamma \vdash_3 U\rangle$, $x^L \notin \mathsf{fv}(M)$ and $\mathsf{deg}(U) = K$ then $U = \omega^K$ or $U = \sqcap_{i=1}^{p} \vec{\mathsf{e}}_K(V_i{\to}T_i)$ where $p \geq 1$ and $\forall i \in \{1, \ldots, p\}$. $M : \langle \Gamma \vdash_3 \vec{\mathsf{e}}_K T_i\rangle$.*
4. *If $Mx^L : \langle \Gamma, (x^L : U) \vdash_3 T\rangle$ and $x^L \notin \mathsf{fv}(M)$, then $M : \langle \Gamma \vdash_3 U{\to}T\rangle$.* □

Proof. 1. By induction on the derivation $x^L : \langle \Gamma \vdash_3 U\rangle$. 2. By induction on the derivation $\lambda x^L.M : \langle \Gamma \vdash_3 U\rangle$. 3. Same proof as that of 2. 4. By induction on the derivation $Mx^L : \langle \Gamma, x^L : U \vdash_3 T\rangle$. □

Lemma 7.4.8 (Substitution for $\vdash_3$)**.** *If $M : \langle \Gamma, x^L : U \vdash_3 V\rangle$, $N : \langle \Delta \vdash_3 U\rangle$ and $M \diamond N$ then $M[x^L := N] : \langle \Gamma \sqcap \Delta \vdash_3 V\rangle$.* □

Proof. By induction on the derivation $M : \langle \Gamma, x^L : U \vdash_3 V\rangle$. □

Since $\vdash_3$ does not allow weakening, we need the next definition since when a term is reduced, it may lose some of its free variables and hence will need to be typed in a smaller environment.

Definition 7.4.9. Let $\Gamma\upharpoonright_s$ stand for $s \lhd \Gamma$. We write $\Gamma\upharpoonright_M$ instead of $\Gamma\upharpoonright_{\mathsf{fv}(M)}$. □

Now we are ready to prove the main result of this section:

Theorem 7.4.10 (Subject reduction for $\vdash_3$)**.** *If $M : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrow\!\!\!\triangleright^*_{\beta\eta} N$ then $N : \langle \Gamma\!\upharpoonright_N \vdash_3 U \rangle$.* □

Proof. By induction on the reduction $M \rightarrow\!\!\!\triangleright^*_{\beta\eta} N$. □

Corollary 7.4.11.

1. *If $M : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrow\!\!\!\triangleright^*_{\beta} N$ then $N : \langle \Gamma\!\upharpoonright_N \vdash_3 U \rangle$.*
2. *If $M : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrow\!\!\!\triangleright^*_{h} N$ then $N : \langle \Gamma\!\upharpoonright_N \vdash_3 U \rangle$.* □

The next lemma is needed for expansion.

Lemma 7.4.12. *If $M[x^L := N] : \langle \Gamma \vdash_3 U \rangle$, $\mathsf{deg}(N) = L$, $x^L \in \mathsf{fv}(M)$, and $M \diamond N$ then there exist a type V and two type environments Γ_1, Γ_2 such that $\mathsf{deg}(V) = L$, $M : \langle \Gamma_1, x^L : V \vdash_3 U \rangle$, $N : \langle \Gamma_2 \vdash_3 V \rangle$, and $\Gamma = \Gamma_1 \sqcap \Gamma_2$.* □

Proof. By induction on the derivation $M[x^L := N] : \langle \Gamma \vdash_3 U \rangle$. □

Since more free variables might appear in the β-expansion of a term, the next definition gives a possible enlargement of an environment.

Definition 7.4.13. Let $m \geq n$, $\Gamma = (x_i^{L_i} : U_i)_n$ and $X = \{x_1^{L_1}, \ldots, x_m^{L_m}\}$. We write $\Gamma\!\uparrow^X$ for $x_1^{L_1} : U_1, \ldots, x_n^{L_n} : U_n, x_{n+1}^{L_{n+1}} : \omega^{L_{n+1}}, \ldots, x_m^{L_m} : \omega^{L_m}$. If $\mathsf{dom}(\Gamma) \subseteq \mathsf{fv}(M)$, we write $\Gamma\!\uparrow^M$ instead of $\Gamma\!\uparrow^{\mathsf{fv}(M)}$. □

We are now ready to establish that subject β-expansion holds in $\vdash_3$ (Theorem. 7.4.14) and that subject η-expansion fails (Lemma 7.4.16).

Theorem 7.4.14 (Subject β-expansion holds in $\vdash_3$)**.** *If $N : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrow\!\!\!\triangleright^*_{\beta} N$ then $M : \langle \Gamma\!\uparrow^M \vdash_3 U \rangle$.* □

Proof. By induction on the length of the derivation $M \rightarrow\!\!\!\triangleright^*_{\beta} N$ using the fact that if $\mathsf{fv}(P) \subseteq \mathsf{fv}(Q)$ then $(\Gamma\!\uparrow^P)\!\uparrow^Q = \Gamma\!\uparrow^Q$. □

Corollary 7.4.15. *If $N : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrow\!\!\!\triangleright^*_{h} N$ then $M : \langle \Gamma\!\uparrow^M \vdash_3 U \rangle$.* □

Lemma 7.4.16 (Subject η-expansion fails in $\vdash_3$)**.** *Let a be a type variable and let $x \neq y$. We have:*

1. *$\lambda y^{\oslash}.\lambda x^{\oslash}.y^{\oslash} x^{\oslash} \rightarrow\!\!\!\triangleright_{\eta} \lambda y^{\oslash}.y^{\oslash}$.*
2. *$\lambda y^{\oslash}.y^{\oslash} : \langle () \vdash_3 a \to a \rangle$.*
3. *It is not possible that: $\lambda y^{\oslash}.\lambda x^{\oslash}.y^{\oslash} x^{\oslash} : \langle () \vdash_3 a \to a \rangle$. Hence, subject η-expansion fails in $\vdash_3$.* □

Proof. 1. and 2. are easy. For 3., assume $\lambda y^{\oslash}.\lambda x^{\oslash}.y^{\oslash} x^{\oslash} : \langle () \vdash_3 a \to a \rangle$. By Lemma 7.4.7.2, $\lambda x^{\oslash}.y^{\oslash} x^{\oslash} : \langle (y : a) \vdash_3 a \rangle$. Again, by Lemma 7.4.7.2, $a = \omega^{\oslash}$ or there exists $n \geq 1$ such that $a = \sqcap_{i=1}^{n} (U_i \to T_i)$, absurd. □

Chapter 8

Realisability semantics and their completeness

8.1 Realisability

Crucial to a realisability semantics is the notion of a saturated set:

Definition 8.1.1 (Saturated sets)**.** Let $i \in \{1,2,3\}$ and $\overline{M}, \overline{M}_1, \overline{M}_2 \subseteq \mathcal{M}_i$.

1. Let $\overline{M}_1 \rightsquigarrow \overline{M}_2 = \{M \in \mathcal{M}_i \mid \forall N \in \overline{M}_1.\ M \diamond N \Rightarrow MN \in \overline{M}_2\}$.
2. Let $\overline{M}_1 \wr \overline{M}_2$ iff $\forall M \in \overline{M}_1 \rightsquigarrow \overline{M}_2.\ \exists N \in \overline{M}_1.\ M \diamond N$.
3. For $r \in \{\beta, \beta\eta, h\}$, let $\mathsf{SAT}^r = \{\overline{M} \subseteq \mathcal{M}_i \mid (M \rightarrow\!\!\!\triangleright_r^* N \wedge N \in \overline{M}) \Rightarrow M \in \overline{M}\}$. If $\overline{M} \in \mathsf{SAT}^r$ then we say that $\overline{M}$ is r-saturated. □

Saturation is closed under intersection, lifting and arrows:

Lemma 8.1.2. *Let $i \in \{1,2,3\}$, $r \in \{\beta, \beta\eta, h\}$, and $\overline{M}_1, \overline{M}_2 \subseteq \mathcal{M}_i$.*

1. *If $\overline{M}_1, \overline{M}_2$ are r-saturated sets then $\overline{M}_1 \cap \overline{M}_2$ is r-saturated.*
2. *If $\overline{M}_1 \subseteq \mathcal{M}_2$ is r-saturated then $\overline{M}_1{}^+$ is r-saturated.*
3. *If $\overline{M}_1 \subseteq \mathcal{M}_3$ is r-saturated then $\overline{M}_1^{+i}$ is r-saturated.*
4. *If $\overline{M}_2$ is r-saturated then $\overline{M}_1 \rightsquigarrow \overline{M}_2$ is r-saturated.*
5. *If $\overline{M}_1, \overline{M}_2 \subseteq \mathcal{M}_2$ then $(\overline{M}_1 \rightsquigarrow \overline{M}_2)^+ \subseteq \overline{M}_1{}^+ \rightsquigarrow \overline{M}_2{}^+$.*
6. *If $\overline{M}_1, \overline{M}_2 \subseteq \mathcal{M}_3$ then $(\overline{M}_1 \rightsquigarrow \overline{M})^{+i} \subseteq \overline{M}_1^{+i} \rightsquigarrow \overline{M}_2^{+i}$.*
7. *Let $\overline{M}_1, \overline{M}_2 \subseteq \mathcal{M}_2$. If $\overline{M}_1{}^+ \wr \overline{M}_2{}^+$, then $\overline{M}_1{}^+ \rightsquigarrow \overline{M}_2{}^+ \subseteq (\overline{M}_1 \rightsquigarrow \overline{M}_2)^+$.*
8. *Let $\overline{M}_1, \overline{M}_2 \subseteq \mathcal{M}_3$. If $\overline{M}_1^{+i} \wr \overline{M}_2^{+i}$, then $\overline{M}_1^{+i} \rightsquigarrow \overline{M}_2^{+i} \subseteq (\overline{M}_1 \rightsquigarrow \overline{M}_2)^{+i}$.*
9. *For every $n \in \mathbb{N}$, the set $\mathbb{M}^n$ is r-saturated.* □

The interpretations and meanings of types are crucial to a realisability semantics:

Definition 8.1.3 (Interpretations and meaning of types)**.** Let $\mathsf{Var} = \mathsf{Var}_1 \cup \mathsf{Var}_2$ such that $\mathsf{dj}(\mathsf{Var}_1, \mathsf{Var}_2)$ and $\mathsf{Var}_1, \mathsf{Var}_2$ are both countably infinite. Let $i \in \{1, 2, 3\}$.

1. Let $x \in \mathsf{Var}_i$ and I an index. We define the following family of sets:

$$\mathsf{VAR}_x^I = \{M \in \mathcal{M}_i \mid \exists N_1, \ldots, N_n \in \mathcal{M}_i.\ M = x^I N_1 \ldots N_n\}.$$

2. In $\lambda I^{\mathbb{N}}$, let $r = \beta$ and $I_0 = 0$. In $\lambda^{\mathcal{L}_{\mathbb{N}}}$, let $r \in \{\beta, \beta\eta, h\}$ and $I_0 = \oslash$.

   (a) An r_i-interpretation $\mathcal{I}$ is a function in $\mathsf{TyVar} \to \mathbb{P}(\mathcal{M}_i^{I_0})$ such that for all $a \in \mathsf{TyVar}$:

$$\mathcal{I}(a) \in \mathsf{SAT}^r \qquad \forall x \in \mathsf{Var}_1.\ \mathsf{VAR}_x^{I_0} \subseteq \mathcal{I}(a) \qquad \text{In } \lambda I^{\mathbb{N}},\ \mathcal{I}(a) \subseteq \mathbb{M}^0$$

   (b) We extend $\mathcal{I}$ to ITy_1 in case of $\lambda I^{\mathbb{N}}$ and to ITy_3 in case of $\lambda^{\mathcal{L}_{\mathbb{N}}}$ as follows:

In $\lambda I^{\mathbb{N}}$ and $\lambda^{\mathcal{L}_{\mathbb{N}}}$:	$\mathcal{I}(U_1 \sqcap U_2) = \mathcal{I}(U_1) \cap \mathcal{I}(U_2)$	$\mathcal{I}(U \to T) = \mathcal{I}(U) \rightsquigarrow \mathcal{I}(T)$
In $\lambda I^{\mathbb{N}}$:	$\mathcal{I}(eU) = \mathcal{I}(U)^+$	
In $\lambda^{\mathcal{L}_{\mathbb{N}}}$:	$\mathcal{I}(\mathsf{e}_i U) = \mathcal{I}(U)^{+i}$	$\mathcal{I}(\omega^L) = \mathcal{M}_3^L$

   Let $\mathsf{Interp}^{r_i} = \{\mathcal{I} \mid \mathcal{I} \text{ is a } r_i\text{-interpretation}\}$[1].

   (c) Let $U \in \mathsf{ITy}_i$. We define $[U]_{r_i}$, the r_i-interpretation of U as follows:

$$[U]_{r_i} = \{M \in \mathcal{M}_i \mid \mathsf{closed}(M) \wedge M \in \textstyle\bigcap_{\mathcal{I} \in \mathsf{Interp}^{r_i}} \mathcal{I}(U)\}$$

Because $\cap$ is commutative, associative, idempotent, $(\overline{M}_1 \cap \overline{M}_2)^+ = \overline{M}_1{}^+ \cap \overline{M}_2{}^+$ in $\lambda I^{\mathbb{N}}$, $(\overline{M}_1 \cap \overline{M}_2)^{+i} = \overline{M}_1^{+i} \cap \overline{M}_2^{+i}$ in $\lambda^{\mathcal{L}_{\mathbb{N}}}$, and $\mathcal{I}$ is well defined. □

Type interpretations are saturated and interpretations of good types contain only good terms.

Lemma 8.1.4. *Let $r \in \{\beta, \beta\eta, h\}$. Let $i \in \{1, 2, 3\}$.*

1. *(a) For all $U \in \mathsf{ITy}_i$ and $\mathcal{I} \in \mathsf{Interp}^{r_i}$, we have $\mathcal{I}(U) \in \mathsf{SAT}^r$.*

   (b) If $\mathsf{deg}(U) = L$ and $\mathcal{I} \in \mathsf{Interp}^{r_3}$ then $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}_x^L \subseteq \mathcal{I}(U) \subseteq \mathcal{M}_3^L$.

   (c) On ITy_1 (hence also on ITy_2), if $U \in \mathsf{GITy}$, $\mathsf{deg}(U) = n$, and $\mathcal{I} \in \mathsf{Interp}^{r_2}$ then $\forall x \in \mathsf{Var}_1.\ x^n \in \mathsf{VAR}_x^n \subseteq \mathcal{I}(U) \subseteq \mathbb{M}^n$.

2. *Let $i \in \{2, 3\}$. If $\mathcal{I} \in \mathsf{Interp}^{r_i}$ and $U \sqsubseteq V$ then $\mathcal{I}(U) \subseteq \mathcal{I}(V)$.* □

Proof. 1a . By induction on U using Lemma 8.1.2. 1b. By induction on U. 1c. By definition, $x^n \in \mathsf{VAR}_x^n$. We prove $\mathsf{VAR}_x^n \subseteq \mathcal{I}(U) \subseteq \mathbb{M}^n$ by induction on $U \in \mathsf{GITy}$. 2. By induction of the derivation $U \sqsubseteq V$. □

[1] We effectively define five interpretation sets $\mathsf{Interp}^{\beta_1}$, $\mathsf{Interp}^{\beta_2}$, $\mathsf{Interp}^{\beta_3}$, $\mathsf{Interp}^{\beta\eta_3}$, and Interp^{h_3}

Corollary 8.1.5 (Meanings of good types consist of good terms)**.** *On* ITy_1 *(hence also on* ITy_2*), if* $U \in \mathsf{GITy}$ *such that* $\mathsf{deg}(U) = n$ *then* $[U]_{\beta_2} \subseteq \mathbb{M}^n$*.* □

Proof. By Lemma 8.1.4.1c, for any interpretation $\mathcal{I} \in \mathsf{Interp}^{\beta_2}$, $\mathcal{I}(U) \subseteq \mathbb{M}^n$. □

Lemma 8.1.6 (Soundness of $\vdash_1$, $\vdash_2$, and $\vdash_3$)**.** *Let* $i \in \{1, 2, 3\}$*,* $r \in \{\beta, \beta\eta, h\}$*,* $\mathcal{I} \in \mathsf{Interp}^{r_i}$*. If* $M : \langle (x_j^{I_j} : U_j)_n \vdash_i U\rangle$*,* $\forall j \in \{1, \ldots, n\}$*.* $N_j \in \mathcal{I}(U_j)$*, and* $\diamond\{M, N_1, \ldots, N_n\}$ *then* $M[(x_j^{I_j} := N_j)_n] \in \mathcal{I}(U)$*.* □

Proof. By induction on the derivation $M : \langle (x_j^{I_j} : U_j)_n \vdash_i U\rangle$. □

Corollary 8.1.7. *Let* $r \in \{\beta, \beta\eta, h\}$ *and* $i \in \{1, 2, 3\}$*. If* $M : \langle () \vdash_i U\rangle$ *then* $M \in [U]_{r_i}$*.* □

Proof. By Lemma 8.1.6, $M \in \mathcal{I}(U)$ for any $\mathcal{I} \in \mathsf{Interp}^{r_i}$. By Theorem 7.3.5, $\mathsf{fv}(M) = \mathsf{dom}(()) = \varnothing$ and hence M is closed. Therefore, $M \in [U]_{r_i}$. □

Lemma 8.1.8 (The meaning of types is closed under type operations)**.** *Let* $r \in \{\beta, \beta\eta, h\}$ *and* $j \in \{1, 2, 3\}$*. The following hold:*

1. $[\mathsf{e}_i U]_{r_3} = [U]_{r_3}^{+i}$ *and if* $j \neq 3$ *then* $[eU]_{r_j} = [U]_{r_j}{}^{+}$*.*

2. $[U \sqcap V]_{r_j} = [U]_{r_j} \cap [V]_{r_j}$*.*

3. *If* $U \to T \in \mathsf{ITy}_3$ *then* $\forall \mathcal{I} \in \mathsf{Interp}^{r_3}$*.* $\mathcal{I}(U) \wr \mathcal{I}(T)$*.*

4. *If* $U \to T \in \mathsf{GITy}$ *then* $\forall \mathcal{I} \in \mathsf{Interp}^{\beta_2}$*.* $\mathcal{I}(U) \wr \mathcal{I}(T)$*.*

5. *On* ITy_1 *only (since* $eU \to eT \notin \mathsf{ITy}_2$*), we have: if* $U \to T \in \mathsf{GITy}$ *then* $[e(U \to T)]_{\beta_2} = [eU \to eT]_{\beta_2}$*.* □

Proof. 1. and 2. are easy.

3. Let $\mathsf{deg}(U) = L$, $M \in \mathcal{I}(U) \rightsquigarrow \mathcal{I}(T)$ and $x \in \mathsf{Var}_1$ such that $\forall K.\ x^K \notin \mathsf{fv}(M)$, hence $M \diamond x^L$ and by Lemma 8.1.4, $x^L \in \mathcal{I}(U)$.

4. Let $\mathsf{deg}(U) = n$ and $M \in \mathcal{I}(U) \rightsquigarrow \mathcal{I}(T)$. Take $x \in \mathsf{Var}_1$ such that $\forall p.\ x^p \notin \mathsf{fv}(M)$. Hence, $M \diamond x^n$. By Lemma 7.2.3, $U \in \mathsf{GITy}$ and by Lemma 8.1.4, $x^n \in \mathcal{I}(U)$.

5. Since $U \to T \in \mathsf{GITy}$ then, by Lemma 7.2.3, $U, T \in \mathsf{GITy}$ and $\mathsf{deg}(U) \geq \mathsf{deg}(T)$. Again by Lemma 7.2.3, $eU, eT \in \mathsf{GITy}$, $\mathsf{deg}(eU) \geq \mathsf{deg}(eT)$ and $eU \to eT \in \mathsf{GITy}$. Hence by 4., $\mathcal{I}(U)^+ \wr \mathcal{I}(T)^+$. Thus, by Lemma 8.1.2.5 and Lemma 8.1.2.7, $\forall \mathcal{I} \in \mathsf{Interp}^{\beta_2}$. $\mathcal{I}(e(U \to T)) = \mathcal{I}(eU \to eT)$. □

Let us now put the realisability semantics in use.

Example 8.1.9. Let a and b be two distinct type variables in TyVar. We define:

- $\mathsf{id}_0 = \mathtt{a}{\rightarrow}\mathtt{a}$ and $\mathtt{id}_1 = \mathtt{e}_1(\mathsf{id}_0)$.
- $\mathsf{d} = (\mathtt{a} \sqcap (\mathtt{a}{\rightarrow}\mathtt{b})){\rightarrow}\mathtt{b}$.
- $\mathsf{nat}_0 = (\mathtt{a}{\rightarrow}\mathtt{a}){\rightarrow}(\mathtt{a}{\rightarrow}\mathtt{a})$, $\mathtt{nat}_1 = \mathtt{e}_1(\mathsf{nat}_0)$, and $\mathtt{nat}'_0 = (\mathtt{e}_1\mathtt{a}{\rightarrow}\mathtt{a}){\rightarrow}(\mathtt{e}_1\mathtt{a}{\rightarrow}\mathtt{a})$.

Moreover, if M, N are terms and $n \in \mathbb{N}$, we define $(M)^n N$ by induction on n as follows: $(M)^0 N = N$ and $(M)^{m+1} N = M((M)^m N)$.

We now illustrate our realisability semantics by providing the meaning of the types defined above:

1. $[(\mathtt{a} \sqcap \mathtt{b}){\rightarrow}\mathtt{a}]_{\beta_1} = \{M \in \mathbb{M}^0 \mid M \rightarrowtriangle^*_\beta \lambda y^0.y^0\}$.
2. It is not possible that $\lambda y^0.y^0 : \langle () \vdash_1 (\mathtt{a} \sqcap \mathtt{b}){\rightarrow}\mathtt{a}\rangle$.
3. $\lambda y^0.y^0 : \langle () \vdash_2 (\mathtt{a} \sqcap \mathtt{b}){\rightarrow}\mathtt{a}\rangle$.
4. $[\mathsf{id}_0]_{\beta_3} = \{M \in \mathcal{M}_3^{\oslash} \mid \mathsf{closed}(M) \wedge M \rightarrowtriangle^*_\beta \lambda y^{\oslash}.y^{\oslash}\}$.
5. $[\mathsf{id}_1]_{\beta_3} = \{M \in \mathcal{M}_3^{(1)} \mid \mathsf{closed}(M) \wedge M \rightarrowtriangle^*_\beta \lambda y^{(1)}.y^{(1)}\}$.
6. $[\mathsf{d}]_{\beta_3} = \{M \in \mathcal{M}_3^{\oslash} \mid \mathsf{closed}(M) \wedge M \rightarrowtriangle^*_\beta \lambda y^{\oslash}.y^{\oslash}y^{\oslash}\}$.
7. $[\mathsf{nat}_0]_{\beta_3} = \{M \in \mathcal{M}_3^{\oslash} \mid \mathsf{closed}(M) \wedge (M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.f^{\oslash} \vee (n \geq 1 \wedge M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.\lambda y^{\oslash}.(f^{\oslash})^n y^{\oslash}))\}$.
8. $[\mathsf{nat}_1]_{\beta_3} = \{M \in \mathcal{M}_3^{(1)} \mid \mathsf{closed}(M) \wedge (M \rightarrowtriangle^*_\beta \lambda f^{(1)}.f^{(1)} \vee (n \geq 1 \wedge M \rightarrowtriangle^*_\beta \lambda f^{(1)}.\lambda x^{(1)}.(f^{(1)})^n y^{(1)}))\}$.
9. $[\mathsf{nat}'_0]_{\beta_3} = \{M \in \mathcal{M}_3^{\oslash} \mid \mathsf{closed}(M) \wedge (M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.f^{\oslash} \vee M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.\lambda y^{(1)}.f^{\oslash}y^{(1)})\}$.

□

8.2 Completeness challenges in $\lambda I^{\mathbb{N}}$

In this document we consider two realisability semantics of types involving E-variables. These semantics are based on a hierarchy of types and terms. Considering how expansions can introduce new substitutions, new expansions and an unbound number of new variables (type variables and E-variables), it was decided to use a hierarchy on types and terms to give meanings to expansions to represent the encapsulation of types by E-variables. An obvious (and naive) approach is to label types and terms with natural numbers. This is the hierarchy we used in $\lambda I^{\mathbb{N}}$. When assigning meanings to types, we ensured that each use of an E-variable in a typing simply changes the indexes of types and terms in the typing and that each E-variable acted as a kind of capsule that isolates parts of the analysed λ-term in a typing. This captured the intuition behind E-variables. However, there are two issues w.r.t.

this indexing: it imposes that the type ω should have all possible indexes (which is impossible[2] and hence we eliminated ω from the type systems for $\mathcal{M}_2$) and it implies that the realisability semantics can only be complete when a single E-variable is used (as we will see in this section). In order to understand the challenges of the semantics of E-variables with ω and the idea behind the hierarchy, we first studied two representative intersection type systems for the λI-calculus. The restriction to λI (where in every $(\lambda x.M)$ the variable x must occur free in M) was motivated by not supporting the ω type while preserving the intuitive indexes made of single natural numbers. For $\vdash_1$, the first of these type systems, we showed that subject reduction and hence completeness do not hold.

8.2.1 Completeness for $\vdash_1$ fails

Remark 8.2.1 (Failure of completeness for $\vdash_1$). Items 1., 2., and 3. of Example 8.1.9 show that we can not have a completeness result (a converse of the soundness Lemma 8.1.6 for closed terms) for $\vdash_1$. To type the term $\lambda y^0.y^0$ by the type $(a \sqcap b) \to a$, we need an elimination rule for $\sqcap$ which we do not have in $\vdash_1$. □

Note that failure of completeness for $\vdash_1$ is related to the failure of its subject reduction. So, one might think that since $\vdash_2$, the second type system for $\lambda I^{\mathbb{N}}$, has subject reduction, its semantics is complete. This is not entirely true.

8.2.2 Completeness for $\vdash_2$ fails with more than one E-variable

Remark 8.2.2 (Failure of completeness for $\vdash_2$ if more than one E-variable are used). Let a be a type variable, e_1 and e_2 be two distinct expansion variable, and $\mathsf{nat}''_0 = (\mathsf{e}_1\mathsf{a} \to \mathsf{a}) \to (\mathsf{e}_2\mathsf{a} \to \mathsf{a})$. Then:

1. $\lambda f^0.f^0 \in [\mathsf{nat}''_0]_{\beta_2}$.

2. it is not possible that $\lambda f^0.f^0 : \langle () \vdash_2 \mathsf{nat}''_0 \rangle$.

Hence $\lambda f^0.f^0 \in [\mathsf{nat}''_0]_{\beta_2}$ but $\lambda f^0.f^0$ is not typable by nat''_0 and we do not have completeness in the presence of more than one expansion variable. □

However, we will see that we have completeness for $\vdash_2$ if only one expansion variable is used.

8.2.3 Completeness for $\vdash_2$ with only one E-variable

The problem shown in remark 8.2.2 comes from the fact that the realisability semantics designed for $\vdash_2$ identifies all expansion variables. In order to give a completeness

[2] Let us assume that that our type language contains the ω type annotated with integers, i.e., of the form ω^n, then we would need $\mathsf{e}_1\omega^n = \omega^{n+1}$ and $\mathsf{e}_2\omega^n = \omega^{n+1}$, and finally we would have $\mathsf{e}_1\omega^n = \mathsf{e}_2\omega^n$.

theorem for $\vdash_2$ we will, in what follows, restrict our system to only one expansion variable. In the rest of this section, we assume that the set ExpVar contains only one expansion variable e_1.

The need of one single expansion variable is clear in item 2. of Lemma 8.2.3 which would fail if we use more than one expansion variable. For example, if $e_1 \neq e_2$ then $(e_1 a)^- = a = (e_2 a)^-$ but $e_1 a \neq e_2 a$. This lemma is crucial for the rest of this section and hence, a single expansion variable is also crucial.

Lemma 8.2.3. *Let $U, V \in \mathsf{ITy}_2$ and $\mathsf{deg}(U) = \mathsf{deg}(V) > 0$.*

1. $\mathsf{e}_1 U^- = U$.
2. *If $U^- = V^-$ then $U = V$.* □

Proof. 1. is by induction on U. 2. goes as follows: if $U^- = V^-$ then $\mathsf{e}_1 U^- = \mathsf{e}_1 V^-$ and by 1., $U = V$. □

Despite the difference in the number of considered expansion variables in the completeness proof presented in the current section and the one of Sec. 8.3, both proofs share some similarities. We still write these two proofs independently to illustrate the method and especially since the proof in the current section is far simpler. Furthermore, in the current section we only show the completeness of our semantics w.r.t. β-reduction.

The first step of the proof is to divide $\{y^n \mid y \in \mathsf{Var}_2\}$ into disjoint subset amongst types of order n.

Definition 8.2.4. Let $U \in \mathsf{ITy}_2$. We define the set of variables DVar_U by induction on $\mathsf{deg}(U)$. If $\mathsf{deg}(U) = 0$ then DVar_U is an infinite set $\{y^0 \mid y \in \mathsf{Var}_2\}$ such that if $U \neq V$ and $\mathsf{deg}(U) = \mathsf{deg}(V) = 0$ then $\mathsf{dj}(\mathsf{DVar}_U, \mathsf{DVar}_V)$. If $\mathsf{deg}(U) = n+1$ then $\mathsf{DVar}_U = \{y^{n+1} \mid y^n \in \mathsf{DVar}_{U^-}\}$. □

Our partition of Var_2 allows useful infinite sets containing type environments that will play a crucial role in one particular type interpretation. These sets and environments are given in the next definition.

Definition 8.2.5.

- Let $\mathsf{IPreEnv}^n = \{(\!|y^n, U|\!) \mid U \in \mathsf{ITy}_2 \wedge \mathsf{deg}(U) = n \wedge y^n \in \mathsf{DVar}_U\}$ and $\mathsf{BPreEnv}^n = \bigcup_{m \geq n} \mathsf{IPreEnv}^m$ (where "I" stands for "index" and "B" stands for "bound"). Note that $\mathsf{IPreEnv}^n$ and $\mathsf{BPreEnv}^n$ are not type environments because they are not functions.
- If $M \in \mathcal{M}_2$ and $U \in \mathsf{ITy}_2$ then we write $M : \langle \mathsf{BPreEnv}^n \vdash_2 U \rangle$ iff there is a type environment $\Gamma \subseteq \mathsf{BPreEnv}^n$ where $M : \langle \Gamma \vdash_2 U \rangle$. □

Now, for every n, we define the set of the good terms of order n which contain some free variable x^i where $x \in \mathsf{Var}_1$ and $i \geq n$.

Definition 8.2.6. Let $\mathsf{OPEN}^n = \{M \in \mathbb{M}^n \mid x^i \in \mathsf{fv}(M) \wedge x \in \mathsf{Var}_1 \wedge i \geq n\}$. □

Obviously, if $x \in \mathsf{Var}_1$ then $\mathsf{VAR}^n_x \subseteq \mathsf{OPEN}^n$.

Here is the crucial β_2-interpretation $\mathbb{I}$ for the proof of completeness:

Definition 8.2.7. Let $\mathbb{I}$ be the β_2-interpretation defined as follows: for all type variables a, $\mathbb{I}(a) = \mathsf{OPEN}^0 \cup \{M \in \mathcal{M}^0_2 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 a\rangle\}$. □

The function $\mathbb{I}$ is indeed a β_2-interpretation and the interpretation of a type of order n contains the good terms of order n which are typable in the special environments which are parts of the infinite sets of definition 8.2.5:

Lemma 8.2.8.

1. *$\mathbb{I}$ is a β_2-interpretation, i.e., for all $a \in \mathsf{TyVar}$, $\mathbb{I}(a)$ is β-saturated and $\forall x \in \mathsf{Var}_1$, $\mathsf{VAR}^0_x \subseteq \mathbb{I}(a) \subseteq \mathbb{M}^0$.*

2. *If $U \in \mathsf{ITy}_2 \cap \mathsf{GITy}$ and $\mathsf{deg}(U) = n$ then $\mathbb{I}(U) = \mathsf{OPEN}^n \cup \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U\rangle\}$.* □

Proof. We prove 1. by first showing that $\mathbb{I}(a)$ is saturated: if $M \rightarrow\!\!\triangleright^*_\beta N$ then if $N \in \mathsf{OPEN}^0$ we prove that $M \in \mathsf{OPEN}^0$ and if $N \in \{M \in \mathcal{M}^0_2 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 a\rangle\}$ then $M \in \{M \in \mathcal{M}^0_2 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 a\rangle\}$. We then show $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}^0_x \subseteq \mathbb{I}(a) \subseteq \mathbb{M}^0$. We prove 2. by induction on $U \in \mathsf{GITy}$. □

$\mathbb{I}$ is used to prove completeness (see Appendix B for the proof).

Theorem 8.2.9 (Completeness)**.** *Let $U \in \mathsf{ITy}_2 \cap \mathsf{GITy}$ such that $\mathsf{deg}(U) = n$. The following hold:*

1. *$[U]_{\beta_2} = \{M \in \mathbb{M}^n \mid M : \langle () \vdash_2 U\rangle\}$.*

2. *$[U]_{\beta_2}$ is stable by reduction: if $M \in [U]_{\beta_2}$ and $M \rightarrow\!\!\triangleright^*_\beta N$ then $N \in [U]_{\beta_2}$.*

3. *$[U]_{\beta_2}$ is stable by expansion: if $N \in [U]_{\beta_2}$ and $M \rightarrow\!\!\triangleright^*_\beta N$ then $M \in [U]_{\beta_2}$.* □

Proof. The first item follows by Lemmas 8.2.8 and 8.1.6. We obtain the second item using subject reduction and the third one using subject expansion. □

8.3 Completeness for $\lambda^{\mathcal{L}_{\mathbb{N}}}$

Having understood the challenges of E-variables and the difficulty of representing the type ω using natural numbers as indices for the hierarchy, we moved to the presentation of indices as sequences of natural numbers and we provided our third type system $\vdash_3$. We developed a realisability semantics where we allow the full λ-calculus (i.e., where K-redexes are allowed) indexed with lists of natural numbers, an arbitrary (possibly infinite) number of expansion variables and where ω is present, and we showed its soundness. Now, we show its completeness.

We need the following partition of the set of indexed variables $\{y^L \mid y \in \mathsf{Var}_2\}$.

Definition 8.3.1.

- Let $\mathsf{ITy}_3^L = \{U \in \mathsf{ITy}_3 \mid \mathsf{deg}(U) = L\}$ and $\mathsf{Var}^L = \{x^L \mid x \in \mathsf{Var}_2\}$.
- We inductively define, for every $U \in \mathsf{ITy}_3$, a set of variables DVar_U as follows:
  - If $\mathsf{deg}(U) = \oslash$ then:
    - DVar_U is an infinite set of indexed variables of degree $\oslash$.
    - If $U \neq V$ and $\mathsf{deg}(U) = \mathsf{deg}(V) = \oslash$ then $\mathsf{dj}(\mathsf{DVar}_U, \mathsf{DVar}_V)$.
    - $\bigcup_{U \in \mathsf{ITy}_3^{\oslash}} \mathsf{DVar}_U = \mathsf{Var}^{\oslash}$.
  - If $\mathsf{deg}(U) = i :: L$ then $\mathsf{DVar}_U = \{y^{i::L} \mid y^L \in \mathsf{DVar}_{U^{-i}}\}$.

  Therefore, if $\mathsf{deg}(U) = L$ then $\mathsf{DVar}_U = \{y^L \mid y^{\oslash} \in \mathsf{DVar}_{U^{-L}}\}$. □

Let us now provide some simple results concerning the DVar_U sets:

Lemma 8.3.2.

1. *If* $\mathsf{deg}(U) \succeq L$, $\mathsf{deg}(V) \succeq L$, *and* $U^{-L} = V^{-L}$ *then* $U = V$.
2. *If* $\mathsf{deg}(U) = L$ *then* DVar_U *is an infinite subset of* Var^L.
3. *If* $U \neq V$ *and* $\mathsf{deg}(U) = \mathsf{deg}(V) = L$ *then* $\mathsf{dj}(\mathsf{DVar}_U, \mathsf{DVar}_V)$.
4. $\bigcup_{U \in \mathsf{ITy}_3^L} \mathsf{DVar}_U = \mathsf{Var}^L$.
5. *If* $y^L \in \mathsf{DVar}_U$ *then* $y^{i::L} \in \mathsf{DVar}_{\mathsf{e}_i U}$.
6. *If* $y^{i::L} \in \mathsf{DVar}_U$ *then* $y^L \in \mathsf{DVar}_{U^{-i}}$. □

Proof. 1. goes as follows: if $L = (n_i)_m$ then we have $U = \mathsf{e}_{n_1} \dots \mathsf{e}_{n_m} U'$ and $V = \mathsf{e}_{n_1} \dots \mathsf{e}_{n_m} V'$; then $U^{-L} = U'$, $V^{-L} = V'$ and $U' = V'$; thus $U = V$. 2., 3. and 4. are by induction on L and using 1. We obtain 5. because $(\mathsf{e}_i U)^{-i} = U$. 6. is by definition. □

The set Var_2 as defined above allows us to give in the next definition useful infinite sets containing type environments that will play a crucial role in one particular type interpretation.

Definition 8.3.3.

- Let $L \in \mathcal{L}_{\mathbb{N}}$. We denote $\mathsf{IPreEnv}^L = \{(\!| y^L, U |\!) \mid U \in \mathsf{ITy}_3^L \wedge y^L \in \mathsf{DVar}_U\}$ and $\mathsf{BPreEnv}^L = \bigcup_{K \succeq L} \mathsf{IPreEnv}^K$. Note that $\mathsf{IPreEnv}^L$ and $\mathsf{BPreEnv}^L$ are not type environments because they are not functions.
- Let $L \in \mathcal{L}_{\mathbb{N}}$, $M \in \mathcal{M}_3$ and $U \in \mathsf{ITy}_3$, we write:
  - $M : \langle \mathsf{BPreEnv}^L \vdash_3 U \rangle$ iff there exists a type environment $\Gamma \subseteq \mathsf{BPreEnv}^L$ such that $M : \langle \Gamma \vdash_3 U \rangle$.
  - $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U \rangle$ iff $M \rightarrow\!\!\!\triangleright_{\beta\eta}^* N$ and $N : \langle \mathsf{BPreEnv}^L \vdash_3 U \rangle$.

□

Let us now provide some results concerning the $\mathsf{BPreEnv}^L$ sets:

Lemma 8.3.4.

1. *If* $\Gamma \subseteq \mathsf{BPreEnv}^L$ *then* $\mathsf{ok}(\Gamma)$.
2. *If* $\Gamma \subseteq \mathsf{BPreEnv}^L$ *then* $\mathsf{e}_i\Gamma \subseteq \mathsf{BPreEnv}^{i::L}$.
3. *If* $\Gamma \subseteq \mathsf{BPreEnv}^{i::L}$ *then* $\Gamma^{-i} \subseteq \mathsf{BPreEnv}^L$.
4. *If* $\Gamma_1 \subseteq \mathsf{BPreEnv}^L$, $\Gamma_2 \subseteq \mathsf{BPreEnv}^K$, *and* $L \preceq K$ *then* $\Gamma_1 \sqcap \Gamma_2 \subseteq \mathsf{BPreEnv}^L$. □

Proof. 1. is by definition. 2. and 3. are by Lemma 8.3.2. 4. First, by 1., $\Gamma_1 \sqcap \Gamma_2$ is well defined. Also, $\mathsf{BPreEnv}^K \subseteq \mathsf{BPreEnv}^L$. Let $(\Gamma_1 \sqcap \Gamma_2)(x^{L'}) = U_1 \sqcap U_2$ where $\Gamma_1(x^{L'}) = U_1$ and $\Gamma_2(x^{L'}) = U_2$, then $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = L'$ and $x^{L'} \in \mathsf{DVar}_{U_1} \cap \mathsf{DVar}_{U_2}$. Hence, by Lemma 8.3.2.3, $U_1 = U_2$ and $\Gamma_1 \sqcap \Gamma_2 = \Gamma_1 \cup \Gamma_2 \subseteq \mathsf{BPreEnv}^L$. □

For every $L \in \mathcal{L}_{\mathbb{N}}$, we define the set of terms of degree L which contain some free variable x^K where $x \in \mathsf{Var}_1$ and $K \succeq L$.

Definition 8.3.5. For every $L \in \mathcal{L}_{\mathbb{N}}$, let $\mathsf{OPEN}^L = \{M \in \mathcal{M}_3^L \mid x^K \in \mathsf{fv}(M) \wedge x \in \mathsf{Var}_1 \wedge K \succeq L\}$. It is easy to see that, for every $L \in \mathcal{L}_{\mathbb{N}}$ and $x \in \mathsf{Var}_1$, $\mathsf{VAR}_x^L \subseteq \mathsf{OPEN}^L$. □

Let us now provide some results on the OPEN^L sets:

Lemma 8.3.6.

1. $(\mathsf{OPEN}^L)^{+i} = \mathsf{OPEN}^{i::L}$.

2. *If* $y \in \mathsf{Var}_2$ *and* $My^K \in \mathsf{OPEN}^L$ *then* $M \in \mathsf{OPEN}^L$.

3. *If* $M \in \mathsf{OPEN}^L$, $M \diamond N$, *and* $L \preceq K = \mathsf{deg}(N)$ *then* $MN \in \mathsf{OPEN}^L$.

4. *If* $\mathsf{deg}(M) = L$, $L \preceq K$, $M \diamond N$, *and* $N \in \mathsf{OPEN}^K$ *then* $MN \in \mathsf{OPEN}^L$. □

Proof. Easy using Def. 8.3.5. □

The crucial interpretation $\mathbb{I}$ (the three interpretations $\mathbb{I}_{\beta\eta}$, $\mathbb{I}_{\beta}$, and $\mathbb{I}_h$ for our three reduction relations) used in the completeness proof is given as follows:

Definition 8.3.7.

1. Let $\mathbb{I}_{\beta\eta}$ be the $\beta\eta_3$-interpretation defined by: for all type variables a, $\mathbb{I}_{\beta\eta}(a) = \mathsf{OPEN}^{\oslash} \cup \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* a\rangle\}$.

2. Let $\mathbb{I}_{\beta}$ be the β_3-interpretation defined by: for all type variables a, $\mathbb{I}_{\beta}(a) = \mathsf{OPEN}^{\oslash} \cup \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 a\rangle\}$.

3. Let $\mathbb{I}_h$ be the h_3-interpretation defined by: for all type variables a, $\mathbb{I}_h(a) = \mathsf{OPEN}^{\oslash} \cup \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 a\rangle\}$. □

The next crucial lemma shows that $\mathbb{I}$ (the three functions $\mathbb{I}_{\beta\eta}$, $\mathbb{I}_{\beta}$, and $\mathbb{I}_h$) is an interpretation and that the interpretation of a type of order L contains terms of order L which are typable in these special environments which are parts of the infinite sets of Def. 8.3.3.

Lemma 8.3.8. *Let* $r \in \{\beta\eta, \beta, h\}$ *and* $r' \in \{\beta, h\}$.

1. *If* $\mathbb{I}_r \in \mathsf{Interp}^{r_3}$ *and* $a \in \mathsf{TyVar}$ *then* $\mathbb{I}_r(a) \in \mathsf{SAT}^r$ *and* $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}_x^{\oslash} \subseteq \mathbb{I}_r(a)$.

2. *If* $U \in \mathsf{ITy}_3$ *and* $\mathsf{deg}(U) = L$ *then* $\mathbb{I}_{\beta\eta}(U) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U\rangle\}$.

3. *If* $U \in \mathsf{ITy}_3$ *and* $\mathsf{deg}(U) = L$ *then* $\mathbb{I}_{r'}(U) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U\rangle\}$. □

Proof. We prove the first item by first showing that $\mathbb{I}_r(a)$ is saturated: if $M \rightarrowtriangle_r^* N$ then if $N \in \mathsf{OPEN}^{\oslash}$ we prove that $M \in \mathsf{OPEN}^{\oslash}$ and if $N \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* a\rangle\}$ then $M \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* a\rangle\}$. We then show that for all $x \in \mathsf{Var}_1$, $\mathsf{VAR}_x^{\oslash} \subseteq \mathsf{OPEN}^{\oslash} \subseteq \mathbb{I}_r(a)$. We prove the second and third items by induction on U. □

Now, we use this crucial $\mathbb{I}$ to establish completeness of our semantics.

Theorem 8.3.9 (Completeness of $\vdash_3$)**.** *Let* $U \in \mathsf{ITy}_3$ *such that* $\mathsf{deg}(U) = L$.

1. $[U]_{\beta\eta_3} = \{M \in \mathcal{M}_3^L \mid \mathsf{closed}(M) \wedge M \rightarrowtriangle_{\beta\eta}^* N \wedge N : \langle () \vdash_3 U\rangle\}$.

2. $[U]_{\beta_3} = [U]_{h_3} = \{M \in \mathcal{M}_3^L \mid M : \langle () \vdash_3 U \rangle\}$.

3. $[U]_{\beta\eta_3}$ *is stable by reduction: if* $M \in [U]_{\beta\eta_3}$ *and* $M \rightarrow_{\beta\eta} N$ *then* $N \in [U]_{\beta\eta_3}$. □

Proof.

1. Let $M \in [U]_{\beta\eta_3}$. Then M is closed and $M \in \mathbb{I}_{\beta\eta}(U)$. By Lemma 8.3.8.2, $M \in \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U \rangle\}$. Since M is closed, $M \notin \mathsf{OPEN}^L$. Hence, $M \in \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U \rangle\}$ and so, $M \rightarrow_{\beta\eta}^* N$ and $N : \langle \Gamma \vdash_3 U \rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^L$. By Theorem 7.1.11.2, N is closed and, by Lemma 7.3.5.2a, $N : \langle () \vdash_3 U \rangle$.

   Conversely, take M closed such that $M \rightarrow_{\beta}^* N$ and $N : \langle () \vdash_3 U \rangle$. Let $\mathcal{I} \in \mathsf{Interp}^{\beta_3}$. By Lemma 8.1.6, $N \in \mathcal{I}(U)$. By Lemma 8.1.4.1, $\mathcal{I}(U)$ is $\beta\eta$-saturated. Hence, $M \in \mathcal{I}(U)$. Thus $M \in [U]_{\beta\eta_3}$.

2. Let $M \in [U]_{\beta_3}$. Then M is closed and $M \in \mathbb{I}_{\beta}(U)$. By Lemma 8.3.8.3, $M \in \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U \rangle\}$. Since M is closed, $M \notin \mathsf{OPEN}^L$. Hence, $M \in \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U \rangle\}$ and so, $M : \langle \Gamma \vdash_3 U \rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^L$. By Lemma 7.3.5.2a, $N : \langle () \vdash_3 U \rangle$.

   Conversely, take M such that $M : \langle () \vdash_3 U \rangle$. By Lemma 7.3.5.2a, M is closed. Let $\mathcal{I} \in \mathsf{Interp}^{\beta_3}$. By Lemma 8.1.6, $M \in \mathcal{I}(U)$. Thus $M \in [U]_{\beta_3}$.

   It is easy to see that $[U]_{\beta_3} = [U]_{h_3}$.

3. Let $M \in [U]_{\beta\eta_3}$ and $M \rightarrow_{\beta\eta} N$. By 1., M is closed, $M \rightarrow_{\beta\eta}^* P$, and $P : \langle () \vdash_3 U \rangle$. By confluence Theorem 7.1.13, there is Q such that $P \rightarrow_{\beta\eta}^* Q$ and $N \rightarrow_{\beta\eta}^* Q$. By subject reduction Theorem 7.4.10, $Q : \langle () \vdash_3 U \rangle$. By Theorem 7.1.11.2, N is closed and, by 1., $N \in [U]_{\beta\eta_3}$. □

Chapter 9

Conclusion and future work

Expansion may be viewed to work like a multi-layered simultaneous substitution. Moreover, expansion is a crucial part of a procedure for calculating principal typings and helps support compositional type inference. Because the early definitions of expansion were complicated, expansion variables (E-variables) were introduced to simplify and mechanize expansion. The aim of this document is to give a complete semantics for intersection type systems with expansion variables.

We studied first the $\lambda I^{\mathbb{N}}$-calculus, an indexed version of the λI-calculus. This indexed version was typed using first a basic intersection type system with expansion variables but without an intersection elimination rule, and then using an intersection type system with expansion variables and an elimination rule.

We gave a realisability semantics for both type systems showing that the first type system is not complete in the sense that there are types whose semantics is not the set of $\lambda I^{\mathbb{N}}$-terms having this type. In particular, we showed that $\lambda y^0.y^0$ is in the semantics of $(\mathtt{a} \sqcap \mathtt{b}) \to \mathtt{a}$ but that it is not possible to give $\lambda y^0.y^0$ the type $(\mathtt{a} \sqcap \mathtt{b}) \to \mathtt{a}$ in the type system $\vdash_1$ (see Example 8.1.9 in Ch. 8.1). The main reason for the failure of completeness in the first system is associated with the failure of the subject reduction property for this first type system. Hence, we moved to the second system which we showed to have the desirable properties of subject reduction and expansion and strong normalisation. However, for this second system, we showed again that completeness fails if we use more than one expansion variable but that completeness succeeds if we restrict the system to a single expansion variable.

In order to overcome the problems of completeness, we changed our realisability semantics from one which uses natural numbers as indices to one that uses lists of natural numbers as indices. The new semantics is more complex and we lose the elegance of the first (especially in being able to define the good terms and good types). However, we consider a third type system for this new indexed calculus and we show that is has all the desirable properties of a type system and it handles all of the λ-calculus (not simply the λI-calculus). We also show that this second semantics is complete when any number (including infinite) of expansion variables is used w.r.t. our third type system. As far as we know, our work constitutes the first study of a realisability semantics of intersection type systems with E-variables and of the difficulties involved.

Note that a restricted version (restricted to normalised types[1]), which we call RCDV, of the well known CDV intersection type system (see Sec.2.4.2), both systems introduced by Coppo, Dezani and Venneri [27, 28] and recalled by Van Bakel [4], can be embedded in our type system $\vdash_3$ without making use of expansion variables (a more detailed remark can be found in Sec. B.3). We can then restrain the range of our interpretations (see Def. 8.1.3) from $\mathcal{M}_3$ to the "space of meaning" $\mathcal{M}_3^{\oslash}$ (see Def. 7.1.9) which is then the only necessary set because expansion variables are not used and therefore they do not allow one to change the index of terms. Unfortunately, we do not believe that it would be possible to embed RCDV in our system such that we would make use of the expansion variables "as much as possible" (everywhere where an expansion might be needed). For example, if $M : \langle \Gamma \vdash_3 U_1 \sqcap U_2 \rangle$ is derivable from $M : \langle \Gamma \vdash_3 U_1 \rangle$ and $M : \langle \Gamma \vdash_3 U_2 \rangle$ using the intersection introduction rule and we apply the expansion introduction rule to each of the branches of the derivation then we obtain the two following typing judgements: $M^{+i} : \langle \mathsf{e}_i \Gamma \vdash_3 \mathsf{e}_i U \rangle$ and $M^{+j} : \langle \mathsf{e}_j \Gamma \vdash_3 \mathsf{e}_j U \rangle$. If we use two different expansion variables ($i \neq j$) then, given these two new typing judgements, we cannot use the intersection introduction rule because $\mathsf{e}_i U \sqcap \mathsf{e}_j U$ is not a ITy_3 type ($\mathsf{deg}(\mathsf{e}_i U) = i :: \mathsf{deg}(U) \neq j :: \mathsf{deg}(U) = \mathsf{deg}(\mathsf{e}_j U)$). This might be overcome by considering trees instead of lists as indices in our semantics. We let the investigation of such a system to future work.

In the present document we are not interested in a denotational semantics of the presented calculus. We are neither interested in an extensional λ-model interpreting the terms of the untyped λ-calculus. Instead, we are interested in building a realisability semantics by defining sets of realisers (programs satisfying the requirements of some specification) of types. We believe such a model would help highlighting the relation between terms of the untyped λ-calculus and types involving expansion variables w.r.t. a type system. Moreover, interpreting types in a model helps understanding the meaning of types (w.r.t. the model) which are defined as purely syntactic forms and are clearly used as meaningful expressions. For example, the integer type (whatever its notation is) is always used as the type of each integer. An arrow type expresses functionality. In that way, models based on λ-models have been built for intersection type systems [69, 8, 35]. In these models, intersection types were interpreted by set-theoretical intersections of meanings. Even though E-variables have been introduced to give a simple formalisation of the expansion mechanism, i.e., as syntactic objects, we are interested in the meaning of such syntactic objects. We are particularly interested in answering a number of questions

[1] Normalised types are types strongly related to normalisable (typable) terms.

such as:

1. Can we find a second order function, whose range is the set of λ-terms, and which interprets types involving any kind of expansions (any expansion term and not just expansion variables)?
2. How can we characterise the realisers of a type involving expansion terms?
3. How can the relation between terms and types involving expansion terms be described w.r.t. a type system?
4. How can we extend models such as the one given in Kamareddine and Nour [80] to a type system with expansion?

These questions have not yet been answered. We leave their investigation for future work.

Part III

A constraint system for a type error slicer

Chapter 10

Introduction

10.1 Background of type error slicing

As explained in Sec. 2.4.3, SML is a higher-order function-oriented imperative programming language and Milner's W algorithm [32] is the original type-checking algorithm of the functional core of ML. W implementations generally locate errors at or near the syntax tree node being visited when unification fails, and this is unsatisfactory.

10.1.1 Moving the error spot

Following W, other algorithms try to get better locations by arranging that untypability will be discovered when visiting a different syntax tree node. For example, Lee and Yi proved the folklore algorithm M [98] finds errors "earlier" (this measure is based on the number of recursive calls of the algorithm) than W and claimed that their combination "can generate strictly more informative type-error messages than either of the two algorithms alone can". Similar claims are made for W' [104] and UAE [147]. McAdam observes that W suffers a left-to-right bias and tries to eliminate it by replacing the unification algorithm used in the application case of W by another operation called "unification of substitutions". McAdam explains that the left-to-right bias in W arises because in the case of applications, "the substitution from a left-hand subexpression is applied to the type-environment before traversing the right-hand side expression" [104]. His "unification of solutions" allows one "to infer types and substitutions for each subexpression independently" [104]. The "unification of substitutions" operation is then used to comnbine the inferred substitutions. Yang claims that UAE's primary advantage is that it also eliminates this bias. However, all the algorithms mentioned above retain a left-to-right bias in handling of let-bindings and they all blame only one syntax tree node for each type error when in fact a node set is at fault.

When only one node is reported as the error site, it is often far away from the actual programming error. The situation is made worse because the blamed node depends on internal implementation details, i.e., the tree node traversal order and which constraints are accumulated and solved at different times in the traversal. The confusion is worsened because these algorithms usually exhibit in error messages (1) an internal representation of the program subtree at the blamed location which often has been transformed substantially from what the programmer wrote, and (2) inferred type details which were not written by the programmer and which are anyway erroneous and confusing.

10.1.2 Other improved error reporting systems

Constraint-based type inference algorithms [112, 115, 116] separate *constraint generation* and *constraint solving*. Many works use this idea to improve error reporting. A probably incomplete list includes [56, 57, 47, 64, 63, 58, 65, 60, 125, 126, 127]. Independently from this separation, there exist other approaches toward improving errors [149]: error explanation systems [9, 37, 36, 148] which focus on explaining the reasoning steps leading to a type error, and error reporting systems [139, 133] which focus on trying to precisely locate errors in pieces of code. There are also approaches that report type errors together with suggestions for changes that would solve the errors [59, 99]. Some of these approaches are discussed in Ch. 12.

10.2 Type error slicing

Haack and Wells [57] developed a type error reporting method called *type error slicing* (TES). Haack and Wells [57] noted that "*Identifying only one node or subtree of the program as the error location makes it difficult for programmers to understand type errors. To choose the correct place to fix a type error, the programmer must find all of the other program points that participate in the error.*" They locate type errors at *program slices* which include all parts of an untypable piece of code where changes can be made to fix the error and exclude the parts where changes cannot fix the error.

We shall refer to the method of Haack and Wells as HW-TES in this document (the slicer of Haack and Wells as presented in their papers [56, 57] and not its implementation). HW-TES generates a constraint set for a program, enumerates minimal unsatisfiable subsets of the constraint set, and computes type error slices. Generation and solving of constraints are not interleaved. To identify slices responsible for type errors, each constraint is labelled by the location responsible for its generation. Error slices are portions of a program where all blameless subterms are elided (e.g., replaced by dots). Slices can be shown by highlighting the source code.

HW-TES makes use of intersection types and its handling of polymorphism involves heavy constraint and type environment duplications which leads to a combinatorial constraint size explosion at constraint generation.

HW-TES meets the following seven criteria of Yang et al. [149] for good type error reports: it reports only errors for ill-typed code (*correct*), it reports no more than the conflicting portions of code (*precise*), it reports short messages (*succinct*), it does not report internal information such as internal types generated during type inference (*a-mechanical*), it reports only code written by the programmer which has not been transformed as happens with existing SML implementations (*source-based*), it does not privilege any location over the others (*unbiased*), and it reports all the conflicting portions of code (*comprehensive*).

10.3 Contributions

Unfortunately, HW-TES is not practical on real programs and works only for a tiny SML subset barely larger than the λ-calculus. Our goal is a TES method that (1) covers full SML, (2) is practical on real programs, and (3) has a simple and general design. As would happen for any programming language, we faced challenges.

An initial challenge was avoiding a combinatorial constraint size explosion. The naive approach in HW-TES duplicated constraints for code that gets a polymorphic type (e.g., in SML's let-expressions), and thus is unusable beyond small examples. Instead, at constraint solving we simplify constraints before copying them, and copy them as late as possible. We retain compositional initial generation of constraints, but unlike in HW-TES we solve constraints in a strict left-to-right order. Our solution is related in part to earlier constraint systems for ML-style let-bindings [115, 116, 108, 55, 112], which Pottier explains "allow building a constraint of linear size" [115]. Unfortunately, the earlier ideas are inadequate for module systems, so we needed a new constraint representation.

The next challenge was to scale constraint generation while also handling advanced module system features. Like many languages, SML can manipulate namespaces, e.g., with structures (modules), signatures (module types), functors (functions from modules to modules), etc. We achieve this with our novel hybrid *constraint/environments* (metavariable e in Fig. 11.2 in Sec. 11.2). They are constraints because they are satisfiable (or not) depending on variable values, and environments because they bind program names to information. Furthermore, some bindings are *polymorphic* to support some the kinds of polymorphism in SML: polymorphic functions, datatype constructors, named structure signatures, and functors.

The remaining challenges were using the novel constraint machinery for a full programming language, with all its features and warts. Ch. 11 presents full details for a core of language features large enough to show the essence of the mechanism, and Ch. 14 presents a larger feature set towards *Full-TES* which is the TES we are aiming at but which we have not yet achieved. This core includes polymorphic functions, datatypes and pattern matching, and structures (including the difficult `open` operation). We call this core system, *Core-TES*. The larger set of features/warts we present includes SML's value/constructor identifier-status ambiguity, local declarations, type functions/abbreviations, structure signatures, functors, type annotations, and the value polymorphism restriction. We generally refer to this formalised TES as *Form-TES* (the formalism we have achieved so far). Even though the implementation of our TES covers nearly full SML, it is not quite Full-TES. Some TES features have not yet been implemented and some SML features are not yet supported. We generally refer to the implementation of our TES as *Impl-TES*. Impl-TES is usable via a web demo and installable packages [132]. Note that neither Impl-TES is a superset of Form-TES and nor is Form-TES a superset of Impl-TES because Impl-TES supports some features that are not supported by Form-TES (e.g., many cases of records or the `fun` SML forms to write recursive functions) and vice versa (e.g., Form-TES has a better support for functors). We plan to have both Form-TES and Impl-TES converge with Full-TES in the future. We will often write *our TES* to encompass both Form-TES and Impl-TES.

The most challenging feature for full SML was the `open` declaration, which splices another structure into the current environment (example in Sec. 10.4.3), and has been criticized in the literature [2, 11, 12, 61]. Harper writes [61]: "*it is hard to control its behaviour, since it incorporates the entire body of a structure, and hence may inadvertently shadow identifiers that happen to be also used in the structure*". Blume [11] shows that certain automatic dependency analyses become NP-complete in the presence of `open`, and writes: "*Programs are not only read by analysis tools; human read them as well. A language construct like open that serves to confuse the analysis tool is also likely to confuse the human reader*". We believe `open` is one of the most difficult programming language features to analyze, but our constraint/environments make it easy and simple, and we believe this highlights the generality of our machinery. Our TES clarifies otherwise obscure type errors involving `open` and enhances its usability.

10.4 Key motivating examples

This section gives examples extracted from our testcase database motivating TES. Our testcase database is distributed with the packages and archives we provide [132]. Type error slices are highlighted with red. Purple and blue highlight error *end points* (sources of conflict). End points are discussed in Sec. 15.2.

```
fun g x y =
  let val f = if y
              then fn _ => fn z => z
              else fn z => z
      val u = (f, true)
  in (#1 u) y
  end
```

Figure 10.1 Conditionals, pattern matching, tuples (testcase 121)

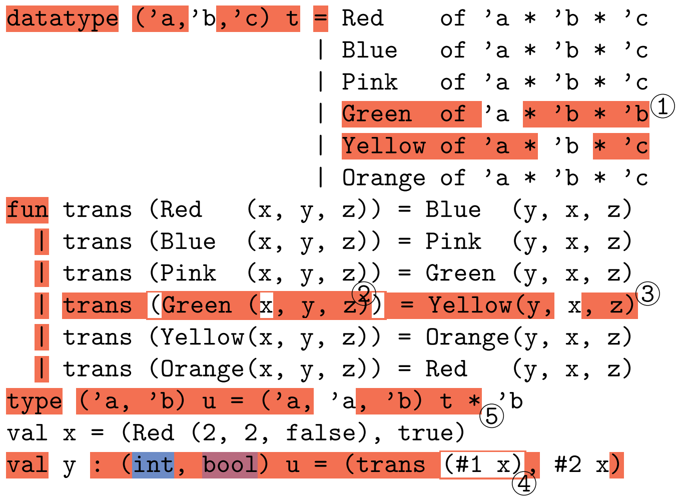
10.4.1 Conditionals, pattern matching, records

Fig. 10.1 shows an untypable piece of code involving, among other things, the following derived forms: a conditional, a record selector (`# u`). Derived forms are syntactic sugar for core of module forms. For example, `if exp1 then exp2 else exp3`, where `exp1`, `exp2`, and `exp3` are expressions, is not a core expression itself but is equivalent to the core expression `case exp1 of true => exp2 | false => exp3`. Suppose the programming error in the code presented in Fig. 10.1 is that we wrote `y` (the circled one in Fig. 10.1) instead of `x`. We call the programming error location, the real error location. The function `g` can be used to perform computations on integers. For example `(g true (fn x => x + 1) 2)` evaluates to `2` and `(g false (fn x => x + 1) 2)` evaluates to `3`. This piece of code is untypable because of the following reasons (highlighted in Fig. 10.1): `y`, being a parameter of a function, has a monomorphic type; `y` is constrained to be a Boolean via the conditional; and finally, `u`'s first component is applied to `y`, where `u`'s first component is the function `f` which is constrained by the two branches of the conditional to take a function as argument. SML's compiler SML/NJ (version 110.72) reports a type constructor clash in line 6 (more precisely, the circled portion of code `(#1 u) y` in Fig. 10.1 is blamed) as follows:

```
Error:  operator and operand don't agree [tycon mismatch]
  operator domain: 'Z -> 'Z
  operand:         bool
  in expression:
    ((fn {1=<pat>,...} => 1) u) y
```

In the above example, because of the small size of the piece of code, the programmer's error is not too far away from the location reported by SML/NJ. It is not always the case. The real error location might even be in another file. Nonetheless, note that SML/NJ reports only one location which is far from the real error location w.r.t. the size of the piece of code. Also, note that the type `'Z -> 'Z` reported by SML/NJ is an internal type made up during type inference. Finally, the reported expression does not match the source code[1].

[1] SML/NJ has transformed the code because the derived form `#1` is equivalent to the function `(fn {1=y,...} => y)` in SML. Note also that `(fn {1=<pat>,...} => 1)` is SML/NJ's pretty



```
datatype ('a,'b,'c) t = Red    of 'a * 'b * 'c
                      | Blue   of 'a * 'b * 'c
                      | Pink   of 'a * 'b * 'c
                      | Green  of 'a * 'b * 'b①
                      | Yellow of 'a * 'b * 'c
                      | Orange of 'a * 'b * 'c
fun trans (Red   (x, y, z)) = Blue  (y, x, z)
  | trans (Blue  (x, y, z)) = Pink  (y, x, z)
  | trans (Pink  (x, y, z)) = Green (y, x, z)
  | trans (Green (x, y, z)②) = Yellow(y, x, z)③
  | trans (Yellow(x, y, z)) = Orange(y, x, z)
  | trans (Orange(x, y, z)) = Red   (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t *⑤'b
val x = (Red (2, 2, false), true)
val y : (int, bool) u = (trans (#1 x)④, #2 x)
```

Figure 10.2 Datatypes, pattern matching, type functions (testcase 114)

Fig. 10.1 highlights a slice for the type error described above. This highlighting contains the minimal amount of information necessary to understand and fix the type error. Also, it highlights the real error location. Note that the fact that most of the piece of code is highlighted is due to the small size of the piece of code. We present below larger examples where a smaller percentage of the pieces of code is highlighted[2].

10.4.2 Datatypes, pattern matching, type functions

Fig. 10.2 shows how TES helps for intricate errors. The code declares the datatype `t` and the function `trans` to deal with user defined colours. This function is then applied to an instance of a colour (the first element in the pair `x`). Suppose the programming error is that we wrote `'b` instead of `'c` in `Green`'s definition at location ①. SML/NJ (version 110.72) reports a type constructor clash at ④ as follows:

```
operator domain: (int,int,int) t
operand:         (int,int,bool) t
in expression:
  trans ((fn {1=<pat>,...} => 1) x)
```

The reported code is far from the actual error and does not match the source code. SML/NJ gives the same error message if, instead of the error described above,

printing of `#1`, but the two functions are different because `(fn {1=<pat>,...} => 1)` returns always `1` while `#1` takes a record and returns the field of field name `1` in the record, which is confusing. SML's compilers MLton and Poly/ML do not transform the code.

[2] A slice for a type error will always contain exactly the portion of the program required to explain the error. We have no choice on how much or how little of a piece of code is included in a type error slice. The choice is made by the type error itself. In our experience in using TES, the size of slices does not vary much depending on the size of the program but it varies mainly depending on the kind of error.

```
structure S = struct
  structure Y = struct
    structure A = struct val x = false end
    structure X = struct val x = false end
    structure M = struct val x = true end
  end
  open Y
  val m = M.x
  val x = if m then true else false
end
structure T = struct
  structure X = struct val x = 1 end
  open S
  open X
  val y = if m then 1 else x
end
```

Figure 10.3 Chained *open*s and nested structures (testcase 450)

one writes `x` instead of `z` in the right-hand-side of any branch of `trans`. Thus, one might need to inspect the entire program to find the error.

Fig. 10.2 highlights a slice for this error. The programming error location being in the slice, we track it down by considering only the highlighted code, starting from the clashing types on the last line. The type annotation `(int, bool) u` constrains the result type of `trans`'s application. The part of the `trans` function in the slice is the case handling a `Green` object. At ①, `Green`'s second and third arguments are constrained to be of the same type. At ②, `y` is therefore constrained to be of the same type as `z`. At ③, because `y` and `z` are respectively `Yellow`'s first and third arguments and using `Yellow`'s definition, we infer that the type of `Yellow`'s application to its three arguments (returned by `trans`) is `t` where its first and third parameters have to be equal. At ④ and ⑤ we can see that `trans` is constrained to return a `t` where its first (`int`) and third (`bool`) parameters differ.

10.4.3 Chained *open*s and nested structures

Fig. 10.3 has an intricate type error with chained `open`s. Let us describe what the code was meant to do. Structure `T` declares structure `X` declaring integer `x`. Structure `S` is opened to access the Boolean `m`. Then, `X` is opened to access the integer `x`. Finally, if `m` is true then we return `1` otherwise we return `x`. This is untypable and SML/NJ blames `y`'s body as follows:

```
Error:  types of if branches do not agree [literal]
  then branch:  int
  else branch:  bool
  in expression:
    if m then 1 else x
```

The programming error, as our type error slice shows, is that opening `S` causes `S`'s declarations to shadow the current typing environment. Because `Y` is opened in `S`, the structures `A`, `X` and `M` are part of `S`'s declarations. Hence, when opening `S` in `T`, the structure `X` which was in our current typing environment is shadowed by the one defined in `Y`. If the programmer's intent is as described above (and only then), this error can be solved by replacing "`open S open X`" by "`open S X`", which opens `X` and `Y` simultaneously (opening `X` results then to the opening of the structure `X` declared in `T` because it is then not shadowed by the one declared in `Y`).

Our type error slice rules out `x`'s declarations in `X` and `S` and clearly shows why `x` does not have the expected type. The traditional report leaves us to track down `x`'s binding by hand.

Chapter 11

Technical design of Core-TES

This chapter introduces Core-TES and its different modules: initial constraint generator (Sec. 11.5), constraint solver (Sec. 11.6), minimiser (Sec. 11.7), enumerator (Sec. 11.7), and slicer (Sec. 11.8). The reader might (or might not) want to peek ahead at Sec. 11.7.3 which motivates the need of a minimiser. Sec. 11.1 defines the overall algorithm. Sec. 11.2 presents a fragment of SML syntax handled by Core-TES. Sec. 11.3 defines the constraint syntax of Core-TES and Sec. 11.4 their semantics. Sec. 11.10 discusses the principles of our approach. The reader might (or might not) want to peek ahead at Sec. 11.10 while reading the sections below.

11.1 TES' overall algorithm

Fig. 11.1 informally presents how the different modules of our TES interact with each other. We use different colours to differentiate different parts of our TES. The green parts are user interface related. The red parts are related to slicing. The purple parts are related to constraint generation. These parts are external language related. The blue parts are related to the enumeration of type errors. These parts are external language unrelated.

Formally, given a SML structure declaration *strdec* (see Fig. 11.8), the initial constraint generation algorithm defined in Fig. 11.7 and extended in Fig. 11.14 to dot terms (see Sec. 11.8.1), generates a constraint/environment e (see Fig. 11.3). Then, the enumerator defined in Fig. 11.12 enumerates the type errors of e. Each error found by the enumerator is minimised by the minimiser also defined in Fig. 11.12. From each minimised error and *strdec*, the slicing algorithm defined in Sec. 11.8 computes a type error slice. Both enumeration and minimisation rely on the constraint solver defined in Fig. 11.10. The computed type error slices are finally reported to the user. A type error report includes a type error slice, a highlighting of the slice directly in the SML user code, and a message explaining the kind of the error (see Fig. 11.8). Formally, our overall algorithm tes is defined as follows (the undefined

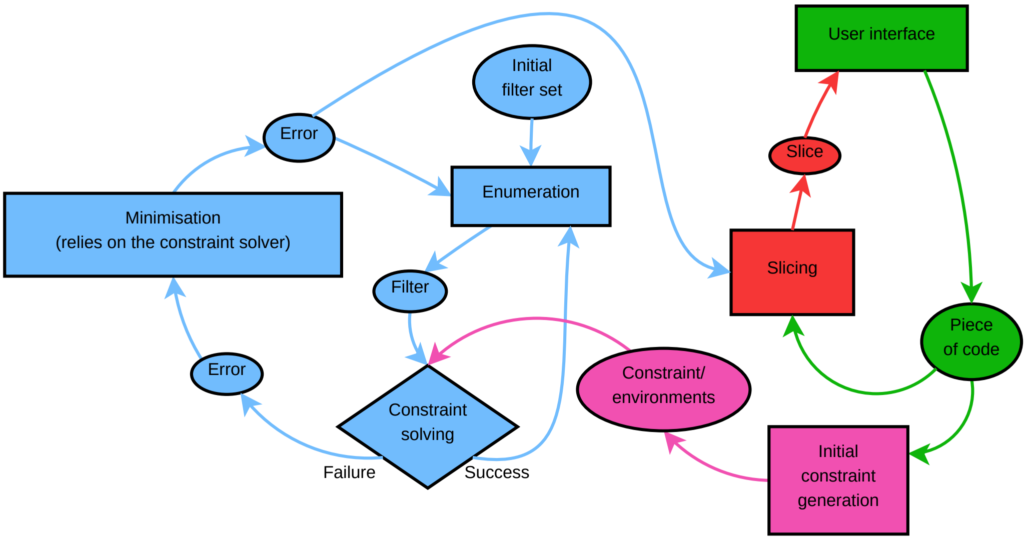


Figure 11.1 Interaction between the different modules of our TES

relations, functions, and other syntactic forms used in this definition of TES' overall algorithm are all defined in the remaining sections of the current chapter):

$$\begin{aligned}\mathsf{tes}(\mathit{strdec}) = \{\langle \mathit{strdec}', \mathit{ek}, \overline{\mathit{vid}}\rangle \mid\ & \mathit{strdec} \mathrel{\rightarrowtriangle} e \\ & \wedge \mathtt{enum}(e) \rightarrow^{*}_{\mathsf{e}} \mathtt{errors}(\overline{\mathit{er}}) \\ & \wedge \langle \mathit{ek}, \overline{l} \cup \overline{\mathit{vid}}\rangle \in \overline{\mathit{er}} \\ & \wedge \mathsf{sl}(\mathit{strdec}, \overline{l}) = \mathit{strdec}'\}\end{aligned}$$

Note that Core-TES does not have value identifier dependencies. These dependencies are introduced in Sec. 14.1. We anticipate this addition in the definition of our overall algorithm above (see the computation of the $\overline{\mathit{vid}}$ sets).

11.2 External syntax

Fig. 11.2 defines a fragment of SML syntax used to present the core ideas. Most syntactic forms have labels (l), which are generated to track blame for errors. To provide a visually convenient place for labels, some terms such as function applications are surrounded by $\lceil\ \rceil$ which are not written by programmers but are part of an internal representation used to avoid confusion with ( ) as part of SML syntax. Value identifiers (vid) are subscripted to disambiguate rules for expression ($\mathit{vid}^{l}_{\mathsf{e}}$), datatype constructor definitions ($\mathit{dcon}^{l}_{\mathsf{c}}$), and pattern ($\mathit{vid}^{l}_{\mathsf{p}}$) occurrences. Note that the only non-subscripted value identifiers are those occurring at unary positions in patterns and datatype declarations.

Although SML distinguishes value variables and datatype constructors by assigning statuses in the type system, we distinguish them by defining two disjoint sets ValVar and DatCon. For fully correct minimal error slices, we discuss the needed handling of identifier statuses in Sec. 14.1.

external syntax (what the programmer sees, plus labels)

$$
\begin{array}{rll}
l \in \mathsf{Label} & & \text{(labels)} \\
tv \in \mathsf{TyVar} & & \text{(type variables)} \\
tc \in \mathsf{TyCon} & & \text{(type constructors)} \\
strid \in \mathsf{StrId} & & \text{(structure identifiers)} \\
vvar \in \mathsf{ValVar} & & \text{(value variables)} \\
dcon \in \mathsf{DatCon} & & \text{(datatype constructors)} \\
vid \in \mathsf{VId} & ::= & vvar \mid dcon \\
ltc \in \mathsf{LabTyCon} & ::= & tc^{l} \\
ldcon \in \mathsf{LabDatCon} & ::= & dcon^{l} \\
ty \in \mathsf{Ty} & ::= & tv^{l} \mid ty_1 \xrightarrow{l} ty_2 \mid \lceil ty\ ltc \rceil^{l} \\
cb \in \mathsf{ConBind} & ::= & dcon^{l}_{\mathtt{c}} \mid dcon\ \mathtt{of}^{\,l}\ ty \\
dn \in \mathsf{DatName} & ::= & \lceil tv\ tc \rceil^{l} \\
dec \in \mathsf{Dec} & ::= & \mathtt{val\ rec}\ pat \overset{l}{=} exp \mid \mathtt{open}^{l}\ strid \mid \mathtt{datatype}\ dn \overset{l}{=} cb \\
atexp \in \mathsf{AtExp} & ::= & vid^{l}_{\mathtt{e}} \mid \mathtt{let}^{l}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \\
exp \in \mathsf{Exp} & ::= & atexp \mid \mathtt{fn}\ pat \overset{l}{\Rightarrow} exp \mid \lceil exp\ atexp \rceil^{l} \\
atpat \in \mathsf{AtPat} & ::= & vid^{l}_{\mathtt{p}} \\
pat \in \mathsf{Pat} & ::= & atpat \mid \lceil ldcon\ atpat \rceil^{l} \\
strdec \in \mathsf{StrDec} & ::= & dec \mid \mathtt{structure}\ strid \overset{l}{=} strexp \\
strexp \in \mathsf{StrExp} & ::= & strid^{l} \mid \mathtt{struct}^{l}\ strdec_1 \cdots strdec_n\ \mathtt{end}
\end{array}
$$

extra metavariables

$id \in \mathsf{Id} ::= vid \mid strid \mid tv \mid tc$ $\quad$ $term \in \mathsf{Term} ::= ltc \mid ldcon \mid ty \mid cb \mid dn \mid exp \mid pat \mid strdec \mid strexp$

Figure 11.2 External labelled syntax

To simplify the presentation of Core-TES, all datatypes have one constructor and one type argument.

Note that we do not enforce all the syntactic restrictions of the SML syntax [107]. For example, in SML, in a recursive declaration such as `val rec` $pat \overset{l}{=} exp$, the expression exp must be a `fn`-expression.

In this chapter we are going to consider the following simple running example:

(EX1)

```
structure X = struct
  structure S = struct datatype 'a u = U end
  datatype 'a t = T
  val rec f = fn T => T
  val rec g = let open S in f U end
  end
end
```

This piece of code is untypable because f is defined as taking a `'a t` and is applied to a `'a u`. The labelled version of this piece of code is as follows:

$$
\begin{array}{l}
\mathtt{structure\ X} \overset{l_1}{=} \mathtt{struct}^{l_2} \\
\qquad \mathtt{structure\ S} \overset{l_3}{=} \mathtt{struct}^{l_4}\ \mathtt{datatype}\ \lceil \mathtt{'a\ u} \rceil^{l_6} \overset{l_5}{=} \mathtt{U}^{l_7}_{\mathtt{c}}\ \mathtt{end} \\
\qquad \mathtt{datatype}\ \lceil \mathtt{'a\ t} \rceil^{l_9} \overset{l_8}{=} \mathtt{T}^{l_{10}}_{\mathtt{c}} \\
\qquad \mathtt{val\ rec\ f}^{l_{12}}_{\mathtt{p}} \overset{l_{11}}{=} \mathtt{fn\ T}^{l_{14}}_{\mathtt{p}} \overset{l_{13}}{\Rightarrow} \mathtt{T}^{l_{15}}_{\mathtt{e}} \\
\qquad \mathtt{val\ rec\ g}^{l_{17}}_{\mathtt{p}} \overset{l_{16}}{=} \mathtt{let}^{l_{18}}\ \mathtt{open}^{l_{19}}\ \mathtt{S\ in}\ \lceil \mathtt{f}^{l_{21}}_{\mathtt{e}}\ \mathtt{U}^{l_{22}}_{\mathtt{e}} \rceil^{l_{20}}\ \mathtt{end} \\
\quad \mathtt{end}
\end{array}
$$

We call this structure declaration $strdec_{\text{EX}}$.

constraint terms (syntax of entities used internally by TES and which the programmer never sees)
$ev \in \mathsf{EnvVar}$ (environment variables)
$\delta \in \mathsf{TyConVar}$ (type constructor variables)
$\gamma \in \mathsf{TyConName}$ (type constructor names)
$\alpha \in \mathsf{ITyVar}$ (internal type variables)
$d \in \mathsf{Dependency} ::= l$
$\mu \in \mathsf{ITyCon} ::= \delta \mid \gamma \mid \mathtt{ar} \mid \langle \mu, \overline{d} \rangle$
$\tau \in \mathsf{ITy} ::= \alpha \mid \tau\,\mu \mid \tau_1 {\to} \tau_2 \mid \langle \tau, \overline{d} \rangle$
$\sigma \in \mathsf{Scheme} ::= \tau \mid \forall \overline{\alpha}.\,\tau \mid \langle \sigma, \overline{d} \rangle$
$bind \in \mathsf{Bind} ::= \downarrow tc{=}\mu \mid \downarrow strid{=}e \mid \downarrow tv{=}\alpha \mid \downarrow vid{=}\sigma$
$acc \in \mathsf{Accessor} ::= \uparrow tc{=}\delta \mid \uparrow strid{=}ev \mid \uparrow tv{=}\alpha \mid \uparrow vid{=}\alpha$
$c \in \mathsf{EqCs} ::= \mu_1{=}\mu_2 \mid e_1{=}e_2 \mid \tau_1{=}\tau_2$
$e \in \mathsf{Env} ::= \top \mid ev \mid bind \mid acc \mid c \mid \mathtt{poly}(e) \mid e_2;e_1 \mid \langle e, \overline{d} \rangle$

extra metavariables
$v \in \mathsf{Var} ::= \alpha \mid \delta \mid ev$
$dep \in \mathsf{Dependent} ::= \langle \tau, \overline{d} \rangle \mid \langle \mu, \overline{d} \rangle \mid \langle e, \overline{d} \rangle$

Figure 11.3 Syntax of constraint terms

11.3 Constraint syntax

11.3.1 Terms

Fig. 11.3 defines *constraint terms*, those pieces of syntax that can occur anywhere inside a constraint. In our system, this is any μ, τ, σ, or e.

Some forms, called *dependent forms*, are annotated by dependencies: $\langle x, \overline{d} \rangle$. In Core-TES, a dependency d must be a label l (but in Impl-TES, d can also be a value identifier *vid* for handling identifier statuses in Sec. 14.1). During analysis, a form $\langle x, \overline{d} \rangle$ depends on the program nodes with labels in $\overline{d}$. For example, the dependent equality constraint $\langle \tau_1{=}\tau_2, \overline{d} \cup \{l\} \rangle$ might be generated for the labelled function application $\lceil exp\ atexp \rceil^l$, indicating the equality constraint $\tau_1{=}\tau_2$ need only be true if node l has not been sliced out. Let strip be the function that strips off the outer dependencies of any syntactic form: $\mathsf{strip}(x) = \mathsf{strip}(y)$ if $x = \langle y, \overline{d} \rangle$, x otherwise. Let collapse be the function that combines nested outermost dependencies: $\mathsf{collapse}(x) = \mathsf{collapse}(\langle y, \overline{d}_1 \cup \overline{d}_2 \rangle)$ if $x = \langle (\langle y, \overline{d}_1 \rangle), \overline{d}_2 \rangle$, x otherwise.

An internal type $\tau\,\mu$ is a *type construction* and is built from an internal type constructor μ and its argument τ (such as the polymorphic list type `'a list`, where `'a` is an explicit type variable in SML). To simplify the formalisation of Core-TES, external (tc) and internal (μ) type constructors both take exactly one argument. We present how to handle non-unary type constructors in Sec. 14.10. The special internal type constructor `ar` represents the binary arrow type constructor (→) during constraint solving solely to allow constraints between → and any unary type constructor. This allows one to compute the necessary portions of code when generating type errors. A type scheme can either be a universal quantification, or an internal type, or a dependent type scheme. Our type schemes are subject to alpha-conversion. For example, $\forall\{\alpha\}.\,\alpha$ is convertible to $\forall\{\alpha'\}.\,\alpha'$. These two terms are considered equal.

A *constraint/environment* e is a hybrid that acts as both a *constraint* and an *environment*, and we will freely switch between these terms when discussing them. A major novelty is three of the constraint/environment forms, and their interaction: *binders* ($\downarrow id{=}x$, with metavariable *bind*), *composition environments* ($e_1;e_2$), and *accessors* ($\uparrow id{=}x$, with metavariable *acc*). A binder $\downarrow id{=}x$ or an accessor $\uparrow id{=}x$ is used for program occurrences of *id* that are respectively binding or bound. The composition $e_1;e_2$ is used when the accessors of e_2 are in the scope of the binders of e_1, and acts like a logical conjunction requiring e_1 to be satisfied, and e_2 to be satisfied when the bindings of e_1 are in scope. For example, in $\downarrow vid{=}\sigma;\uparrow vid{=}\alpha$, the type variable α is constrained to be an instance of σ through the binding of *vid*. Note that the binders and accessors do not need to be next to each other. For example, in $\downarrow vid{=}\forall\overline{\alpha}.\,\tau;\cdots;\uparrow vid{=}\alpha_1;\cdots;\uparrow vid{=}\alpha_2$, if the ellipses do not shadow *vid*'s binder (e.g., if they are equality constraints) then this constraint/environment has same solvability as $\downarrow vid{=}\forall\overline{\alpha}.\,\tau;\cdots;\tau[ren_1]{=}\alpha_1;\cdots;\tau[ren_2]{=}\alpha_2$ where the two accessors have been resolved by accessing the corresponding binder, and where the two renamings ren_1 and ren_2 rename the type variables in $\overline{\alpha}$ to fresh variables in order to instantiate the type scheme $\forall\overline{\alpha}.\,\tau$. We have $\mathsf{dom}(ren_1) = \mathsf{dom}(ren_2) = \overline{\alpha}$ and, among other properties it holds that $\mathsf{dj}(\mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$. The shadowing mechanism is further discussed in Sec. 11.6. The motivation for these constraint/environments is to have a general mechanism to build environments for sequential declarations that avoids duplications at initial constraint generation or during constraint solving.

The operator ; is used to compose environments. We consider ; to be associative (i.e., $(e_1;(e_2;e_3))$ is considered to be equivalent to $((e_1;e_2);e_3)$) with unit $\top$ (i.e., $(\top;e)$, $(e;\top)$ and e are all equivalent).

A constraint/environment can also be (1) the empty environment and satisfied constraint $\top$, (2) a constraint/environment variable ev, (3) an equality constraint c, (4) a special form $\mathtt{poly}(e)$ which promotes bindings in e to be polymorphic (see below), or (5) a conditional environment $\langle e, \overline{d}\rangle$ which acts like e if the dependencies in $\overline{d}$ are satisfied and otherwise acts (mostly) like $\top$. The semantics of our constraint/environments is provided in Sec. 11.4.

Binders and accessors are related to ideas in earlier systems, e.g., Pottier and Rémy's let-constraints and type scheme instantiations [116]. The earlier systems are too restrictive to easily represent module systems because they only support very limited cases of what our binders do and they lack environment variables. We know of no other system with these features. With our constraints we can easily define a compositional constraint generation algorithm. A comparison with related constraint systems is provided in Sec. 12.1.

Note that in Fig. 11.3, $\mathsf{Sub} = \mathsf{Unifier}$. These two sets will be extended in Ch. 14 such that they will be different. The set $\mathsf{Unifier}$ is generally the set of unifiers generated by our constraint solver defined in Sec. 11.6. We also use the distinct set Sub because we sometimes need to substitute more syntactic forms than allowed by unifiers. For example, in Sec. 14.7 we need to to substitute rigid type variables (introduced in Sec. 14.7 as well) when instantiating type schemes (type schemes are also extended in Sec. 14.7). Rigid type variables are not allowed to be in the domain of a unifier during constraint solving (because, as explained in Sec. 14.7, they act as constant types).

11.3.2 "Atomic" syntactic forms

Let $\mathsf{atoms}(x)$ be the syntactic form set belonging to $\mathsf{Var} \cup \mathsf{TyConName} \cup \mathsf{Dependency}$ and occurring in x whatever x is. We define the following functions:

$$\begin{array}{lll} \mathsf{vars}(x) & = \mathsf{atoms}(x) \cap \mathsf{Var} & \text{(set of variables)} \\ \mathsf{labs}(x) & = \mathsf{atoms}(x) \cap \mathsf{Label} & \text{(set of labels)} \\ \mathsf{deps}(x) & = \mathsf{atoms}(x) \cap \mathsf{Dependency} & \text{(set of dependencies)} \end{array}$$

11.3.3 Freshness

We use distinguished dummy variables: $\mathsf{Dum} = \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}\}$. Each use of a dummy variable acts like a fresh variable. These variables are used to generate dummy environments and constraints. For example, in $(\alpha_{\mathtt{dum}}{=}\alpha_1);(\alpha_{\mathtt{dum}}{=}\alpha_2)$, the two occurrences of $\alpha_{\mathtt{dum}}$ can be thought of as type variables different from each other and also different from α_1 and α_2. Note that variable freshness is not handled via existential constraints as in other systems [55, 108, 116]. Instead the relation dja ensures the freshness of the generated variables and type constructor names: $\mathsf{dja}(x_1, \ldots, x_n) \Leftrightarrow \mathsf{dj}(f(x_1), \ldots, f(x_n), \mathsf{Dum})$, where $f(x) = \mathsf{atoms}(x) \setminus \mathsf{VId}$. This also ensures that each label occurs at most once in a labelled program. Let us define $\mathsf{nonDums}$ as follows: $\mathsf{nonDums}(x) = \mathsf{vars}(x) \setminus \mathsf{Dum}$.

11.3.4 Syntactic sugar

We write $\langle x, d\rangle$ for $\langle x, \{d\}\rangle$. If y is a d or a $\overline{d}$, then x^y abbreviates $\langle x, y\rangle$, and $x_1 \overset{y}{=} x_2$ abbreviates $\langle x_1{=}x_2, y\rangle$, and similarly for *bind*s and *acc*s. Let $[e]$ abbreviate $(ev_{\mathtt{dum}}{=}e)$, an equality constraint that enforces the logical constraint nature of e while limiting the scope of its bindings (they can still have an effect if e constrains some environment variable ev). This is used for local bindings by rules (G2) and (G4) of our constraint generation algorithm defined in Fig. 11.7.

$$
\begin{array}{rcl}
\mathit{ren} \in \mathsf{Ren} & = & \{\mathit{ren} \in \mathsf{ITyVar} \to \mathsf{ITyVar} \mid \mathit{ren} \text{ is injective} \land \mathsf{dj}(\mathsf{dom}(\mathit{ren}), \mathsf{ran}(\mathit{ren}), \mathsf{Dum})\} \\
u \in \mathsf{Unifier} & = & \{f_1 \cup f_2 \cup f_3 \mid f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \land f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \land f_3 \in \mathsf{EnvVar} \to \mathsf{Env}\} \\
\mathit{sub} \in \mathsf{Sub} & = & \mathsf{Unifier} \\
\mathit{\Delta} \in \mathsf{Context} & ::= & \langle u,\ e\rangle
\end{array}
$$

Figure 11.4 Renamings, unifiers, and substitutions

11.4 Semantics of constraint/environments

11.4.1 Renamings, unifiers, and substitutions

Fig. 11.4 defines renamings, unifiers and substitutions. One can observe that $\mathsf{Ren} \subset \mathsf{Unifier} = \mathsf{Sub}$. Renamings are used to instantiate type schemes. Substitutions will be extended in Ch. 14 (see Sec.14.7 and Sec.14.9) such that $\mathsf{Unifier} \subset \mathsf{Sub}$. It will always be the case that $\mathsf{Unifier} \subseteq \mathsf{Sub}$.

The application of a substitution *sub* (and therefore of a renaming *ren* and a unifier u) to a constraint term is defined as follows:

$$
\begin{array}{ll}
v[\mathit{sub}] = \begin{cases} x, \text{if } \mathit{sub}(v) = x \\ v, \text{otherwise} \end{cases} & (\uparrow \mathit{id}{=}v)[\mathit{sub}] = \begin{cases} (\uparrow \mathit{id}{=}v[\mathit{sub}]), \\ \quad \text{if } v[\mathit{sub}] \in \mathsf{Var} \\ \text{undefined, otherwise} \end{cases} \\
(\tau\ \mu)[\mathit{sub}] = \tau[\mathit{sub}]\ \mu[\mathit{sub}] & \\
(\tau_1{\to}\tau_2)[\mathit{sub}] = \tau_1[\mathit{sub}]{\to}\tau_2[\mathit{sub}] & (\downarrow \mathit{id}{=}x)[\mathit{sub}] = (\downarrow \mathit{id}{=}x[\mathit{sub}]) \\
x^{\overline{d}}[\mathit{sub}] = x[\mathit{sub}]^{\overline{d}} & (x_1{=}x_2)[\mathit{sub}] = (x_1[\mathit{sub}]{=}x_2[\mathit{sub}]) \\
(\forall \overline{v}.\, x)[\mathit{sub}] = \begin{cases} \forall \overline{v}.\, x[\overline{v} \lhd \mathit{sub}], \\ \quad \text{if } \mathsf{dj}(\overline{v}, \mathsf{vars}(\overline{v} \lhd \mathit{sub})) \\ \text{undefined, otherwise} \end{cases} & \begin{array}{l} (e_1;e_2)[\mathit{sub}] = e_1[\mathit{sub}];e_2[\mathit{sub}] \\ \mathtt{poly}(e)[\mathit{sub}] = \mathtt{poly}(e[\mathit{sub}]) \\ x[\mathit{sub}] = x, \text{ otherwise} \end{array}
\end{array}
$$

Fig. 11.4 also defines constraint solving contexts. A *constraint solving context* $\mathit{\Delta} = \langle u,\ e\rangle$ is used as the context in which the meaning of constraint/environments is checked in the semantic rules provided below in Sec. 11.4.3. Such forms are also used in our constraint solver defined in Sec. 11.6 as contexts in which the solvability of constraint/environments is checked. In our system unifiers and environments are complementary: unifiers contain information on internal type variables and environments on external identifiers. This is further stressed in Sec. 11.4.2, in the definition of the application of a constraint solving context to an identifier.

Let $\langle u,\ e\rangle(v)$ be $u(v)$, let $\langle u,\ e\rangle;e'$ be $\langle u,\ e;e'\rangle$.

11.4.2 Shadowing and constraint solving context application

In a constraint solving context (of the form $\langle u,\ e\rangle$) some parts might be shadowed and so inaccessible. For example, in the constraint solving context $\langle u,\ \mathit{bind}_2;\mathit{ev};\mathit{bind}_1\rangle$ where $u = \varnothing$, the binder bind_1 is "visible" and *ev* shadows bind_2 because *ev* is not bound in u ($\mathit{ev} \notin \mathsf{dom}(u)$) and an environment variable stands for any environment

$$\frac{}{u, e \rhd \top \hookrightarrow \top}\ (\top) \qquad \frac{}{u, e \rhd ev \hookrightarrow ev[u]}\ (\mathsf{evar})$$

$$\frac{x_1[u] = x_2[u] \quad x_1, x_2 \notin \mathsf{Env}}{u, e \rhd (x_1{=}x_2) \hookrightarrow \top}\ (\mathsf{eqc}) \qquad \frac{\forall i \in \{1, 2\}.\ u, e \rhd e_i \hookrightarrow e'_i \quad e'_1 = e'_2}{u, e \rhd (e_1{=}e_2) \hookrightarrow \top}\ (\mathsf{eqe})$$

$$\frac{e(id) \xrightarrow{\mathsf{instance}} x \quad u, e \rhd (x{=}v) \hookrightarrow \top}{u, e \rhd (\uparrow id{=}v) \hookrightarrow \top}\ (\mathsf{acc}) \qquad \frac{e(id)\ \text{undefined}}{u, e \rhd (\uparrow id{=}v) \hookrightarrow \top}\ (\mathsf{acc}')$$

$$\frac{}{u, e \rhd (\downarrow id{=}x) \hookrightarrow (\downarrow id{=}x[u])}\ (\mathsf{bind}) \qquad \frac{u, e \rhd e' \hookrightarrow e''}{u, e \rhd \mathtt{poly}(e') \hookrightarrow \mathsf{toPoly}(\langle \varnothing, e\rangle, e'')}\ (\mathsf{poly})$$

$$\frac{u, e \rhd e_1 \hookrightarrow e'_1 \quad u, (e;e'_1) \rhd e_2 \hookrightarrow e'_2}{u, e \rhd (e_1;e_2) \hookrightarrow (e'_1;e'_2)}\ (\mathsf{comp})$$

Figure 11.5 Semantics of the constraint/environments, ignoring dependencies

and could potentially bind any identifier. Let the predicate shadowsAll be defined as follows:

$$\begin{array}{ll}
\mathsf{shadowsAll}(\langle u,\ e\rangle) \Leftrightarrow & \begin{cases} \ \ (e = ev & \wedge\, (\mathsf{shadowsAll}(\langle u,\ u(ev)\rangle) \vee ev \notin \mathsf{dom}(u))) \\ \vee\, (e = (e_1;e_2) & \wedge\, (\mathsf{shadowsAll}(\langle u,\ e_1\rangle) \vee \mathsf{shadowsAll}(\langle u,\ e_2\rangle))) \\ \vee\, (e = e'^{\overline{d}} & \wedge\, \mathsf{shadowsAll}(\langle u,\ e'\rangle)) \end{cases} \\
\mathsf{shadowsAll}(e) & \Leftrightarrow \mathsf{shadowsAll}(\langle \varnothing,\ e\rangle)
\end{array}$$

If shadowsAll(e) then it means that some of the binders in e might be shadowed, and especially it means that in $(e';e)$, the environment e shadows the entire environment e' (no binder from e' is accessible in $(e;e)$).

Let us now present how to access the semantics of an identifier in an environment. The applications $\Delta(id)$ and $e(id)$ to access identifiers' static semantics are defined as follows:

$$\begin{array}{ll}
\langle u,\ \downarrow id{=}x\rangle(id) & = x \\
\langle u,\ e^{\overline{d}}\rangle(id) & = \mathsf{collapse}(\langle u,\ e\rangle(id)^{\overline{d}}) \\
\langle u,\ (e_1;e_2)\rangle(id) & = \begin{cases} x, \text{if } \langle u,\ e_2\rangle(id) = x \text{ or } \mathsf{shadowsAll}(\langle u,\ e_2\rangle) \\ \langle u,\ e_1\rangle(id), \text{otherwise} \end{cases} \\
\langle u,\ ev\rangle(id) & = \begin{cases} \langle u,\ e\rangle(id), \text{if } u(ev) = e \\ \text{undefined, otherwise} \end{cases} \\
e(id) & = \langle \varnothing,\ e\rangle(id)
\end{array}$$

For example, $(\downarrow vid{=}\forall\overline{\alpha}.\,\tau;\downarrow strid{=}e)(vid) = \forall\overline{\alpha}.\,\tau$ but $(e';ev;\downarrow strid{=}e)(vid)$ and $(\downarrow vid{=}\sigma;\downarrow strid{=}e)(tc)$ are undefined.

Let us now present another example involving a unifier:

$$\langle\{ev \mapsto (\downarrow vid{=}\forall\overline{\alpha}.\,\tau)\},\ (e';ev;\downarrow strid{=}e)\rangle(vid) = \forall\overline{\alpha}.\,\tau$$

11.4.3 Semantic rules

We will now present the semantics of our constraint/environments.

First, let us define the relation $\mathsf{instance}$, which allows one to generate instances of type schemes. This predicate is defined as follows:

$$\begin{array}{lll} x \xrightarrow{\mathsf{instance}} y^{\overline{d}}[sub] & \text{if} & \mathsf{collapse}(x^{\varnothing}) = (\forall \overline{v}_0.\, y)^{\overline{d}} \text{ and } \mathsf{dom}(sub) = \overline{v}_0 \\ x \xrightarrow{\mathsf{instance}} x & \text{if} & \mathsf{collapse}(x^{\varnothing}) \text{ is not of the form } (\forall \overline{v}_0.\, y)^{\overline{d}} \end{array}$$

Let us define semantic judgements as follows:

$$\Phi \in \mathsf{SemanticsJudgement} ::= u, e \rhd e_1 \hookrightarrow e_2$$

Fig. 11.5 defines the semantics of our constraint/environments, ignoring dependencies at first. Note that this figure uses the function toPoly which is formally defined below in Fig. 11.9 in Sec. 11.6.4. The function toPoly allows one to transform a monomorphic environment into a polymorphic one. The function toPoly used in Core-TES (i.e., defined in Fig. 11.9) can only be applied to a single dependent value identifier binder. Note that this function is extended in Fig. 14.2 in Sec. 14.1.4 to deal with environments composed of more than one binder.

We say that an environment e is satisfiable iff there exist u and e' such that $u, \top \rhd e \hookrightarrow e'$. The environment e' is the semantics of e in the context $\langle u,\ \top\rangle$.

Let us now consider the following environment which we call e_1

$$\mathtt{poly}(\downarrow vid{=}\alpha_0);(\uparrow vid{=}\alpha_2);(\alpha_2{=}\alpha\,\gamma);(\alpha_1{=}\alpha_3{\to}\alpha_4)$$

Let $u = \{\alpha_2 \mapsto \alpha\,\gamma, \alpha_1 \mapsto \alpha_3{\to}\alpha_4\}$ and $e' = (\downarrow vid{=}\forall\{\alpha_0\}.\,\alpha_0)$. Let $\Phi = u, \top \rhd e_1 \hookrightarrow e'$. Then, one can derive Φ. Let us show how to derive this judgement.

Let $\Phi_1 = (u, \top \rhd \mathtt{poly}(\downarrow vid{=}\alpha_0) \hookrightarrow e')$. This judgement can be derived as follows:

$$\frac{\dfrac{}{u, \top \rhd (\downarrow vid{=}\alpha_0) \hookrightarrow (\downarrow vid{=}\alpha_0)} \qquad \mathsf{toPoly}(\langle\varnothing,\ \top\rangle, \downarrow vid{=}\alpha_0) = e'}{\Phi_1}$$

Let $\Phi_2 = (u, \top \rhd \mathtt{poly}(\downarrow vid{=}\alpha_0);(\uparrow vid{=}\alpha_2) \hookrightarrow e')$. This judgement can be derived as follows:

$$\frac{\Phi_1 \qquad \dfrac{e'(vid) \xrightarrow{\mathsf{instance}} \alpha\,\gamma \qquad \dfrac{\alpha_2[u] = (\alpha\,\gamma)[u] = \alpha\,\gamma}{u, e' \rhd (\alpha\,\gamma{=}\alpha_2) \hookrightarrow \top}}{u, e' \rhd (\uparrow vid{=}\alpha_2) \hookrightarrow \top}}{\Phi_2}$$

Finally, the judgement Φ can be derived as follows:

$$\frac{\dfrac{\Phi_2 \qquad \dfrac{\alpha_2[u] = (\alpha\,\gamma)[u] = \alpha\,\gamma}{u, e' \rhd (\alpha_2{=}\alpha\,\gamma) \hookrightarrow \top}}{u, \top \rhd \mathtt{poly}(\downarrow vid{=}\alpha_0);(\uparrow vid{=}\alpha_2);(\alpha_2{=}\alpha\,\gamma) \hookrightarrow e'} \qquad \dfrac{\alpha_1[u] = (\alpha_3{\to}\alpha_4)[u] = \alpha_3{\to}\alpha_4}{u, e' \rhd (\alpha_1{=}\alpha_3{\to}\alpha_4) \hookrightarrow \top}}{\Phi}$$

Let us mention an issue concerning the semantics of our constraint/environments and our constraint solver defined below in Sec. 11.6. Let us consider the following environment, similar to e_1, which we call e_2:

$$\frac{}{u, e, de \rhd \top \hookrightarrow \top}\;(\top) \qquad \frac{}{u, e, de \rhd ev \hookrightarrow ev[u]}\;(\mathsf{evar})$$

$$\frac{x_1[u] = x_2[u] \quad x_1, x_2 \notin \mathsf{Env}}{u, e, de \rhd (x_1{=}x_2) \hookrightarrow \top}\;(\mathsf{eqc}) \qquad \frac{\forall i \in \{1,2\}.\ u, e, de \rhd e_i \hookrightarrow e'_i \quad e'_1 = e'_2}{u, e, de \rhd (e_1{=}e_2) \hookrightarrow \top}\;(\mathsf{eqe})$$

$$\frac{e(id) \xrightarrow{\mathsf{instance}} x \quad u, e, de \rhd (x{=}v) \hookrightarrow \top}{u, e, de \rhd (\uparrow id{=}v) \hookrightarrow \top}\;(\mathsf{acc}) \qquad \frac{e(id) \text{ undefined}}{u, e, de \rhd (\uparrow id{=}v) \hookrightarrow \top}\;(\mathsf{acc'})$$

$$\frac{}{u, e, de \rhd (\downarrow id{=}x) \hookrightarrow (\downarrow id{=}x)}\;(\mathsf{bind}) \qquad \frac{u, e, de \rhd e' \hookrightarrow e''}{u, e, de \rhd \mathtt{poly}(e') \hookrightarrow \mathsf{toPoly}(\langle\varnothing, e\rangle, e'')}\;(\mathsf{poly})$$

$$\frac{u, e, de \rhd e_1 \hookrightarrow e'_1 \quad u, (e;e'_1), de \rhd e_2 \hookrightarrow e'_2}{u, e, de \rhd (e_1;e_2) \hookrightarrow (e'_1;e'_2)}\;(\mathsf{comp}) \qquad \frac{u, e, de \rhd e' \hookrightarrow e'' \quad de(\overline{d}) = \{\mathtt{keep}\}}{u, e, de \rhd \langle e', \overline{d}\rangle \hookrightarrow \langle e'', \overline{d}\rangle}\;(\mathsf{keep})$$

$$\frac{\mathtt{drop} \in de(\overline{d})}{u, e, de \rhd \langle e', \overline{d}\rangle \hookrightarrow \mathsf{dum}(e')}\;(\mathsf{drop}) \qquad \frac{\{\mathtt{keep\text{-}only\text{-}binders}\} = de(\overline{d}) \setminus \{\mathtt{keep}\}}{u, e, de \rhd \langle e', \overline{d}\rangle \hookrightarrow \top}\;(\mathsf{keep\text{-}only\text{-}binders})$$

Figure 11.6 Semantics of the constraint/environments, considering dependencies

$$\mathtt{poly}(\downarrow vid{=}\alpha_1);(\uparrow vid{=}\alpha_2);(\alpha_2{=}\alpha\,\mu);(\alpha_1{=}\alpha_3{\to}\alpha_4)$$

The environment e_2 only differs from e_1 by the replacement of α_0 by α_1. Note that there are now two occurrences of α_1 in e_2. Note that e_2 uses α_1 at two separate unrelated places. Because of these two occurrences of α_1, the environment e_2 fails to be satisfiable w.r.t. the rules defined in Fig. 11.5. However, e_2 is satisfiable w.r.t. our constraint solver defined below in Sec. 11.6. The issue is that our constraint solver considers the two occurrences of α_1 to be different when with the semantics defined in this section, these two occurrences are considered to be the same. Note that e_2 cannot be generated by our initial constraint generation algorithm defined below in Sec. 11.5, so this bug is not triggered. (Not initially generating environments such as e_2 is currently our only way of forbidding them.)

Let us define semantic judgements considering dependencies as follows:

$$\begin{array}{lll} ds \in \mathsf{DepStatus} & ::= & \mathtt{keep} \mid \mathtt{drop} \mid \mathtt{keep\text{-}only\text{-}binders} \\ de \in \mathsf{DepEnv} & = & \mathsf{Dependency} \to \mathsf{DepStatus} \\ \Psi \in \mathsf{SemanticsJudgementDep} & ::= & u, e, de \rhd e_1 \hookrightarrow e_2 \end{array}$$

We define des on dependency sets as follows:

$$de(\overline{d}) = \{de(d) \mid d \in \overline{d}\}$$

Fig. 11.6 adds dependencies to the rules from Fig. 11.5. Semantic judgements are now of the form $u, e, de \rhd e_1 \hookrightarrow e_2$. Except for these additions, rules ($\top$), (eqc), (eqe), (acc), ($\mathsf{acc'}$), (bind), (evar), (comp), and (poly) do not differ from the ones defined in Fig. 11.5. In addition to these rules, Fig. 11.6 defines three new rules: (keep), (drop), and ($\mathsf{keep\text{-}only\text{-}binders}$) to deal with dependencies. Note that this figure also uses the function toPoly and in addition uses the function dum which is formally defined below in Fig. 11.11 in Sec. 11.7.2. The function dum allows one to transform an environment e into a similar dummy environment e' which cannot participate in any error but contains dummy versions of the binders from e.

We say that an environment e is satisfiable w.r.t. the dependency environment de iff there exist u and e' such that $u, \top, de \triangleright e \hookrightarrow e'$. Given a dependency environment de, a dependency d is said to be satisfied if $de(d) = \texttt{keep}$, and it is said to be unsatisfied if $de(d) = \texttt{drop}$. The dependency status `keep-only-binders` is more complicated. This status is needed for scoping issues which are further discussed below in Sec. 11.7.2. If an environment e is annotated by a dependency which has status `keep-only-binders` then e's binders and environment variables (which could potentially bind any identifier) are turned into dummy binders and dummy environment variables respectively. Other environments, such as equality constraints, are discarded. The environment e' is the semantics of e in the context $\langle u, \top, de \rangle$.

11.5 Constraint generation

11.5.1 Algorithm

Fig. 11.7 defines our *initial constraint generator* which is the relation $\Rightarrow$ defined as the smallest relation satisfying the rules in Fig. 11.7. We use the word "initial" to distinguish it from our constraint solver defined in Sec. 11.6 which, while solving constraints, is also responsible for the generation of some constraints. Let the forms associated with terms (in Term) by our initial constraint generator be defined as follows:

$$cg \in \mathsf{InitGen} ::= e \mid \langle v, e \rangle$$

The relation $\Rightarrow$ is a binary relation defined on $\mathsf{Term} \times \mathsf{InitGen}$, i.e., $\Rightarrow \subset \mathsf{Term} \times \mathsf{InitGen}$. This relation is extended below in Sec. 14.

The rules of our constraint generator return cgs which can either be environments e (rules ($\mathsf{G17}$)-($\mathsf{G20}$)) or constrained variables of the form $\langle v, e \rangle$ where e constrains v. Such a constrained variable v is in some cases an internal type variable α (rules ($\mathsf{G1}$)-($\mathsf{G8}$),($\mathsf{G10}$)-($\mathsf{G16}$)), in some other cases a type constructor variable δ (rule ($\mathsf{G9}$)), and in some other cases an environment variable ev (rules ($\mathsf{G21}$)-($\mathsf{G22}$)). We chose not to have a constructor of constrained types that would build an internal type from an environment and an internal type (as a composition environment of the form $e_1;e_2$ builds a constrained environment from two environments because e_1 constrains e_2), because it simplifies the presentation of our system by not having deep types. Such a system with constrained types could be investigated (see also Sec. 12.1.2 on this matter). Having chosen to return pairs of the form $\langle \alpha, e \rangle$ for expressions, we then decided to follow the same pattern for structure expressions and return pairs of the form $\langle ev, e \rangle$ instead of returning composition environments of the form $e;ev$.

All rules of the form $P \Leftarrow Q$ have to be read as $P \Leftarrow (Q \wedge \mathsf{dja}(e, e_1, e_2, \alpha, \alpha', ev, ev'))$

$\boxed{\textsf{\textbf{Expressions}}\ (exp \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle)}$

$(\mathsf{G1})\ vid^{l}_{\mathtt{e}} \mathrel{-\!\triangleright} \langle\alpha,\ \uparrow vid \overset{l}{=} \alpha\rangle$

$(\mathsf{G2})\ \mathtt{let}^{l}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \mathrel{-\!\triangleright} \langle\alpha,\ [e_1;e_2;(\alpha \overset{l}{=} \alpha_2)]\rangle \Leftarrow dec \mathrel{-\!\triangleright} e_1 \wedge exp \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$(\mathsf{G3})\ \lceil exp\ \ atexp\rceil^{l} \mathrel{-\!\triangleright} \langle\alpha,\ e_1;e_2;(\alpha_1 \overset{l}{=} \alpha_2{\to}\alpha)\rangle \Leftarrow exp \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge atexp \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$(\mathsf{G4})\ \mathtt{fn}\ pat \overset{l}{\Rightarrow} exp \mathrel{-\!\triangleright} \langle\alpha,\ [(ev{=}e_1);ev^{l};e_2;(\alpha \overset{l}{=} \alpha_1{\to}\alpha_2)]\rangle \Leftarrow pat \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge exp \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$\boxed{\textsf{\textbf{Labelled datatype constructors}}\ (ldcon \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle)}$

$(\mathsf{G5})\ dcon^{l} \mathrel{-\!\triangleright} \langle\alpha,\ \uparrow dcon \overset{l}{=} \alpha\rangle$

$\boxed{\textsf{\textbf{Patterns}}\ (pat \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle)}$

$(\mathsf{G6})\ vvar^{l}_{\mathtt{p}} \mathrel{-\!\triangleright} \langle\alpha,\ \downarrow vvar \overset{l}{=} \alpha\rangle \qquad (\mathsf{G7})\ dcon^{l}_{\mathtt{p}} \mathrel{-\!\triangleright} \langle\alpha,\ \uparrow dcon \overset{l}{=} \alpha\rangle$

$(\mathsf{G8})\ \lceil ldcon\ \ atpat\rceil^{l} \mathrel{-\!\triangleright} \langle\alpha,\ e_1;e_2;(\alpha_1 \overset{l}{=} \alpha_2{\to}\alpha)\rangle \Leftarrow ldcon \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge atpat \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$\boxed{\textsf{\textbf{Labelled type constructors}}\ (ltc \mathrel{-\!\triangleright} \langle\delta,\ e\rangle)}$

$(\mathsf{G9})\ tc^{l} \mathrel{-\!\triangleright} \langle\delta,\ \uparrow tc \overset{l}{=} \delta\rangle$

$\boxed{\textsf{\textbf{Types}}\ (ty \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle)}$

$(\mathsf{G10})\ tv^{l} \mathrel{-\!\triangleright} \langle\alpha,\ \uparrow tv \overset{l}{=} \alpha\rangle$

$(\mathsf{G11})\ \lceil ty\ \ ltc\rceil^{l} \mathrel{-\!\triangleright} \langle\alpha',\ e_1;e_2;(\alpha' \overset{l}{=} \alpha\,\delta)\rangle \Leftarrow ty \mathrel{-\!\triangleright} \langle\alpha,\ e_1\rangle \wedge ltc \mathrel{-\!\triangleright} \langle\delta,\ e_2\rangle$

$(\mathsf{G12})\ ty_1 \overset{l}{\to} ty_2 \mathrel{-\!\triangleright} \langle\alpha,\ e_1;e_2;(\alpha \overset{l}{=} \alpha_1{\to}\alpha_2)\rangle \Leftarrow ty_1 \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge ty_2 \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$\boxed{\textsf{\textbf{Datatype names}}\ (dn \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle)}$

$(\mathsf{G13})\ \lceil tv\ \ tc\rceil^{l} \mathrel{-\!\triangleright} \langle\alpha',\ (\alpha' \overset{l}{=} \alpha\,\gamma);(\downarrow tc \overset{l}{=} \gamma);(\downarrow tv \overset{l}{=} \alpha)\rangle \Leftarrow \alpha \neq \alpha'$

$\boxed{\textsf{\textbf{Constructor bindings}}\ (cb \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle)}$

$(\mathsf{G14})\ dcon^{l}_{\mathtt{c}} \mathrel{-\!\triangleright} \langle\alpha,\ \downarrow dcon \overset{l}{=} \alpha\rangle$

$(\mathsf{G16})\ dcon\ \mathtt{of}^{\,l}\ ty \mathrel{-\!\triangleright} \langle\alpha,\ e_1;(\alpha' \overset{l}{=} \alpha_1{\to}\alpha);(\downarrow dcon \overset{l}{=} \alpha')\rangle \Leftarrow ty \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle$

$\boxed{\textsf{\textbf{Declarations}}\ (dec \mathrel{-\!\triangleright} e)}$

$(\mathsf{G17})\ \mathtt{val\ rec}\ pat \overset{l}{=} exp \mathrel{-\!\triangleright} (ev{=}\mathtt{poly}(e_1;e_2;(\alpha_1 \overset{l}{=} \alpha_2)));ev^{l} \Leftarrow pat \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge exp \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$(\mathsf{G18})\ \mathtt{datatype}\ dn \overset{l}{=} cb \mathrel{-\!\triangleright} (ev{=}((\alpha_1 \overset{l}{=} \alpha_2);e_1;\mathtt{poly}(e_2)));ev^{l} \Leftarrow dn \mathrel{-\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge cb \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle$

$(\mathsf{G19})\ \mathtt{open}^{l}\ strid \mathrel{-\!\triangleright} (\uparrow strid \overset{l}{=} ev);ev^{l}$

$\boxed{\textsf{\textbf{Structure declarations}}\ (strdec \mathrel{-\!\triangleright} e)}$

$(\mathsf{G20})\ \mathtt{structure}\ strid \overset{l}{=} strexp \mathrel{-\!\triangleright} [e];(ev'{=}(\downarrow strid \overset{l}{=} ev));ev'^{l} \Leftarrow strexp \mathrel{-\!\triangleright} \langle ev,\ e\rangle$

$\boxed{\textsf{\textbf{Structure expressions}}\ (strexp \mathrel{-\!\triangleright} \langle ev,\ e\rangle)}$

$(\mathsf{G21})\ strid^{l} \mathrel{-\!\triangleright} \langle ev,\ \uparrow strid \overset{l}{=} ev\rangle$

$(\mathsf{G22})\ \mathtt{struct}^{l}\ strdec_1 \cdots strdec_n\ \mathtt{end} \mathrel{-\!\triangleright} \langle ev,\ (ev \overset{l}{=} ev');(ev'{=}(e_1;\cdots;e_n))\rangle$
$\qquad \Leftarrow strdec_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge strdec_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, ev, ev')$

Figure 11.7 Constraint generation rules

11.5.2 Shape of the generated environments

Our initial constraint generator defined in Fig. 11.7 only generates restricted forms of environments (ge defined below, where "g" stands for "generation"). Let us present these restricted forms, where sit is a restriction of τ, and the other forms are restrictions of e (where "p" stands for "poly" and "l" for "labelled"):

$$\begin{array}{lll}
\mathit{sit} & \in \mathsf{ShallowITy} & ::= \alpha \mid \alpha\,\delta \mid \alpha\,\gamma \mid \alpha_1{\rightarrow}\alpha_2 \\
\mathit{lbind} & \in \mathsf{LabBind} & ::= {\downarrow}\mathit{tc} \overset{l}{=} \gamma \mid {\downarrow}\mathit{strid} \overset{l}{=} \mathit{ev} \mid {\downarrow}\mathit{tv} \overset{l}{=} \alpha \mid {\downarrow}\mathit{vid} \overset{l}{=} \alpha \\
\mathit{lc} & \in \mathsf{LabCs} & ::= \mathit{ev}_1 \overset{l}{=} \mathit{ev}_2 \mid \alpha \overset{l}{=} \mathit{sit} \\
\mathit{lacc} & \in \mathsf{LabAcc} & ::= \mathit{acc}^l \\
\mathit{lev} & \in \mathsf{LabEnvVar} & ::= \mathit{ev}^l \\
\mathit{ipe} & \in \mathsf{InPolyEnv} & ::= \mathit{lacc} \mid \mathit{lc} \mid \mathit{ipe}_1;\mathit{ipe}_2 \\
\mathit{pe} & \in \mathsf{PolyEnv} & ::= {\downarrow}\mathit{vid} \overset{l}{=} \alpha \mid \mathit{pe};\mathit{ipe} \mid \mathit{ipe};\mathit{pe} \\
\mathit{ge} & \in \mathsf{GenEnv} & ::= \top \mid \mathit{lev} \mid \mathit{lbind} \mid \mathit{lacc} \mid \mathit{lc} \mid \mathit{ev}{=}\mathit{ge} \mid \mathtt{poly}(\mathit{pe}) \mid \mathit{ge}_1;\mathit{ge}_2
\end{array}$$

At initial constraint generation, the only labelled (dependent) environments are equality constraints (c), binders (bind), accessors (acc), and environment variables (ev). Also, note that a pe contains exactly one binder and can also contain equality constraints as well as accessors.

11.5.3 Complexity of constraint generation

Inspection reveals the generated constraint's size is linear in the program size. Unlike HW-TES' constraint generation [57], for a polymorphic (let-bound) function (see the combination of rules (G2), (G6) and (G17)) we do not eagerly copy constraints for the function body. Instead, we generate (among other things) `poly` environments, composition environments, and binders, and force solving (constraint solving is defined below in Sec. 11.6) the constraints for the body before copying its type for each use of the function. This type is a simplified form of the constraints generated for the function body.

11.5.4 Discussion of some constraint generation rules

In rule (G17), the environment e_1 generated for pat constrains e_2 generated for exp. This order is necessary to handle the recursivity of such declarations. The binders in e_1 are monomorphic. Polymorphic type schemes are generated at constraint solving when dealing with the `poly` constraint. Within the `poly` environment, binders need to be monomorphic because SML does not allow polymorphic recursion. Allowing `poly` constraints on environments other than just a single binder (e.g., allowing `poly` on a binder constraining equality constraints and accessors such as in $\mathit{bind};c;\mathit{acc}$ where acc could potentially refer to bind) allows one to delay the generation of polymorphic types. Therefore, given a recursive function declaration, one can generate only one binder for the function (in a naive approach two would be needed: one monomorphic for the function's body and one polymorphic for the function's scope as mentioned in Sec. 12.1.7 below).

In rule (G18) for datatype declarations, the environment e_1 generated for the declared type constructor constrains the environment $\mathtt{poly}(e_2)$ generated for the datatype constructor of the declared type constructor. This order is necessary to handle the recursivity of such datatype declarations. For example, in the declaration `datatype nat = z | s of nat`, `nat`'s second occurrence refers to its first occurrence. Note that e_1 also binds explicit type variables in Core-TES. This extends the scope of the bound external type variable further than needed, but causes no harm in Core-TES, in which all type variables only occur inside datatype constructor bindings. This will be changed in Sec. 14.3, after introducing internal local environments in Sec. 14.2.

Rules (G4), (G17), (G18), (G19) and (G20) label environment variables to prevent sliced out declarations from shadowing their context (e.g., in our constraint system if ev is unconstrained, it shadows e in $e;ev$ which is something we do not want to happen in these rules). In each of these rules, such an environment variable represents the entire declaration. For example, in rule (G19), ev represents the entire analysed opening declaration. Our initial constraint generation algorithm labels ev using l, the label associated with the analysed opening declaration. In rule (G19) (as in any of the other rules mentioned above), without l the environment variable ev would be a constraint that always has to be satisfied, even when the corresponding opening declaration has been sliced out. For example, slicing out `open S` in `structure S = struct end; val x = 1; open S; val y = x 1` would result in the environment variable generated for `open S` shadowing its context which contains the declaration `val x = 1`. Failing from labelling ev using l in rule (G19) would therefore prevent from finding the error that `x` is declared as an integer in the piece of code presented above, and is also applied to an argument in `y`'s body. With the label, the environment variable is a constraint that has to be satisfied only when the declaration is not sliced out. Note that in rule (G19), the link between the environment variable and the structure to open is made via the labelled accessor.

Rules (G4), (G17), (G18), (G20) and (G22) generate unlabelled equality constraints. Those generated by rule (G22) are of the form $ev'=(e_1;\cdots;e_n)$. Such a constraint needs to be unlabelled because each e_i does not depend on the analysed structure expression $\mathtt{struct}^l\ strdec_1 \cdots strdec_n\ \mathtt{end}$ itself, but only on the corresponding declaration $strdec_i$ packed together with other declarations in the structure expression. Therefore when slicing out the packaging created by this structure expression (i.e., when slicing out l above) we do not want to discard all the e_is as well (which is what would happen if we were to label $ev'=(e_1;\cdots;e_n)$ with l and entirely discard it when slicing out l). The information related to the structure expression $\mathtt{struct}^l\ strdec_1 \cdots strdec_n\ \mathtt{end}$, carried by the unlabelled equality constraint $ev'=(e_1;\cdots;e_n)$, is the fact that a sequence of declarations (corresponding to the composition environment $e_1;\cdots;e_n$) is packed into a structure. This information depends on the structure expression via the extra labelled equality constraint $ev\overset{l}{=}ev'$. In rules (G4), (G17), (G18) and (G20), we use labelled environment variables of the form ev^l for this purpose.

11.5.5 Constraints generated for example (EX1)

We now present the constraints generated for example (EX1) presented in Sec. 11.2. First, let us repeat the labelled version of example (EX1) which is called $strdec_{\mathrm{EX}}$:

$$\begin{array}{l}
\texttt{structure X} \overset{l_1}{=} \texttt{struct}^{l_2} \\
\qquad\qquad \texttt{structure S} \overset{l_3}{=} \texttt{struct}^{l_4}\ \texttt{datatype}\ \lceil\texttt{'a u}\rceil^{l_6} \overset{l_5}{=} \texttt{U}_{\texttt{c}}^{l_7}\ \texttt{end} \\
\qquad\qquad \texttt{datatype}\ \lceil\texttt{'a t}\rceil^{l_9} \overset{l_8}{=} \texttt{T}_{\texttt{c}}^{l_{10}} \\
\qquad\qquad \texttt{val rec f}_{\texttt{p}}^{l_{12}} \overset{l_{11}}{=} \texttt{fn T}_{\texttt{p}}^{l_{14}} \overset{l_{13}}{\Rightarrow} \texttt{T}_{\texttt{e}}^{l_{15}} \\
\qquad\qquad \texttt{val rec g}_{\texttt{p}}^{l_{17}} \overset{l_{16}}{=} \texttt{let}^{l_{18}}\ \texttt{open}^{l_{19}}\ \texttt{S in}\ \lceil\texttt{f}_{\texttt{e}}^{l_{21}}\ \texttt{U}_{\texttt{e}}^{l_{22}}\rceil^{l_{20}}\ \texttt{end} \\
\qquad\quad \texttt{end}
\end{array}$$

We assume in this section that the generated variables and type constructor names are all distinct from each other.

The environment generated for `datatype 'a u = U` which we call e_0 is as follows:

$$\begin{array}{rcl}
e_0 & = & (ev_4{=}((\alpha_1 \overset{l_5}{=} \alpha_2);e''_0;e'_0));ev_4^{l_5} \\
& & \text{such that} \begin{cases} e'_0 & \text{is}\ \ \texttt{poly}(\downarrow\texttt{U} \overset{l_7}{=} \alpha_2) \\ e''_0 & \text{is}\ \ (\alpha_1 \overset{l_6}{=} \alpha'_1\,\gamma_1);(\downarrow\texttt{u} \overset{l_6}{=} \gamma_1);(\downarrow\texttt{'a} \overset{l_6}{=} \alpha'_1) \end{cases}
\end{array}$$

The environment generated for `structure S = struct datatype 'a u = U end`, which we call e_1 is as follows:

$$e_1 \ = \ [(ev_2 \overset{l_4}{=} ev_3);(ev_3{=}e_0)];(ev_1{=}(\downarrow\texttt{S} \overset{l_3}{=} ev_2));ev_1^{l_3}$$

The environment generated for `datatype 'a t = T`, which we call e_2 is as follows:

$$\begin{array}{rcl}
e_2 & = & (ev_5{=}((\alpha_3 \overset{l_8}{=} \alpha_4);e''_2;e'_2));ev_5^{l_8} \\
& & \text{such that} \begin{cases} e'_2 & \text{is}\ \ \texttt{poly}(\downarrow\texttt{T} \overset{l_{10}}{=} \alpha_4) \\ e''_0 & \text{is}\ \ (\alpha_3 \overset{l_9}{=} \alpha'_3\,\gamma_2);(\downarrow\texttt{t} \overset{l_9}{=} \gamma_2);(\downarrow\texttt{'a} \overset{l_9}{=} \alpha'_3) \end{cases}
\end{array}$$

The environment generated for `val rec f = fn T => T`, which we call e_3 is as follows:

$$\begin{array}{rcl}
e_3 & = & (ev_6{=}\texttt{poly}((\downarrow\texttt{f} \overset{l_{12}}{=} \alpha_5);e'_3;(\alpha_5 \overset{l_{11}}{=} \alpha_6)));ev_6^{l_{11}} \\
& & \text{such that } e'_3 = [(ev_7{=}(\uparrow\texttt{T} \overset{l_{14}}{=} \alpha_7));ev_7^{l_{13}};(\uparrow\texttt{T} \overset{l_{15}}{=} \alpha_8);(\alpha_6 \overset{l_{13}}{=} \alpha_7{\rightarrow}\alpha_8)]
\end{array}$$

The environment generated for `val rec g = let open S in f U end`, which we call e_4 is as follows:

$$\begin{array}{rcl}
e_4 & = & (ev_8{=}\texttt{poly}((\downarrow\texttt{g} \overset{l_{17}}{=} \alpha_9);[e'_4;e''_4;(\alpha_{10} \overset{l_{18}}{=} \alpha_{11})];(\alpha_9 \overset{l_{16}}{=} \alpha_{10})));ev_8^{l_{16}} \\
& & \text{such that} \begin{cases} e'_4 & \text{is}\ \ (\uparrow\texttt{S} \overset{l_{19}}{=} ev_9);ev_9^{l_{19}} \\ e''_4 & \text{is}\ \ (\uparrow\texttt{f} \overset{l_{21}}{=} \alpha_{12});(\uparrow\texttt{U} \overset{l_{22}}{=} \alpha_{13});(\alpha_{12} \overset{l_{20}}{=} \alpha_{13}{\rightarrow}\alpha_{11}) \end{cases}
\end{array}$$

Finally, the environment generated for the entire piece of code is the following environment which we call e_{EX}:

$$e_{\mathrm{EX}} \ = \ [(ev_{11} \overset{l_2}{=} ev_{12});(ev_{12}{=}(e_1;e_2;e_3;e_4))];(ev_{10}{=}(\downarrow\texttt{X} \overset{l_1}{=} ev_{11}));ev_{10}^{l_1}$$

$$\begin{array}{rl} er \in \mathsf{Error} & ::= \langle ek, \overline{d} \rangle \\ ek \in \mathsf{ErrKind} & ::= \mathtt{tyConsClash}(\mu_1, \mu_2) \mid \mathtt{circularity} \\ state \in \mathsf{State} & ::= \mathtt{slv}(\mathit{\Delta}, \overline{d}, e) \mid \mathtt{succ}(\mathit{\Delta}) \mid \mathtt{err}(er) \end{array}$$

Figure 11.8 Syntactic forms used by the constraint solver

11.6 Constraint solving

11.6.1 Syntax

Fig. 11.8 defines additional syntactic forms used by our constraint solver (Fig. 11.10) where a constraint solving step is defined by the relation →, and where $\rightarrow^*$ is its reflexive (w.r.t. State) and transitive closure. A constraint solving process always starts in a state of the form $\mathtt{slv}(\langle \varnothing, \top \rangle, \varnothing, e)$ where $\top$ is called the *initial environment*. Given such a state, our constraint solver either succeeds with final state $\mathtt{succ}(\mathit{\Delta})$ returning its current constraint solving context $\mathit{\Delta}$, or fails with final state $\mathtt{err}(er)$ returning an error which can be a type constructor clash or a circularity error (see ek in Fig. 11.8). Given a state $\mathtt{slv}(\mathit{\Delta}, \overline{d}, e)$, if the dependencies in $\overline{d}$ are satisfied and e is solvable in the context $\mathit{\Delta}$ then the constraint solver will succeed with final state $\mathtt{succ}(\mathit{\Delta}')$ for some $\mathit{\Delta}'$.

11.6.2 Building of constraint terms

We defined a substitution operation in Sec. 11.3. Let us now define a new substitution operation called "build" that differs from the one defined in Sec. 11.3 by the facts that: it is recursively called in the variable case, it is undefined on ∀ schemes and environments, and it collapses dependencies. The constraint solver defined in Fig. 11.10 uses build to generate, w.r.t. a given constraint solving context, polymorphic types from monomorphic ones (build is called by toPoly in Fig. 11.9) and check circularity errors (in order not to generate a unifier where, e.g., $\alpha = \tau \rightarrow \alpha$):

$$\begin{array}{ll} \mathsf{build}(u, v) = \begin{cases} \mathsf{build}(u, x), \text{if } u(v) = x \\ v, \quad\quad\quad \text{otherwise} \end{cases} & \begin{array}{ll} \mathsf{build}(u, \tau_1 \rightarrow \tau_2) & = \mathsf{build}(u, \tau_1) \rightarrow \mathsf{build}(u, \tau_2) \\ \mathsf{build}(u, x^{\overline{d}}) & = \mathsf{collapse}(\mathsf{build}(u, x)^{\overline{d}}) \end{array} \\ \mathsf{build}(u, \tau\, \mu) = \mathsf{build}(u, \tau)\, \mathsf{build}(u, \mu) & \mathsf{build}(u, x) \quad\quad\ = x, \text{ otherwise} \end{array}$$

We also extend the build function to constraint solving contexts as follows:

$$\mathsf{build}(\langle u,\ e \rangle, x) = \mathsf{build}(u, x).$$

Types have to be built up when generating polymorphic environments (see Sec. 11.6.4) for efficiency issues (to avoid duplicating constraints). Also, because SML does not allow infinite types, we use build to detect circularity issues. During constraint solving, before augmenting any constraint solving context, we check if the augmentation could lead to the generation of infinite types (see rule (U1) in Fig. 11.10). For example, given the unifier $\{\alpha_1 \mapsto \alpha_2^{\overline{d_1}}, \alpha_2 \mapsto \langle \alpha_3^{\overline{d_3}} \rightarrow \alpha_4^{\overline{d_4}}, \overline{d_2} \rangle\}$, we do

$$\begin{array}{l} \mathsf{toPoly}(\mathit{\Delta}, {\downarrow}\mathit{vid} \overset{\overline{d}}{=} \tau) \\ = \mathit{\Delta};({\downarrow}\mathit{vid} \overset{\overline{d}\cup\overline{d}'}{=\!=\!=} \forall\overline{\alpha}.\,\tau') \end{array} \quad \text{where} \begin{cases} \tau' = \mathsf{build}(\mathit{\Delta}, \tau) \\ \overline{\alpha} = (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\mathit{\Delta})) \cup \{\alpha_{\mathsf{dum}}\}) \\ \overline{d}' = \{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\mathit{\Delta}) \wedge \alpha \in \mathsf{vars}(\tau') \setminus \overline{\alpha}\} \end{cases}$$

Figure 11.9 Monomorphic to polymorphic environment

not allow its augmentation with, e.g., $\{\alpha_3 \mapsto \langle \alpha_5^{\overline{d}_6} \to \alpha_1^{\overline{d}_7}, \overline{d}_5 \rangle\}$ because it would allow one to generate infinite types.

Note that $\tau[u]$ and $\mathsf{build}(u, \tau)$ do not always yield the same result. Consider $u = \{\alpha_1 \mapsto \alpha_2, \alpha_2 \mapsto \alpha_3\}$ where $\mathsf{dja}(\alpha_1, \alpha_2, \alpha_3)$. Then $\alpha_1[u] = \alpha_2$ but $\mathsf{build}(u, \alpha_1) = \alpha_3$. The result would be the same if u was idempotent (i.e., if we had $u[u] = u$).

11.6.3 Environment extraction

The function diff is used by rules (U4) and (P1) of our constraint solver (see Fig. 11.10) to extract environments generated during solving. It is defined as follows:

$$\begin{array}{ll} \mathsf{diff}(e, e) & = \top \\ \mathsf{diff}(e_1, (e_2;e_3)) & = \mathsf{diff}(e_1, e_2);e_3 \text{ if } e_1 \neq (e_2;e_3) \end{array}$$

When solving an environment, it allows one to get back its "solved version" once all of its constraints have been dealt with. By "solved version" of an environment e, we mean the sequence of environments that has been added to the constraint solving context of the state in which the constraint solving process was when it started to solve e. For example, if $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_0) \to^* \mathtt{succ}(\langle u',\ e'\rangle)$ then $e' = e;e_1;\cdots;e_n$ and $\mathsf{diff}(e, e') = \top;e_1;\cdots;e_n$ which is the "solved version" of e_0 w.r.t. e.

11.6.4 Polymorphic environments

The function monos computes the set of dependent monomorphic type variables occurring in an environment w.r.t. a unifier as follows (the type variables occurring in the types of the monomorphic binders):

$$\mathsf{monos}(\mathit{\Delta}) = \{\alpha^{\mathsf{deps}(\tau)} \mid \exists \mathit{vid}.\ \tau = \mathsf{build}(\mathit{\Delta}, \mathit{\Delta}(\mathit{vid})) \wedge \alpha \in \mathsf{nonDums}(\tau)\}$$

Note that in monos' definition, $\tau = \mathsf{build}(\mathit{\Delta}, \mathit{\Delta}(\mathit{vid}))$ enforces that vid has a monomorphic binder in $\mathit{\Delta}$. For example, in $\langle u,\ e;({\downarrow}\mathit{vid} \overset{\overline{d}}{=} \tau)\rangle$, vid has a monomorphic binder because τ is not a $\forall$ (dependent or not) type scheme.

Fig. 11.9 defines toPoly which, given a constraint solving context $\mathit{\Delta}$ and a dependent monomorphic value identifier binder, generates a polymorphic binder by quantifying the type variables not occurring in the types of the monomorphic binders of $\mathit{\Delta}$. The function toPoly is used by the semantic rule (poly) and by the constraint solving rule (P1).

In Fig. 11.9, τ is the type from which a type scheme is generated. First, we build up τ, using the constraint solving context ($\mathit{\Delta}$) of the current state, to obtain the type τ'. The set $\overline{\alpha}$ is the set of type variables that are quantified over because they do not depend on the types of monomorphic binders. The dependencies set $\overline{d}'$ "explains" why the type variables not in $\overline{\alpha}$ but occurring in τ' (therefore depending on monomorphic binders) are not allowed to be quantified over. Roughly speaking, $\overline{\alpha}$ is the set of polymorphic type variables in τ' and $\mathsf{vars}(\tau')\backslash\overline{\alpha}$ is the set of monomorphic type variables in τ'.

Let us illustrate this mechanism using the fn-expression *exp* defined as follows: `fn x => let val rec f = fn z => x z in f end`. At initial constraint generation, an environment of the form $\mathtt{poly}(e_1)$[1] is generated for the recursive declaration `val rec f = fn z => x z`. When solving the constraints generated for *exp*, the constraint solver eventually applies toPoly to a constraint solving context $\langle u,\ e\rangle$ and a binder of the form $\langle\downarrow\mathtt{f}{=}\alpha_1, \overline{d}\rangle$ (which is the "solved version" of e_1). Building up α_1 results in a type τ' of the form $\langle\alpha_2^{\overline{d}_2}{\rightarrow}\alpha_3^{\overline{d}_3}, \overline{d}_1\rangle$. Because `x`'s type is monomorphic, a monomorphic binder (the only one) of the form $\downarrow\mathrm{x}{=}\alpha_0$ occurs in e and so $\mathsf{vars}(\mathsf{monos}(\langle u,\ e\rangle)) = \mathsf{vars}(\tau_0)$ where τ_0 is obtained by building up α_0 and is of the form $\langle\alpha_2^{\overline{d}_5}{\rightarrow}\alpha_3^{\overline{d}_6}, \overline{d}_4\rangle$ (equivalent to τ' up to dependencies because `f` eta-reduces to `x`). We therefore build a $\overline{\alpha}$ (see Fig. 11.9) of the form $\varnothing$ because α_2 and α_3 both occur in τ_0. We also build a $\overline{d}'$ of the form $\overline{d}_4\cup\overline{d}_5\cup\overline{d}_6$ which are the "reasons" for not allowing α_2 and α_3 to be in $\overline{\alpha}$ (type variable set allowed to be generalised over when building the type scheme returned by toPoly). Finally, e is augmented with $\langle\downarrow\mathtt{f}{=}\forall\varnothing.\,\langle\alpha_2^{\overline{d}_2}{\rightarrow}\alpha_3^{\overline{d}_3}, \overline{d}_1\rangle, \overline{d}\cup\overline{d}'\rangle$.

When solving constraints generated by our constraint generator, toPoly is only applied to $\mathit{bind}^{\overline{d}}$'s resulting from the solving of an environment wrapped by `poly` which in turn is only used to wrap environments built from: dependencies, a single monomorphic binder, equality constraints, and accessors (see $\mathsf{PolyEnv}$'s definition in Sec. 11.5).

Extracting the monomorphic type variables of a binder's type is expensive. We only perform it once per polymorphic binder by, given a constraint solving context, first building the type of a given binder and by then looking up in the constraint solving context which type variables are monomorphic. When accessing the type of a polymorphic binder we then only have to generate an instance of its type scheme (see rule ($\mathsf{A1}$) of our constraint solver).

In Fig. 11.9, the computation of $\overline{d}'$ and our constraining of the generated type scheme with $\overline{d}'$, even though a correct approximation (that cannot generate false errors and that will eventually allow one to obtain minimal type errors), could be refined, thereby speeding up minimisation. This refinement is presented in Sec. 11.6.7.

[1] When considering the following labelling: $\mathtt{val\ rec}\ \underset{\mathtt{p}}{\mathtt{f}}^{l_6} \overset{l_7}{=} \mathtt{fn}\ \underset{\mathtt{p}}{\mathtt{z}}^{l_4} \overset{l_5}{\Rightarrow} \lceil \underset{\mathtt{e}}{\mathtt{x}}^{l_1}\ \underset{\mathtt{e}}{\mathtt{z}}^{l_2}\rceil^{l_3}$, the environment e_1 is of the form $\downarrow\mathtt{f} \overset{l_6}{=} \alpha_6;[ev{=}(\downarrow\mathtt{z} \overset{l_4}{=} \alpha_4);ev^{l_5};\uparrow\mathtt{x} \overset{l_1}{=} \alpha_1;\uparrow\mathtt{z} \overset{l_2}{=} \alpha_2;\alpha_1\overset{l_3}{=}\alpha_2{\rightarrow}\alpha_3;\alpha_5\overset{l_5}{=}\alpha_4{\rightarrow}\alpha_3];\alpha_6\overset{l_7}{=}\alpha_5$.

11.6.5 Algorithm

Let $u_1 \oplus u_2$ be $\{(v \mapsto x) \in u_1 \cup u_2 \mid \mathsf{dj}(\{v, \mathsf{strip}(x)\}, \mathsf{Dum})\}$ if $\mathsf{dj}(\mathsf{Dum} \lhd \mathsf{dom}(u_1), \mathsf{Dum} \lhd \mathsf{dom}(u_2))$, and undefined otherwise. This function allows us, at constraint solving (see rules $(\mathsf{U3})$ and rule $(\mathsf{U4})$ of our constraint solved defined in Fig. 11.10), to generate unifiers which do not constrain dummy variables. For example, $u \oplus \{\alpha_{\mathsf{dum}} \mapsto \tau\} = u \oplus \{\alpha \mapsto \alpha^{\overline{d}}_{\mathsf{dum}}\} = u$.

Fig. 11.10 defines our constraint solver which can be regarded as a rewriting system. A finite computation is then a finite sequence of states $\langle \mathit{state}_1, \ldots, \mathit{state}_n \rangle$ such that for each $i \in \{1, \ldots, n-1\}$, the state state_{i+1} is obtained by applying one of the constraint solving rules as defined in Fig. 11.10 to the state state_i (i.e., the pair $\langle \mathit{state}_i, \mathit{state}_{i+1} \rangle$ is obtained by instantiating one of the constraint solving rules, where state_i is the instantiation of the left-hand-side of the rule and state_{i+1} is the instantiation of the right-hand-side).

Rule $(\mathsf{A3})$ can be used to report free identifiers. If $\mathtt{slv}(\Delta, \overline{d}, \uparrow id{=}v) \to \mathtt{succ}(\Delta)$ and $\neg\mathsf{shadowsAll}(\Delta)$ then it means that there is no binder for id and so that it is a free identifier. Free identifiers are in any case important to report, but it is especially vital for structure identifiers in `open` declarations. In our approach, a free opened structure is considered as potentially redefining its entire context. Hence, `val x = 1; open S; val y = x 1` does not have an error involving `x` because `x`'s first occurrence is hidden by the declaration `open S`. This might be confusing if `S` was not reported as being free. Let us explain how a free opened structure shadows its context. Given a declaration `open S` labelled by l, our initial constraint generation algorithm generates an environment of the form $(\uparrow\mathtt{S} \overset{l}{=} ev); ev^l$. Because `S` is free, rule $(\mathsf{A3})$ applies when solving $\uparrow\mathtt{S}{=}ev$. The environment variable ev is then unconstrained. Hence, when solving ev, rule (V) applies and $\Delta; ev$ (from the right-hand-side of rule (V)) results in the shadowing of all the binders in Δ by ev.

Let the relations isErr and $\mathsf{solvable}$ be defined as follows:

$$\begin{array}{ll}
e \xrightarrow{\mathsf{isErr}} er & \Leftrightarrow \mathtt{slv}(\langle \varnothing, \top \rangle, \varnothing, e) \to^* \mathtt{err}(er) \\
\mathsf{solvable}(e) & \Leftrightarrow \exists \Delta.\ \mathtt{slv}(\langle \varnothing, \top \rangle, \varnothing, e) \to^* \mathtt{succ}(\Delta) \\
\mathsf{solvable}(\mathit{strdec}) & \Leftrightarrow \exists e.\ \mathit{strdec} \mathrel{-\!\!\triangleright} e \wedge \mathsf{solvable}(e)
\end{array}$$

These relations are used, among other things, to define our minimisation and enumeration algorithms in Sec. 11.7.

11.6.6 Shape of the environments generated during constraint solving

During constraint solving (see Fig. 11.10), a constraint solving context of the form $\langle u,\ e \rangle$ is maintained. No c or acc occurs in e because they are transformed instead into unifiers u (rules $(\mathsf{U3})$ and $(\mathsf{U4})$ in Fig. 11.10). Similarly, the $\mathtt{poly}(e')$ forms are

equality constraint reversing

(R) $\mathtt{slv}(\Delta, \overline{d}, x{=}y) \rightarrow \mathtt{slv}(\Delta, \overline{d}, y{=}x)$, if $s = \mathsf{Var} \cup \mathsf{Dependent} \wedge y \in s \wedge x \notin s$

equality simplification

(S1) $\mathtt{slv}(\Delta, \overline{d}, x{=}x) \rightarrow \mathtt{succ}(\Delta)$

(S2) $\mathtt{slv}(\Delta, \overline{d}, x^{\overline{d}'}{=}y) \rightarrow \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', x{=}y)$

(S3) $\mathtt{slv}(\Delta, \overline{d}, \tau\,\mu{=}\tau'\,\mu') \rightarrow \mathtt{slv}(\Delta, \overline{d}, (\mu{=}\mu');(\tau{=}\tau'))$

(S4) $\mathtt{slv}(\Delta, \overline{d}, \tau_1{\rightarrow}\tau_2{=}\tau_3{\rightarrow}\tau_4) \rightarrow \mathtt{slv}(\Delta, \overline{d}, (\tau_1{=}\tau_3);(\tau_2{=}\tau_4))$,

(S5) $\mathtt{slv}(\Delta, \overline{d}, \tau_1{=}\tau_2) \rightarrow \mathtt{slv}(\Delta, \overline{d}, \mu{=}\mathtt{ar})$, if $\{\tau_1, \tau_2\} = \{\tau\,\mu, \tau_3{\rightarrow}\tau_4\}$

(S6) $\mathtt{slv}(\Delta, \overline{d}, \mu_1{=}\mu_2) \rightarrow \mathtt{err}(\langle \mathtt{tyConsClash}(\mu_1, \mu_2), \overline{d}\rangle)$, if $\{\mu_1, \mu_2\} \in \{\{\gamma, \gamma'\}, \{\gamma, \mathtt{ar}\}\}$
$\wedge\ \gamma \neq \gamma'$

unifier access

Rules (U1) through (U6) have also these common side conditions: $v \neq x$ and $y = \mathsf{build}(u, x^{\overline{d}})$.

(U1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, v{=}x) \rightarrow \mathtt{err}(\langle \mathtt{circularity}, \mathsf{deps}(y)\rangle)$,
if $v \in \mathsf{vars}(y) \setminus (\mathsf{dom}(u) \cup \mathsf{Env} \cup \mathsf{Dum}) \wedge \mathsf{strip}(y) \neq v$

(U2) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, v{=}x) \rightarrow \mathtt{succ}(\langle u,\ e\rangle)$,
if $v \in \mathsf{vars}(y) \setminus (\mathsf{dom}(u) \cup \mathsf{Env}) \wedge \mathsf{strip}(y) = v$

(U3) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, v{=}x) \rightarrow \mathtt{succ}(\langle u{\oplus}\{v \mapsto y\},\ e\rangle)$,
if $v \notin (\mathsf{vars}(y) \setminus \mathsf{Dum}) \cup \mathsf{dom}(u) \cup \mathsf{Env}$

(U4) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, v{=}x) \rightarrow \mathtt{succ}(\langle u'{\oplus}\{v \mapsto \mathsf{diff}(e, e')\},\ e\rangle)$,
if $v \in \mathsf{Env} \setminus \mathsf{dom}(u) \wedge \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, x) \rightarrow^* \mathtt{succ}(\langle u',\ e'\rangle)$

(U5) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, v{=}x) \rightarrow \mathtt{err}(er)$,
if $v \in \mathsf{Env} \setminus \mathsf{dom}(u) \wedge \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, x) \rightarrow^* \mathtt{err}(er)$

(U6) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, v{=}x) \rightarrow \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, z{=}x)$,
if $u(v) = z$

binders

(B1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, {\downarrow}id{=}x) \rightarrow \mathtt{succ}(\langle u,\ e;({\downarrow}id \overset{\overline{d}}{=} x)\rangle)$

empty/dependent/variables

(E) $\mathtt{slv}(\Delta, \overline{d}, \top) \rightarrow \mathtt{succ}(\Delta)$

(D) $\mathtt{slv}(\Delta, \overline{d}, e^{\overline{d}'}) \rightarrow \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', e)$

(V) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, ev) \rightarrow \mathtt{succ}(\langle u,\ e;ev^{\overline{d}}\rangle)$

composition environments

(C1) $\mathtt{slv}(\Delta, \overline{d}, e_1;e_2) \rightarrow \mathtt{slv}(\Delta', \overline{d}, e_2)$, if $\mathtt{slv}(\Delta, \overline{d}, e_1) \rightarrow^* \mathtt{succ}(\Delta')$

(C2) $\mathtt{slv}(\Delta, \overline{d}, e_1;e_2) \rightarrow \mathtt{err}(er)$, if $\mathtt{slv}(\Delta, \overline{d}, e_1) \rightarrow^* \mathtt{err}(er)$

accessors

(A1) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \mathtt{slv}(\Delta, \overline{d} \cup \overline{d}', v{=}\tau[ren])$,
if $\Delta(id) = (\forall\overline{\alpha}.\,\tau)^{\overline{d}'} \wedge \mathsf{dom}(ren) = \overline{\alpha} \wedge \mathsf{dj}(\mathsf{vars}(\langle \Delta, v\rangle), \mathsf{ran}(ren))$

(A2) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \mathtt{slv}(\Delta, \overline{d}, v{=}x)$,
if $\Delta(id) = x \wedge \mathsf{strip}(x)$ is not of the form $\forall\overline{\alpha}.\,\tau$

(A3) $\mathtt{slv}(\Delta, \overline{d}, {\uparrow}id{=}v) \rightarrow \mathtt{succ}(\Delta)$,
if $\Delta(id)$ undefined

polymorphic environments

(P1) $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, \mathtt{poly}(e)) \rightarrow \mathtt{succ}(\mathsf{toPoly}(\langle u_2,\ e_1\rangle, \mathsf{diff}(e_1, e_2)))$,
if $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, e) \rightarrow^* \mathtt{succ}(\langle u_2,\ e_2\rangle)$

(P2) $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, \mathtt{poly}(e)) \rightarrow \mathtt{err}(er)$,
if $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, e) \rightarrow^* \mathtt{err}(er)$

Figure 11.10 Constraint solver

eliminated. Moreover, given that a constraint solving process always starts with the initial environment $\top$, the environment e is then of the form $\top;e_1 \cdots ;e_n$, where each e_i is built from dependencies, binders, environment variables, composition environments, and $\top$. Such an environment e is of the form se defined as follows:

$$
\begin{array}{lll}
\mathit{sbind} \in \mathsf{SolvBind} & ::= & {\downarrow}\mathit{tc}{=}\mu \mid {\downarrow}\mathit{strid}{=}\mathit{se} \mid {\downarrow}\mathit{tv}{=}\alpha \mid {\downarrow}\mathit{vid}{=}\sigma \\
\mathit{serhs} \in \mathsf{SolvEnvRHS} & ::= & \top \mid \mathit{ev} \mid \mathit{sbind} \mid \mathit{serhs}_1;\mathit{serhs}_2 \mid \mathit{serhs}^{\overline{d}} \\
\mathit{se} \quad \in \mathsf{SolvEnv} & ::= & \top \mid \top;\mathit{serhs}
\end{array}
$$

It is also the case that, for any environment variable $ev \in \mathsf{dom}(u)$, $u(ev) \in \mathsf{SolvEnv}$. This is obtained by a simple inspection of the constraint solving rules.

We sometimes call an environment of the form se, a “solved” environment.

11.6.7 Improved generation of polymorphic environments

Fig. 11.9 defines the simple toPoly function which is used by rule ($\mathsf{P1}$) of our constraint solver to generate a polymorphic environment from a monomorphic one by quantifying the type variables not occurring in the types of the monomorphic bindings of the current constraint solving context. In this figure $\overline{\alpha}$ is the set of type variables occurring in τ' (the type that we want to generalise to a $\forall$ scheme) that can be generalised and quantified over. The dependencies in the dependency set $\overline{d}'$ are the reasons for not generalising the type variables occurring in τ' that are not in $\overline{\alpha}$ (these dependencies are the reasons why some type variables are not allowed to be quantified over).

As mentioned in Sec. 11.6.4, the computation of $\overline{d}'$ and our constraining of the generated type scheme with $\overline{d}'$, even though a correct approximation, could be refined, thereby speeding up minimisation. We now present how this can be done.

First, we define functions from internal type variables to dependency sets as follows:

$$\mathit{tvdeps} \in \mathsf{ITyvarToDeps} = \mathsf{ITyVar} \to \mathbb{P}(\mathsf{Dependency})$$

Let us now define the two functions $\mathsf{getDeps}$ and $\mathsf{putDeps}$. The application $\mathsf{getDeps}(\alpha, \tau, \varnothing)$ results in the dependency set occurring in τ on the paths from its root node to any occurrence of α. The application $\mathsf{putDeps}(\tau, \mathit{tvdeps})$ results in the constraining, for each variable α in $\mathsf{dom}(\mathit{tvdeps})$, of the occurrences of α in τ with the dependency set $\mathit{tvdeps}(\alpha)$. The function $\mathsf{getDeps}$ is defined as follows:

$$
\begin{array}{ll}
\mathsf{getDeps}(\alpha, \alpha', \overline{d}) & = \begin{cases} \overline{d}, \text{ if } \alpha = \alpha' \\ \varnothing, \text{otherwise} \end{cases} \\
\mathsf{getDeps}(\tau\,\mu, \alpha, \overline{d}) & = \mathsf{getDeps}(\tau, \alpha, \overline{d}) \\
\mathsf{getDeps}(\tau_1{\to}\tau_2, \alpha, \overline{d}) & = \mathsf{getDeps}(\tau_1, \alpha, \overline{d}) \cup \mathsf{getDeps}(\tau_2, \alpha, \overline{d}) \\
\mathsf{getDeps}(\tau^{\overline{d}}, \alpha, \overline{d}') & = \mathsf{getDeps}(\tau, \alpha, \overline{d} \cup \overline{d}')
\end{array}
$$

The function $\mathsf{putDeps}$ is defined as follows:

$$
\begin{array}{ll}
\mathsf{putDeps}(\alpha, \mathit{tvdeps}) & = \begin{cases} \alpha^{\overline{d}}, \text{if } \mathit{tvdeps}(\alpha) = \overline{d} \\ \alpha, \ \ \text{otherwise} \end{cases} \\
\mathsf{putDeps}(\tau\,\mu, \mathit{tvdeps}) & = \mathsf{putDeps}(\tau, \mathit{tvdeps})\,\mu \\
\mathsf{putDeps}(\tau_1{\to}\tau_2, \mathit{tvdeps}) & = \mathsf{putDeps}(\tau_1, \mathit{tvdeps}){\to}\mathsf{putDeps}(\tau_2, \mathit{tvdeps}) \\
\mathsf{putDeps}(\tau^{\overline{d}}, \mathit{tvdeps}) & = \mathsf{putDeps}(\tau, \mathit{tvdeps})^{\overline{d}}
\end{array}
$$

Let us now present another way of constraining τ' from the one in Fig. 11.9 (different from constraining it with $\overline{d}'$). In the following, τ' and $\overline{\alpha}$ are the same as in Fig. 11.9. First, we define a variant of monos, called monos′ that gathers labels more precisely as follows:

$$\mathsf{monos}'(\mathit{\Delta}) = \{\alpha^{\overline{d}} \mid \exists \mathit{vid}.\ \tau = \mathsf{build}(\mathit{\Delta}, \mathit{\Delta}(\mathit{vid})) \wedge \alpha \in \mathsf{nonDums}(\tau) \wedge \overline{d} = \mathsf{getDeps}(\tau, \alpha, \varnothing)\}$$

Let e be the monomorphic binder $(\downarrow \mathit{vid}{=}\alpha)$, u be the set $\{\alpha \mapsto \langle \alpha_1^{\overline{d}_1} {\to} \alpha_2^{\overline{d}_2}, \overline{d}_0\rangle\}$, and $\mathit{\Delta} = \langle u,\ e\rangle$. Then, $\mathsf{monos}(\mathit{\Delta}) = \{\alpha_1^{\overline{d}_0 \cup \overline{d}_1 \cup \overline{d}_2}, \alpha_2^{\overline{d}_0 \cup \overline{d}_1 \cup \overline{d}_2}\}$, while $\mathsf{monos}'(\mathit{\Delta}) = \{\alpha_1^{\overline{d}_0 \cup \overline{d}_1}, \alpha_2^{\overline{d}_0 \cup \overline{d}_2}\}$. As a matter of fact, α_1 occurring in the monomorphic type associated with vid, does not depend on the dependency set $\overline{d}_2$ but only on the dependency set $\overline{d}_0 \cup \overline{d}_1$ (and similarly for α_2).

We can then compute the set of type variables occurring in τ' that are not allowed to be quantified over in the generated type scheme (because they occur in $\mathsf{monos}'(\mathit{\Delta})$) along with the precise reasons as why they are not allowed to be quantified over:

$$\mathit{tvdeps} = \{\alpha \mapsto \cup_{i=1}^{m} \overline{d}'_i \mid \mathsf{monos}'(\mathit{\Delta}) = \{\alpha^{\overline{d}'_1}, \dots, \alpha^{\overline{d}'_m}\} \uplus \overline{\tau} \wedge \alpha \in \mathsf{vars}(\tau') \setminus (\overline{\alpha} \cup \mathsf{vars}(\overline{\tau}))\}$$

Finally, toPoly would generate the following type scheme: $\forall \overline{\alpha}.\, \mathsf{putDeps}(\tau', \mathit{tvdeps})$.

11.6.8 Solving of the constraint generated for example (EX1)

Sec. 11.2 introduced the untypable piece of code (EX1). Let us repeat the labelled version of example (EX1) (called $\mathit{strdec}_{\mathrm{EX}}$):

$$
\begin{array}{l}
\mathtt{structure\ X} \overset{l_1}{=} \mathtt{struct}^{l_2}\\
\qquad\qquad \mathtt{structure\ S} \overset{l_3}{=} \mathtt{struct}^{l_4}\ \mathtt{datatype}\ \lceil\mathtt{'a\ u}\rceil^{l_6} \overset{l_5}{=} \mathtt{U}^{l_7}_{\mathtt{c}}\ \mathtt{end}\\
\qquad\qquad \mathtt{datatype}\ \lceil\mathtt{'a\ t}\rceil^{l_9} \overset{l_8}{=} \mathtt{T}^{l_{10}}_{\mathtt{c}}\\
\qquad\qquad \mathtt{val\ rec\ f}^{l_{12}}_{\mathtt{p}} \overset{l_{11}}{=} \mathtt{fn}\ \mathtt{T}^{l_{14}}_{\mathtt{p}} \overset{l_{13}}{\Rightarrow} \mathtt{T}^{l_{15}}_{\mathtt{e}}\\
\qquad\qquad \mathtt{val\ rec\ g}^{l_{17}}_{\mathtt{p}} \overset{l_{16}}{=} \mathtt{let}^{l_{18}}\ \mathtt{open}^{l_{19}}\ \mathtt{S\ in}\ \lceil \mathtt{f}^{l_{21}}_{\mathtt{e}}\ \mathtt{U}^{l_{22}}_{\mathtt{e}}\rceil^{l_{20}}\ \mathtt{end}\\
\qquad\quad \mathtt{end}
\end{array}
$$

Sec. 11.5.5 presented the environment, called e_{EX}, that our initial constraint generation algorithm generates given example (EX1). Let us now present the solving of e_{EX}. We present below the solving of the environment $(e_1;e_2;e_3;e_4)$ which is part of e_{EX} as defined in Sec. 11.5.5. Let us consider the solving of e_1 generated for `structure S = struct datatype 'a u = U end` Let us repeat e_1's definition:

$$
\begin{array}{rcl}
e_1 & = & [(ev_2 \overset{l_4}{=} ev_3);(ev_3{=}e_0)];(ev_1{=}(\downarrow\mathtt{S} \overset{l_3}{=} ev_2));ev_1^{l_3}\\
 & & \text{such that} \begin{cases}
e'_0 & \text{is}\ \ \mathtt{poly}(\downarrow\mathtt{U} \overset{l_7}{=} \alpha_2)\\
e''_0 & \text{is}\ \ (\alpha_1 \overset{l_6}{=} \alpha'_1\, \gamma_1);(\downarrow\mathtt{u} \overset{l_6}{=} \gamma_1);(\downarrow\mathtt{'a} \overset{l_6}{=} \alpha'_1)\\
e_0 & \text{is}\ \ (ev_4{=}((\alpha_1 \overset{l_5}{=} \alpha_2);e''_0;e'_0));ev_4^{l_5}
\end{cases}
\end{array}
$$

The solved version of e_1 is as follows:

$$ev_1^{l_3} \text{ such that } \begin{cases} ev_1 \mapsto {\downarrow}\texttt{S} \overset{l_3}{=} ev_2 \\ ev_2 \mapsto ev_3^{l_4} \\ ev_3 \mapsto (({\downarrow}\texttt{u} \overset{l_6}{=} \gamma_1);({\downarrow}\texttt{'a} \overset{l_6}{=} \alpha'_1);({\downarrow}\texttt{U} \overset{l_7}{=} \forall\{\alpha'_1\}.\,(\alpha'_1\,\gamma_1)^{\{l_5,l_6\}}))^{l_5} \end{cases}$$

Note that ev_3 is mapped to an environment that contains a binder for `'a` because we have not yet introduced any mechanism to only export partial environments (we want a mechanism other than $e_1;e_2$ that exports, e.g., the binders of e_1 but not those of e_2). This is done in Sec. 14.2 below.

Let us now consider the solving of e_2 generated for `datatype 'a t = T`. First, let us repeat e_2's definition:

$$\begin{array}{rcl} e_2 & = & (ev_5{=}((\alpha_3 \overset{l_8}{=} \alpha_4);e''_2;e'_2));ev_5^{l_8} \\ & & \text{such that } \begin{cases} e'_2 \ \text{ is } \ \texttt{poly}({\downarrow}\texttt{T} \overset{l_{10}}{=} \alpha_4) \\ e''_0 \ \text{ is } \ (\alpha_3 \overset{l_9}{=} \alpha'_3\,\gamma_2);({\downarrow}\texttt{t} \overset{l_9}{=} \gamma_2);({\downarrow}\texttt{'a} \overset{l_9}{=} \alpha'_3) \end{cases} \end{array}$$

The solved version of e_2 is a follows:

$$ev_5^{l_8} \text{ such that } ev_5 \mapsto ({\downarrow}\texttt{t} \overset{l_9}{=} \gamma_2);({\downarrow}\texttt{'a} \overset{l_9}{=} \alpha'_3);({\downarrow}\texttt{T} \overset{l_{10}}{=} \forall\{\alpha'_3\}.\,(\alpha'_3\,\gamma_2)^{\{l_8,l_9\}})$$

Let us now consider the solving of e_3 generated for `val rec f = fn T => T`. First, let us repeat e_3's definition:

$$\begin{array}{rcl} e_3 & = & (ev_6{=}\texttt{poly}(({\downarrow}\texttt{f} \overset{l_{12}}{=} \alpha_5);e'_3;(\alpha_5 \overset{l_{11}}{=} \alpha_6)));ev_6^{l_{11}} \\ & & \text{such that } e'_3 = [(ev_7{=}({\uparrow}\texttt{T} \overset{l_{14}}{=} \alpha_7));ev_7^{l_{13}};({\uparrow}\texttt{T} \overset{l_{15}}{=} \alpha_8);(\alpha_6 \overset{l_{13}}{=} \alpha_7{\rightarrow}\alpha_8)] \end{array}$$

The solved version of e_3 is a follows:

$$ev_6^{l_{11}} \text{ such that } ev_6 \mapsto ({\downarrow}\texttt{f} \overset{l_{12}}{=} \forall\{\alpha''_3,\alpha'''_3\}.\,((\alpha''_3\,\gamma_2)^{\{l_8,l_9,l_{10},l_{14}\}}{\rightarrow}(\alpha'''_3\,\gamma_2)^{\{l_8,l_9,l_{10},l_{15}\}})^{\{l_{11},l_{13}\}})$$

Note that in the binder generated at constraint solving for `f`, l_{15} only labels $(\alpha'''_3\,\gamma_2)$ and does not label the whole binder. Having dependencies on types as well as on environments allows a precise blaming (dependency tracking).

Let us present the solving of e_4 generated for `val rec g = let open S in f U end`. First, let us repeat e_4's definition:

$$\begin{array}{rcl} e_4 & = & (ev_8{=}\texttt{poly}(({\downarrow}\texttt{g} \overset{l_{17}}{=} \alpha_9);[e'_4;e''_4;(\alpha_{10} \overset{l_{18}}{=} \alpha_{11})];(\alpha_9 \overset{l_{16}}{=} \alpha_{10})));ev_8^{l_{16}} \\ & & \text{such that } \begin{cases} e'_4 \ \text{ is } \ ({\uparrow}\texttt{S} \overset{l_{19}}{=} ev_9);ev_9^{l_{19}} \\ e''_4 \ \text{ is } \ ({\uparrow}\texttt{f} \overset{l_{21}}{=} \alpha_{12});({\uparrow}\texttt{U} \overset{l_{22}}{=} \alpha_{13});(\alpha_{12} \overset{l_{20}}{=} \alpha_{13}{\rightarrow}\alpha_{11}) \end{cases} \end{array}$$

We start by solving e'_4. Its solved version is as follows:

$$ev_9^{l_{19}} \text{ such that } ev_9 \mapsto ev_2^{\{l_3,l_{19}\}}$$

Then, we solve e''_4. The dependent accessor $({\uparrow}\texttt{f} \overset{l_{21}}{=} \alpha_{12})$ accesses `f`'s binder through ev_6. It leads to the generation of the following mapping:

$$\alpha_{12} \mapsto ((\alpha''_4\,\gamma_2)^{\{l_8,l_9,l_{10},l_{14}\}}{\rightarrow}(\alpha'''_4\,\gamma_2)^{\{l_8,l_9,l_{10},l_{15}\}})^{\{l_{11},l_{12},l_{13},l_{21}\}}$$

filtering function

$$\mathsf{filt}(e^{l},\overline{l}_1,\overline{l}_2) = \begin{cases} e^{l}, & \text{if } l \in \overline{l}_1 \setminus \overline{l}_2 \\ \mathsf{dum}(e)^{\varnothing}, & \text{if } l \in \overline{l}_2 \\ \top, & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathsf{filt}(ev{=}e,\overline{l}_1,\overline{l}_2) &= (ev{=}\mathsf{filt}(e,\overline{l}_1,\overline{l}_2)) \\
\mathsf{filt}(e_1;e_2,\overline{l}_1,\overline{l}_2) &= \mathsf{filt}(e_1,\overline{l}_1,\overline{l}_2);\mathsf{filt}(e_2,\overline{l}_1,\overline{l}_2) \\
\mathsf{filt}(\mathtt{poly}(e),\overline{l}_1,\overline{l}_2) &= \mathtt{poly}(\mathsf{filt}(e,\overline{l}_1,\overline{l}_2)) \\
\mathsf{filt}(\top,\overline{l}_1,\overline{l}_2) &= \top
\end{aligned}$$

conversion of environments into dummy environments

$$\begin{aligned}
\mathsf{dum}(\downarrow id{=}x) &= (\downarrow id{=}\mathsf{toDumVar}(x)) & \mathsf{dum}(c) &= \top & \mathsf{toDumVar}(\sigma) &= \alpha_{\mathtt{dum}} \\
\mathsf{dum}(ev) &= ev_{\mathtt{dum}} & \mathsf{dum}(acc) &= \top & \mathsf{toDumVar}(\mu) &= \delta_{\mathtt{dum}} \\
\mathsf{dum}(e_1;e_2) &= \mathsf{dum}(e_1);\mathsf{dum}(e_2) & \mathsf{dum}(\top) &= \top & \mathsf{toDumVar}(e) &= ev_{\mathtt{dum}} \\
\mathsf{dum}(e^{\overline{d}}) &= \mathsf{dum}(e)
\end{aligned}$$

Figure 11.11 Constraint filtering

The dependent accessor ($\uparrow\mathtt{U} \overset{l_{22}}{=} \alpha_{13}$) accesses to `U`'s binder through ev_9, ev_2, and ev_3. It leads to the generation of the following mapping:

$$\alpha_{13} \mapsto (\alpha_1''\,\gamma_1)^{\{l_3,l_4,l_5,l_6,l_7,l_{19},l_{22}\}}$$

Finally, our constraint solver returns a type error (terminates in an error state) when dealing with the equality constraint ($\alpha_{12} \overset{l_{20}}{=} \alpha_{13} \to \alpha_{11}$), because $\gamma_1 \neq \gamma_2$. The error is as follows: $\langle \mathtt{tyConsClash}(\gamma_1,\gamma_2), \{l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11}, l_{12}, l_{13}, l_{14}, l_{19}, l_{20}, l_{21}, l_{22}\}\rangle$. We call this error er_{EX}. Let $er_{\mathrm{EX}} = \langle ek_{\mathrm{EX}}, \overline{d}_{\mathrm{EX}}\rangle$.

11.7 Minimisation and enumeration

11.7.1 Extraction of environment labels

Given an environment e, lBinds extracts the labels labelling binders occurring in e. It is used during the first phase of our minimisation algorithm which consists in trying to remove entire sections of code (datatype declarations, functions, structures, ...) by "disconnecting" accessors from their binders:

$$\mathsf{lBinds}(e) = \{l \mid bind^{l} \text{ occurs in } e\}$$

11.7.2 Constraint filtering

Fig. 11.11 defines the constraint filtering function filt, used to check the solvability of constraints in which some constraints are discarded. Note that our filtering function is not defined on all environments. The forms on which the filtering function is defined are the ones generated by our initial constraint generator (these forms are defined in Sec. 11.5.2). When applied to unlabelled equality constraints on environments, our filtering function is only applied to unlabelled equality constraints of the form $ev{=}e$ (and not of the general form $e_1{=}e_2$) because our initial constraint generator only generates variables as the left-hand-side of unlabelled equality constraints on environments (see the definition of GenEnv in Sec. 11.5.2). Similarly, we only apply our filtering function to dependent environments of the form e^l, i.e., depending on a single label. In $\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)$, $\overline{l}_1$ is the label set for which we want to keep the annotated environments (first case of the filtering rule for e^l), and $\overline{l}_2$ is the label set for which we do not want to keep the equality constraints and accessors but for which we want to turn the binders into dummy ones and keep the environment variables (second case of the filtering rule for e^l). The environments annotated by labels not in $\overline{l}_1 \cup \overline{l}_2$ are discarded (third case of the filtering rule for e^l). In the constraint filtering context, label sets are sometimes called *filters*. The distinction between binders to discard (not labelled by a label in $\overline{l}_1 \cup \overline{l}_2$) and binders to turn into dummy ones (labelled by a label in $\overline{l}_2$) is necessary because at minimisation, throwing away any environment might result in different bindings in the filtered constraints (corresponding to a different SML code). For example, removing the binder labelled by l_2 in $(\downarrow \mathtt{x} \overset{l_1}{=} \tau_1);(\downarrow \mathtt{x} \overset{l_2}{=} \tau_2);(\uparrow \mathtt{x} \overset{l}{=} \tau)$ results in `x`'s accessor being bound to `x`'s first binder instead of its second. Similarly, removing the binder labelled by `f`'s second occurrence's label in the environment generated for

```
let val rec f = fn x => x 1
in let val rec f = fn x => x + 1 in f true end
end
```

results in `f`'s third occurrence being bound to its first occurrence and so to a non-existing (false) type error to be found at enumeration. When a binder is labelled by a label from $\overline{l}_2$, it is turned into a dummy unlabelled one that cannot be involved in any error and it results that the same holds for its accessors.

The intended meaning of a labelled constraint is that it only must hold if the condition represented by the label is true. The machinery presented in this document is designed to implement this intended semantics. We therefore allow our filtering function to entirely discard labelled equality constraints, bindings, accessors and environment variables because when generated, these forms are always shallow. As a matter of fact, by definition, the right-hand-side of an accessor can only be a variable v. When generated, the right-hand-side of a binder is either a variable v or a type constructor name γ (see LabBind's definition in Sec. 11.5.2). Concerning the generated equality constraints, by shallow we mean a *lc* constraint as defined in Sec. 11.5.2. The non-shallow generated equality constraints are the non-labelled ones generated by rules (G4), (G17), (G18), (G19), (G20) and (G22). Because these constraints are not labelled, they are then never filtered out but the filtering function is recursively called on the right-hand-sides of these constraints as they can be non shallow.

11.7.3 Why is minimisation necessary?

Given an environment generated for a piece of code (given e such that $strdec \rightarrowtriangle e$ for a given $strdec$), our enumeration algorithm works as follows: it selects a filter from its search space, it filters out the constraints labelled by the filter in the environment and runs the constraint solver on the filtered environment. If the constraint solver succeeds (terminates in a success state) then the enumerator keeps searching for type errors using the rest of the search space. If the constraint solver fails (terminates in an error state) then the enumerator has found a new error. This new error might not be minimal. The enumerator runs then the minimiser on the found error and once a minimal error has been found, keeps searching for other type errors. The minimiser is necessary because when the constraint solver returns an error at enumeration, this error might not be minimal. An obvious example is as follows:

```
val rec f = fn x => (x (fn z => z), x (fn () => ()))
val rec g = fn y => y true
val u = f g
```

This piece of code is untypable and the highlighting of one of the type errors of this piece of code is as follows:

```
val rec f = fn x => (x (fn z => z), x (fn () => ()))
val rec g = fn y => y true
val u = f g
```

The corresponding type error slice is as follows (we have adapted the slice returned by Impl-TES to the restricted language presented in this document):

```
⟨..val rec f = fn x => ⟨..x (fn () => ⟨..⟩)..⟩
 ..val rec g = fn y => ⟨..y true..⟩
 ..f g..⟩
```

The issue is that because of the first component returned by the function `f` (the application `x (fn z => z)`) and because of `x`'s monomorphism, when the error presented above is first found at enumeration, it is not minimal. The error first found by the enumeration algorithm, before minimisation, is as follows:

```
⟨..val rec f = fn x => ⟨..x (fn z => ⟨..⟩)..x (fn () => ⟨..⟩)..⟩
 ..val rec g = fn y => ⟨..y true..⟩
 ..f g..⟩
```

Because `x` is monomorphic, it is constrained by both `z` and `()`. This is a typical example that shows the necessity of the minimisation algorithm. We have not yet found a way to directly obtain the first slice presented above without the help of the minimiser. The investigation of such a system is left for future work.

11.7.4 Minimisation algorithm

Fig. 11.12 defines our minimisation algorithm: the relation min that uses the relation $\rightarrow_{\mathsf{test}}$ which tests if a label can be removed from a slice and where $\rightarrow^{*}_{\mathsf{test}}$ is its reflexive (w.r.t. $\mathsf{Env} \times \mathsf{Error}$) and transitive closure. Minimisation consists of two main phases. The first one (phase1) tries to remove entire sections of code at once by turning bindings into dummy ones using lBinds (defined in Sec. 11.7.1). In a fine-grained second phase (phase2) the algorithm tries to remove the remaining labels ($\overline{l}_1$ in rule ($\mathsf{MIN3}$) in Fig. 11.12) one at a time.

A step of our minimisation algorithm is as follow: $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2 \rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_3, \overline{l}_4 \rangle$ where $\overline{l}_3$ and $\overline{l}_4$ depend on the solvability of $\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\})$. Let $e' = \mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\})$. The set $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$ is the label set of the error that the minimisation algorithm is minimising at this step, and $\{l\} \uplus \overline{l}_2$ is the label set yet to try to discard. The environment e' is obtained from e by filtering out the constraints that are not labelled by $\overline{l}_1 \cup \overline{l}_2 \cup \{l\}$, by filtering out the accessors and equality constraints that are labelled by l, and by turning the binders labelled by l into dummy ones (and similarly for the environment variables labelled by l). If e' is solvable it means that l is necessary for an error to occur, and therefore $\overline{l}_3 = \overline{l}_1 \cup \{l\}$ and $\overline{l}_4 = \overline{l}_2$. If e' is unsolvable (the solver failed and we obtained a new smaller error, i.e., which contains strictly less labels), it means that l is unnecessary for an error to occur and that any environment labelled by l can be completely filtered out in the next step. The label sets $\overline{l}_3$ and $\overline{l}_4$ are then restricted to the newly found error (see rule ($\mathsf{MIN1}$)).

Environments (bindings, environment variables, ...) can be completely filtered out from one step to another because the labelled internal syntax, our constraint generator and solver, together ensure that if a binder is turned into a dummy one then none of its accessors will be part of any error (see Sec. 11.7.6 for more on this matter). This invariant could explicitly be enforced during constraint solving by adding side conditions to rules ($\mathsf{A1}$) and ($\mathsf{A2}$) checking that the accessed identifiers' types are not dummy variables (in Dum). This enforcement is not necessary.

Note that the minimisation algorithm fails if at the end of the second phase, in rule ($\mathsf{MIN3}$), the label set $\overline{l}_2$ does not correspond to an error in e: because of $\mathsf{filt}(e, \overline{l}_2, \varnothing) \xrightarrow{\mathsf{isErr}} er'$, rule ($\mathsf{MIN3}$) is only defined if $\mathsf{filt}(e, \overline{l}_2, \varnothing)$ is unsolvable.

11.7.5 Enumeration algorithm

Enumeration states are defined as follows:

$$\mathsf{EnumState} ::= \mathtt{enum}(e) \mid \mathtt{enum}(e, \overline{er}, \overline{\overline{l}}) \mid \mathtt{errors}(\overline{er})$$

minimisation

(MIN1) $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2\rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_1 \cap \overline{d}, \overline{l}_2 \cap \overline{d}\rangle$, if $\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\}) \xrightarrow{\mathsf{isErr}} \langle ek, \overline{d}\rangle$

(MIN2) $\langle e, \overline{l}_1, \{l\} \uplus \overline{l}_2\rangle \rightarrow_{\mathsf{test}} \langle e, \overline{l}_1 \cup \{l\}, \overline{l}_2\rangle$, if $\mathsf{solvable}(\mathsf{filt}(e, \overline{l}_1 \cup \overline{l}_2, \{l\}))$

(MIN3) $\langle e, er\rangle \xrightarrow{\mathsf{min}} er'$, if $\mathsf{lBinds}(e) = \overline{l}$
$\wedge \langle e, \mathsf{labs}(er) \setminus \overline{l}, \mathsf{labs}(er) \cap \overline{l}\rangle \rightarrow^{*}_{\mathsf{test}} \langle e, \overline{l}_1, \varnothing\rangle$ (phase1)
$\wedge \langle e, \varnothing, \overline{l}_1\rangle \rightarrow^{*}_{\mathsf{test}} \langle e, \overline{l}_2, \varnothing\rangle$ (phase2)
$\wedge\, \mathsf{filt}(e, \overline{l}_2, \varnothing) \xrightarrow{\mathsf{isErr}} er'$

enumeration

(ENUM1) $\mathtt{enum}(e) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \varnothing, \{\varnothing\})$

(ENUM2) $\mathtt{enum}(e, \overline{er}, \varnothing) \rightarrow_{\mathsf{e}} \mathtt{errors}(\overline{er})$

(ENUM3) $\mathtt{enum}(e, \overline{er}, \overline{\overline{l}} \uplus \{\overline{l}\}) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \overline{er}, \overline{\overline{l}})$, if $\mathsf{solvable}(\mathsf{filt}(e, \mathsf{labs}(e), \overline{l}))$

(ENUM4) $\mathtt{enum}(e, \overline{er}, \overline{\overline{l}} \uplus \{\overline{l}\}) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \overline{er} \cup \{\langle ek, \overline{d}\rangle\}, \overline{\overline{l}}' \cup \overline{\overline{l}})$,
if $\mathsf{filt}(e, \mathsf{labs}(e), \overline{l}) \xrightarrow{\mathsf{isErr}} er$
$\wedge \langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \overline{d}\rangle$
$\wedge\, \overline{\overline{l}}' = \{\overline{l} \cup \{l\} \mid l \in \overline{d} \wedge \forall \overline{l}_0 \in \overline{\overline{l}}.\ \overline{l}_0 \not\subseteq \overline{l} \cup \{l\}\}$

Figure 11.12 Minimisation and enumeration algorithms

Fig. 11.12 also defines our enumeration algorithm: the relation $\rightarrow_{\mathsf{e}}$ where $\rightarrow^{*}_{\mathsf{e}}$ is its reflexive (w.r.t. EnumState) and transitive closure. Assume that *strdec* $\twoheadrightarrow$ e for a given piece of code *strdec*. An enumeration process always starts in a state of the form $\mathtt{enum}(e)$ and stops in a state of the form $\mathtt{errors}(\overline{er})$. Enumerating the minimal type errors in a piece of code consists of trying to solve diverse results of filtering the constraints generated for the piece of code. The tested filters (label sets) form the search space which is built while searching for errors. The enumeration algorithm starts with a single filter: the empty set, so that the constraint solver is called on all the generated constraints. Then, when an error is found and minimised, the labels of the error are used to build new filters (see $\overline{\overline{l}}'$ in rule (ENUM4)). Once the filters are exhausted the enumeration algorithm stops. The found errors are then all the minimal type errors of the analysed piece of code (see rule (ENUM2)). In an enumeration process, the second enumeration state is always (see rule (ENUM1)): $\mathtt{enum}(e, \varnothing, \{\varnothing\})$ where the first empty set is the set of found errors (empty at the beginning) and where the second empty set is the first filter. If *strdec* is untypable, the constraint solver fails and returns a type error er_1 of the form $\langle ek_1, \overline{d}_1\rangle$. The minimisation algorithm minimises er_1 and returns a minimal error er_2 of the form $\langle ek_2, \overline{d}_2\rangle$ such that $\overline{d}_2 \subseteq \overline{d}_1$. The error er_2 can be er_1 if it was already in a minimal form when found by the enumerator. New filters are computed based on the filter used to find this new error ($\varnothing$ in our example) and the new error er_2 itself: $\{\{l\} \mid l \in \overline{d}_2\}$. The enumerator keeps searching for errors using this updated search space: the new state is $\mathtt{enum}(e, \{er_2\}, \{\{l\} \mid l \in \overline{d}_2\})$. At the next step, one of the $\{l\}$ where $l \in \overline{d}_2$ will be picked as the filter to try to find another error. When a filter leads to a solvable filtered environment, the filter is discarded (rule (ENUM3)) otherwise it is used to update the search space as explained above (rule (ENUM4)).

11.7.6 Minimisation and binding discarding

Let us describe a step of the first phase of our minimisation algorithm. We test if we can remove a label l associated with a binder *bind* from the slice we want to minimise (and still obtain a type error slice) by first filtering the constraints of the original piece of code as follows: $\mathsf{filt}(e, \overline{l}, \{l\})$, to obtain e' and where e is the environment generated for the original piece of code and $\overline{l} \cup \{l\}$ is the label set labelling the slice that is being minimised. In order not to mix up the bindings, at constraint filtering, the binder *bind* associated with l is not discarded but is replaced by a non labelled dummy binder $bind'$ (such that $bind' = \mathsf{dum}(bind)$) that cannot participate to any error but that still acts as a binder. If we then solve e' and obtain an error then no label labelling in e' an accessor to $bind'$ will occur in the found error (we give below an informal argument as why none of these accessors will be part of the new error). The bindings in this new error are then not mixed up. (Note that bindings can be mixed up in a filtered environment if and only if an accessor refers to a binder to which it does not refer to in the non filtered environment.) The found error is then the new slice to try to minimise further and next time the constraints will be filtered w.r.t. this new slice, the binder *bind* and its accessors will be completely thrown away (as well as the other constraints not participating in the new error).

Let us consider the following unsolvable environment which we call e:

$$\alpha_1 \overset{l_1}{=} \mathtt{int}; \alpha_2 \overset{l_2}{=} \mathtt{unit}; \downarrow vid \overset{l_3}{=} \alpha_1; \downarrow vid \overset{l_4}{=} \alpha_2; \alpha_3 \overset{l_5}{=} \mathtt{unit}; \uparrow vid \overset{l_6}{=} \alpha_3; \alpha_1 \overset{l_7}{=} \alpha_3$$

The only labels necessary for an error to occur are l_1, l_5 and l_7. Note that *vid*'s accessor refers to *vid*'s binder labelled by l_4 (second binder) and not to the one labelled by l_3 (first binder). Let us run our minimisation algorithm on e and let the first step be to try to discard l_4. First the filtering function is called on e as follows: $\mathsf{filt}(e, \{l_1, l_2, l_3, l_5, l_6, l_7\}, \{l_4\})$, which results in the following environment, called e':

$$\alpha_1 \overset{l_1}{=} \mathtt{int}; \alpha_2 \overset{l_2}{=} \mathtt{unit}; \downarrow vid \overset{l_3}{=} \alpha_1; \downarrow vid{=}\alpha_{\mathtt{dum}}; \alpha_3 \overset{l_5}{=} \mathtt{unit}; \uparrow vid \overset{l_6}{=} \alpha_3; \alpha_1 \overset{l_7}{=} \alpha_3$$

At constraint solving, running on e', when dealing with the accessor $\uparrow vid \overset{l_6}{=} \alpha_3$, the dummy binder $\downarrow vid{=}\alpha_{\mathtt{dum}}$ is accessed and the equality constraint $\alpha_3{=}\alpha_{\mathtt{dum}}$ is generated by rule (A2). This constraint is then discarded by rule (U3) thanks to the definition of $\oplus$ and because $\alpha_{\mathtt{dum}} \in \mathsf{Dum}$. Therefore, the accessor and its label are discarded at constraint solving and cannot occur in any error. In our example, on e', the constraint solver terminates in an error state, which means that l_4 is unnecessary for an error to occur. The error returned by the constraint solver does not involve l_4 or l_6 and especially, in the next step of the minimisation process, *vid*'s accessor cannot refer to *vid*'s first binder.

Note that filtering itself does not prevent bindings to get mixed up because, e.g., filtering allows one to throw away the binder generated for the second occurrence of `x` in `fn x => fn x => x` while not throwing away the binder generated for the first occurrence of `x` and not throwing away its accessor. However, we give below an informal argument as why we never filter a binder without filtering its accessor.

Let us now present an informal argument as why when our constraint solver returns an error, the error does not involve accessors to dummy binders or accessors without their corresponding binders.

According to rules (A1)-(A3), during constraint solving the label labelling an accessor only gets recorded in a constraint solving context $\mathit{\Delta}$ of the form $\langle u, e\rangle$ if the accessed identifier is in the type environment e stored in $\mathit{\Delta}$ in the current state (the state in which the constraint solving process is when the rule applies). There are two possible scenarios. In the environment e (1) either the accessed identifier has a dummy static semantics (resulting from filtering) and then, according to rules (U3) and (U4), the label of the accessor does not get recorded into the constraint solving context $\mathit{\Delta}$. In more details, given an accessor $\uparrow id{=}v$, according to rules (A1) and (A2), a constraint of the form $v{=}v'$ is generated, where $v' \in \mathsf{Dum}$ comes from the accessed id binder. Then (U3) or (U4) applies and the newly generated constraint is discarded without generating anything. (2) Or the accessed identifier has a labelled non-dummy static semantics, and the labels associated with the binder and the label associated with the bound occurrence will always occur together in the constraint solving context. The main point being that in our system if a binder is not a dummy binder then it is labelled.

This is why we strongly believe that an identifier occurring at a non-binding position in a piece of code (represented by an accessor in our constraint language) only occurs in a slice if it is bound and its binder occurs in the slice as well.

This argumentation relies on the fact that our labelled external syntax together with our initial constraint generation algorithm enforce that each bound occurrence of an identifier is labelled by a unique label that does not label a larger piece of code and therefore the label labelling an accessor does not label any other constraint term (see principle (DP6) presented in Sec. 11.10). Therefore in case (1) described above, once the accessor and the generated equality constraint have been dealt with and discarded, the label labelling the accessor does not occur anymore in the state in which the constraint solver is. This label cannot then be part of any error. Note that this would not necessarily be the case with an initial constraint generation rule that would generate $\langle\alpha, ((\alpha \overset{l}{=} \alpha_1{\to}\alpha_2);(\uparrow id \overset{l}{=} \alpha_1))\rangle$ for some term. As a matter of fact, we could imagine a scenario where α is further constrained to, e.g., `int`. We would therefore obtain a type constructor clash (between `int` and the arrow type constructor) that involves l but that does not require the accessor to be resolved. The accessor being kept alive in this error, at the next step of the minimisation algorithm, we would have no guarantee that it does not refer to a different binder from the one it refers to (if referring to any) in the non filtered environment.

Thanks to the invariant that if a binder is filtered out then its bound occurrences are also filtered out, we can then easily compute free identifiers thanks to rule (A3) which is the rule for an accessor for which no binder exists in the current constraint solving context (i.e., for free identifiers) or for which the binder is hidden.

11.7.7 Discussion of the search space used by our enumerator

The search space used by our enumeration algorithm is a set of filters (where a filter is a label set). For example, given an environment e, if using[2] the filter $\overline{l}$ the enumeration algorithm finds a minimal error labelled by the set $\{l_1, l_2\}$ then to search for other type errors, it generates the two filters $\overline{l} \cup \{l_1\}$ and $\overline{l} \cup \{l_2\}$ (if no smaller filter is already in the search space). As a matter of fact, if another error (different from the one labelled by $\{l_1, l_2\}$) can be found in $\mathsf{filt}(e, \mathsf{labs}(e), \overline{l})$ then this other error cannot be labelled by both l_1 and l_2, otherwise it has to be the minimal type error $\{l_1, l_2\}$. So we want to search for errors that are not labelled by $\overline{l} \cup \{l_1\}$ and for errors that are not labelled by $\overline{l} \cup \{l_2\}$ (this allows one to obtain a correct and terminating enumeration algorithm).

A particularity of the enumeration algorithm presented in Sec. 11.7.5 is that the search space stays "relatively" small. However, because of the strategy used by the enumeration algorithm to build new filters, it can happen that the same error is generated twice (using two different filters). Note that even though an error can be generated twice using the algorithm presented in Sec. 11.7.5, this cannot happen using Impl-TES' enumeration algorithm which differs as follows: before using a filter $\overline{l}$, Impl-TES' enumeration algorithm checks whether it has already found an error *er* using a filter at least as big as $\overline{l}$ (superset of $\overline{l}$), and if it did it does not use $\overline{l}$ (does not run the constraint solver) again but instead directly generates new filters because *er* can also be found using $\overline{l}$.

Note that even though the enumeration algorithm presented in Sec. 11.7.5 can enumerate an error twice (using two different filters), it terminates because no filter can be generated twice. (We strongly believe that the termination of our algorithm follows from the one of HW-TES' algorithm [57].)

Let us explain why the enumeration algorithm presented in Sec. 11.7.5 can generate an error twice. We describe a highly possible scenario. Assume that e has been generated by our initial constraint generator for the structure declaration *strdec*, i.e., *strdec* $\Rightarrow$ e. Then, the enumeration algorithm starts with the following transition:

$$(\mathsf{TR1}) \qquad \mathtt{enum}(e) \rightarrow \mathtt{enum}(e, \varnothing, \{\varnothing\})$$

[2] Given an environment e, by using a filter $\overline{l}$ we mean running the constraint solver on $\mathsf{filt}(e, \mathsf{labs}(e), \overline{l})$ which is the environment e where, among other things, the equality constraints labelled by $\overline{l}$ are filtered out.

using rule (ENUM1) (see Fig. 11.12), and where the first $\varnothing$ is the set of found errors and $\{\varnothing\}$ is the initial search space which only contains the empty filter $\varnothing$ at the beginning of the computation.

Let us now assume that the first found minimal error (using rule (ENUM4) in Fig. 11.12) is labelled by the label set $\{l_1, l_4\}$ ($\overline{d} = \{l_1, l_4\}$ in rule (ENUM4) in Fig. 11.12). The enumeration algorithm generates then the filters $\{l_1\}$ (which is the union of the filter $\varnothing$ and the set $\{l_1\}$) and $\{l_4\}$ (which is the union of the filter $\varnothing$ and the set $\{l_4\}$). We obtain the following transition:

$$\text{(TR2)} \qquad \texttt{enum}(e, \varnothing, \{\varnothing\}) \rightarrow \texttt{enum}(e, \{\langle ek_1, \{l_1, l_4\}\rangle\}, \{\{l_1\}, \{l_4\}\})$$

Using the filter $\{l_1\}$ let us assume that the enumeration algorithm finds an other minimal error labelled by the set $\{l_2, l_3\}$. From the filter $\{l_1\}$, the following filters are then generated: $\{l_1, l_2\}$ and $\{l_1, l_3\}$. The search space (set of filters yet to try) is now $\{\{l_1, l_2\}, \{l_1, l_3\}, \{l_4\}\}$. At this stage, the minimal type error set is $\{\{l_1, l_4\}, \{l_2, l_3\}\}$. We obtain the following transition:

$$\text{(TR3)} \qquad \begin{array}{l} \texttt{enum}(e, \{\langle ek_1, \{l_1, l_4\}\rangle\}, \{\{l_1\}, \{l_4\}\}) \\ \rightarrow \\ \texttt{enum}(e, \{\langle ek_1, \{l_1, l_4\}\rangle, \langle ek_2, \{l_2, l_3\}\rangle\}, \{\{l_1, l_2\}, \{l_1, l_3\}, \{l_4\}\}) \end{array}$$

Let us assume now that using the filter $\{l_1, l_2\}$ the enumeration algorithm finds an error labelled by the set $\{l_4, l_5\}$. The enumeration algorithm generates from the filter $\{l_1, l_2\}$, the two following filters: $\{l_1, l_2, l_4\}$ which is immediately discarded because it is a superset of the filter $\{l_4\}$ which is already in our search space, and $\{l_1, l_2, l_5\}$ which is not discarded and then added to the search space. The search space is then $\{\{l_1, l_2, l_5\}, \{l_1, l_3\}, \{l_4\}\}$. At this stage, the minimal type error set is $\{\{l_1, l_4\}, \{l_2, l_3\}, \{l_4, l_5\}\}$. We obtain the following transition:

$$\text{(TR4)} \quad \begin{array}{l} \texttt{enum}(e, \{er_1, er_2\}, \{\{l_1, l_2\}, \{l_1, l_3\}, \{l_4\}\}) \\ \rightarrow \\ \texttt{enum}(e, \{er_1, er_2, er_3\}, \{\{l_1, l_2, l_5\}, \{l_1, l_3\}, \{l_4\}\}) \end{array} \quad \text{where} \begin{cases} er_1 = \langle ek_1, \{l_1, l_4\}\rangle \\ er_2 = \langle ek_2, \{l_2, l_3\}\rangle \\ er_3 = \langle ek_3, \{l_4, l_5\}\rangle \end{cases}$$

Let us assume that the enumeration algorithm does not generate any error with the filter $\{l_1, l_2, l_5\}$. It is then discarded. Assume that the enumeration algorithm uses then the filter $\{l_1, l_3\}$. The enumeration algorithm has already found an error that can be obtained using this filter: er_3 which is labelled by $\{l_4, l_5\}$, and might then generate this error a second time. If it does, we obtain the following transitions:

$$\text{(TR5)} \quad \begin{array}{l} \texttt{enum}(e, \{er_1, er_2, er_3\}, \{\{l_1, l_2, l_5\}, \{l_1, l_3\}, \{l_4\}\}) \\ \rightarrow \\ \texttt{enum}(e, \{er_1, er_2, er_3\}, \{\{l_1, l_3\}, \{l_4\}\}) \\ \rightarrow \\ \texttt{enum}(e, \{er_1, er_2, er_3\}, \{\{l_1, l_3, l_4\}, \{l_1, l_3, l_5\}, \{l_4\}\}) \end{array} \quad \text{where} \begin{cases} er_1 = \langle ek_1, \{l_1, l_4\}\rangle \\ er_2 = \langle ek_2, \{l_2, l_3\}\rangle \\ er_3 = \langle ek_3, \{l_4, l_5\}\rangle \end{cases}$$

$$
\begin{aligned}
&(\mathsf{ENUM4})\ \mathtt{enum}(e, \overline{er}, \overline{\overline{l}} \uplus \{\overline{l}\}) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \overline{er} \cup \{\langle ek, \overline{d}\rangle\}, \overline{\overline{l}}' \cup \overline{\overline{l}}),\\
&\qquad \text{if } (((\langle ek, \overline{d}\rangle \in \overline{er} \wedge \mathsf{dj}(\overline{l}, \overline{d}))\\
&\qquad\quad \vee ((\forall \langle ek_0, \overline{d}_0\rangle \in \overline{er}.\ \neg\mathsf{dj}(\overline{l}, \overline{d}_0)) \wedge \mathsf{filt}(e, \mathsf{labs}(e), \overline{l}) \xrightarrow{\mathsf{isErr}} er \wedge \langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \overline{d}\rangle))\\
&\qquad \wedge \overline{\overline{l}}' = \{\overline{l} \cup \{l\} \mid l \in \overline{d} \wedge \forall \overline{l}_0 \in \overline{\overline{l}}.\ \overline{l}_0 \not\subseteq \overline{l} \cup \{l\}\}
\end{aligned}
$$

(a) Enumeration algorithm of Impl-TES

$$
\begin{aligned}
&(\mathsf{ENUM4})\ \mathtt{enum}(e, \overline{er}, \overline{\overline{l}} \uplus \{\overline{l}\}) \rightarrow_{\mathsf{e}} \mathtt{enum}(e, \overline{er} \cup \{\langle ek, \overline{d}\rangle\}, \overline{\overline{l}}_1 \cup \overline{\overline{l}}_2 \cup \overline{\overline{l}}_3),\\
&\qquad \text{if } \mathsf{filt}(e, \mathsf{labs}(e), \overline{l}) \xrightarrow{\mathsf{isErr}} er \wedge \langle e, er\rangle \xrightarrow{\mathsf{min}} \langle ek, \overline{d}\rangle\\
&\qquad \wedge \overline{\overline{l}}_1 = \{\overline{l} \cup \{l\} \mid l \in \overline{d} \wedge \forall \overline{l}_0 \in \overline{\overline{l}}.\ \overline{l}_0 \not\subseteq \overline{l} \cup \{l\}\}\\
&\qquad \wedge \overline{\overline{l}}_2 = \{\overline{l}_0 \cup \{l\} \mid l \in \overline{d} \wedge \overline{l}_0 \in \overline{\overline{l}} \wedge \mathsf{dj}(\overline{l}_0, \overline{d})\}\\
&\qquad \wedge \overline{\overline{l}}_3 = \{\overline{l}_0 \mid \overline{l}_0 \in \overline{\overline{l}} \wedge \neg\mathsf{dj}(\overline{l}_0, \overline{d})\}
\end{aligned}
$$

(b) Variant to generate each error exactly once

Figure 11.13 Variants of our enumeration algorithm

Now, let us present an alternative strategy to generate new filters. In transition (TR3) above, instead of only generating the filters $\{l_1, l_2\}$ and $\{l_1, l_3\}$ from the filter $\{l_1\}$ and the error $\{l_2, l_3\}$, we also could generate the extra filters $\{l_4, l_2\}$ and $\{l_4, l_3\}$ (and remove $\{l_4\}$ from the search space) because $\{l_4\}$ is a filter which is yet to try (is in the search space) and which is disjoint from the error $\{l_2, l_3\}$ (the error $\{l_2, l_3\}$ can be found using the filter $\{l_4\}$). Then, as before when using the filter $\{l_1, l_2\}$ (see transition (TR4) above), this variant of our enumeration algorithm finds an error labelled by the set $\{l_4, l_5\}$. As before, the filter $\{l_1, l_2, l_5\}$ is generated and we also replace the filter $\{l_1, l_3\}$ by the filter $\{l_1, l_3, l_5\}$ because the filter $\{l_1, l_3\}$ and the error $\{l_4, l_5\}$ are disjoint (we do not generate the filter $\{l_1, l_3, l_4\}$ because it is a superset of the already existing filter $\{l_3, l_4\}$). The error labelled by $\{l_4, l_5\}$ is then not going to be found again. We would then, instead of the transitions as described above, obtain the following transitions (transitions (TR1) and (TR2) stay the same):

$$
\begin{array}{ll}
\mathtt{enum}(e, \{er_1\}, \{\{l_1\}, \{l_4\}\}) & \\
\rightarrow & \\
\mathtt{enum}(e, \{er_1, er_2\}, \{\{l_1, l_2\}, \{l_1, l_3\}, \{l_4, l_2\}, \{l_4, l_3\}\}) & \text{where } \begin{cases} er_1 = \langle ek_1, \{l_1, l_4\}\rangle \\ er_2 = \langle ek_2, \{l_2, l_3\}\rangle \\ er_3 = \langle ek_3, \{l_4, l_5\}\rangle \end{cases} \\
\rightarrow & \\
\mathtt{enum}(e, \{er_1, er_2, er_3\}, \{\{l_1, l_2, l_5\}, \{l_1, l_3, l_5\}, \{l_4, l_2\}, \{l_4, l_3\}\}) &
\end{array}
$$

Finally, let us formally present two alternatives of the enumeration algorithm presented in Fig. 11.12. We only present variants of rule (ENUM4) because the other rules stay unchanged. Fig. 11.13a presents a first variant which is used by Impl-TES, and Fig. 11.13b presents a second variant which is the one described above.

11.7.8 Enumerating all the errors in example (EX1)

First, let us repeat the labelled version of example (EX1) defined in Sec. 11.2:

$$\begin{array}{l}
\mathtt{structure\ X} \overset{l_1}{=} \mathtt{struct}^{l_2}\\
\qquad\qquad \mathtt{structure\ S} \overset{l_3}{=} \mathtt{struct}^{l_4}\ \mathtt{datatype}\ \lceil\mathtt{'a\ u}\rceil^{l_6} \overset{l_5}{=} \mathtt{U}_{\mathtt{c}}^{l_7}\ \mathtt{end}\\
\qquad\qquad \mathtt{datatype}\ \lceil\mathtt{'a\ t}\rceil^{l_9} \overset{l_8}{=} \mathtt{T}_{\mathtt{c}}^{l_{10}}\\
\qquad\qquad \mathtt{val\ rec\ f}_{\mathtt{p}}^{l_{12}} \overset{l_{11}}{=} \mathtt{fn}\ \mathtt{T}_{\mathtt{p}}^{l_{14}} \overset{l_{13}}{\Rightarrow} \mathtt{T}_{\mathtt{e}}^{l_{15}}\\
\qquad\qquad \mathtt{val\ rec\ g}_{\mathtt{p}}^{l_{17}} \overset{l_{16}}{=} \mathtt{let}^{l_{18}}\ \mathtt{open}^{l_{19}}\ \mathtt{S\ in}\ \lceil\mathtt{f}_{\mathtt{e}}^{l_{21}}\ \mathtt{U}_{\mathtt{e}}^{l_{22}}\rceil^{l_{20}}\ \mathtt{end}\\
\qquad \mathtt{end}
\end{array}$$

It turns out that example (EX1) has only one minimal type error which is er_{EX} defined in Sec. 11.6.8 as the pair $\langle ek_{\mathrm{EX}}, \overline{d}_{\mathrm{EX}}\rangle$ where $\overline{d}_{\mathrm{EX}}$ is the following set:

$$\{l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11}, l_{12}, l_{13}, l_{14}, l_{19}, l_{20}, l_{21}, l_{22}\}$$

This error is already minimal when found by the enumeration algorithm and therefore the minimiser does not do anything in this case, but is still called by the enumerator. Therefore we obtain the following enumeration steps (we superscript $\rightarrow_{\mathsf{e}}$ and $\rightarrow_{\mathsf{e}}^{*}$ with the names of the rules used to obtain the provided enumeration steps):

$$\begin{array}{ll}
 & \mathtt{enum}(e_{\mathrm{EX}})\\
\rightarrow_{\mathsf{e}}^{(\mathsf{ENUM1})} & \mathtt{enum}(e_{\mathrm{EX}}, \varnothing, \{\varnothing\})\\
\rightarrow_{\mathsf{e}}^{(\mathsf{ENUM4})} & \mathtt{enum}(e_{\mathrm{EX}}, \{er_{\mathrm{EX}}\}, \{\{l\} \mid l \in \overline{d}_{\mathrm{EX}}\})\\
\rightarrow_{\mathsf{e}}^{*(\mathsf{ENUM3})} & \mathtt{enum}(e_{\mathrm{EX}}, \{er_{\mathrm{EX}}\}, \varnothing)\\
\rightarrow_{\mathsf{e}}^{(\mathsf{ENUM2})} & \mathtt{errors}(\{er_{\mathrm{EX}}\})
\end{array}$$

11.8 Slicing

11.8.1 Dot terms

Our TES' last phase consists of computing minimal type error slices from untypable pieces of code and minimal errors found by the enumerator. This is performed by the slicing function sl (defined below in Fig. 11.17). The nodes labelled by the labels not involved in the error are discarded and replaced by "dot" terms. For example, if we remove the node associated with the label l_2 (the unit expression) in $\lceil 1^{l_1}\ ()^{l_2}\rceil^{l_3}$ then we obtain $\lceil 1^{l_1}\ \mathtt{dot\text{-}e}(\varnothing)\rceil^{l_3}$, displayed as `1 ⟨..⟩` in our implementation. Dots are visually convenient to show that information has been discarded. Fig. 11.14 extends our syntax and constraint generator to dot terms. Our constraint generator is extended to dot terms so that every piece of our extended syntax can be type checked (by generating constraints and by then solving the constraints), which is needed to define type error slices and to state our minimality criteria in Sec. 11.9. We call *slice*, any syntactic form that can be produced using the grammar rules defined in Fig. 11.2 and Fig. 11.14 combined (i.e., a *term* as defined in Fig. 11.2). We call *type error slice*, any slice for which our constraint generation algorithm (defined in Fig. 11.7 and Fig. 11.14 combined) only generates unsolvable constraints. If we

extension of the syntax

$$\begin{array}{rlrlrl}
\mathsf{LabTyCon} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{DatName} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{AtPat} ::= \cdots \mid \texttt{dot-p}(\overrightarrow{pat}) & \\
\mathsf{LabDatCon} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{Dec} ::= \cdots \mid \texttt{dot-d}(\overrightarrow{term}) & & \mathsf{Pat} ::= \cdots \mid \texttt{dot-p}(\overrightarrow{pat}) & \\
\mathsf{Ty} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{AtExp} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{StrDec} ::= \cdots \mid \texttt{dot-d}(\overrightarrow{term}) & \\
\mathsf{ConBind} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{Exp} ::= \cdots \mid \texttt{dot-e}(\overrightarrow{term}) & & \mathsf{StrExp} ::= \cdots \mid \texttt{dot-s}(\overrightarrow{term}) &
\end{array}$$

extension of the constraint generator

$(\mathsf{G23})\ ept \mathrel{-\!\triangleright} e \Leftarrow ept \mathrel{-\!\triangleright} \langle v,\ e\rangle$

$(\mathsf{G24})\ \texttt{dot-d}(\langle term_1,\ldots,term_n\rangle) \mathrel{-\!\triangleright} [e_1;\cdots;e_n] \Leftarrow term_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge term_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n)$

$(\mathsf{G25})\ \texttt{dot-p}(\langle pat_1,\ldots,pat_n\rangle) \mathrel{-\!\triangleright} \langle \alpha,\ e_1;\cdots;e_n\rangle \Leftarrow pat_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge pat_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,\alpha)$

$(\mathsf{G26})\ \texttt{dot-s}(\langle term_1,\ldots,term_n\rangle) \mathrel{-\!\triangleright} \langle ev,\ [e_1;\cdots;e_n]\rangle$
$\quad\Leftarrow term_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge term_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,ev)$

$(\mathsf{G27})\ \texttt{dot-e}(\langle term_1,\ldots,term_n\rangle) \mathrel{-\!\triangleright} \langle \alpha,\ [e_1;\cdots;e_n]\rangle$
$\quad\Leftarrow term_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge term_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,\alpha)$

Figure 11.14 Extension of our syntax and constraint generator to "dot" terms

restrict ourselves to structure declarations, formally a slice is a *strdec* and a type error slice is a *strdec* such that $\neg\mathsf{solvable}(strdec)$.

11.8.2 Remark about the constraint generation rules for dot terms

Fig. 11.14 presents constraint generation rules for the different dot terms of our syntax. Rules (G24), (G26), and (G27) all wrap the environments generated for the *term*s wrapped into the dot constructors, into a local environment not visible from the outside of the form $[e]$. Rule (G25) for dot patterns stands out by not generating an environment of the form $[e]$. As of matter of fact a dot pattern constructor as a different meaning as the other dot constructors. Such a dot pattern term means that information has been sliced away but that the remaining information might still be in a pattern at a binding position. Such a pattern dot term does not define a local scope as the other dot terms do.

11.8.3 Alternative definition of the labelled external syntax

We will now provide an alternative generic definition of the external labelled syntax presented in Fig. 11.2. This definition helps defining our slicing algorithm in a compact way. First, Fig. 11.15 defines our labelled abstract syntax trees. A node in a tree *tree* can either be a labelled node of the form $\langle node, l, \overrightarrow{tree}\rangle$, an unlabelled "dot" node of the form $\langle dot, \overrightarrow{tree}\rangle$, or a leaf of the form *id*.

Fig. 11.16 defines the function toTree which associates a *tree* with each *term* (defined in Fig. 11.2). We also define toTree on sequences of *term*s.

The function getDot generates dot markers (terms in Dot) from nodes as follows:

$$
\begin{array}{lll}
\mathit{class} \in \mathsf{Class} & ::= & \mathtt{lTc} \mid \mathtt{lDcon} \mid \mathtt{ty} \mid \mathtt{conbind} \mid \mathtt{datname} \\
 & & \mid \mathtt{dec} \mid \mathtt{atexp} \mid \mathtt{exp} \\
 & & \mid \mathtt{atpat} \mid \mathtt{pat} \mid \mathtt{strdec} \mid \mathtt{strexp} \\
\mathit{prod} \in \mathsf{Prod} & ::= & \mathtt{tyArr} \mid \mathtt{tyCon} \\
 & & \mid \mathtt{conbindOf} \mid \mathtt{datnameCon} \\
 & & \mid \mathtt{decRec} \mid \mathtt{decDat} \mid \mathtt{decOpn} \\
 & & \mid \mathtt{atexpLet} \mid \mathtt{expFn} \\
 & & \mid \mathtt{strdecDec} \mid \mathtt{strdecStr} \\
 & & \mid \mathtt{strexpSt} \\
 & & \mid \mathtt{id} \mid \mathtt{app} \mid \mathtt{seq}
\end{array}
\qquad
\begin{array}{lll}
\mathit{dot} \in \mathsf{Dot} & ::= & \mathtt{dotE} \mid \mathtt{dotP} \\
 & & \mid \mathtt{dotD} \mid \mathtt{dotS} \\
\mathit{node} \in \mathsf{Node} & ::= & \langle \mathit{class}, \mathit{prod} \rangle \\
\mathit{tree} \in \mathsf{Tree} & ::= & \langle \mathit{node}, l, \overrightarrow{\mathit{tree}} \rangle \\
 & & \mid \langle \mathit{dot}, \overrightarrow{\mathit{tree}} \rangle \\
 & & \mid \mathit{id}
\end{array}
$$

Figure 11.15 Labelled abstract syntax trees

Labelled type constructors

$$\mathsf{toTree}(\mathit{tc}^{l}) = \langle\langle \mathtt{lTc}, \mathtt{id}\rangle, l, \langle \mathit{tc}\rangle\rangle$$

Labelled datatype constructors

$$\mathsf{toTree}(\mathit{dcon}^{l}) = \langle\langle \mathtt{lDcon}, \mathtt{id}\rangle, l, \langle \mathit{dcon}\rangle\rangle$$

Types

$$
\begin{aligned}
\mathsf{toTree}(\mathit{tv}^{l}) &= \langle\langle \mathtt{ty}, \mathtt{id}\rangle, l, \langle \mathit{tv}\rangle\rangle \\
\mathsf{toTree}(\mathit{ty}_1 \overset{l}{\rightarrow} \mathit{ty}_2) &= \langle\langle \mathtt{ty}, \mathtt{tyArr}\rangle, l, \langle \mathsf{toTree}(\mathit{ty}_1), \mathsf{toTree}(\mathit{ty}_2)\rangle\rangle \\
\mathsf{toTree}(\lceil \mathit{ty}\ \mathit{ltc}\rceil^{l}) &= \langle\langle \mathtt{ty}, \mathtt{tyCon}\rangle, l, \langle \mathsf{toTree}(\mathit{ty}), \mathsf{toTree}(\mathit{ltc})\rangle\rangle
\end{aligned}
$$

Constructor bindings

$$
\begin{aligned}
\mathsf{toTree}(\mathit{dcon}^{l}_{\mathtt{c}}) &= \langle\langle \mathtt{conbind}, \mathtt{id}\rangle, l, \langle \mathit{dcon}\rangle\rangle \\
\mathsf{toTree}(\mathit{dcon}\ \mathtt{of}^{\,l}\ \mathit{ty}) &= \langle\langle \mathtt{conbind}, \mathtt{conbindOf}\rangle, l, \langle \mathit{dcon}, \mathsf{toTree}(\mathit{ty})\rangle\rangle
\end{aligned}
$$

Datatype names

$$\mathsf{toTree}(\lceil \mathit{tv}\ \mathit{tc}\rceil^{l}) = \langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l, \langle \mathit{tv}, \mathit{tc}\rangle\rangle$$

Declarations

$$
\begin{aligned}
\mathsf{toTree}(\mathtt{val\ rec}\ \mathit{pat} \overset{l}{=} \mathit{exp}) &= \langle\langle \mathtt{dec}, \mathtt{decRec}\rangle, l, \langle \mathsf{toTree}(\mathit{pat}), \mathsf{toTree}(\mathit{exp})\rangle\rangle \\
\mathsf{toTree}(\mathtt{datatype}\ \mathit{dn} \overset{l}{=} \mathit{cb}) &= \langle\langle \mathtt{dec}, \mathtt{decDat}\rangle, l, \langle \mathsf{toTree}(\mathit{dn}), \mathsf{toTree}(\mathit{cb})\rangle\rangle \\
\mathsf{toTree}(\mathtt{open}^{l}\ \mathit{strid}) &= \langle\langle \mathtt{dec}, \mathtt{decOpn}\rangle, l, \langle \mathit{strid}\rangle\rangle
\end{aligned}
$$

Expressions

$$
\begin{aligned}
\mathsf{toTree}(\mathit{vid}^{l}_{\mathtt{e}}) &= \langle\langle \mathtt{atexp}, \mathtt{id}\rangle, l, \langle \mathit{vid}\rangle\rangle \\
\mathsf{toTree}(\mathtt{let}^{l}\ \mathit{dec}\ \mathtt{in}\ \mathit{exp}\ \mathtt{end}) &= \langle\langle \mathtt{atexp}, \mathtt{atexpLet}\rangle, l, \langle \mathsf{toTree}(\mathit{dec}), \mathsf{toTree}(\mathit{exp})\rangle\rangle \\
\mathsf{toTree}(\mathtt{fn}\ \mathit{pat} \overset{l}{\Rightarrow} \mathit{exp}) &= \langle\langle \mathtt{exp}, \mathtt{expFn}\rangle, l, \langle \mathsf{toTree}(\mathit{pat}), \mathsf{toTree}(\mathit{exp})\rangle\rangle \\
\mathsf{toTree}(\lceil \mathit{exp}\ \mathit{atexp}\rceil^{l}) &= \langle\langle \mathtt{exp}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(\mathit{exp}), \mathsf{toTree}(\mathit{atexp})\rangle\rangle
\end{aligned}
$$

Patterns

$$
\begin{aligned}
\mathsf{toTree}(\mathit{vid}^{l}_{\mathtt{p}}) &= \langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l, \langle \mathit{vid}\rangle\rangle \\
\mathsf{toTree}(\lceil \mathit{ldcon}\ \mathit{atpat}\rceil^{l}) &= \langle\langle \mathtt{pat}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(\mathit{ldcon}), \mathsf{toTree}(\mathit{atpat})\rangle\rangle
\end{aligned}
$$

Structure declarations

$$\mathsf{toTree}(\mathtt{structure}\ \mathit{strid} \overset{l}{=} \mathit{strexp}) = \langle\langle \mathtt{strdec}, \mathtt{strdecStr}\rangle, l, \langle \mathit{strid}, \mathsf{toTree}(\mathit{strexp})\rangle\rangle$$

Structure expressions

$$
\begin{aligned}
\mathsf{toTree}(\mathit{strid}^{l}) &= \langle\langle \mathtt{strexp}, \mathtt{id}\rangle, l, \langle \mathit{strid}\rangle\rangle \\
\mathsf{toTree}(\mathtt{struct}^{l}\ \mathit{strdec}_1 \cdots \mathit{strdec}_n\ \mathtt{end}) &= \langle\langle \mathtt{strexp}, \mathtt{strexpSt}\rangle, l, \mathsf{toTree}(\langle \mathit{strdec}_1, \ldots, \mathit{strdec}_n\rangle)\rangle
\end{aligned}
$$

Term sequences

$$\mathsf{toTree}(\langle \mathit{term}_1, \ldots, \mathit{term}_n\rangle) = \langle \mathsf{toTree}(\mathit{term}_1), \ldots, \mathsf{toTree}(\mathit{term}_n)\rangle$$

Dot terms

$$
\begin{aligned}
\mathsf{toTree}(\mathsf{dot\text{-}e}(\overrightarrow{\mathit{term}})) &= \langle \mathtt{dotE}, \mathsf{toTree}(\overrightarrow{\mathit{term}})\rangle \\
\mathsf{toTree}(\mathsf{dot\text{-}d}(\overrightarrow{\mathit{term}})) &= \langle \mathtt{dotD}, \mathsf{toTree}(\overrightarrow{\mathit{term}})\rangle \\
\mathsf{toTree}(\mathsf{dot\text{-}p}(\overrightarrow{\mathit{pat}})) &= \langle \mathtt{dotP}, \mathsf{toTree}(\overrightarrow{\mathit{pat}})\rangle \\
\mathsf{toTree}(\mathsf{dot\text{-}s}(\overrightarrow{\mathit{term}})) &= \langle \mathtt{dotS}, \mathsf{toTree}(\overrightarrow{\mathit{term}})\rangle
\end{aligned}
$$

Figure 11.16 From terms to trees

$$
\begin{array}{ll@{\qquad}ll}
\mathsf{getDot}(\langle \mathtt{lTc}, \mathit{prod}\rangle) & = \mathtt{dotE} & \mathsf{getDot}(\langle \mathtt{atexp}, \mathit{prod}\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{lDcon}, \mathit{prod}\rangle) & = \mathtt{dotE} & \mathsf{getDot}(\langle \mathtt{exp}, \mathit{prod}\rangle) & = \mathtt{dotE} \\
\mathsf{getDot}(\langle \mathtt{ty}, \mathit{prod}\rangle) & = \mathtt{dotE} & \mathsf{getDot}(\langle \mathtt{atpat}, \mathit{prod}\rangle) & = \mathtt{dotP} \\
\mathsf{getDot}(\langle \mathtt{conbind}, \mathit{prod}\rangle) & = \mathtt{dotE} & \mathsf{getDot}(\langle \mathtt{pat}, \mathit{prod}\rangle) & = \mathtt{dotP} \\
\mathsf{getDot}(\langle \mathtt{datname}, \mathit{prod}\rangle) & = \mathtt{dotE} & \mathsf{getDot}(\langle \mathtt{strdec}, \mathit{prod}\rangle) & = \mathtt{dotD} \\
\mathsf{getDot}(\langle \mathtt{dec}, \mathit{prod}\rangle) & = \mathtt{dotD} & \mathsf{getDot}(\langle \mathtt{strexp}, \mathit{prod}\rangle) & = \mathtt{dotS}
\end{array}
$$

This function is, among other things, used by rule (SL1) of our slicing algorithm defined below in Fig. 11.17 to generate dot nodes from labelled nodes.

11.8.4 Tidying

In addition to turning nodes not participating in type errors into dot nodes, our slicing algorithm uses two tidying functions flat and tidy. The flattening function flat flattens sequences of terms (*term*). For example, flattening $\langle\mathtt{..1..}\langle\mathtt{..()..}\rangle\mathtt{..}\rangle$ results in $\langle\mathtt{..1..()..}\rangle$. Not all nested dot terms are flattened. In order not to mix up bindings in a slice, we do not let declarations escape dot terms. For example, we do not flatten $\langle\mathtt{..val\ x\ =\ false..}\langle\mathtt{..val\ x\ =\ 1..}\rangle\mathtt{..x\ +\ 1..}\rangle$ to $\langle\mathtt{..val\ x\ =\ false..val\ x\ =\ 1..x\ +\ 1..}\rangle$ because they have different semantics. The first slice is not typable but the second is. In the first slice x's last occurrence is bound to x's first occurrence while in the second slice x's last occurrence is bound to x's second occurrence.

Let $\mathsf{isClass}(\mathit{tree}, \{\mathit{class}\} \cup \overline{\mathit{class}})$ be true iff $\mathit{tree} = \langle\langle \mathit{class}, \mathit{prod}\rangle, l, \overrightarrow{\mathit{tree}}\rangle$. This predicate is used to check the class of the root node of a tree. Let $\mathsf{declares}(\mathit{tree})$ be true iff $\mathsf{isClass}(\mathit{tree}, \{\mathtt{dec}, \mathtt{strdec}, \mathtt{datname}, \mathtt{conbind}\})$ and let $\mathsf{pattern}(\mathit{tree})$ be true iff $\mathsf{isClass}(\mathit{tree}, \{\mathtt{atpat}, \mathtt{pat}\})$. The classes dec, strdec, datname, and conbind are associated (using the toTree function) with terms for which our initial constraint generation algorithm generates binders.

Let us define our flattening function flat as follows:

$$
\begin{array}{ll}
\mathsf{flat}(\langle\rangle) & = \langle\rangle \\
\mathsf{flat}(\langle \mathit{tree}\rangle @ \overrightarrow{\mathit{tree}}) & = \begin{cases} \langle \mathit{tree}_1, \dots, \mathit{tree}_n\rangle @ \mathsf{flat}(\overrightarrow{\mathit{tree}}), \\ \quad \text{if } \mathit{tree} = \langle \mathit{dot}, \langle \mathit{tree}_1, \dots, \mathit{tree}_n\rangle\rangle \\ \quad \text{and } (\forall i \in \{1, \dots, n\}.\ \neg\mathsf{declares}(\mathit{tree}_i) \text{ or } \overrightarrow{\mathit{tree}} = \langle\rangle) \\ \langle \mathit{tree}\rangle @ \mathsf{flat}(\overrightarrow{\mathit{tree}}), \text{otherwise} \end{cases}
\end{array}
$$

The condition "$\forall i \in \{1, \dots, n\}.\ \neg\mathsf{declares}(\mathit{tree}_i)$" ensures that bindings are not mixed up as explained above. However, flattening the last dot term (if it actually is a dot term) cannot mix up the bindings because there is no identifier left to bind. Therefore, flattening $\langle\mathtt{..val\ x\ =\ 1..}\langle\mathtt{..val\ x\ =\ true..}\rangle\mathtt{..}\rangle$ would lead to $\langle\mathtt{..val\ x\ =\ 1..val\ x\ =\ true..}\rangle$. We however have not yet found a concrete example where this situation occurs.

We also define the function tidy to tidy sequences of declarations in structure expressions as follows:

$$\begin{array}{lll}
(\mathsf{SL1})\ \mathsf{sl}(\langle \mathit{node}, l, \overrightarrow{\mathit{tree}}\rangle, \overline{l}) & = & \begin{cases} \langle \mathit{node}, l, \mathsf{sl}_1(\overrightarrow{\mathit{tree}}, \overline{l})\rangle, & \text{if } l \in \overline{l} \text{ and } \mathsf{getDot}(\mathit{node}) \neq \mathtt{dotS} \\ \langle \mathit{node}, l, \mathsf{tidy}(\mathsf{sl}_1(\overrightarrow{\mathit{tree}}, \overline{l}))\rangle, & \text{if } l \in \overline{l} \text{ and } \mathsf{getDot}(\mathit{node}) = \mathtt{dotS} \\ \langle \mathsf{getDot}(\mathit{node}), \mathsf{flat}(\mathsf{sl}_2(\overrightarrow{\mathit{tree}}, \overline{l}))\rangle, & \text{otherwise} \end{cases} \\
(\mathsf{SL2})\ \mathsf{sl}_1(\langle \mathit{dot}, \langle \mathit{tree}_1, \ldots, \mathit{tree}_n\rangle\rangle, \overline{l}) & = & \langle \mathit{dot}, \mathsf{flat}(\langle \mathsf{sl}_2(\mathit{tree}_1, \overline{l}), \ldots, \mathsf{sl}_2(\mathit{tree}_n, \overline{l})\rangle)\rangle \\
(\mathsf{SL3})\ \mathsf{sl}_2(\langle \mathit{dot}, \langle \mathit{tree}_1, \ldots, \mathit{tree}_n\rangle\rangle, \overline{l}) & = & \langle \mathit{dot}, \mathsf{flat}(\langle \mathsf{sl}_2(\mathit{tree}_1, \overline{l}), \ldots, \mathsf{sl}_2(\mathit{tree}_n, \overline{l})\rangle)\rangle \\
(\mathsf{SL4})\ \mathsf{sl}_1(\langle \mathit{node}, l, \overrightarrow{\mathit{tree}}\rangle, \overline{l}) & = & \mathsf{sl}(\langle \mathit{node}, l, \overrightarrow{\mathit{tree}}\rangle, \overline{l}) \\
(\mathsf{SL5})\ \mathsf{sl}_2(\langle \mathit{node}, l, \overrightarrow{\mathit{tree}}\rangle, \overline{l}) & = & \mathsf{sl}(\langle \mathit{node}, l, \overrightarrow{\mathit{tree}}\rangle, \overline{l}) \\
(\mathsf{SL6})\ \mathsf{sl}_1(\langle \mathit{tree}_1, \ldots, \mathit{tree}_n\rangle, \overline{l}) & = & \langle \mathsf{sl}_1(\mathit{tree}_1, \overline{l}), \ldots, \mathsf{sl}_1(\mathit{tree}_n, \overline{l})\rangle \\
(\mathsf{SL7})\ \mathsf{sl}_2(\langle \mathit{tree}_1, \ldots, \mathit{tree}_n\rangle, \overline{l}) & = & \langle \mathsf{sl}_2(\mathit{tree}_1, \overline{l}), \ldots, \mathsf{sl}_2(\mathit{tree}_n, \overline{l})\rangle \\
(\mathsf{SL8})\ \mathsf{sl}_1(\mathit{id}, \overline{l}) & = & \mathit{id} \\
(\mathsf{SL9})\ \mathsf{sl}_2(\mathit{id}, \overline{l}) & = & \langle \mathtt{dotE}, \langle\rangle\rangle
\end{array}$$

Figure 11.17 Slicing algorithm

$$\begin{array}{l}
\mathsf{tidy}(\langle\rangle) = \langle\rangle \\
\mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{\mathit{tree}_1}\rangle, \langle \mathtt{dotD}, \overrightarrow{\mathit{tree}_2}\rangle\rangle @ \overrightarrow{\mathit{tree}}) \\
\quad = \mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{\mathit{tree}_1} @ \overrightarrow{\mathit{tree}_2}\rangle\rangle @ \overrightarrow{\mathit{tree}}), \text{ if } \forall \mathit{tree} \in \mathsf{ran}(\overrightarrow{\mathit{tree}_1}).\ \neg\mathsf{declares}(\mathit{tree}) \\
\mathsf{tidy}(\langle\langle \mathtt{dotD}, \varnothing\rangle\rangle @ \overrightarrow{\mathit{tree}}) \\
\quad = \mathsf{tidy}(\overrightarrow{\mathit{tree}}), \text{ if none of the above applies} \\
\mathsf{tidy}(\langle \mathit{tree}\rangle @ \overrightarrow{\mathit{tree}}) \\
\quad = \langle \mathit{tree}\rangle @ \mathsf{tidy}(\overrightarrow{\mathit{tree}}), \text{ if none of the above applies}
\end{array}$$

11.8.5 Algorithm

Fig. 11.17 formally defines our slicing algorithm. Note that rule (SL9) generates the dot marker dotE, but we could have used any of the terms in Dot because the flattening function flat discards such terms. The functions sl_1 and sl_2 are defined on trees but also on sequences of trees in rules (SL6) and (SL7). Finally, let $\mathsf{sl}(\mathit{strdec}, \overline{l})$ be $\mathsf{sl}(\mathsf{toTree}(\mathit{strdec}), \overline{l})$.

11.8.6 Generating type error slices for example (EX1)

First, let us repeat the labelled version of example (EX1) called $\mathit{strdec}_{\mathrm{EX}}$ and defined in Sec. 11.2:

$$\begin{array}{l}
\texttt{structure X} \overset{l_1}{=} \texttt{struct}^{l_2} \\
\qquad\qquad \texttt{structure S} \overset{l_3}{=} \texttt{struct}^{l_4}\ \texttt{datatype}\ \lceil\texttt{'a u}\rceil^{l_6} \overset{l_5}{=} \texttt{U}_{\texttt{c}}^{l_7}\ \texttt{end} \\
\qquad\qquad \texttt{datatype}\ \lceil\texttt{'a t}\rceil^{l_9} \overset{l_8}{=} \texttt{T}_{\texttt{c}}^{l_{10}} \\
\qquad\qquad \texttt{val rec f}_{\texttt{p}}^{l_{12}} \overset{l_{11}}{=} \texttt{fn T}_{\texttt{p}}^{l_{14}} \overset{l_{13}}{\Rightarrow} \texttt{T}_{\texttt{e}}^{l_{15}} \\
\qquad\qquad \texttt{val rec g}_{\texttt{p}}^{l_{17}} \overset{l_{16}}{=} \texttt{let}^{l_{18}}\ \texttt{open}^{l_{19}}\ \texttt{S in}\ \lceil\texttt{f}_{\texttt{e}}^{l_{21}}\ \texttt{U}_{\texttt{e}}^{l_{22}}\rceil^{l_{20}}\ \texttt{end} \\
\qquad \texttt{end}
\end{array}$$

We saw in Sec. 11.5.5, that, given example (EX1) (i.e., given $\mathit{strdec}_{\mathrm{EX}}$), our initial constraint generation algorithm generates the environments e_{EX}. We saw in Sec. 11.7.8, that, given e_{EX}, our enumeration algorithm enumerates only one error, namely er_{EX}.

$$\begin{array}{l}
\langle\langle\texttt{strdec},\texttt{strdecStr}\rangle,l_1,\langle\texttt{X},\langle\langle\texttt{strexp},\texttt{strexpSt}\rangle,l_2,\langle \mathit{tree}_1,\mathit{tree}_2,\mathit{tree}_3,\mathit{tree}_4\rangle\rangle\rangle\rangle \\
\text{where } \mathit{tree}_1 = \langle\langle\texttt{strdec},\texttt{strdecStr}\rangle,l_3, \\
\qquad\qquad\quad \langle\texttt{S}, \\
\qquad\qquad\quad\; \langle\langle\texttt{strexp},\texttt{strexpSt}\rangle,l_4, \\
\qquad\qquad\qquad\; \langle\langle\langle\texttt{dec},\texttt{decDat}\rangle,l_5, \\
\qquad\qquad\qquad\qquad \langle\langle\langle\texttt{datname},\texttt{datnameCon}\rangle,l_6,\langle\texttt{'a},\texttt{u}\rangle\rangle,\langle\langle\texttt{conbind},\texttt{id}\rangle,l_7,\langle\texttt{U}\rangle\rangle\rangle\rangle\rangle\rangle\rangle\rangle \\
\qquad\quad \mathit{tree}_2 = \langle\langle\texttt{dec},\texttt{decDat}\rangle,l_8,\langle\langle\langle\texttt{datname},\texttt{datnameCon}\rangle,l_9,\langle\texttt{'a},\texttt{t}\rangle\rangle,\langle\langle\texttt{conbind},\texttt{id}\rangle,l_{10},\langle\texttt{T}\rangle\rangle\rangle\rangle \\
\qquad\quad \mathit{tree}_3 = \langle\langle\texttt{dec},\texttt{decRec}\rangle,l_{11}, \\
\qquad\qquad\quad \langle\langle\langle\texttt{atpat},\texttt{id}\rangle,l_{12},\langle\texttt{f}\rangle\rangle, \\
\qquad\qquad\quad\; \langle\langle\texttt{exp},\texttt{expFn}\rangle,l_{13},\langle\langle\langle\texttt{atpat},\texttt{id}\rangle,l_{14},\langle\texttt{T}\rangle\rangle,\langle\langle\texttt{atexp},\texttt{id}\rangle,l_{15},\langle\texttt{T}\rangle\rangle\rangle\rangle\rangle\rangle \\
\qquad\quad \mathit{tree}_4 = \langle\langle\texttt{dec},\texttt{decRec}\rangle,l_{16}, \\
\qquad\qquad\quad \langle\langle\langle\texttt{atpat},\texttt{id}\rangle,l_{17},\langle\texttt{g}\rangle\rangle, \\
\qquad\qquad\quad\; \langle\langle\texttt{atexp},\texttt{atexpLet}\rangle,l_{18}, \\
\qquad\qquad\qquad\; \langle\langle\langle\texttt{dec},\texttt{decOpn}\rangle,l_{19},\langle\texttt{S}\rangle\rangle, \\
\qquad\qquad\qquad\;\; \langle\langle\texttt{exp},\texttt{app}\rangle,l_{20},\langle\langle\langle\texttt{atexp},\texttt{id}\rangle,l_{21},\langle\texttt{f}\rangle\rangle,\langle\langle\texttt{atexp},\texttt{id}\rangle,l_{22},\langle\texttt{U}\rangle\rangle\rangle\rangle\rangle\rangle\rangle
\end{array}$$

Figure 11.18 Result of applying toTree to $\mathit{strdec}_{\mathrm{EX}}$

In Sec. 11.6.8, $\mathit{er}_{\mathrm{EX}}$ is defined as $\langle \mathit{ek}_{\mathrm{EX}}, \overline{d}_{\mathrm{EX}}\rangle$ where $\overline{d}_{\mathrm{EX}}$ is the dependency set $\{l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11}, l_{12}, l_{13}, l_{14}, l_{19}, l_{20}, l_{21}, l_{22}\}$. Let us present the slice that our slicing algorithm computes when given $\mathit{er}_{\mathrm{EX}}$, i.e., we compute $\mathsf{sl}(\mathit{strdec}_{\mathrm{EX}}, \overline{d}_{\mathrm{EX}})$.

Fig. 11.18 shows the tree (which we call $\mathit{tree}_{\mathrm{EX}}$) obtained when applying toTree to $\mathit{strdec}_{\mathrm{EX}}$. Finally, $\mathsf{sl}(\mathsf{toTree}(\mathit{strdec}_{\mathrm{EX}}), \overline{d}_{\mathrm{EX}})$ returns the following tree where tree_1 and tree_2 are the ones defined above, and tree'_3 and tree'_4, are obtained from tree_3 and tree_4 respectively:

$$\begin{array}{l}
\langle\texttt{dotD},\langle\mathit{tree}_1,\mathit{tree}_2,\mathit{tree}'_3,\mathit{tree}_4\rangle\rangle \\
\text{where } \mathit{tree}'_3 = \langle\langle\texttt{dec},\texttt{decRec}\rangle,l_{11}, \\
\qquad\qquad\qquad \langle\langle\langle\texttt{atpat},\texttt{id}\rangle,l_{12},\langle\texttt{f}\rangle\rangle, \\
\qquad\qquad\qquad\; \langle\langle\texttt{exp},\texttt{expFn}\rangle,l_{13},\langle\langle\langle\texttt{atpat},\texttt{id}\rangle,l_{14},\langle\texttt{T}\rangle\rangle,\langle\texttt{dotE},\langle\rangle\rangle\rangle\rangle\rangle\rangle \\
\qquad\quad\;\; \mathit{tree}'_4 = \langle\texttt{dotE},\langle\langle\langle\texttt{dec},\texttt{decOpn}\rangle,l_{19},\langle\texttt{S}\rangle\rangle, \\
\qquad\qquad\qquad\qquad \langle\langle\texttt{exp},\texttt{app}\rangle,l_{20},\langle\langle\langle\texttt{atexp},\texttt{id}\rangle,l_{21},\langle\texttt{f}\rangle\rangle,\langle\langle\texttt{atexp},\texttt{id}\rangle,l_{22},\langle\texttt{U}\rangle\rangle\rangle\rangle\rangle\rangle
\end{array}$$

This slice is displayed as follows:

```
⟨..structure S = struct datatype 'a u = U end
 ..datatype 'a t = T
 ..val rec f = fn T => ⟨..⟩
 ..⟨..open S..f U..⟩..⟩
```

11.9 Minimality

Informally, bindings is a function on environments that extracts the bindings between accessors and binders (by keeping track of the bindings generated at constraint solving by rules (A1) and (A2)). We extend this function to a function on our external labelled syntax (this extension uses our constraint generator). For example, if *exp* is `let val x = true in let val x = 1 in x end end`, and the label l_i is associated with the ith occurrence of `x` then $\mathsf{bindings}(exp) = \{\langle l_2, l_3\rangle\}$.

We define the sub-slice relation as follows: $strdec_1 \sqsubseteq_{\overline{l}} strdec_2$ iff $\mathsf{sl}(strdec_2, \overline{l}) = strdec_1$ and $\mathsf{bindings}(strdec_1) \subseteq \mathsf{bindings}(strdec_2)$.

Let $strdec_2$ be a *minimal type error slice* of $strdec_1$ iff $\neg\mathsf{solvable}(strdec_2)$, $strdec_2 \sqsubseteq_{\overline{l}} strdec_1$, and for all $strdec'$ if $strdec' \sqsubseteq_{\overline{l}'} strdec_2$ for some $\overline{l}'$ and $strdec' \neq strdec_2$ then $\mathsf{solvable}(strdec')$.

We consider minimality as a design principle for our TES even though minimal slices do not always seem to be the correct answer to type error reporting (e.g., as explained in Sec. 15.1, for record field name clashes we report merged minimal type error slices).

For Core-TES (the subset of our TES presented in this section), we believe the following holds: a slice $strdec'$ is a minimal slice of $strdec$ iff $\langle strdec', ek, \overline{vid}\rangle \in \mathsf{tes}(strdec)$. We have not formally proved this statement for diverse reasons. First, our TES (Form-TES as well as Impl-TES) is in constant change and proving the minimality of one of its versions would not guarantee the minimality of the others. Moreover proving the minimality of Core-TES would not guarantee the minimality of TES (of Form-TES or of Impl-TES) and proving the minimality of TES is beyond the scope of this thesis. Then, as mentioned above, minimality is only a design principle. Let us finally stress that we feel improving the range and quality of our slices is more important than ensuring their minimality in particular.

Note that, given an untypable piece of code, a type error slice will always contain exactly the portion of the piece of code required to explain the error reported by the type error slice. Moreover, if a part of a slice is not necessary to explain the error, minimisation will remove it. Therefore the minimality of a type error slice is not related to its size. The size of a minimal type error slice depends on the error itself.

11.10 Design principles

While developing our TES we discovered, developed, and followed the following principles.

(DP1). Each syntactic sort of constraint terms should have a case ranging over an infinite variable set. This allows incomplete information everywhere, which allows one to consider every possible way of slicing out parts of the program. This is essential to get precise slices that include all relevant details and exclude the irrelevant. Thus, the sorts μ, τ, and e have the variable cases δ, α, and ev.

(DP2). Each syntactic sort of constraint terms should support dependencies. This allows precise blame, which enables precise slicing. Thus, sorts μ, τ, σ, and e have dependency cases $\langle\mu, \overline{d}\rangle$, $\langle\tau, \overline{d}\rangle$, $\langle\sigma, \overline{d}\rangle$, and $\langle e, \overline{d}\rangle$.

(DP3). Our initial constraint generation rules return a main result (a type or an environment) and sometimes also an environment result (used for constraints and bindings), i.e., our initial constraint generation rules return cgs as defined in Fig. 11.5.1. The generated constraints may connect information from the results for a program node's subtrees to the other subtrees or to the node's results.

The principle is that these connections should generally be via constraints that carry the syntax tree node's label and that are "shallow", i.e., contain only connection details and not constraints from program subtrees (see LabCs's definition in Sec. 11.5.2). Fresh variables should be used as needed. This allows a program syntax node to be "disconnected" for type errors that depend on the node's details, while still keeping type errors that arise solely due to connections between environment accessors and bindings that pass through the node.

For example, rule (G22) of our initial constraint generation algorithm defined in Fig. 11.7 in Sec. 11.5.1 builds the unlabelled constraint $ev'{=}(e_1;\cdots;e_n)$. This "deep" unlabelled constraint packs together a sequence of environments from the declarations that are the structure's body. The resulting environment is connected to the main result by the labelled shallow constraint $ev \overset{l}{=} ev'$.

(DP4). Duplicating constraints should be unnecessary. This seems obvious, but some previous formalisms seem too weak for the needed sharing. For example, rule (G22) of our initial constraint generation algorithm defined in Fig. 11.7 in Sec. 11.5.1 builds a structure's environment as the sequential composition of its component declarations' environments: $e_1;\cdots;e_n$. Here, the first declaration's environment e_1 is available for subsequent declarations and also in the result (if its bindings are not shadowed) which avoids duplicating it. A previous version of our system had a weaker constraint system with let-constraints similar to those of Pottier and Rémy [116], and the best solution we could find duplicated the constraints for each declaration's bindings, causing severe performance problems. Sec. 12.1.7 discusses further this issue.

(DP5). Dependencies must be propagated during solving exactly where needed. If dependencies are not propagated where they should go, minimisation could overminimise yielding non-errors. This can be detected. More insidiously, propagating dependencies where they are unneeded can keep alive unneeded parts of error slices much longer during minimisation, resulting in severe slowdowns. Because correct results happen eventually, detecting such bugs is harder so this requires great care. For example, an earlier version of our solver copied dependencies from declarations in a structure to the structure's main result. The minimiser had to remove declarations one at a time. Debugging this was hard because only speed suffered. Furthermore, the system should yield error slices (before minimisation) that are as close to minimal as can be reasonably achieved. If constraint solving yields a non-minimal error slice, then solving steps must have annotated a constraint with a location on which it does not uniquely depend.

(DP6) Sec. 11.7.6 already mentioned this principle. In the labelled external syntax, identifiers which can occur at bound positions must be labelled by a unique label that does not label a piece of code larger than the identifier itself. Moreover, for those labelled identifiers, the constraint generator should in general generate no more than a labelled accessor. (Note that to simplify the presentation of Core-TES we do otherwise for structure openings (see constraint generation rule (G19) in Fig. 11.7) but this is in general unsafe.) The risk of not following this principle is that during minimisation, a bound occurrence of an identifier can be kept in a slice while its binding occurrence is discarded. This can then result in the identifier at a bound position being bound to a different binding occurrence than the one to which it is originally bound in the original piece of code. This can then lead to generating wrong identifier bindings and finding false errors.

(DP7) Environment variables, when not generated as part of a shallow environment in an equality constraint (e.g., as the direct left or right-hand-side of an equality constraint), should always be labelled. As explained in Sec. 11.3, an unlabelled environment variable is a constraint that can never be filtered out and has to always be satisfied (independently from any program location). Because an environment variable shadows its context (i.e., in $(ev;e)$, the environment variable ev shadows e), if such an environment variable is unlabelled and is not constrained to be equal to anything, it can only shadow its context whatever filtering is applied on it. This behaviour is undesirable because the shadowing of an environment should in general be dependent on a program location (see, e.g., constraint generation rule (G19) in Fig. 11.7 for `open` declarations).

However, in our TES, at constraint generation, it happens that most of the environment variables not generated as part of a shallow environment in an equality constraint cannot shadow their environments. It is the case for rules (G4), (G17) and (G18). (Note that in these rules, each generated environment variable has to be labelled to carry the dependency on the program point responsible for its generation.) Each of these rules generates an environment variable that is constrained by an unlabelled equality constraint on the environment variable itself (these unlabelled equality constraints cannot be filtered out). If these equality constraints were labelled, but the environment variables were not, the equality constraints could be filtered out and the environment variables could then be unconstrained and therefore shadow their contexts. Given a piece of code, for rule (G17), e.g., this would mean that filtering out the constraints associated with a recursive value declaration in the piece of code would allow this declaration to shadow its entire context in the analysed piece of code which is undesirable. For example, when slicing out the recursive value declaration in `val x = 1 val rec f = fn x => x val y = x x`, we do not want it do shadow `val x = 1` (i.e., we do not want the environment generated for `val rec f = fn x => x` to shadow the environment generated for `val x = 1` when the label associated with `val rec f = fn x => x` is sliced out in the environment generated for the entire piece of code). Rule (G19) stands out by generating environment variables that are constrained by labelled accessors. Hence, if this rule was generating $((\uparrow strid \overset{l}{=} ev);ev)$ instead $((\uparrow strid \overset{l}{=} ev);ev^{l})$ (where the environment variable is unlabelled), ev would then be totally unconstrained when filtering out the accessor. This would disallow one to slice out `open` declarations. Worse, this could lead to finding typable type error slices. Let us illustrate this last point with the following example:

```
structure S = struct end
val x = 1
open S
val y = x 1
```

Note that the structure `S` is empty, so `open S` does not do anything and especially `x` is not rebound. Let us assume that our constraint generation algorithm generates the environment e for this sequence of declarations. Our enumeration algorithm would find a slice as follows:

```
⟨..val x = 1
 ..x ⟨..⟩..⟩
```

Now, filtering out the constraints in e w.r.t. this slice would lead to an environment e' where the unlabelled environment variable generated for `open S` (assuming that unlabelled environment variables are generated for `open` declarations instead of labelled environment variables as we do in our TES) shadows the environment generated for `x`'s declarations. The environment e' would then be solvable.

Chapter 12

Related work

12.1 Related work on constraint systems

12.1.1 Constraint based type inference algorithm

Milner [105] proved the soundness of the semantics of a small language (application, abstraction, conditional, recursion, local declaration) w.r.t. a typing relation. We refer to this language in this document as core ML. This result allows Milner to state that the well typed property is enough to prove the well-defined behaviour of pieces of code, for a certain notion of behaviour. Milner's method is based on three steps. First he provides a denotational semantics of his language. Milner defines *wrong* as a value in his denotational semantics. Milner points out that *wrong* "corresponds to the detection of a failure at run-time" where in his language "the only failures are the occurrences of a non-Boolean value as a condition of a conditional, and the occurrence of a nonfunctional value as the operator of an application"[1]. This semantics allows one to check some type constraints such as: the first parameter of a conditional expression has to be a Boolean. However, this semantics does not allow one to check some other constraints such as: the two branches of a conditional must have the same type. The second step of Milner's method consists in defining types and a typing relation between the values of his semantics and types to ensure the consistency of the typing of an expression, meaning that, e.g., a function cannot sometimes return a Boolean and sometimes return an integer when applied to, say, an integer. Milner provides an example of values that do not have types (such as the value wrong). One of them can be explained as follows: the value (semantics) of the function "$\lambda x.\text{if } x \text{ then } 1 \text{ else true}$" does not have any type. The third step of Milner's method is to define a type assignment system that assigns types to expressions. Finally, Milner's soundness results expresses that if a type can be assigned to an expression (if the expression is well-typed) then this type can also be

[1] Milner's theorem is well known under the slogan "well-typed expressions do not go wrong" where wrong is a value of his semantics with which no type can be associated.

assigned to the semantics of the expression (so the semantics of the expression cannot be the *wrong* value). An interesting aspect in Milner's paper is that when giving an informal presentation of his type inference algorithm (W) he separates constraint generation and constraint solving (these are interleaved in the W algorithm which leads to the well-known left-to-right bias).

Aiken [1] provides three reasons in favour of constraint-based program analyses (even though Aiken does not restrict himself to type constraints and to the type inference problem we provide our understanding of the advantages Aiken describes in the context of type inference). (1) "Constraints separate specification from implementation". This says that one obtains a clear separation between constraint generation and constraint solving where the constraint generation phase is regarded as producing a specification of the information that one wishes to analyse, and where the constraint solving phase is regarded as the implementation to compute this information. (2) "Constraints yield natural specifications". This says that each analysed piece of syntax is usually translated into (local) primitive constraints, each expressing a particular feature of the analysed piece of syntax. Moreover, let us add that in many constraint systems (see below for examples of such systems), new forms of constraints are sometimes introduced to deal with particular features of the analysed language and to deal with them in a particular way, and these constraints are usually used to translate more than one feature of the analysed language. Given a piece of code, the generated constraints are packed in a way that gives a constraint representation of the piece of code. (3) "Constraints enable sophisticated implementations". For example, various constraint solvers extending the Martelli-Montanari algorithm [103] have been designed to define different implementations.

As early as 1987, Wand [140] introduced a constraint based type inference algorithm for the simply typed λ-calculus to provide an alternative proof of the decidability of the type inference problem for the simply typed λ-calculus. Wand reduced the type inference problem to a unification problem by first converting λ-terms into constraint sets and by then solving the constraints. Wand's system is simple, he does not consider polymorphism and his constraints are only equality constraints (the only constraints required in his setting). His constraint generation algorithm is based on a type environment that associates types (type variables) with identifiers.

Henglein [66] considers the type inference problem for two calculi: the Milner calculus [105, 32] and the Milner-Mycroft calculus [110]. As in the original systems, the considered languages contain a fixpoint operator and a non-recursive "let" construct (the two calculi differ on the semantics of the fixpoint operator which only allows monomorphic recursion in the Milner calculus and polymorphic recursion in the Milner-Mycroft calculus). Henglein formulates the type inference problem in these calculi using a constraint based approach. First equality and inequality constraints are generated. Inequalities are used to deal with polymorphism (to encode type schemes) and therefore to enforce the monomorphism of λ-bindings (`fn`-bindings in SML). For example, using SML's syntax, in `fn z => let val rec f = fn x => z x in (f (), f true) end`, `f`'s first occurrence binds both `f`'s second and third occurrences. For each of the bindings, Henglein generates inequalities on `z`'s (monomorphic) type which eventually lead to an error because through the generated equality and inequality constraints, `z`'s type is constrained to be both a function that takes a `unit` (thanks to a first inequality set generated for the binding of `f`'s second occurrence to `f`'s first occurrence) and a `bool` (thanks to a second inequality set generated for the binding of `f`'s third occurrence to `f`'s first occurrence). Then, Henglein presents how to compute most general semi-unifiers from equality and inequality constraints. Unfortunately, Henglein's algorithm, based on semi-unification, is undecidable in the general case [87, 67].

Kanellakis, Mairson and Mitchell [86] consider the same algorithm as Wand [140]. They propose a type inference (they instead use the terminology "type reconstruction") algorithm for the λ-calculus extended with polymorphic (non-recursive) let-expressions (core ML) which consists of reducing an expression to a let-free expression (by reducing all the let-expressions) and then use Wand's algorithm on the obtained λ-expression. This algorithm, obviously inefficient in practice, intuitively gives the DEXPTIME-completeness of the type inference problem for core ML.

Pottier [114] defines a type system which is based on, among other things, constrained types, which are types depending on subtyping constraints. These forms are not allowed in types but only in type schemes and in type judgements (a constrained type is a component of a type judgement). The language considered by Pottier is a core ML-like language with (non-recursive) let-polymorphic expressions and subtyping. Pottier's system is based on a similar system by Eifrig, Smith and Trifnov [39] (they use a notion of *recursively constrained type* which is a type constrained by a set of inequality constraints which can themselves be recursive). Pottier mentions that Eifrig, Smith and Trifnov's system, "although theoretically correct, depends on type simplification in order to be usable in practice" (this is due to the fact that their polymorphic variable rule duplicates the constraints generated for polymorphic values without simplifying them first). Pottier's solution to avoid a combinatorial explosion in the number of constraints is to allow the simplification of constraints during constraint generation. Moreover, Pottier does not use a notion of solvability of generated constraints but instead uses a notion of consistency. With the notion of consistency, no "solution" of a constraint set is computed[2]. Pottier proves that the notion of consistency is equivalent to the notion of solvability. He defines a notion of entailment which is used by his substitution and subtyping rules. An issue

[2] Eifrig, Smith and Trifnov [39] write: "we expect general union and intersection types would be required to express the solution of constraints as types, but we do not wish to pay the penalty of having these types in our languages". The notion of consistency is then expected to be simpler to deal with than the notion of solvability.

with Pottier's approach is that, as in many other approaches, to avoid constraint duplication, constraint generation and constraint solving are mixed.

Sulzmann, Odersky and Wehr [112] define a generic type inference algorithm for the HM(X) system. This system is a "general framework for Hindley/Milner style type systems with constraints". Sulzmann, Odersky and Wehr say about their system that "particular type systems can be obtained by instantiating the parameter X to a specific constraint system" and that "the Hindley/Milner system itself is obtained by instantiating X to the trivial constraint system" (the standard Herbrand constraint system). They also extend their framework with subtyping. Their type inference algorithm mixes constraint generation and constraint solving. Constraint solving is performed via a "normalization" function. Each time an already generated constraint is extended with a new constraint (constraints are packed together via a conjunction operator which can be seen as the union operator in their context), the extended constraint is normalised. Type schemes in their system can either be monomorphic types or constrained type schemes of the form $\forall\overline{\alpha}.C \Rightarrow \sigma$ where $\overline{\alpha}$ is a type variable set, C is a constraint and σ is a type scheme (similar forms are used by, e.g., Eifrig, Smith, and Trifonov [39], Pottier [114], or Duggan [36]). Because of the way normalisation is used, during type inference, the constraints of the generated type schemes are already simplified. Sulzmann [129] calls such a use of normalisation, an *eager* use. Sulzmann [129] defines variants of the generic type inference mentioned above where normalisation is only used before inferring the type of let-expression's bodies and at the end of the type inference process only. This is achieved by defining an extra rule (and relation) that normalises constraints and which is to be used when needed (such a use of normalisation is called *by need*). In their system, normalisation is required before inferring the type of let-expression's bodies because using normalisation only at the end of the type inference process leads to the separation of the constraint generation and the constraint solving phases but also to an inefficient type inference algorithm. Sulzmann, Muller and Zenger [128, 129] present a variant of the inference algorithm mentioned above where constraints are preferred over terms. For example, informally, constraint-based systems are more expressive because one can devise a simple constraint language and a simple constraint generation algorithm that associates the constrained type $\langle\{\alpha_1{=}\alpha_2{\rightarrow}\alpha, \alpha_1{=}\texttt{int}, \alpha_2{=}\texttt{int}\}, \alpha\rangle$ (where, using our notation, the first component of the pair is a constraint set that constrains the second component of the pair which is a type variable) with the application `(1 1)`. However, for this expression to be typable, one needs more complex type constructors such as the ones used by Neubauer and Thiemann [111]. Also, because Sulzmann, Muller and Zenger's type inference algorithm is not based on substitutions anymore (but on constraints), they obtain simpler results (e.g., their completeness of inference) than with Sulzmann, Odersky and Wehr's system [112]. Müller [109] claims that an advantage of HM(X) is that “it provide generic proofs of correctness, principality, and completeness of type inference”.

We discuss other constrained based systems below, by Hage and Heeren [65, 63, 58, 60], by Müller [108], by Gustavsson and Svenningsson [55], and by Pottier and Rémy [116, 115].

12.1.2 Constrained types

Pottier defines a system [114], similar to the one used by Eifrig, Smith and Trifonov [39], that uses constrained types of the form $\tau|C$, where τ is a type and C is a (subtyping) constraint set. These forms are not allowed in types but only in type judgements and in type schemes which are of the following form: $\forall\overline{\alpha}.\tau|C$ (similar to those used by Pottier and Rémy [116]) where $\overline{\alpha}$ is a set of type variables. As opposed to other systems [112, 78], Pottier allows constrained types in typing judgement because in his system a typing judgement is of the form $A \vdash e : \tau|C$ where A is a type environment and e is an expression of the external syntax.

Odersky, Sulzmann and Wehr [112] and Kaes [78] also consider constrained types in their type schemes. However, because they use a different presentation style of their constraint generation algorithm, constrained types are not allowed in type judgements (a constrained type is not a component of a type judgement). Instead of writing $A \vdash e : \tau|C$ (using Pottier’s syntax) they would write such a typing judgement as follows: $C, A \vdash e : \tau$ where C also constrains τ but where such a constrained form is not explicitly defined.

In our constraint system, types can only be constrained via equality constraints as in the following environment: $e;(\tau_1{=}\tau_2)$ where both τ_1 and τ_2 are constrained by the environment e. For example, our constraint generation rule (G3) for expression applications generates an environment of the form $e_1;e_2;(\alpha_1 \overset{l}{=} \alpha_2{\to}\alpha)$ where e_1 and α_1 are generated for the function part of the application, and where e_2 and α_2 are generated for the argument part of the application. In this environment, both e_1 and e_2 constrain both α_1 and α_2 even though α_1 only depends on e_1 and α_2 only depends on e_2. We could then imagine a constraint system where we allow constrained types to be types. Constrained types could be of the form $(e;\tau)$. This would allow one to generate instead, for expression applications, an environment of the form $(e_1;\alpha_1) \overset{l}{=} (e_2;\alpha_2){\to}\alpha$. The drawback of such a system is that types are not shallow anymore which complicates constraint filtering and solving.

12.1.3 Comparison with Haack and Wells’ constraint system

The method of Haack and Wells (HW-TES) makes use of intersection types. A type ty in HW-TES can either be a type variable, the integer type or an arrow type. A type set is denoted by S. An intersection type is denoted $\wedge S$. HW-TES' constraint generation algorithm gathers the types of bound occurrences of identifiers in type environments which associate intersection types with identifiers.

Let us consider the following simple piece of code: `x x`. Given this piece of code, HW-TES generates the triple $\langle \Gamma_x, a_x, C_x \rangle$, where the type environments Γ_x, the type variable a_5, and the constraint set C_x are described below. First, the type environment Γ_x is of the form $\{\mathtt{x} \mapsto \wedge\{a_1, a_2\}\}$[3] where $a_1 \neq a_2$, a_1 is a type variable generated for `x`'s first occurrence, and a_2 is a type variable generated for `x`'s second occurrence. The constraint set C_x contains, among other things, constraints on a_1 and a_2, and is of the following form: $\{a_1 \overset{l_1}{=} a'_1, a_2 \overset{l_2}{=} a'_2, a'_1 \overset{l_3}{=} a_3 \rightarrow a_4, a'_2 \overset{l_3}{=} a_3, a_{\mathtt{x}} \overset{l_3}{=} a_4\}$ where l_1 is `x`'s first occurrence's label, l_2 is `x`'s second occurrence's label, and l_3 is the label associated with the application.

Let us now consider a monomorphic binding of these two occurrences of `x`. Let `x` be bound via a monomorphic `fn`-binding as follows: `fn x => x x`. Given this piece of code, HW-TES' constraint generation algorithm generates the triple $\langle \Gamma_m, a_m, C_m \rangle$ (where "m" stands for "monomorphic"). The type environment Γ_m is $\varnothing$ and C_m is of the following form: $C_x \cup \{a \overset{l}{=} a_1, a \overset{l}{=} a_2, a \rightarrow a_x \overset{l}{=} a_m\}$, where l is the label labelling the `fn`-expression, and where a_1 and a_2 are obtained from Γ_x.

Let us now consider the polymorphic case. First, assume that given `fn y => z y` (this piece of code is reused in the let-expression presented below), where `z` is a free variable, HW-TES' constraint generation algorithm generates the following triple: $\langle \Gamma_z, a_z, C_z \rangle$. The type environment Γ_z is of the form $\{\mathtt{z} \mapsto \wedge\{a_5\}\}$. Let us now consider the following polymorphic let-binding of `x`: `let val x = fn y => z y in x x end`. Now, because Γ_x (defined above) associates two type variables with `x`, HW-TES' constraint generation algorithm generates two "fresh" copies of $\langle \Gamma_z, a_z, C_z \rangle$ namely $\langle \Gamma'_z, a'_z, C'_z \rangle$ and $\langle \Gamma''_z, a''_z, C''_z \rangle$. The type environments Γ'_z and Γ''_z are of the form $\{\mathtt{z} \mapsto \wedge\{a'_5\}\}$ and $\{\mathtt{z} \mapsto \wedge\{a''_5\}\}$ respectively. It finally generates the following triple for the entire let-expression: $\langle \Gamma'_z \wedge \Gamma''_z, a', C_x \cup C'_z \cup C''_z \cup \{a'_z \overset{l}{=} a_1, a''_z \overset{l}{=} a_2, a' \overset{l}{=} a_x\} \rangle$ where l is the label labelling the let-expression, where a_1 and a_2 are obtained from Γ_x, and where $\Gamma'_z \wedge \Gamma''_z = \{x \mapsto \wedge S_1 \cup S_2 \mid \Gamma'_z(x) = \wedge S_1 \wedge \Gamma''_z(x) = \wedge S_2\} = \{\mathtt{z} \mapsto \wedge\{a'_5, a''_5\}\}$ (x is Haack and Wells' notation for program variables). Note that polymorphism involves heavy constraint and type environment duplications which leads to a combinatorial constraint size explosion at constraint generation.

[3] Environments in HW-TES are total functions from identifiers to intersection types. Therefore, the environment $\{\mathtt{x} \mapsto \wedge\{a_1, a_2\}\}$ denotes the total function that associates $\wedge\{a_1, a_2\}$ with `x` and that associates $\wedge\{\}$ with any identifier different from `x`.

12.1.4 Comparison with Hage and Heeren's constraint system

The approach followed by Hage and Heeren [65, 63, 58, 60] is as follows: given a piece of code, first a constraint tree is generated, then this constraint tree is converted into a list (many conversions are possible which result in different lists), and finally the constraints are solved. Because different conversions of trees into lists are allowed, their system allows them to emulate algorithms such as W [32], M [98] or UAE [147].

In their system, a constraint tree can among other things (we only present some of their constructs), be a strict node as follows: $T_1 \ll T_2$ where T_1 and T_2 are constraint trees. A constraint can be attached to a tree using for example the following construct: $c \diamond T$, which makes the constraint c "part of the constraint associated with the root of T" [60]. A tree can also pack together trees as follows: $\{\!\!\{ T_1, \ldots, T_n \}\!\!\}$. A constraint itself can among other things be: an equality constraint $\tau_1 \equiv \tau_2$, a generalisation constraint $\sigma{:=}\texttt{GEN}(M, \tau)$ where M is a (monomorphic) type variable set and σ is a scheme variable, or a instantiation constraint $\tau \preceq \sigma$. Hage and Heeren [60] say about their generalisation and instantiation constraints: "The reason we have constraints to explicitly represent generalization and instantiation is the same as why, e.g., Pottier and Rémy do [116]: otherwise we would be forced to (make a fresh) duplicate of the set of constraints every single time we use a polymorphically defined identifier".

Their equality types are similar to ours. Their generalisation constraints are related to `poly` environments but are restricted to types. Another difference is that the monomorphic type variable set that are not allowed to be quantified over when generating a type scheme is part of a generalisation constraint in their system while in our system, such a set is computed at constraint solving. Their instantiation constraints are related to our accessors but they do not mention external syntax (external identifiers) and do not have identifier bindings in their constraint language.

Trees in their system can be regarded as sophisticated constraints. They are used to provide extra structure on constraint sets. In our system a single equality constraint can be an environment. Similarly, in their system a single constraint can be a tree. Their strict nodes of the form $T_1 \ll T_2$ can be seen as a restricted version of our composition environments of the form $e_1;e_2$. Environments of the form $e_1;e_2$ also enforce e_1 to be solved before e_2. A major difference is that in our system, not only in an environment $e_1;e_2$, the environment e_1 has to be solved before e_2 but also e_2 looks up in e_1 to access binders. Also a major difference between trees and constraint/environments is that in their system trees do not act as environments, they do not allow one to associate static semantics with identifiers. We do not allow non-strict nodes (such as their nodes of the form $\{\!\!\{ T_1, \ldots, T_n \}\!\!\}$) because our system does not rearrange the order in which constraints are initially generated. Their constraint rearrangement mechanism can be seen as a restriction of our enumeration algorithm.

Enforcing to solve constraints before other introduces a bias. Our TES is unbiased thanks to our enumeration algorithm which, given an environment e, run our constraint solver on the different environments that can be obtain from e using our filtering function. We believe that Hage and Heeren only partially remove the bias thanks to their ordering strategies.

The main difference between their transformation of a type inference problem into a constraint solving problem and ours (and so the main difference between their constraint system and our constraint system) is that we also encode the bindings of identifiers into our constraint system. Bindings of identifiers are solved at constraint solving in our system while they are solved at constraint generation in Hage and Heeren's system. We do so thanks to our binders and accessors. We moved from a binding resolution at initial constraint generation to a binding resolution at constraint solving in order to handle SML features such as the `open` feature. Thanks to our binders and accessors, we can generate a "faithful" representation of a SML program, that uses intricate features such as `open`, into our constraint language.

Moreover, we believe that in addition to the motivation of generating "faithful" representations of SML programs in our constraint language, binders and accessors are necessary to distinctly separate the constraint generation and constraint solving phases of a constraint based type inference algorithm for SML. To illustrate this point let us consider the following typable SML program:

```
structure S = struct val c = fn () => () end
structure T = S
structure U = T
open U
val d = c ()
```

Without binders and accessors, one needs to use type environments at constraint generation to be able to access identifiers' static semantics when analysing identifiers at bound positions. At constraint generation, in order to be able to generate a proper environment for the declaration `open U` so that it can be used when dealing with the declaration `val d = c ()`, one needs to resolve the chain of structure equalities. This means that solving structures' static semantics at constraint generation becomes necessary which goes against a clear separation between constraint generation (generation of constraints on the static semantics of the analysed piece of code) and constraint solving.

The necessity of having bindings solved at constraint solving rather than at constraint generation is also motivated by the will of having a compositional constraint generation algorithm while dealing with the inherent identifier status ambiguity in

SML which is dealt with in Sec. 14.1. Here we anticipate Sec. 14.1 where unconfirmed binders of the form $\updownarrow vid{=}\alpha$ are introduced to deal with SML's identifier status ambiguity. When initially generated, such unconfirmed binders are neither binders nor accessors but lie between the two. As a matted of fact, for a piece of code such as `fn x => fn c => x c`, from Sec. 14.1 on, the binders generated for `x` and `c` are unconfirmed binders and the static semantics of `x`'s second occurrence does not depend on the static semantics of `x`'s first occurrence until the unconfirmed binder generated for `x` is turned into a confirmed one (and similarly for `c`). If it turns out at constraint solving that, e.g., `c` is a datatype constructor then `c`'s unconfirmed binder is turned into an accessor. Otherwise `c`'s unconfirmed binder turns into a dependent or independent (on `c`'s status) confirmed binder (still at constraint solving only and not at constraint generation). Note that a similar argument holds about `open` declarations. Compositionality is further discussed in Sec. 16.1.

12.1.5 Comparison with Müller's constraint system

Müller [108] defines the *relational calculus* ρ_{deep} to "implement Damas-Milner polymorphic type inference". This calculus allows one to generate constraints of linear size. It does that by generating identifier binders with which are associated static semantics. The semantics attached to an identifier binder can then be simplified before being "used", i.e., before instantiating the polymorphic type. The language considered by Müller is the λ-calculus extended with polymorphic let-expressions (core ML). Müller also forces bound variables in λ-expressions to be "pairwise different and distinct from the free variables". His constraint language is a two layer language. He first defines a constraint set and then an expression set containing the constraint set. What Müller calls an expression will sometimes be called a constraint expression in this discussion when we need to distinguish between a λ-expression (an external expression) and an expression (an internal or constraint expression). Müller's syntax of constraints and expressions is defined as follows:

$$\begin{aligned}\phi, \psi &::= \top \mid \bot \mid \exists\alpha\ \phi \mid \phi \wedge \psi \mid \alpha = \beta \mid \alpha = \beta \rightarrow \gamma\\ E, F &::= \phi \mid E \wedge F \mid \exists\alpha\ E \mid x{:}\alpha/E \mid [\![M]\!]\alpha\end{aligned}$$

where M is a λ-expression and α, β and γ are type variables. The two constant constraints are the satisfied constraint $\top$ and the unsatisfied constraint $\bot$. Constraints and expressions of the forms $\exists\alpha\ \phi$ and $\exists\alpha\ E$ introduce fresh variables. Constraints and expressions of the form $\phi \wedge \psi$ and $E \wedge F$ are conjunctions. The two last forms of constraints are shallow equality constraints.

The most interesting forms in Müller's constraint system are: $x{:}\alpha/E$ and $[\![M]\!]\alpha$.

An expression $x{:}\alpha/E$ is called an *abstraction* and associates the constrained static semantics α, constrained by E, with the identifier x. Such expressions are called abstractions because, e.g., $x{:}\alpha/E$ abstracts the type variable α. The polymorphism of such forms comes from the fact that expressions can be existential expressions. If `id` is the polymorphic identity function, one can then generate the following abstraction (binder) for `id` (where some expressions are omitted for clarity): `id`:$\gamma/\exists\beta\ \gamma = \beta \rightarrow \beta$. Let us now assume a bound occurrence of `id` with which is associated the static semantics α. One has then to apply the abstraction generated for `id` to α which results in $\exists\beta\ \alpha = \beta \rightarrow \beta$. A particularity of ρ_{deep} is that computations can occur within the nested expression of an abstraction, which is within E in an abstraction of the form $x{:}\alpha/E$.

Intuitively, we believe that an abstraction of the form $x{:}\alpha/E$ would be represented in our system by an environment of the form poly(e;$\downarrow x$=α) where E is represented by e.

Note that because of the restriction on free and bound variables, Müller does not need to define local constraints to restrict the scope of abstractions. Given such a restriction on the λ-expressions, Müller's inference algorithm cannot generate two abstractions for the same identifier.

An expression of the form $[\![M]\!]\alpha$ is called a *proof obligation* and it "represent the constraint $\alpha = \tau$ for the principal type τ of M", where τ is an internal type in Müller's system. A constraint expression of the form $[\![M]\!]\alpha$ is used to analyse (infer a type for) the lambda expression M.

The constraint based type inference algorithm defined by Müller does not distinguish between constraint generation and constraint solving and no specific constraint solving strategy is presented (constraint generation and solving interleave). Especially, it seems that Müller's system does not enforce simplifying the constraints generated for a polymorphic identifier x before applying the abstraction generated for x. This can therefore lead to the exponential growth of the size of the constraint expression generated for a λ-expression. Let us consider the following simple let-expression called M (where `fn x => x` is written as λ`x.x` using Müller's λ-expressions' syntax):

```
let id = fn x => x
in let f = id id in f f end
end
```

Let M' be `let f = id id in f f end`. Fig. 12.1 presents the inference of M'type using Müller's type inference algorithm. One can observe the duplication of the constraint expression generated for `id`'s body.

$$
\begin{array}{ll}
 & \llbracket M \rrbracket\alpha \\
\rightarrow & \llbracket M' \rrbracket\alpha \wedge F \wedge \texttt{id}{:}\gamma/E,\ \text{where } E = \llbracket \texttt{fn x => x} \rrbracket\gamma \text{ and } F = \exists\beta\ (\llbracket \texttt{id} \rrbracket\beta) \\
\rightarrow & \llbracket \texttt{f f} \rrbracket\alpha \wedge F' \wedge \texttt{f}{:}\gamma'/\llbracket \texttt{id id} \rrbracket\gamma' \wedge F \wedge \texttt{id}{:}\gamma/E,\ \text{where } F' = \exists\beta'\ (\llbracket \texttt{f} \rrbracket\beta') \\
\rightarrow & (\exists\beta''\ \exists\gamma''\ (\llbracket \texttt{f} \rrbracket\beta'' \wedge \llbracket \texttt{f} \rrbracket\gamma'' \wedge \beta'' = \gamma'' \rightarrow \alpha)) \wedge F' \wedge \texttt{f}{:}\gamma'/\llbracket \texttt{id id} \rrbracket\gamma' \wedge F \wedge \texttt{id}{:}\gamma/E \\
\rightarrow & (\exists\beta''\ \exists\gamma''\ (\llbracket \texttt{f} \rrbracket\beta'' \wedge \llbracket \texttt{f} \rrbracket\gamma'' \wedge \beta'' = \gamma'' \rightarrow \alpha)) \wedge F' \wedge \texttt{f}{:}\gamma'/E' \wedge F \wedge \texttt{id}{:}\gamma/E \\
 & \quad \text{where } E' = \exists\beta'''\ \exists\gamma'''\ (\llbracket \texttt{id} \rrbracket\beta''' \wedge \llbracket \texttt{id} \rrbracket\gamma''' \wedge \beta''' = \gamma''' \rightarrow \gamma') \\
\rightarrow^{*} & (\exists\beta''\ \exists\gamma''\ (\llbracket \texttt{f} \rrbracket\beta'' \wedge \llbracket \texttt{f} \rrbracket\gamma'' \wedge \beta'' = \gamma'' \rightarrow \alpha)) \wedge F' \wedge \texttt{f}{:}\gamma'/E'' \wedge F \wedge \texttt{id}{:}\gamma/E \\
 & \quad \text{where } E'' = \exists\beta'''\ \exists\gamma'''\ (E\{\beta'''/\gamma\} \wedge E\{\gamma'''/\gamma\} \wedge \beta''' = \gamma''' \rightarrow \gamma') \\
\rightarrow^{*} & (\exists\beta''\ \exists\gamma''\ (E''\{\beta''/\gamma'\} \wedge E''\{\gamma''/\gamma'\} \wedge \beta'' = \gamma'' \rightarrow \alpha)) \wedge F' \wedge \texttt{f}{:}\gamma'/E'' \wedge F \wedge \texttt{id}{:}\gamma/E
\end{array}
$$

Figure 12.1 Derivation using Müller's type inference algorithm

12.1.6 Comparison with Gustavsson and Svenningsson's constraint system

Gustavsson and Svenningsson [55] defined a constraint system where solutions can be found in cubic time. Their constraint syntax is based on: the satisfied constraint $\top$, inequality constraints on variables of the form $a \leq b$ where a and b are variables, conjunctions of constraints of the form $M \wedge N$ where M and N are constraint terms, and existential constraints of the form $\exists a.M$. They also add to their syntax, abstractions and applications.

Constraint abstractions are inspired by let-expressions and are of the form: $f\ \vec{a} = M$, where f is a constraint abstraction variable (the name of an abstraction), $\vec{a}$ is a set[4] of variables, and M is a constraint term. Constraint abstractions are used in let-constraint terms. A let-constraint term is of the form: $\texttt{let}\ \{\vec{F}\}\ \texttt{in}\ M$, where $\vec{F}$ is a set of abstractions and M is a constraint term. Abstractions in a let-constraint are mutually recursive so in a let-constraint $\texttt{let}\ \{\vec{F}\}\ \texttt{in}\ M'$, if $f\ \vec{a} = M$ is a constraint abstraction in $\vec{F}$, then all the uses of f in $\vec{F}$ and M' all refer to this occurrence of f.

We believe a let-constraint as follows:

$$\texttt{let}\ \{f_1\ \vec{a}_1 = M_1, \ldots, f_n\ \vec{a}_n = M_n\}\ \texttt{in}\ M$$

would be represented in our system by an environment as follows:

$$[\texttt{poly}(\downarrow f_1{=}\alpha_1;\cdots;\downarrow f_n{=}\alpha_n;e_1;\cdots;e_n);e]$$

where M_i would be represented by e_i for each $i \in \{1, \ldots, n\}$, where M would be represented by e, and where $\vec{a}_i$, for each $i \in \{1, \ldots, n\}$, would be computed when dealing at constraint solving with the `poly` constraint.

Abstractions are applied thanks to application constraint terms of the form $f\ \vec{a}$. An abstraction of the form $f\ \vec{a}$ would be represented in our system by an accessor of the form $\uparrow f{=}\alpha$.

Gustavsson and Svenningsson define a constraint solving algorithm and prove it to be of cubic complexity. Such a result is obtained by enforcing that abstractions are simplified before being applied. Their constraint solver is based on a rewriting

[4] Even though it is not explicitly stated in their paper, vectors seem to be used for sets.

system that allows four kinds of reductions: a transitivity reduction rule and three reduction rules allowing reducing abstractions at various places in a let-constraint (in the body of the let-constraint, in the body of the abstraction that is applied or in the body of another abstraction declared in the same let-constraint).

These reduction rules do not allow one to copy the whole body of an abstraction when it is applied. Only the "live" inequality constraints are allowed to be copied at an application location, where an inequality constraint is said to be "live" in a constraint term if it does not use a variable which is bound in the term.

12.1.7 Comparison with Pottier and Rémy's let-constraints

Our constraint system has evolved through many versions. One earlier version of our constraint system had a kind of constraint that was very close to the let-constraints[5] of systems of Pottier and Rémy [116, 115]. Pottier and Rémy define a constraint system [116] which allows one "to reduce type inference problems for HM(X) to constraint solving problems". Pottier defines a very similar system [115]. Using their let-constraints Pottier and Rémy "achieve the desired separation between constraint generation, on the one hand, and constraint solving and simplification, on the other hand, without compromising efficiency" [116]. In our discussion, we will collectively refer to these two systems as the PR (Pottier/Rémy) system and ignore their technical differences, although our presentation will follow more closely the presentation of Pottier and Rémy [116].

In PR, a constraint can, among other things, be a let-constraint, a subtyping constraint, a type scheme instantiation constraint, a conjunction of constraints, or the constant (and satisfied) `true` constraint. A PR let-constraint looks like **let** $id{:}\dot{\sigma}$ **in** C where $\dot{\sigma}$ ranges over type schemes, and C ranges over constraints. In PR, type schemes are of the form $\forall\overline{X}[C].T$ where $\overline{X}$ is a type variable set, C is a constraint, and T is a type. We borrow for our discussion two abbreviations that Pottier and Rémy define: (1) the form $\forall\overline{X}.T$ stands for the type scheme $\forall\overline{X}[\mathtt{true}].T$, and (2) the form **let** $id{:}T$ **in** C stands for **let** $id{:}\forall\varnothing.T$ **in** C.

The idea of let-constraints is that a constraint of the form

$$\textbf{let } id{:}\forall\overline{X}[C].T \textbf{ in } (id = T_1 \wedge id = T_2)$$

is (roughly) equivalent to a constraint of this form:

$$(\exists\overline{X}.(C \wedge T = T_1)) \wedge (\exists\overline{X}.(C \wedge T = T_2)) \wedge (\exists\overline{X}.C)$$

The key point is that one can get the effect of making the appropriate number

[5] Technically, the let-constraints of Pottier and Rémy are based on their more primitive def-constraints.

of copies of C and T while keeping the size of the constraint proportional to the program size. The constraints will need to be copied and each copy solved independently, but each copy can be solved immediately before the next copy is made, avoiding an exponential increase in the amount of memory used during constraint solving. To get the full benefit of this, an implementation should be eager in simplifying C and calculating T as much as possible before making any copies. (In our application, it could be good to also be lazy in simplifying and calculating only those portions of C and T that are actually needed by the uses of id, because our TES needs to spend most of its time finding minimal portions of unsatisfiable constraints. We leave investigating this idea for future work.)

Identifier bindings occurring in let-constraints are similar to abstractions as defined by Müller [108]. A binding as defined by Pottier and Rémy is of the form $id{:}\forall\overline{X}[C].T$ where the type scheme $\forall\overline{X}[C].T$ associated with id is a constrained type scheme where the constraint C constrains the type T. An abstraction as defined by Müller [108] is of the form $x{:}\alpha/E$ where the static semantics associated with the identifier x is the type variable α which is constrained by the expression E.

In our latest system, the equivalent of let-constraints can be represented as a special case of what our system supports. Informally, a let-constraint of the form $\mathbf{let}\ id{:}\forall\overline{X}[C_1].T\ \mathbf{in}\ C_2$ generated for a SML recursive `let`-binding would be represented in our system by (using a combination of rules (G2) and (G17) in Fig. 11.7)

$$[\mathtt{poly}((\downarrow id{=}\tau);e_1);e_2]$$

where C_i is represented by e_i and T is represented by τ. (Let-constraints generated for other SML forms would not necessarily get the same representation.) There is no explicit representation of $\overline{X}$ in the representation in our system; instead the correct set of type variables that can be quantified is calculated by toPoly which generates type schemes when it handles environments of the form $\mathtt{poly}(e)$ (see Fig. 11.9).

Let us have a closer look at the different components of a let-constraint. A let-constraint is of the form $\mathbf{let}\ id{:}\forall\overline{X}[C_1].T\ \mathbf{in}\ C_2$. Such a constraint: (1) assigns static semantics to the identifier id (thanks to the form $id{:}\dot{\sigma}$), (2) quantifies the static semantics associated with id over a set of variables (generates a polymorphic type), (3) makes the access to id's semantics local to C_2, and (4) defines an order in which the constraints have to be solved (C_1 before C_2). Such a constraint can then be seen as the combination of (at least) four primitive constraints. The first one is a binder in our system, the second one is a `poly` environment in our system, the third one is an environment of the form $[e]$ in our system, and the fourth one is an environment of the form $e_1;e_2$ in our system.

We now give an example comparing the constraints that would be generated for

SML recursive value declarations in the PR system and our system. Consider the SML expression

$$\texttt{let val rec f = fn z => } \mathit{exp}_1 \texttt{ in } \mathit{exp}_2$$

where exp_1 and exp_2 are two sub-expressions. The constraint generated in PR for this let-expression would be

$$\mathbf{let}\ \texttt{f}{:}\forall XY[\mathbf{let}\ \texttt{f}{:}X \to Y\ \mathbf{in}\ \mathbf{let}\ \texttt{z}{:}X\ \mathbf{in}\ C_1].X \to Y\ \mathbf{in}\ C_2$$

where X and Y are internal type variables, where XY is PR notation for the set $\{X, Y\}$, where C_i for $i \in \{1, 2\}$ is the constraint generated for exp_i, and where Y is the result type of exp_1. Due to the way let-constraints declare a local environment, the PR system needs two binders for `f`. The outer one polymorphically binds the occurrences of `f` in exp_2 and the inner one monomorphically binds the occurrences of `f` in exp_1.

Some of the differences between PR and our system can be seen when comparing how this example is handled. Our constraint generator builds roughly[6] the following constraint (technically, an environment) for the example let-expression:

$$[\texttt{poly}(\downarrow\texttt{f}{=}\alpha_1{\to}\alpha_2;[(\updownarrow\texttt{z}{=}\alpha_1);e_1]);e_2]$$

In contrast to how PR handles this example, only one binder for `f` is needed in our system. Two features of our system interact to allow this. First, in a composition environment $(e_1;e_2)$, the bindings from e_1 are available in e_2, but also form part of the result (except where bindings in e_2 shadow them). Second, in an environment of the form $\texttt{poly}(e)$, the `poly` operator changes the status of binders in the result from the status they had internally. In the example constraint (environment) above, `f`'s binder is monomorphic within the scope of the `poly` operator (in e_1) and polymorphic outside (in e_2).

There is a sense in which what the PR system does is similar to what would happen in our system if the `poly` operator worked on just single types or single bindings rather than entire environments. It is significant that we can form environments of the form $\texttt{poly}(\downarrow\mathit{vid}{=}\tau;e_1);e_2$, in which the type for vid is available monomorphically in e_1 and polymorphically in e_2.

The differences between the PR system and our system gain greater significance when we consider how to handle the SML module system. The most basic construct

[6] We have omitted labels and simplified a bit. The actual constraint that is generated (still omitting labels though) is

$$[(\mathit{ev}_2{=}\texttt{poly}(\downarrow\texttt{f}{=}\alpha_1;[(\mathit{ev}_1{=}(\updownarrow\texttt{z}{=}\alpha_2));\mathit{ev}_1;e_1;c_1];c_2));\mathit{ev}_2;e_2;c_3]$$

where $c_1 = (\alpha_3{=}\alpha_2{\to}\alpha_4)$, $c_2 = (\alpha_1{=}\alpha_3)$, $c_3 = (\alpha_5{=}\alpha_6)$, $\langle\alpha_4, e_1\rangle$ is generated for exp_1, $\langle\alpha_6, e_2\rangle$ is generated for exp_2, and α_5 is the type of the entire let-expression.

of the module system is what forms the body of a structure, namely a sequence of declarations $\mathit{dec}_1 \cdots \mathit{dec}_n$. For this discussion, assume each dec_i declares exactly one identifier x_i. Consider how declaration sequences can be handled by the PR system and our system. PR can handle such a sequence with nested let-constraints as follows:

$$\textbf{let } x_1{:}\dot{\sigma}_1 \textbf{ in } (\cdots \textbf{let } x_n{:}\dot{\sigma}_n \textbf{ in } C_0 \cdots)$$

The constraints must be nested as indicated because each x_i is only visible in the "**in**" part of the corresponding let-constraint, where an identifier binding occurrence is visible when constraints can refer to it. In contrast, our system handles the same declaration sequence with the environment

$$e_1; \cdots ; e_n$$

where e_i is the environment generated for the declaration dec_i for each $i \in \{1, \ldots, n\}$.

The importance of the difference becomes clearer when we consider how to represent full structures and structure bindings. Take the above example declaration sequence and wrap it up in a structure definition:

$$\texttt{structure } \mathit{strid} = \texttt{struct } \mathit{dec}_1 \cdots \mathit{dec}_n \texttt{ end}$$

A structure expression packs into a unit a sequence of declarations. The normal scope of the declarations ends at the end of the structure, and subsequent access to the declarations must go through the structure itself, which must first be bound to a name via either a structure declaration like above or a functor application. When performing type inference for SML structure expressions, it is most natural and straightforward that the type inferred for a structure will be a sequence of individual mappings from declared names to their types[7]. Such sequences are often called *environments*. It seems clear that any type inference method will need to handle environments.

The PR system has never been extended to handle ML-style structures[8], but let us imagine how it might be extended to do this. First, let us point out that Pottier and Rémy abbreviate the above example of nested let-constraints as follows:

$$\textbf{let } \Gamma_{\mathrm{d}} \textbf{ in } C_0, \text{ where } \Gamma_{\mathrm{d}} = x_1{:}\dot{\sigma}_1; \cdots ; x_n{:}\dot{\sigma}_n$$

Let us call this constraint C_{d} where the "d" means "declarations". Given an SML structure definition, this kind of constraint can represent the constraints required

[7] The order of the sequence is important because a type scheme for one value identifier in a structure can refer to a type constructor name defined by the structure, while at the same time a type scheme for a different value identifier can use the same type constructor name to refer to a definition outside the structure.

[8] François Pottier told us this on 2010-08-09.

for typability of the sequence of declarations in the structure body, and it is the only easy way to do so in the context of the PR system.

Now, how do we represent the connection of the structure's body to the structure's name? The immediately (and naively) obvious idea is to extend PR with let-constraints of a form similar to **let** $strid{:}\Gamma_{\mathrm{s}}$ **in** C, where $strid$ is a structure identifier, and Γ_{s} is an environment (the type of a structure). Let us call this new constraint C_{s}. This is not enough, because there needs to be some way to connect the constraint C_{d} to the environment Γ_{s}. In fact, the environment Γ_{d} inside C_{d} is just what we need, but there is no easy way to get at it, because there is no mechanism in PR for generating an environment from a constraint. The easiest thing to do is to nest the entire constraint C_{s} inside the constraint C_0 inside of C_{d}, because the types of the x_i's are not accessible from outside C_{d}, but this seems like turning the program inside out, because the entire rest of the program must be nested inside the scope of the constraints for just the structure's body.

So one might then want to extend the PR constraint system with an exporting mechanism and generate a constrained environment of the form $[C_{\mathrm{d}}].\Gamma_{\mathrm{s}}$ for the structure expression where C_{d} would export the type schemes of the x_is and where Γ_{s} would refer to these exported type schemes. But, all this technicality really should not be needed because Γ_{d} is already the environment that we would want to generate for the structure expression.

The way our constraint system achieves that is by instead of having only one mechanism (the let-constraints) to bind identifiers and to restrict their scope (let-constraints define a local scope), it has two separate mechanisms: one for bindings that does not restrict the scope of the binders (we obtain this behaviour by having binding constraints of form $\downarrow id{=}x$ and by having our general composition environment forms $e_1;e_2$ where the accessors occurring in e_2 can depend on the binders occurring in e_1), and another one for constraining the scope of a type environment (obtained thanks to our environments of the form $[e]$). The environment we generate for the structure expression presented above is then similar to the environment Γ_{d}.

12.2 Related work on presenting type errors and types

12.2.1 Methods making use of slices

After the first version of TES presented by Haack and Wells [56, 57], many researchers began to present type errors as program slices obtained from unsolvable sets of constraints.

Tip and Dinesh [133] report type error slices for a Pascal-like language called

CLaX, which is an explicitly typed language (where explicit types are enforced, e.g., on function parameters). Their method consists of defining the type checker of the CLaX language as a rewriting system. This rewriting system rewrites a piece of code into either a type if the piece of code is typable, or into a list of error messages if the piece of code is untypable. To compute slices they use "dependence tracking" [41, 42]. Tip and Dinesh explain that "Dependence tracking is a method for computing term slices that relies on an analysis of rewriting rules to determine how the application of rewriting rules causes *creation* of new function symbols, and the *residuation* (i.e., copying, moving around, or erasing) of previously existing subterms" [133]. Developments (w.r.t. a sequence of rewriting steps on a piece of code) are trimmed to retain only the necessary symbols of a piece of code, i.e., the ones responsible for an error to occur. Tip and Dinesh also applied their techniques to Mini-ML [25] which is a subset of ML ("a simple typed λ-calculus with constants, products, conditionals, and recursive function definitions" [25]). However, Tip and Dinesh face some minimality issues when applying their method to Mini-ML ("in some cases slices are computed that seem larger than necessary" [133]). This issue is related to the lack of a minimisation algorithm.

Neubauer and Thiemann [111] use flow analysis to compute type dependencies for a small ML-like language to report type errors. Their system uses discriminative sum types and can analyze any term. Their first step ("collecting phase") labels the studied term and infers type information. This analysis generates a set of program point sets. These program points are directly stored in the discriminative sum types. A conflicting type ("multivocal") is then paired with the locations responsible for its generation. Their second step ("reporting phase") consists of generating error reports from the conflicts generated during the first phase. Slices are built from which highlighting are produced. An interesting detail is that a type derivation can be viewed as the description of all type errors in an untypable piece of code, from which another step computes error reports.

Similar to ours is work by Stuckey, Sulzmann and Wazny [127, 141] (based on earlier work without slices [125, 126]). They do type inference, type checking and report type errors for the Chameleon language (a modified Haskell subset). Chameleon includes algebraic data types, type-class overloading, and functional dependencies. They code the typing problem into a constraint problem and attach labels to constraints to track program locations and highlight parts of untypable pieces of code. First they compute a minimal unsatisfiable set of generated constraints from which they select one of the type error locations to provide their type explanation. They finally provide a highlighting and an error message depending on the selected location. They provide slice highlighting but using a different strategy from ours. They focus on explaining conflicts in the inferred types at one program point inside the error location set. It is not completely clear, but they do not seem to worry much about whether the program text they are highlighting is exactly (no more and no less) a complete explanation of the type error. For example, they do not highlight applications because "they have no explicit tokens in the source code". It is then stated: "We leave it to the user to understand when we highlight a function position we may also refer to its application". This differs from our strategy because we think it is preferable to highlight all the program locations responsible for an error even if these are white spaces. Moreover, they do not appear to highlight the parts of datatype declarations relevant to type errors.

When running on a translation of the code presented in Sec. 10.4.2 into Haskell, ChameleonGecko outputs the error report partially displayed below (the rest of the output seems to be internal information from their solver).

```
ERROR: Type error; conflicting sites:
y = (trans x1, x2)
```

This highlighting identifies the same location as SML/NJ and would not help solve the error.

Significantly, because they handle a Haskell-like language, they face challenges for accurate type error location that are different from the ones for SML.

Gast [47] generates "detailed explanations of ML type errors in terms of data flows". His method is in three steps: generation of subtyping constraints annotated by reasons for their generation; gathering of reasons during constraint solving; transformation of the gathered reasons into explanations by data flows. He provides a visually convenient display of the data flows with arrows in XEmacs. Gast's method (which seems to be designed only for a small portion of OCaml) can be considered as a slicing method with data flow explanations.

Braßel [16] presents a generic approach (implemented for the language Curry) for type error reporting that consists of two different procedures. The first one tries to replace portions of code by dummy terms that can be assigned any type. If an untypable piece of code becomes typable when one of its subtrees has been replaced by a dummy term then the process goes on to apply the same strategy inside the subtree. The second procedure consists in using of a heuristic to guide the search of type errors. The heuristic is based on two principles: it will always "prefer an inner correction point to an outer one" and will always "prefer the point which is located in a function farther away in the call graph from the function which was reported by the type checker as the error location". Braßel's method does not seem to compute proper slices but instead singles out different locations that might be the cause of a type error inside a piece of code.

12.2.2 Significant non-slicing type explanation methods

Heeren et al. designed a method used in the Helium project [64, 62, 65, 59] to provide error messages for the Haskell language relying on a constraint-based type inference. First, a constraint graph is generated from a piece of code. For an ill-typed piece of code, a conflicting path called an inconsistency is extracted from the constraint graph. Such a conflicting path is a structured unsolvable set of type constraints. Heuristics are used to remove inconsistencies. A trust value is associated with each type constraint and depending on these values and the defined heuristics, some constraints are discarded until the inconsistency is removed. They also propose some "program correcting heuristics" used to search for a typable piece of code from an untypable one. Such a heuristic is for example the permutation of parameters which is a common mistake in programming. Their approach has been used with students learning functional programming. Using pieces of code written by students and their expertise of the language they refined their heuristics. They also designed a system of "directives" which are commands specified by the programmer to constrain the set of types derivable from a type class. This approach differs from ours by privileging locations over others by the use of some heuristics. They do not compute minimal slices and highlightings.

We present below the most interesting part of the error report obtained using Helium on a translation of the code presented in Sec. 10.4.2 into Haskell. It comes with some warnings (which are not displayed here) on the bindings of identifiers such as the binding of `y` in `trans` (some of these warnings explain, for example, that `y`'s declaration at the end of the code does not bind any of the `y`'s in `trans`'s definition).

```
(16,6):  Type error in application
expression       :  trans x1
term             :  trans
  type           :  T a   a   a    -> T a   a   a
  does not match :  T Int Int Bool -> T Int Int Bool

Compilation failed with 1 error
```

It is reported that `x1` and `trans` don't have the expected types. The application, which is at the end of the code, is then blamed when our programming error is at the very beginning of the code.

Also, they have tackled the task to report type errors for Java [14, 15]. Error reports provided by usual compilers can be of little help, especially in the presence of generics. El Boustani and Hage try to do a better job by keeping track of more information during type checking. When analysing an untypable piece of code, it allows a more global view of its type errors and leads to more informative error reports. The main difference between type error reporting for SML and for Java is that in Java "types are instantiated based on local information only and not through a long and complicated sequence of unifications" [14].

Lerner, Flower, Grossman and Chambers [99] present type error messages by constructing well-typed programs from ill-typed ones using different techniques (like Heeren et al. [59]), e.g., switching two parameters. Automatically conceived modifications to the ill-typed piece of code are checked for typability. They target Caml, and also developed a prototype for C++. The new typable generated code is presented as possible code that the programmer might have intended. It could be interesting to study the combination of this with TES.

Chapter 13

Case studies

13.1 Modification of user data types using TES

Our TES is generally of great help when coding in SML. It is particularly helpful when one wants to modify a user data type in a well-typed program. Let us consider the very simple program provided in Fig. 13.1a (this is testcase 577 in our testcase database) where we define a structure `Id` to deal with labelled identifiers (see the type `idlab`). In this structure we define some functions to handle labelled identifiers such as a function to compare two labelled identifiers (`compare`), or a function to build a labelled identifier from a label and an identifier (`cons`).

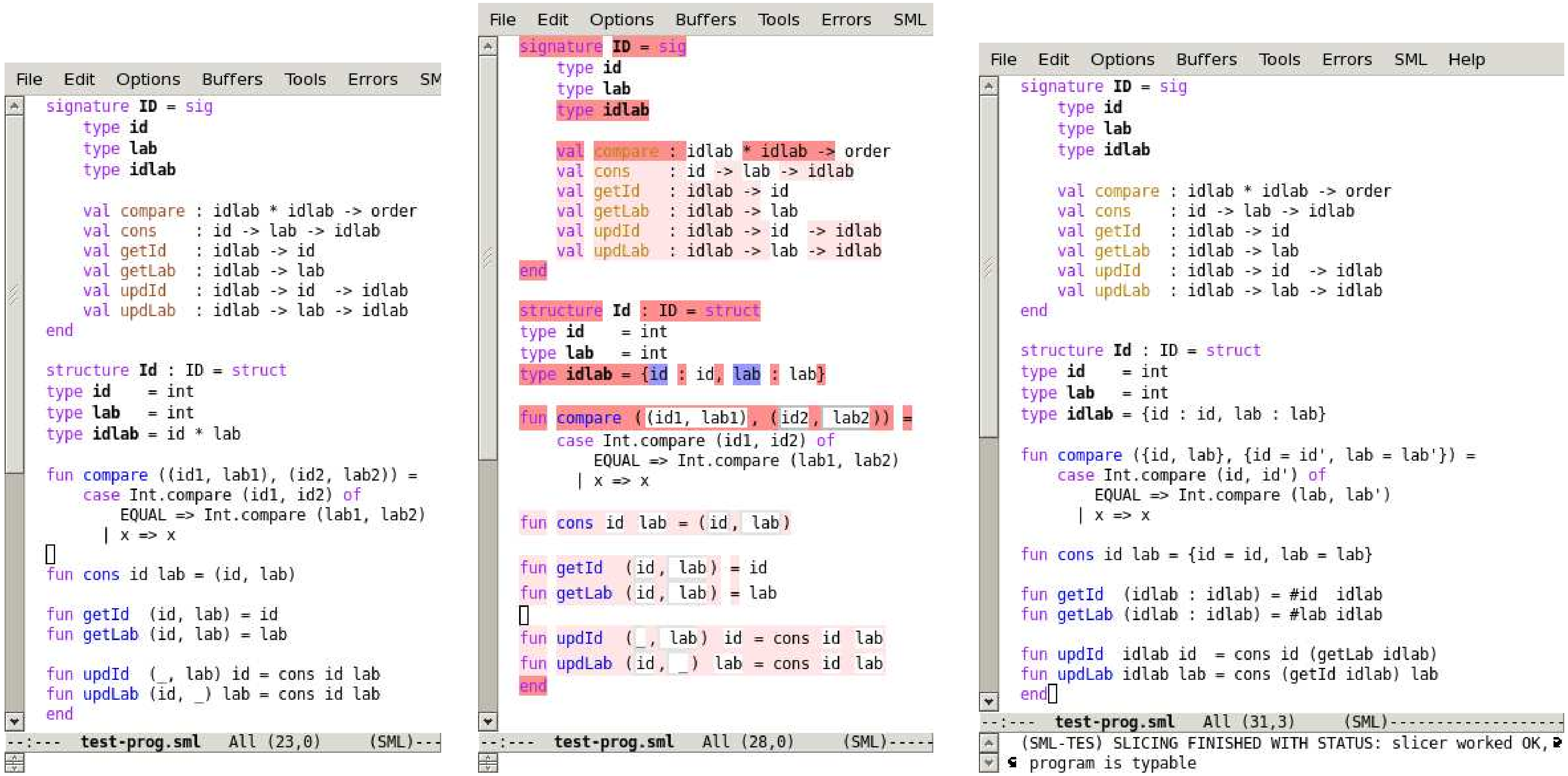
Now, let us change `idlab`'s declaration, for a more convenient one as follows: `type idlab = {id : id, lab : lab}`. The type `idlab` is now a record type containing two fields, one named `id` of type `id` and a second one named `lab` of type `lab`. Records are usually preferred over tuples because they are more flexible and meaningful thanks to the field names.

For example, one can access the field named `id` in an expression `x` of type `idlab` (the new type `idlab`) as follows: `#id(x:idlab)`. Records are more flexible than tuples because the order of the fields does not matter in a record. For example, `{id = 0, lab = 0}` is equivalent to `{lab = 0, id = 0}`. Note that a tuple `(id, lab)` is equivalent to a record `{1 = id, 2 = lab}`.

First of all, let us mention that when compiling the updated code with SML/NJ v.110.72, one obtains a type error report for each function defined in the structure `Id`. The report concerning the `compare` function is as follows:

```
test-prog.sml:14.1-31.4 Error: value type in structure doesn't match signature spec
    name: compare
  spec:   ?.Id.idlab * ?.Id.idlab -> order
  actual: (int * int) * (int * int) -> order
```

Note that the reported region is the entire structure `Id`.


File Edit Options Buffers Tools Errors SM
File Edit Options Buffers Tools Errors SML
File Edit Options Buffers Tools Errors SML Help

```
signature ID = sig
    type id
    type lab
    type idlab

    val compare : idlab * idlab -> order
    val cons    : id -> lab -> idlab
    val getId   : idlab -> id
    val getLab  : idlab -> lab
    val updId   : idlab -> id  -> idlab
    val updLab  : idlab -> lab -> idlab
end

structure Id : ID = struct
type id    = int
type lab   = int
type idlab = id * lab

fun compare ((id1, lab1), (id2, lab2)) =
    case Int.compare (id1, id2) of
        EQUAL => Int.compare (lab1, lab2)
      | x => x

fun cons id lab = (id, lab)

fun getId  (id, lab) = id
fun getLab (id, lab) = lab

fun updId  (_, lab) id = cons id lab
fun updLab (id, _) lab = cons id lab
end
--:---  test-prog.sml   All (23,0)     (SML)---
```

(a) Structure defining labelled identifiers

```
signature ID = sig
    type id
    type lab
    type idlab

    val compare : idlab * idlab -> order
    val cons    : id -> lab -> idlab
    val getId   : idlab -> id
    val getLab  : idlab -> lab
    val updId   : idlab -> id  -> idlab
    val updLab  : idlab -> lab -> idlab
end

structure Id : ID = struct
type id    = int
type lab   = int
type idlab = {id : id, lab : lab}

fun compare ((id1, lab1), (id2, lab2)) =
    case Int.compare (id1, id2) of
        EQUAL => Int.compare (lab1, lab2)
      | x => x

fun cons id lab = (id, lab)

fun getId  (id, lab) = id
fun getLab (id, lab) = lab

fun updId  (_, lab) id  = cons id lab
fun updLab (id, _)  lab = cons id lab
end
--:---  test-prog.sml   All (28,0)     (SML)-----
```

(b) Highlighting obtained after a type change

```
signature ID = sig
    type id
    type lab
    type idlab

    val compare : idlab * idlab -> order
    val cons    : id -> lab -> idlab
    val getId   : idlab -> id
    val getLab  : idlab -> lab
    val updId   : idlab -> id  -> idlab
    val updLab  : idlab -> lab -> idlab
end

structure Id : ID = struct
type id    = int
type lab   = int
type idlab = {id : id, lab : lab}

fun compare ({id, lab}, {id = id', lab = lab'}) =
    case Int.compare (id, id') of
        EQUAL => Int.compare (lab, lab')
      | x => x

fun cons id lab = {id = id, lab = lab}

fun getId  (idlab : idlab) = #id  idlab
fun getLab (idlab : idlab) = #lab idlab

fun updId  idlab id  = cons id (getLab idlab)
fun updLab idlab lab = cons (getId idlab) lab
end
--:---  test-prog.sml   All (31,3)     (SML)------------------
(SML-TES) SLICING FINISHED WITH STATUS: slicer worked OK,
program is typable
```

(c) Program obtained after solving all the type errors

Figure 13.1 Using TES to modify user data types

MLton v.20100608 outputs the following error report concerning `compare`:

```
Error: test-prog.sml 14.16.
  Variable type in structure disagrees with signature.
    variable: compare
    structure: [lab * lab] * [lab * lab] -> _
    signature: [id: lab, lab: lab] * [id: lab, lab: lab] -> _
```

Poly/ML v.5.3 outputs the following error report concerning `compare`:

```
Error-Structure does not match signature.
   Signature: val compare: idlab * idlab -> order
   Structure: val compare: (int * int) * (int * int) -> order
   Reason: Can't match int * int to {id: int, lab: int} (Field 1 missing)
Found near
  struct
   type id = int
   type lab = int
   type idlab = {id: id, ...}
   fun ...
   fun ...
   ...
   ...
   end
```

As for SML/NJ, MLton and Poly/ML both report a conflict between `compare`'s types in the structure `Id` and in its signature `ID`. Also, MLton blames the signature `ID` constraining the structure `Id` and Poly/ML blames the entire structure.

In contrast, Fig. 13.1b presents the highlighting that one obtains when running Impl-TES on the updated piece of code. The error in focus (highlighted with a darker red) shows that the parameter of `compare` is a pair of pairs. The second pair (equivalent to a record with two fields named `1` and `2`) clashes with the type of `compare`'s second parameter given in the signature `ID`, which is `idlab`, declared as a record with field names `id` and `lab` in the structure `Id`. In the parameter of `compare`, the second pair has its elements surrounded by grey boxes. We do so, because tuples do not have explicitly written field names. The first grey box surrounds the first element of a pair that corresponds to a record where the element would be in a field with field name `1` (and similarly for the second box). Note that the number of boxes indicates the arity of the tuple. In addition to the highlighting, we also report a type error slice (not presented here because often, as it is the case in Fig. 13.1b, highlightings are enough to solve type errors) and the following message for this type error:

Record clash, the fields {id,lab} conflict with {1,2}

The light pink corresponds to slices other than the focused one. One can then start solving the errors one at a time by just editing the highlighted portions of code, to get from a well-typed program to another well-typed program (see Fig. 13.1c).

13.2 Adding a new parameter to a function

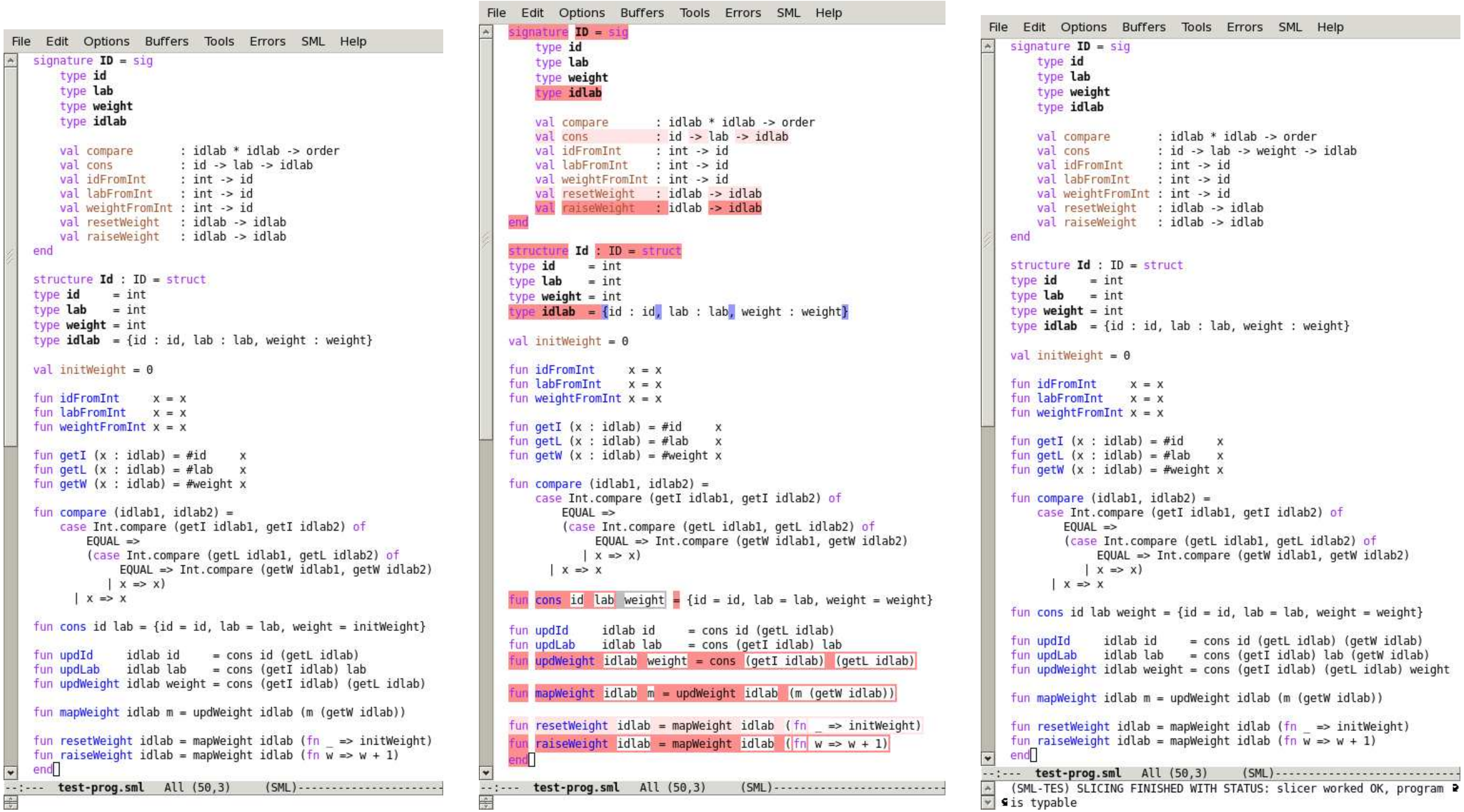
Our TES and its Emacs user interface are also generally useful when one wants to add a new parameter to a function. Starting from the program in Fig. 13.1c, let us consider the program provided in Fig. 13.2a. We have essentially added weights to our labelled identifiers (this is testcase 578 in our testcase database). We have also added some functions (declared in `Id` and sometimes also specified in `ID`) such as functions to deal with weights (e.g., `raiseWeight` raises the weight of a labelled identifier), renamed some functions (e.g., `getId` has been renamed to `getI`), removed some specifications from `ID` (e.g., we removed `getId`'s specification).

Even though in Fig. 13.2a, we still have not made all the necessary changes to deal with weights, the program is well-typed. Let us now add a new parameter to the function `cons`. The new (third) parameter is a weight which allows one to build a labelled identifier by specifying its weight (in Fig. 13.2a, `cons` uses a default weight when building a labelled identifiers from a label and an identifier).

When compiling the updated code with SML/NJ v.110.72, one obtains three type error reports. One reporting that `cons`'s type in `Id` does not match its specification in `ID`. The two other ones are similar but for the two functions `resetWeight` and `raiseWeight`. The three error reports are as follows:

```
test-prog.sml:16.1-50.4 Error: value type in structure doesn't match signature spec
    name: cons
  spec:   Id.id -> Id.lab -> Id.idlab
  actual: 'a -> 'b -> 'c -> {id:'a, lab:'b, weight:'c}
test-prog.sml:16.1-50.4 Error: value type in structure doesn't match signature spec
    name: resetWeight
  spec:   Id.idlab -> Id.idlab
  actual: Id.idlab -> 'a -> {id:Id.id, lab:Id.lab, weight:'a}
test-prog.sml:16.1-50.4 Error: value type in structure doesn't match signature spec
    name: raiseWeight
  spec:   Id.idlab -> Id.idlab
  actual: Id.idlab -> 'a -> {id:Id.id, lab:Id.lab, weight:'a}
```

Note that once again the reported region is the entire structure `Id`. In the report mentioning `raiseWeight`, one can see that SML/NJ derived that the function `raiseWeight` declared in `Id` takes two arguments and that the second argument's type is the same as the type of the weight of the returned labelled identifier. However,



```
signature ID = sig
    type id
    type lab
    type weight
    type idlab

    val compare       : idlab * idlab -> order
    val cons          : id -> lab -> idlab
    val idFromInt     : int -> id
    val labFromInt    : int -> id
    val weightFromInt : int -> id
    val resetWeight   : idlab -> idlab
    val raiseWeight   : idlab -> idlab
end

structure Id : ID = struct
type id     = int
type lab    = int
type weight = int
type idlab  = {id : id, lab : lab, weight : weight}

val initWeight = 0

fun idFromInt     x = x
fun labFromInt    x = x
fun weightFromInt x = x

fun getI (x : idlab) = #id     x
fun getL (x : idlab) = #lab    x
fun getW (x : idlab) = #weight x

fun compare (idlab1, idlab2) =
    case Int.compare (getI idlab1, getI idlab2) of
        EQUAL =>
        (case Int.compare (getL idlab1, getL idlab2) of
             EQUAL => Int.compare (getW idlab1, getW idlab2)
           | x => x)
      | x => x

fun cons id lab = {id = id, lab = lab, weight = initWeight}

fun updId     idlab id     = cons id (getL idlab)
fun updLab    idlab lab    = cons (getI idlab) lab
fun updWeight idlab weight = cons (getI idlab) (getL idlab)

fun mapWeight idlab m = updWeight idlab (m (getW idlab))

fun resetWeight idlab = mapWeight idlab (fn _ => initWeight)
fun raiseWeight idlab = mapWeight idlab (fn w => w + 1)
end
--:---  test-prog.sml   All (50,3)     (SML)--------------------
```

(a) Structure defining labelled identifiers with weights

```
signature ID = sig
    type id
    type lab
    type weight
    type idlab

    val compare       : idlab * idlab -> order
    val cons          : id -> lab -> idlab
    val idFromInt     : int -> id
    val labFromInt    : int -> id
    val weightFromInt : int -> id
    val resetWeight   : idlab -> idlab
    val raiseWeight   : idlab -> idlab
end

structure Id : ID = struct
type id     = int
type lab    = int
type weight = int
type idlab  = {id : id, lab : lab, weight : weight}

val initWeight = 0

fun idFromInt     x = x
fun labFromInt    x = x
fun weightFromInt x = x

fun getI (x : idlab) = #id     x
fun getL (x : idlab) = #lab    x
fun getW (x : idlab) = #weight x

fun compare (idlab1, idlab2) =
    case Int.compare (getI idlab1, getI idlab2) of
        EQUAL =>
        (case Int.compare (getL idlab1, getL idlab2) of
             EQUAL => Int.compare (getW idlab1, getW idlab2)
           | x => x)
      | x => x

fun cons id lab weight = {id = id, lab = lab, weight = weight}

fun updId     idlab id     = cons id (getL idlab)
fun updLab    idlab lab    = cons (getI idlab) lab
fun updWeight idlab weight = cons (getI idlab) (getL idlab)

fun mapWeight idlab m = updWeight idlab (m (getW idlab))

fun resetWeight idlab = mapWeight idlab (fn _ => initWeight)
fun raiseWeight idlab = mapWeight idlab (fn w => w + 1)
end
--:---  test-prog.sml   All (50,3)     (SML)------------------------
```

(b) Highlighting obtained after adding a parameter to a function

```
signature ID = sig
    type id
    type lab
    type weight
    type idlab

    val compare       : idlab * idlab -> order
    val cons          : id -> lab -> weight -> idlab
    val idFromInt     : int -> id
    val labFromInt    : int -> id
    val weightFromInt : int -> id
    val resetWeight   : idlab -> idlab
    val raiseWeight   : idlab -> idlab
end

structure Id : ID = struct
type id     = int
type lab    = int
type weight = int
type idlab  = {id : id, lab : lab, weight : weight}

val initWeight = 0

fun idFromInt     x = x
fun labFromInt    x = x
fun weightFromInt x = x

fun getI (x : idlab) = #id     x
fun getL (x : idlab) = #lab    x
fun getW (x : idlab) = #weight x

fun compare (idlab1, idlab2) =
    case Int.compare (getI idlab1, getI idlab2) of
        EQUAL =>
        (case Int.compare (getL idlab1, getL idlab2) of
             EQUAL => Int.compare (getW idlab1, getW idlab2)
           | x => x)
      | x => x

fun cons id lab weight = {id = id, lab = lab, weight = weight}

fun updId     idlab id     = cons id (getL idlab) (getW idlab)
fun updLab    idlab lab    = cons (getI idlab) lab (getW idlab)
fun updWeight idlab weight = cons (getI idlab) (getL idlab) weight

fun mapWeight idlab m = updWeight idlab (m (getW idlab))

fun resetWeight idlab = mapWeight idlab (fn _ => initWeight)
fun raiseWeight idlab = mapWeight idlab (fn w => w + 1)
end
--:---  test-prog.sml   All (50,3)     (SML)--------------------------
(SML-TES) SLICING FINISHED WITH STATUS: slicer worked OK, program
is typable
```

(c) Program obtained after solving all the type errors

Figure 13.2 Using TES to add a parameter to a function

this report does not make it clear as why SML/NJ constrains raiseWeight to take two arguments. One finally ends up at trying to understand as why SML/NJ generated such type information. Note that the piece of code being untypable, the types generated and reported by SML/NJ are anyway erroneous and therefore confusing.

MLton v.20100608 outputs the following error report concerning raiseWeight:

```
Error: test-prog.sml 16.16.
  Variable type in structure disagrees with signature.
    variable: raiseWeight
    structure: _ -> [???  -> {id: weight, lab: weight, weight: ???}]
    signature: _ -> [{id: weight, lab: weight, weight: weight}]
```

MLton blames the signature constraint on Id, namely, the signature ID. This report is similar to the one generated by SML/NJ. Apart from the blamed region, it also differs by hiding some of the non-conflicting generated internal type information using _.

Poly/ML v.5.3 outputs the following error report concerning raiseWeight:

```
Error-Structure does not match signature.
   Signature: val raiseWeight: idlab -> idlab
   Structure: val raiseWeight: idlab -> 'a -> {id: int, lab: int, weight: 'a}
   Reason:
      Can't match 'a -> {id: int, lab: int, weight: 'a} to
         {id: int, lab: int, weight: int} (Incompatible types)
Found near
  struct
   type id = int
   type lab = int
   type weight = int
   type idlab = ...
   val ...
   ...
   ...
   end
```

Once again Poly/ML blames the entire Id structure. Poly/ML's report is similar to MLton's report. Apart from the blamed region, it also differs by not hiding some of the non-conflicting generated internal type information but by outputting an extra "reason" which explains why the type Poly/ML has generated for raiseWeight in Id conflicts with raiseWeight's specification in ID (using again generated internal type information).

In contrast, Fig. 13.1b presents the highlighting that one obtains when using TES on the updated piece of code. The error in focus (highlighted with a darker red) shows that the function `raiseWeight` is involved in a type error. According to `ID`, `raiseWeight` is meant to return an `idlab` which is defined as a record type in `Id`. In `Id`, `raiseWeight` takes a parameter and applies two arguments to `mapWeight`, which itself takes two parameters and applies `updWeight` to two arguments, which itself takes two parameters and applies `cons` to two arguments, which itself takes three arguments (and not two). This means that `raiseWeight` returns a function and not a record type. We therefore obtain a type constructor clash between a record type and an arrow type. In our case, our programming error only concerns `raiseWeight` through its use of `cons` in `updWeight`. Since `cons` takes three parameters now, we have to update the definitions of `updId`, `updLab` and `updWeight`.

We can then quickly spot our programming error and make the necessary changes to get from a well-typed program to another well-typed program (see Fig. 13.1c).

Chapter 14

More TES features to handle more of SML

Let us now present other interesting features of our TES which allow one to handle SML features such as local declarations, type functions, many cases of signatures, functors, non-recursive declarations, type annotations, and non-unary type constructors. Some of these features were already used in the examples provided above. We will now formally present how to handle them.

In this section will will extend Core-TES presented above with additional features. Also, some syntactic forms will sometimes need to be redefined. In this section, we will sometimes write $x \xrightarrow{s} y$ to mean that in the set s, syntactic forms of the form x are replaced by syntactic forms of the form y.

Many examples are provided in the sections below. For readability purposes, we sometimes omit dependencies and the environment $\top$ in these examples.

14.1 Identifier statuses

In the presentation of Core-TES we have syntactically distinguished between value identifiers and datatype constructors by defining two disjoints sets ValVar and DatCon. In SML there is no lexical distinction between, e.g., value variables and datatype constructors. Only one set exists, the set of value identifiers VId which is redefined below. To distinguish between value variables (the only kind of value identifier considered by Haack and Wells), datatype constructors and exception constructors (omitted in this document), SML assigns statuses to value identifiers. The status of an identifier depends on its context and cannot always be inferred from any context smaller than the entire program.

In the subset of SML presented above, datatype (or exception) constructors are: (1) the value identifiers defined in datatype declarations such as `bot` and `cons` in `datatype ’a list = bot | cons of ’a * ’a list`, (2) the value identifiers occurring in patterns or expressions in the scope of such datatype constructors, and (3) the value identifiers taking arguments in patterns such as `x` in `fn x y => y`. In the subset of SML presented above, Value variable are: (1) the recursive functions such as `f` in `val rec f = fn x => x`, and (2) the value identifiers occurring in patterns or expressions in the scope of such value variables.

For example, all of `c`'s occurrences in `datatype t = c; val rec f = fn c => c` are datatype constructors because of `c`'s declaration as a datatype constructor. Whereas in `val rec c = fn x => x; val f = fn c => c`, all occurrences of `c` are value variables because of `c`'s declaration as a recursive function. The sequence of declarations `val rec c = fn x => x; val rec d = fn c x => x` is not valid SML because `c`'s first occurrence forces `c` to be a value variable in its scope but in the pattern `c x`, `c` must be a datatype constructor.

A challenge in dealing with SML's value identifier statuses is that the status of a value identifier occurring in a pattern, such as `x` in `val rec c = fn x => x`, depends on `x`'s status in its context. If we were analysing a complete piece of code where `x` is not declared in the context of `c`'s declaration, `x` would by default be a value variable. In the context of compositional analysis because `x` does not occur in the context of our declaration, we cannot infer `x`'s status. The identifier `x` could either be defined as a datatype constructor, or as a value variable or undefined in a larger piece of code.

Handling identifier statuses in our constraint system and doing context-independent type checking allows a natural reporting of context-sensitive syntax errors as error slices. For example, `x` occurring twice in the pattern in `fn (x, x) => x` is an error only if `x` has value variable status. Context-sensitive syntax errors are discussed in Sec. 17.1.1.

14.1.1 External syntax

We redefine the sets VId, ConBind and Pat defined in Fig. 11.2 to introduce SML's ambiguity on identifier statuses as follows:

$$\begin{array}{rll} vid \in \mathsf{VId} & & \text{(value identifiers)} \\ lvid \in \mathsf{LabId} & ::= & vid^{l}_{\mathtt{u}} \\ cb \in \mathsf{ConBind} & ::= & vid^{l}_{\mathtt{c}} \mid vid\ \mathtt{of}^{\,l}\ ty \\ atpat \in \mathsf{AtPat} & ::= & vid^{l}_{\mathtt{p}} \\ pat \in \mathsf{Pat} & ::= & atpat \mid \lceil lvid\ atpat \rceil^{l} \end{array}$$

For example, if identifier `c` has value variable status in the context and not datatype constructor status, `fn c => (c 1, c ())` has a unique minimal error which is that `c` has a monomorphic type because it is the parameter of the fn-expression but is applied to two expressions with different types: `int` and `unit`[1]. However,

[1] More specifically, the type `unit` is none of the type on which `1` is overloaded. We do not discuss overloading in this section. Overloading is discussed in Sec. 18.3

this error would not exist if the code was preceded by, e.g., `datatype t = c` because the fn-binding would not bind `c`. Instead there would be a minimal error that `c` is declared as a nullary datatype constructor and is applied to an argument in `c 1`. There would also be another similar error involving `c ()` instead.

In addition to the distinction between value identifiers occurring in expressions, occurring non applied in patterns (at a nullary position), and occurring in datatype constructor definitions, we also make the distinction with value identifiers occurring applied in patterns (at a unary position) using the following subscripted forms: $vid^{l}_{\mathtt{u}}$ (see LabId's definition above), where `u` stands for "unary", because we only use this form for identifiers at unary position in patterns which are unary datatype constructors in SML.

We also entirely discard the sets ValVar, DatCon, and LabDatCon. We replace the *ldcon* forms in Term by the *lvid* forms as follows:

$$ldcon \xrightarrow{\mathsf{Term}} lvid$$

14.1.2 Constraint syntax

To compute correct type error slices, we annotate constraints by context dependencies on identifier statuses (see the extension of the set Dependency below). For the `fn`-binding presented above we generate during constraint solving constraints relating the occurrences of `c` annotated by the dependency that `c` is a value variable and not a datatype constructor. These constraints are not generated if a context confirms that `c` must be a datatype constructor. The constraints but not the context dependency are generated if a context confirms that `c` cannot be a datatype constructor. When handling incomplete programs, we report conditional errors (warnings) that assume a sensible default truth status for the dependencies (value identifiers are assumed to be value variables and not datatype constructors[2]). For example, the type error slice displayed in Fig. 10.2 in Sec. 10.4.2 is context-dependent: it depends on `y` and `z` being value variables and not datatype constructors. Our type error reports are then extended with a set of identifier statuses context dependencies: a type error report is then composed by a type error slice, a highlighting, a message explaining the kind of the error, and a set of identifier statuses context dependencies.

We extend our constraint syntax to deal with identifier statuses as follows:

[2] We do not report errors assuming that these identifiers are datatype constructors because in our experience most of the time these identifiers are value variables. We therefore believe that we would cause a great increase in unhelpful reported slices.

$$\begin{array}{rll}
\eta \in \mathsf{IdStatusVar} & & \text{(status variables)} \\
\mathit{ris} \in \mathsf{RawIdStatus} & ::= & \mathtt{v} \mid \mathtt{c} \mid \mathtt{d} \mid \mathtt{u} \mid \mathtt{p} \\
\mathit{is} \in \mathsf{IdStatus} & ::= & \eta \mid \mathit{ris} \mid \langle \mathit{is}, \overline{d} \rangle \\
d \in \mathsf{Dependency} & ::= & \cdots \mid \mathit{vid} \\
\mathit{bind} \in \mathsf{Bind} & ::= & \cdots \mid \downarrow\!\mathit{vid}{=}\mathit{is} \mid \updownarrow\!\mathit{vid}{=}\alpha \\
\mathit{acc} \in \mathsf{Accessor} & ::= & \cdots \mid \uparrow\!\mathit{vid}{=}\eta \\
c \in \mathsf{EqCs} & ::= & \cdots \mid \mathit{is}_1{=}\mathit{is}_2 \\
\mathit{dep} \in \mathsf{Dependent} & ::= & \cdots \mid \langle \mathit{is}, \overline{d} \rangle
\end{array}$$

In our constraint system, an identifier status can either be a status variable η, a raw status *ris* or a status annotated with dependencies of the form $\mathit{is}^{\overline{d}}$ (this complies with design principles (DP1) and (DP2) defined in Sec. 11.10). The raw status `v` is for value variables, e.g., SML requires the recursive function `f` in `val rec f = fn x => x` to be a value variable and not a datatype constructor. Statuses `c` and `d` are for unary and nullary datatype constructors respectively, e.g., the unary constructor `C` in `datatype 'a t = C of 'a` and the nullary constructor `D` in `datatype 'a t = D`. Status `u` is for unconfirmed context-dependent statuses, e.g., in `fn x => x`, the identifier `x` could be a value variable or a nullary datatype constructor, it is therefore considered as a dependent value variable at constraint solving. Intuitively, `u` is a dependent `v`. Finally, status `p` is for unresolvable statuses, e.g., in `let open S in fn x => x end`, `x` could be declared as a value variable as well as a datatype constructor in the free structure `S`. The difference between `u` and `p` is that `u` is used for identifiers for which we know we do not have enough information to resolve their statuses whereas `p` is used for identifiers for which we do not know whether or not we have enough information to resolve their statuses (because information has been filtered out).

The dependency set Dependency is extended to include the value identifier set. In addition to being dependent on program nodes, constraint terms can now also be dependent on value identifiers. An annotated syntactic term of the form $\langle x, \overline{d} \rangle$ depends on the *vid*s in $\overline{d}$ being in the analysed code, value variables and not datatype constructors (the statuses `v` or `u`). Because identifier statuses are resolved at constraint solving, such dependencies (value identifiers) are only generated during constraint solving and not during initial constraint generation. For example, if constraint solving generates the dependent equality constraint $\langle \tau_1{=}\tau_2, \overline{d} \cup \{\mathit{vid}\} \rangle$, then the equality constraint $\tau_1{=}\tau_2$ need only be true if *vid* cannot be a datatype constructor.

Our binder set is extended with binders of the form $\updownarrow\!\mathit{vid}{=}\alpha$. Such a binder is called an *unconfirmed binder* and can, at constraint solving, either be confirmed to be a binder of a value variable and so be turned into a binder of the form $\downarrow\!\mathit{vid}{=}\alpha$, or be turned into an accessor $\uparrow\!\mathit{vid}{=}\alpha$ if it turns out that *vid* is a datatype constructor. Such unconfirmed binders are initially generated for identifiers occurring in patterns at a nullary positions. The status (and the fact that it binds or is bound) of such an identifier is context dependent. Therefore, in order to design a compositional constraint generation algorithm, thanks to these unconfirmed binders, the resolution of identifier statuses is delayed to be dealt with at constraint solving.

Because we introduced status variables we redefine Dum as follows: $\mathsf{Dum} = \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}, \eta_{\mathtt{dum}}\}$, where $\eta_{\mathtt{dum}}$ is a distinguished dummy status variable.

As a matter of fact, because of the restricted language considered in this document, we do not need any other status variable than the dummy status variable $\eta_{\mathtt{dum}}$. We could therefore discard the status variables and introduce a new constant which would play the role of the dummy status variable. This is not true anymore when considering exceptions. For example, in `exception e = e'`, whether `e` is nullary or unary depends on the status of `e'`. Another reason for introducing status variables is that it simplifies the presentation of our system and makes our TES comply with principle (DP1).

If y is a d or a $\overline{d}$ then we write $\downarrow vid \overset{y}{=} \langle\sigma, is\rangle$ for $\downarrow vid \overset{y}{=} is;\downarrow vid \overset{y}{=} \sigma$, and similarly for accessors.

We extend the application of a substitution to a constraint term as follows:

$$(\updownarrow vid{=}\alpha)[sub] = \begin{cases} (\updownarrow vid{=}\alpha[sub]), \text{if } \alpha[sub] \in \mathsf{ITyVar} \\ \text{undefined}, \qquad \text{otherwise} \end{cases}$$

14.1.3 Constraint generation

In order to deal with identifier statuses, Fig. 14.1 redefines the rules (G5), (G6), (G8), (G14), (G16), and (G17) originally introduced in Fig. 11.7 in Sec. 11.5.1. Rule (G6) now generates unconfirmed binders of the form $\updownarrow vid{=}\alpha$ and no status constraint is generated (as opposed to, e.g., rule (G14) which forces the analysed identifier to be a nullary datatype constructor) because in SML, e.g., in `fn x => x`, without any more context, the identifier `x` could be a value variable or a datatype constructor. The status of `x` is then unknown. Because we do not allow a lexical distinction between datatype constructors and value variables anymore, we then replace the two rules (G6) and (G7) by the generation of unconfirmed binders in a unique rule (the new rule (G6)). Because SML requires recursive functions to be value variables (`v`) even when in the scope of a datatype constructor binding, toV (used by rule (G17)) generates a status constraint:

$$\begin{aligned}
\mathsf{toV}(e_1;e_2) &= \mathsf{toV}(e_1);\mathsf{toV}(e_2)\\
\mathsf{toV}(e^{\overline{d}}) &= \mathsf{toV}(e)^{\overline{d}}\\
\mathsf{toV}(\updownarrow vid{=}\alpha) &= (\downarrow vid{=}\langle\alpha, \mathtt{v}\rangle)\\
\mathsf{toV}(e) &= e, \text{ if none of the above applies}
\end{aligned}$$

This function is used at initial constraint generation because it is not context dependent and therefore we do not need to wait constraint solving to apply it.

If not at constraint generation, at constraint solving unconfirmed binders of the form $\updownarrow vid{=}\alpha$ are eventually turned into binders of the form $\downarrow vid{=}\alpha$ or into accessors

Labelled value identifiers ($\mathit{lvid} \rightarrowtriangle \langle\alpha,\ \eta,\ e\rangle$)

(G5) $\mathit{vid}^{l}_{\mathtt{u}} \rightarrowtriangle \langle\alpha,\ \eta,\ \uparrow\!\mathit{vid} \overset{l}{=} \langle\alpha,\eta\rangle\rangle$

Patterns

(G6) $\mathit{vid}^{l}_{\mathtt{p}} \rightarrowtriangle \langle\alpha,\ \updownarrow\!\mathit{vid} \overset{l}{=} \alpha\rangle$

(G8) $\lceil\mathit{lvid}\ \ \mathit{atpat}\rceil^{l} \rightarrowtriangle \langle\alpha,\ (\alpha_1 \overset{l}{=} \alpha_2\to\alpha);(\eta \overset{l}{=} \mathsf{c});e_1;e_2\rangle$
$\Leftarrow\ \mathit{lvid} \rightarrowtriangle \langle\alpha_1,\ \eta,\ e_1\rangle \wedge \mathit{atpat} \rightarrowtriangle \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\alpha)$

Constructor bindings

(G14) $\mathit{vid}^{l}_{\mathtt{c}} \rightarrowtriangle \langle\alpha,\ \downarrow\!\mathit{vid} \overset{l}{=} \langle\alpha,\mathtt{d}\rangle\rangle$

(G16) $\mathit{vid}\ \mathtt{of}^{\,l}\ \mathit{ty} \rightarrowtriangle \langle\alpha_1,\ e;\alpha_2 \overset{l}{=} \alpha\to\alpha_1;\downarrow\!\mathit{vid} \overset{l}{=} \langle\alpha_2,\mathsf{c}\rangle\rangle\ \Leftarrow\ \mathit{ty} \rightarrowtriangle \langle\alpha,\ e\rangle \wedge \mathsf{dja}(e,\alpha_1,\alpha_2)$

Declarations

(G17) $\mathtt{val\ rec}\ \mathit{pat} \overset{l}{=} \mathit{exp} \rightarrowtriangle (\mathit{ev}{=}\mathtt{poly}(\mathsf{toV}(e_1);e_2;(\alpha_1 \overset{l}{=} \alpha_2)));\mathit{ev}^{l}$
$\Leftarrow\ \mathit{pat} \rightarrowtriangle \langle\alpha_1,\ e_1\rangle \wedge \mathit{exp} \rightarrowtriangle \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\mathit{ev})$

Figure 14.1 Constraint generation rules to handle identifier statuses

of the form $\uparrow\!\mathit{vid}{=}\alpha$. In some cases, a status constraint is also generated from an unconfirmed binder.

Because the new constraint generation rule (G5) generates triples, we extend the set InitGen originally defined in Sec. 11.5.1 as follows:

$$\mathit{cg} \in \mathsf{InitGen} ::= \cdots \mid \langle\alpha,\eta,e\rangle$$

We also extend the set LabBind of initially generated binders and the set LabCs of initially generated labelled equality constraints, originally defined in Sec. 11.5.2, as follows:

$$\begin{array}{lll} \mathit{lbind} \in \mathsf{LabBind} & ::= \cdots \mid \updownarrow\!\mathit{vid} \overset{l}{=} \alpha \mid \downarrow\!\mathit{vid} \overset{l}{=} \mathit{ris} \\ \mathit{lc} \quad\ \in \mathsf{LabCs} & ::= \cdots \mid \eta \overset{l}{=} \mathit{ris} \end{array}$$

We also entirely redefine the set PolyEnv of environment initially generated in a `poly` environment, originally defined in Sec. 11.5.2, as follows (we also discard the set InPolyEnv):

$$\mathit{pe} \in \mathsf{PolyEnv} ::= \mathit{lbind} \mid \mathit{lc} \mid \mathit{lacc} \mid \mathit{pe}_1;\mathit{pe}_2$$

Note that the set PolyEnv is much larger than the set of forms generated in `poly` environments by our initial constraint generation algorithm because it allows, e.g., more than one binder and also other binders than value identifier binders. We do so to anticipate the forms generated to handle other features presented below. Note also that the function toPoly is redefined below to work on such forms.

14.1.4 Constraint solving

In Sec. 11.6, we have defined environment application to access identifier static semantics. Let us now define a similar application to access value identifier statuses. Because the two applications are similar we also redefine the application $e(\mathit{id})$.

$$\begin{array}{lll}
\mathsf{toPoly}(\mathit{\Delta}, \downarrow\! vid{=}\tau) & = \mathit{\Delta};(\downarrow\! vid \overset{\overline{d}}{=} \forall\overline{\alpha}.\,\tau'), & \text{if} \begin{cases} \tau' = \mathsf{build}(\mathit{\Delta}, \tau) \\ \overline{\alpha} = (\mathsf{vars}(\tau') \cap \mathsf{ITyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\mathit{\Delta})) \cup \{\alpha_{\mathtt{dum}}\}) \\ \overline{d} = \{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\mathit{\Delta}) \wedge \alpha \in \mathsf{vars}(\tau') \setminus \overline{\alpha}\} \end{cases} \\
\mathsf{toPoly}(\langle u,\ e\rangle, e_0^{\overline{d}}) & = \langle u',\ (e;\mathsf{diff}(e, e')^{\overline{d}})\rangle, & \text{if } \mathsf{toPoly}(\langle u,\ e\rangle, e_0) = \langle u',\ e'\rangle \\
\mathsf{toPoly}(\mathit{\Delta}, e_1;e_2) & = \mathsf{toPoly}(\mathit{\Delta}', e_2), & \text{if } \mathsf{toPoly}(\mathit{\Delta}, e_1) = \mathit{\Delta}' \\
\mathsf{toPoly}(\mathit{\Delta}, e) & = \mathit{\Delta};e, & \text{if none of the above applies}
\end{array}$$

Figure 14.2 Monomorphic to polymorphic environment function

First, let $k \in \mathsf{AppKind} ::= \mathtt{T} \mid \mathtt{S}$. The applications $e(id)$ to access identifier static semantics, and $e[id]$ and $\mathit{\Delta}[id]$ to access value identifier statuses are defined via the function app as follows:

$$\mathit{\Delta}(id) = \mathsf{app}(\mathit{\Delta}, id, \mathtt{T}) \qquad \mathit{\Delta}[id] = \mathsf{app}(\mathit{\Delta}, id, \mathtt{S}) \qquad e[id] = \langle\varnothing,\ e\rangle[id]$$

$$\begin{array}{ll}
\mathsf{app}(\langle u,\ \downarrow\! id{=}x\rangle, id, \mathtt{T}) & = x, \text{ if } x \notin \mathsf{IdStatus} \\
\mathsf{app}(\langle u,\ \downarrow\! id{=}x\rangle, id, \mathtt{S}) & = x, \text{ if } x \in \mathsf{IdStatus} \\
\mathsf{app}(\langle u,\ e^{\overline{d}}\rangle, id, k) & = \mathsf{collapse}((\mathsf{app}(\langle u,\ e\rangle, id, k))^{\overline{d}}) \\
\mathsf{app}(\langle u,\ (e_1;e_2)\rangle, id, k) & = \begin{cases} x, \text{if } \mathsf{app}(\langle u,\ e_2\rangle, id, k) = x \text{ or } \mathsf{shadowsAll}(\langle u,\ e_2\rangle) \\ \mathsf{app}(\langle u,\ e_1\rangle, id, k), \text{otherwise} \end{cases} \\
\mathsf{app}(\langle u,\ ev\rangle, id, k) & = \begin{cases} \mathsf{app}(\langle u,\ e\rangle, id, k), \text{if } u(ev) = e \\ \text{undefined, otherwise} \end{cases}
\end{array}$$

Because adding statuses to our system can lead to new status errors we extend the set of error kinds as follows:

$$ek \in \mathsf{ErrKind} ::= \cdots \mid \mathtt{statusClash}(is_1, is_2)$$

Because of we have added binders to associate statuses with identifiers, toPoly can now be applied to an environment composed by such binders. We extends toPoly in Fig. 14.2.

Fig. 14.3 extends our constraint solver to deal with our new constraint terms.

Two identifier statuses are incompatible iff a unary datatype constructor, occurring in a pattern, is bound to a (context-dependent or independent) value variable as in `let val rec f = fn x => x in fn (f x) => x end` where `f`'s first occurrence is a value variable and `f`'s second occurrence is a unary datatype constructor (taking an argument in a pattern); or if a nullary value identifier in a pattern is bound to a unary datatype constructor as in `let datatype t = x of int in fn x => x end`. The $\mathsf{compatible}$ relation is defined as follows:

$$\mathsf{compatible}(is_1, is_2) \Leftrightarrow \{is_1, is_2\} \notin \{\{\mathtt{c}, \mathtt{v}\}, \{\mathtt{c}, \mathtt{u}\}, \{\mathtt{c}, \mathtt{p}\}\}$$

Status compatibility is checked by constraint solving rules (S7) and (S8) defined in Fig. 14.3. Rule (S8) is only defined on raw statuses because rule (S2) removes dependencies on, among other things, statuses.

The status `p` is used to catch errors in pieces of code such as the let-expression `let open S in fn x => fn x y => y end` where `x` occurs both at a nullary position and

equality simplification

(S7) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \mathit{is}_1{=}\mathit{is}_2) \quad \rightarrow \mathtt{err}(\langle \mathtt{statusClash}(\mathit{is}_1, \mathit{is}_2), \overline{d}\rangle)$, if $\neg\mathsf{compatible}(\mathit{is}_1, \mathit{is}_2)$

(S8) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \mathit{ris}_1{=}\mathit{ris}_2) \rightarrow \mathtt{succ}(\mathit{\Delta})$, if $\mathsf{compatible}(\mathit{ris}_1, \mathit{ris}_2)$

binders

(B2) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \updownarrow \mathit{vid}{=}\alpha) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \uparrow \mathit{vid}{=}\langle \alpha, \mathsf{ifNotDum}(\alpha, \mathtt{u})\rangle)$,
if $\mathsf{strip}(\mathit{\Delta}[\mathit{vid}]) \in \{\mathtt{c}, \mathtt{d}\}$

(B3) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \updownarrow \mathit{vid}{=}\alpha) \rightarrow \mathtt{succ}(\mathit{\Delta};(\downarrow \mathit{vid} \overset{\overline{d}\cup\overline{d}'}{=\!=\!=} \alpha))$,
if $\mathsf{collapse}(\mathit{\Delta}[\mathit{vid}]^{\varnothing}) = \mathtt{v}^{\overline{d}'}$

(B4) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \updownarrow \mathit{vid}{=}\alpha) \rightarrow \mathtt{succ}(\mathit{\Delta};(\downarrow \mathit{vid} \overset{\overline{d}\cup\{\mathit{vid}\}}{=\!=\!=\!=} \langle \alpha, \mathsf{ifNotDum}(\alpha, \mathtt{u})\rangle))$,
if $\mathsf{strip}(\mathit{\Delta}[\mathit{vid}]) = \mathtt{u} \vee (\neg\mathsf{shadowsAll}(\mathit{\Delta}) \wedge \mathit{\Delta}[\mathit{vid}]$ undefined$)$

(B5) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \updownarrow \mathit{vid}{=}\alpha) \rightarrow \mathtt{succ}(\mathit{\Delta};(\downarrow \mathit{vid} \overset{\overline{d}}{=} \langle \alpha_{\mathtt{dum}}, \mathsf{ifNotDum}(\alpha, \mathtt{p})\rangle))$,
if $\mathsf{strip}(\mathit{\Delta}[\mathit{vid}]) \in \mathsf{Var} \cup \{\mathtt{p}\} \vee (\mathsf{shadowsAll}(\mathit{\Delta}) \wedge \mathit{\Delta}[\mathit{vid}]$ undefined$)$

accessors

(A2) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \uparrow \mathit{id}{=}v) \ \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, v{=}x)$,
if $\mathit{\Delta}(\mathit{id}) = x \wedge \mathsf{strip}(x)$ is not of the form $\forall\overline{\alpha}.\,\tau \wedge v \notin \mathsf{IdStatus}$

(A3) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \uparrow \mathit{id}{=}v) \ \rightarrow \mathtt{succ}(\mathit{\Delta})$,
if $(v \in \mathsf{IdStatus} \wedge \mathit{\Delta}[\mathit{id}]$ undefined$) \vee (v \notin \mathsf{IdStatus} \wedge \mathit{\Delta}(\mathit{id})$ undefined$)$

(A4) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \uparrow \mathit{vid}{=}\eta) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \eta{=}\mathit{is})$, if $\mathit{\Delta}[\mathit{vid}] = \mathit{is}$

Figure 14.3 Constraint solving rules to handle identifier statuses

at a unary position in patterns (applied and not applied). The identifier `x` cannot be a value variable because it is applied in a pattern. It cannot be a datatype constructor either because it would be both nullary and unary.

Context dependencies are solved during constraint solving. An unconfirmed binder of the form $\updownarrow \mathit{vid}{=}\alpha$ either turns into a binder of the form $\downarrow \mathit{vid}{=}\alpha$ or an accessor of the form $\uparrow \mathit{vid}{=}\alpha$ using one of these rules: (B2)-(B5). These rules use the function ifNotDum that ensures that a dummy status binder cannot bind something else than a dummy status and therefore cannot be involved in an error: $\mathsf{ifNotDum}(x, \mathit{is}) = \eta_{\mathtt{dum}}$ if $\mathsf{strip}(x) \in \mathsf{Dum}$, and is otherwise. Rule (B2) discards binders generated under unsatisfied context dependencies, e.g., in `let datatype t = x in fn x => x end`, `x`'s second occurrence does not bind `x`'s third occurrence because of `x`'s declaration as a datatype constructor. The unconfirmed binder is then turned into an accessor. In all three other rules, the unconfirmed binder is turned into a confirmed one. Rule (B3) validates context dependencies, e.g., in `val rec x = fn x => x`, `x` is confirmed to be a value variable because `x`'s second occurrence is in the scope of `x`'s first occurrence which is a recursive function, and so in SML is forced to be a value variable and not a datatype constructor. Rule (B4) generates context dependencies, e.g., in `fn x => x`, because `x` can be a value variable as well as a datatype constructor then `x`'s second occurrence is bound to `x`'s first occurrence under the context dependency that `x` is not a datatype constructor. Rule (B5) generates dummy environments when there is not enough information to check whether a context dependency is satisfied or not, e.g., in `let open S in fn x => x end`, if `S` is free, it might declare `x` as a datatype constructor or as a recursive function. Thus, we do not allow `x` to be a monomorphic binder but we still generate a dummy binder to catch status clashes. For example, if instead of the second occurrence of `x` we had `fn (x y) => y` where `x` is a unary datatype constructor, we would then have `x` occurring in patterns both at a nullary position and a unary position.

Because binders of the form $\downarrow vid{=}is$ can now occur in constraint solving contexts (in e in $\langle u,\ e\rangle$), we extend the binder forms generated at constraint solving, originally defined in Sec. 11.6.6, as follows:

$$sbind \in \mathsf{SolvBind} ::= \cdots \mid \downarrow vid{=}is$$

14.1.5 Constraint filtering (Minimisation and enumeration)

We extend our filtering function as follows:

$$\begin{aligned}\mathsf{dum}(\updownarrow id{=}x) &= (\updownarrow id{=}\mathsf{toDumVar}(x))\\ \mathsf{toDumVar}(is) &= \eta_{\mathtt{dum}}\end{aligned}$$

14.1.6 Slicing

Because our constraint generator generates a triple of the form $\langle \alpha, \eta, e\rangle$ for labelled value identifiers of the form $vid^{l}_{\mathtt{u}}$, we need to introduce a new form of dot term as follows:

$$\mathsf{LabId} ::= \cdots \mid \mathtt{dot\text{-}i}(\overrightarrow{term})$$

We define the new constraint generation rule for terms of the form $\mathtt{dot\text{-}i}(\overrightarrow{term})$ as follows:

$$\begin{gathered}(\mathsf{G28})\ \mathtt{dot\text{-}i}(\langle term_1, \ldots, term_n\rangle) \mathrel{-\!\triangleright} \langle \alpha,\ \eta,\ [e_1;\cdots;e_n]\rangle \Leftarrow\\ term_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge term_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1, \ldots, e_n, \eta, \alpha)\end{gathered}$$

We modify the set of classes Class as follows:

$$\mathtt{lDcon} \xrightarrow{\mathsf{Class}} \mathtt{lVid}$$

We extend the set of dot markers Dot as follows:

$$\mathsf{Dot} ::= \cdots \mid \mathtt{dotI}$$

We extend the function getDot that associates dot markers with node kinds as follows:

$$\mathsf{getDot}(\langle \mathtt{lVid}, prod\rangle) = \mathtt{dotI}$$

Fig. 14.4 extends the function toTree that transforms *terms* into *trees*.

Fig. 14.5 slightly modifies rule ($\mathsf{SL1}$) of our slicing algorithm defined in Fig. 11.17. The only difference with rule ($\mathsf{SL1}$) defined in Fig. 11.17 is the addition of the condition "or $\mathsf{pattern}(\mathsf{sl}_1(\overrightarrow{tree}(0), \overline{l}))$". We add this special treatment for patterns

Labelled value identifiers	$\mathsf{toTree}(vid^{l}_{\mathtt{u}})$	$= \langle\langle \mathtt{lVid}, \mathtt{id}\rangle, l, \langle vid\rangle\rangle$
Constructor bindings	$\mathsf{toTree}(vid^{l}_{\mathtt{c}})$	$= \langle\langle \mathtt{conbind}, \mathtt{id}\rangle, l, \langle vid\rangle\rangle$
	$\mathsf{toTree}(vid\ \mathtt{of}^{\,l}\ ty)$	$= \langle\langle \mathtt{conbind}, \mathtt{conbindOf}\rangle, l, \langle vid, \mathsf{toTree}(ty)\rangle\rangle$
Patterns	$\mathsf{toTree}(vid^{l}_{\mathtt{p}})$	$= \langle\langle \mathtt{atpat}, \mathtt{id}\rangle, l, \langle vid\rangle\rangle$
	$\mathsf{toTree}(\lceil lvid\ \ atpat\rceil^{l})$	$= \langle\langle \mathtt{pat}, \mathtt{app}\rangle, l, \langle \mathsf{toTree}(lvid), \mathsf{toTree}(atpat)\rangle\rangle$
Dot terms	$\mathsf{toTree}(\mathtt{dot\text{-}i}(\overrightarrow{term}))$	$= \langle \mathtt{dotI}, \mathsf{toTree}(\overrightarrow{term})\rangle$

Figure 14.4 Extension of toTree to deal with identifier status

$$(\mathsf{SL1})\ \mathsf{sl}(\langle node, l, \overrightarrow{tree}\rangle, \overline{l})$$
$$= \begin{cases} \langle node, l, \mathsf{sl}_1(\overrightarrow{tree}, \overline{l})\rangle, & \text{if } (l \in \overline{l} \text{ and } \mathsf{getDot}(node) \neq \mathtt{dotS}) \text{ or } \mathsf{pattern}(\mathsf{sl}_1(\overrightarrow{tree}(0), \overline{l})) \\ \langle node, l, \mathsf{tidy}(\mathsf{sl}_1(\overrightarrow{tree}, \overline{l}))\rangle, & \text{if } l \in \overline{l} \text{ and } \mathsf{getDot}(node) = \mathtt{dotS} \\ \langle dot, \mathsf{flat}(\mathsf{sl}_2(\overrightarrow{tree}, \overline{l}))\rangle, & \text{otherwise, and where } dot = \mathsf{getDot}(node) \end{cases}$$

Figure 14.5 Slicing algorithm rule to handle identifier status

because in our system, at constraint solving, we do not record the label associated with the fn-expression when generating the following type error slice (the error being that `x` is declared as a unary datatype constructor and occurs at a nullary position in a pattern):

```
⟨..datatype ⟨..⟩ = x of ⟨..⟩
 ..fn x => ⟨..⟩..⟩
```

This is because the unconfirmed binder generated for `x`'s occurrence in the fn-expression turns into an accessor at constraint solving (`x` being declared as a datatype constructor) and this accessor can directly refer to `x`'s binder without using any constraint labelled by the label associated with the fn-expression. This applies for any accessor generated for an identifier occurring in a pattern.

14.2 Local declarations

14.2.1 External syntax

First, let us extend our external syntax with local declarations as follows:

$$dec ::= \cdots \mid \mathtt{local}^{l}\ dec_1\ \mathtt{in}\ dec_2\ \mathtt{end}$$

For example,

```
val x = true
local val x = 1 in val y = x end
val z = x + 1
```

is untypable because `x`'s last occurrence is bound to its first occurrence and not to its second (assuming that `+` is the one from the Standard ML basis library).

Let us present another example:

```
        val x = true
(EX2)   local val x = 1 in val y = x end
        val z = fn w => (w y, w x)
```

Only `z`'s declaration differs from the previous example. This piece of code is also untypable because `w` has a monomorphic type and is applied to `y` which is an integer and `x` which is a Boolean. This example will be reused later in this section.

14.2.2 Constraint syntax

We extend constraint/environments with local environments as follows:

$$e ::= \cdots \mid \mathtt{loc}\, e_1\, \mathtt{in}\, e_2$$

The meaning of such an environment is that it builds an environment e_2 which depends on e_1 and only exports e_2's binders, i.e., only e_2's binders can be accessed from outside the local environment. Such environments differ from environments of the form $e_1;e_2$ because an environment of the form $e_1;e_2$ builds a new environment from both e_1 and e_2 and exports both e_1's binders not shadowed by e_2 and e_2's binders.

Environments of the form $[e]$ are not enough to handle local declarations because they do not allow one to partially export an environment. The requirement imposed by a local declaration of the from $\mathtt{loc}\, e_1\, \mathtt{in}\, e_2$ is that only e_1 and e_2 should be able to access e_1's binders. Unfortunately, $[e_1;e_2]$ does not export e_2's binders, and $[e_1];e_2$ does not allow e_2's accessors to refer to e_1's binders. The solution was to introduce environments of the form $\mathtt{loc}\, e_1\, \mathtt{in}\, e_2$.

Note that these environments are not only used to generate constraints for local declarations, they are also used to, e.g., handle bindings of external type variables (see Sec. 14.3). In Sec. 11 we allow binding occurrences of explicit type variables to have a larger scope than they should, which is harmless in the small language of Sec. 11, but needs to be (and is) fixed to work for full SML in Sec. 14.3.

We extend the application of a substitution to a constraint term as follows:

$$(\mathtt{loc}\, e_1\, \mathtt{in}\, e_2)[sub] = \mathtt{loc}\, (e_1[sub])\, \mathtt{in}\, (e_2[sub])$$

14.2.3 Constraint generation

Fig. 14.6 extends our constraint generator with a rule to handle local declarations.

Because our initial constraint generation algorithm generates new forms of constraints, we extend the ge forms as follows (see Sec. 11.5.2):

$$ge ::= \cdots \mid \mathtt{loc}\, ge_1\, \mathtt{in}\, ge_2$$

$\boxed{\textsf{Declaration}}$ $(\textsf{G29})\ \texttt{local}^{l}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \mathrel{-\!\!\rhd} (\mathit{ev}{=}e_1);\texttt{loc}\ \mathit{ev}^{l}\ \texttt{in}\ e_2$
$\Leftarrow\ \mathit{dec}_1 \mathrel{-\!\!\rhd} e_1 \wedge \mathit{dec}_2 \mathrel{-\!\!\rhd} e_2 \wedge \textsf{dja}(e_1, e_2, \mathit{ev})$

Figure 14.6 Constraint generation rule for local declarations

local environments

$(\textsf{L1})\ \texttt{slv}(\langle u,\ e\rangle, \overline{d}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{succ}(\varDelta),$ if $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, e_1) \rightarrow^{*} \texttt{succ}(\langle u',\ e'\rangle)$
$\wedge\ \texttt{slv}(\langle u',\ e'\rangle, \overline{d}, e_2) \rightarrow^{*} \texttt{succ}(\langle u'',\ e''\rangle)$
$\wedge\ \varDelta = \langle u'',\ e;\textsf{diff}(e', e'')\rangle$

$(\textsf{L2})\ \texttt{slv}(\langle u,\ e\rangle, \overline{d}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{err}(er),$ if $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, e_1) \rightarrow^{*} \texttt{succ}(\langle u',\ e'\rangle)$
$\wedge\ \texttt{slv}(\langle u',\ e'\rangle, \overline{d}, e_2) \rightarrow^{*} \texttt{err}(er)$

$(\textsf{L3})\ \texttt{slv}(\langle u,\ e\rangle, \overline{d}, \texttt{loc}\ e_1\ \texttt{in}\ e_2) \rightarrow \texttt{err}(er),$ if $\texttt{slv}(\langle u,\ e\rangle, \overline{d}, e_1) \rightarrow^{*} \texttt{err}(er)$

Figure 14.7 Constraint solving rules for local declarations

The forms generated by our initial constraint generator are in fact more restricted than that, but we already anticipate the forms generated by further extensions such as for type functions.

14.2.4 Constraint solving

Fig. 14.7 extends our constraint solver to handle local declarations.

The most important rule is rule (L1). The two other ones are to handle the failure of solving one of the two environments composing a local environment of the form $\texttt{loc}\ e_1\ \texttt{in}\ e_2$.

When solving an environment of this form, first we solve e_1 and if it leads to a success state $\texttt{succ}(\varDelta_1)$, $\varDelta_1$ is used to solve e_2 so that the binders generated while solving e_1 are made available when solving e_2. If solving e_2 leads to a success state $\texttt{succ}(\varDelta_2)$, solving $\texttt{loc}\ e_1\ \texttt{in}\ e_2$ leads then to a success state $\texttt{succ}(\langle u,\ e\rangle)$ where u is the unifier from $\varDelta_2$ and e is the environment from $\varDelta_2$ where we forget the environments generated by the constraint solver while solving e_1.

14.2.5 Constraint filtering (Minimisation and enumeration)

We extend our filtering function as follows:

$$\textsf{filt}(\texttt{loc}\ e_1\ \texttt{in}\ e_2, \overline{l}_1, \overline{l}_2) = \texttt{loc}\,\textsf{filt}(e_1, \overline{l}_1, \overline{l}_2)\ \texttt{in}\,\textsf{filt}(e_1, \overline{l}_1, \overline{l}_2)$$

14.2.6 Slicing

Finally, our slicing algorithm does not need to be extended but we need to update the tree syntax for programs as follows:

$$\textsf{Prod} ::= \cdots \mid \texttt{decLoc}$$

We also need to extend the $\textsf{toTree}$ function that associates trees of the form *tree* with terms of the form *term* as follows:

$$\mathsf{toTree}(\texttt{local}^{l}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end}) = \langle\langle\texttt{dec},\texttt{decLoc}\rangle, l, \langle\mathsf{toTree}(dec_1),\mathsf{toTree}(dec_2)\rangle\rangle$$

14.2.7 Minimality

Let us illustrate what would happen if we were not generating an extra labelled environment variable in rule (G29). Consider example (EX2) presented above. With our current system, we would obtain a type error slice involving the local declaration itself in addition to the nested declarations of `x` and `y` as follows:

```
⟨..val x = true
 ..local val x = 1 in val y = x end
 ..val z = fn w => ⟨..w y..w x..⟩..⟩
```

If we were not to label the environment variable in rule (G29) or if we were to use e_1 instead of ev^l in the local constraint (and omit $ev{=}e_1$ which becomes useless), then we would obtain a type error slice that would look like:

```
⟨..val x = true
 ..val x = 1
 ..val y = x
 ..val z = fn w => ⟨..w y..w x..⟩..⟩
```

which is typable and therefore is not a minimal type error slice of example (EX2). As a matted of fact, in this last slice, both bound occurrences of `x` are bound to `x`'s second declaration.

Therefore, the extra initially generated labelled environment variable is necessary to force, when solving an environment of the form $\texttt{loc}\, e_1\, \texttt{in}\, e_2$, e_1's binders to be dependent on the label of the local declaration for which the local environment has been generated before making them accessible to e_2.

14.3 Type declarations

14.3.1 External syntax

First, let us extend our external syntax with type functions as follows:

$$\mathsf{Dec} ::= \cdots \mid \texttt{type}\ dn \overset{l}{=} ty$$

For example,

```
type 'a t = 'a -> 'a -> 'a
datatype 'a u = U of 'a t
val x = U (fn x => x)
```

is untypable because `U` is applied to the identity function which cannot have the type `'a -> 'a -> 'a`.

Note that in SML, type declarations are not recursive while datatype declarations are. For example, in `type t = t -> t`, the two last occurrences of `t` are free, especially, they are not bound to `t`'s first occurrence. However, in `datatype t = C of t -> t`, the two last occurrences of `t` are bound to `t`'s first occurrence.

We still use *dn* (standing for "datatype name") for type functions. This name is not suitable anymore because it is not only used for datatype declarations only but also for type declarations. However, for lack of a better name, we keep this name in this section.

14.3.2 Constraint syntax

We extend our constraint system with pseudo type functions:

$$\begin{array}{lll} \mathit{tfi} \in \mathsf{TypFunIns} & ::= & \tau_1.\tau_2 \\ \mu \in \mathsf{ITyCon} & ::= & \cdots \mid \Lambda\alpha.\,\tau \end{array}$$

We explain below why, even though we use the symbol Λ, constraint terms of the form $\Lambda\alpha.\,\tau$ are called pseudo type functions and not type functions.

We also introduce quantified internal type constructors as follows:

$$\kappa \in \mathsf{TyConSem} ::= \mu \mid \forall\overline{\alpha}.\,\mu \mid \langle\kappa, \overline{d}\rangle$$

We modify type constructor binders as follows:

$$\downarrow tc{=}\mu \xrightarrow{\mathsf{Bind}} \downarrow tc{=}\kappa$$

A internal type constructor of the form $\Lambda\alpha.\,\tau$ is called a pseudo type function and is not a type function as defined in The Definition of Standard ML [107]. At initial constraint solving, an internal type constructor of the form $\Lambda\alpha.\,\tau$ is a type function only when the constraints on τ have all been solved and when τ is fully built up. As a matter of fact, in $\Lambda\alpha.\,\tau$, the parameter α can be connected to τ via constraints. For example, at initial constraint generation we generate for a type declaration of the form `type 'a t = 'a`, an environment of the form (for readability purposes, we have omitted labels as well as some constraints):

$$(\delta{=}\Lambda\alpha_1.\,\alpha_2);\mathtt{loc}\,(\downarrow\mathtt{'a}{=}\alpha_1)\,\mathtt{in}\,(\uparrow\mathtt{'a}{=}\alpha_2;\downarrow\mathtt{t}{=}\delta)$$

The internal type constructor $\Lambda\alpha_1.\,\alpha_2$ is not a type function. It is a type function only via constraints. However, at constraint solving, if no constraint is filtered out, then the binder $\downarrow\mathtt{t}{=}\forall\varnothing.\,\Lambda\alpha_1.\,\alpha_1$ is eventually generated, where $\Lambda\alpha_1.\,\alpha_1$ is a type function.

We introduce quantified internal type constructors of the form $\forall\overline{\alpha}.\,\mu$ because now internal type variables can occur in internal type constructors via pseudo type functions. For example, the type function $\Lambda\alpha_1.\,\alpha_2$ (where $\alpha_1 \neq \alpha_2$) is generated at constraint solving when solving the constraints generated for the type declaration `type 'a t = ⟨..⟩`. Because α_2 is not bound by the type function, we need to quantify it so that it will be renamed for each accessor to `t`. We then eventually generate the following binder for `t` (where we omit dependencies for readability purposes): $\downarrow$`t`$=\forall\{\alpha_2\}.\,\Lambda\alpha_1.\,\alpha_2$. If we were to not quantify α_2 in our example, we would obtain an error for the following piece of code (because α_2 would be constrained to be equal to `bool` and `unit`):

```
type 'a t = ⟨..⟩
val x = true : bool t
val y = () : unit t
```

But one can observe that this incomplete piece of code becomes typable when replacing ⟨..⟩ by `'a`.

We also define the following forms where $\mathsf{TyFun} \subseteq \mathsf{LabName}$ and $\mathsf{App} \subseteq \mathsf{ITy}$:

$$\begin{aligned} &\mathit{tyf} \in \mathsf{TyFun} ::= \delta \mid \Lambda\alpha.\,\tau \mid \langle \mathit{tyf}, \overline{d}\rangle \\ &\mathit{app} \in \mathsf{App} \quad ::= \tau\ \mathit{tyf} \end{aligned}$$

These forms will be used to state side conditions in the extension of our constraint solver below.

We extend the application of a substitution to a constraint term as follows:

$$(\Lambda\alpha.\,\tau)[\mathit{sub}] = \Lambda\alpha.\,\tau[\{\alpha\} \lhd \mathit{sub}], \text{if } \alpha \notin \mathsf{vars}(\{\alpha\} \lhd \mathit{sub})$$

14.3.3 Constraint generation

Fig. 14.8 modifies the rules for datatype names (G13) and datatype declarations (G18), and defines a new rule (G30) for type function declarations. The environment e_1 is generated before e_2 in rule (G18) to handle the recursivity of datatype declarations and it is generated after e_2 in rule (G30) to handle the non-recursivity of type declarations. Note the use of local environments of the form `loc` e_1 `in` e_2 in rules (G18) and (G30). They are used to handle binding occurrences of explicit type variables. In rule (G30) the environment e_1 is not required to be generated inside the local environment. It could as well be generated after the local environment.

Because the new constraint generation rule (G13) associates tuples of the form $\langle\delta, \alpha, e_1, e_2\rangle$ with dns, we extend the set InitGen originally defined in Sec. 11.5.1 and extended in Sec. 14.1.3 as follows:

$$\mathit{cg} \in \mathsf{InitGen} ::= \cdots \mid \langle\delta, \alpha, e_1, e_2\rangle$$

Because our initial constraint generation algorithm generates new forms of type constructor binders, we replace the initially generated type constructor binders as follows:

$\boxed{\textsf{Datatype names } (dn \rightarrowtriangle \langle \delta, \alpha, e_1, e_2 \rangle)}$

$(\mathsf{G13})\ \lceil tv\ \ tc \rceil^{l} \rightarrowtriangle \langle \delta, \alpha, {\downarrow}tc \overset{l}{=} \delta, {\downarrow}tv \overset{l}{=} \alpha \rangle$

$\boxed{\textsf{Declarations}}$

$(\mathsf{G18})\ \texttt{datatype}\ dn \overset{l}{=} cb \rightarrowtriangle (ev{=}((\delta \overset{l}{=} \gamma);(\alpha_2 \overset{l}{=} \alpha_1\, \gamma);e_1;\texttt{loc}\ e_1'\ \texttt{in}\,\texttt{poly}(e_2)));ev^{l}$
$\quad \Leftarrow\ dn \rightarrowtriangle \langle \delta, \alpha_1, e_1, e_1' \rangle \land cb \rightarrowtriangle \langle \alpha_2,\ e_2 \rangle \land \mathsf{dja}(e_1, e_2, \gamma, ev)$

$(\mathsf{G30})\ \texttt{type}\ dn \overset{l}{=} ty \rightarrowtriangle (ev{=}((\delta \overset{l}{=} \Lambda\alpha_1 .\, \alpha_2);\texttt{loc}\ e_1'\ \texttt{in}\,(e_2;e_1)));ev^{l}$
$\quad \Leftarrow\ dn \rightarrowtriangle \langle \delta, \alpha_1, e_1, e_1' \rangle \land ty \rightarrowtriangle \langle \alpha_2,\ e_2 \rangle \land \mathsf{dja}(e_1, e_2, ev)$

Figure 14.8 Constraint generation rules for type functions

$${\downarrow}tc \overset{l}{=} \gamma \xrightarrow{\mathsf{LabBind}} {\downarrow}tc \overset{l}{=} \delta$$

The extension of our constraint generation algorithm defined in Fig. 14.8 also generates forms of equality constraints that were not generated at initial constraint generation by the algorithm defined so far. We introduce ShallowTyCon and extend LabCs as follows:

$$\begin{array}{lll} stc \in \mathsf{ShallowTyCon} & ::= & \gamma \mid \Lambda\alpha .\, \alpha' \\ lc\ \ \in \mathsf{LabCs} & ::= & \cdots \mid \delta \overset{l}{=} stc \end{array}$$

14.3.4 Constraint solving

Because we added internal type constructors of the form $\Lambda\alpha .\, \tau$, we need to update our building function as follows:

$$\mathsf{build}(u, \Lambda\alpha .\, \tau) = \Lambda\alpha' .\, \mathsf{build}(u, \tau), \text{ if } \mathsf{build}(u, \alpha) = \alpha'$$

We define the free internal type variable of an internal type or an internal type constructor as follows (used by rule (B6) in Fig. 14.9 presented below):

$$\begin{array}{ll} \mathsf{freevars}(\alpha) & = \{\alpha\} \setminus \mathsf{Dum} \\ \mathsf{freevars}(\tau_1 {\rightarrow} \tau_2) & = \mathsf{freevars}(\tau_1) \cup \mathsf{freevars}(\tau_2) \\ \mathsf{freevars}(\tau\, \mu) & = \mathsf{freevars}(\mu) \cup \mathsf{freevars}(\tau) \\ \mathsf{freevars}(\Lambda\alpha .\, \tau) & = \mathsf{freevars}(\tau) \setminus \{\alpha\} \\ \mathsf{freevars}(x^{\overline{d}}) & = \mathsf{freevars}(x) \\ \mathsf{freevars}(x) & = \varnothing, \text{ if none of the above applies} \end{array}$$

Fig. 14.9 extends our constraint solver to handle internal type constructors of the form $\Lambda\alpha .\, \tau$. We replace the two rules (S3) and (S5) defined in Fig. 11.10 by the new rules (S9)-(S13).

Accessor rules (A1) and (A2), originally defined in Fig. 11.10 (rule (A2) is redefined in Fig. 14.3), are redefined to handle universally quantified internal type constructors as well as type schemes. Also, the new binder rule (B6) is introduced to generate universally quantifier internal type constructors.

Note that equality constraints of the forms $(\Lambda\alpha .\, \tau{=}\mu)$ or $(\mu{=}\Lambda\alpha .\, \tau)$, where μ is not a variable, are never generated neither at initial constraint generation nor

equality simplification

(S9) $\mathtt{slv}(\mathit{\Delta},\overline{d},\tau_2\,\mu{=}\tau) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d},\tau'[\{\alpha\mapsto\tau_2\}]{=}\tau)$, if $\mathsf{collapse}(\mu^{\varnothing}) = (\Lambda\alpha.\,\tau_1)^{\overline{d}'}$ $\wedge\ \tau' = \mathsf{build}(\mathit{\Delta},\tau_1^{\overline{d}'})$

(S10) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},\tau_1\,\mu{=}\tau) \rightarrow \mathtt{succ}(\langle u,\ e\rangle)$, if $\mathsf{collapse}(\mu^{\varnothing}) = \delta^{\overline{d}'} \wedge \delta \notin \mathsf{dom}(u)$

(S11) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},\tau_1\,\mu{=}\tau) \rightarrow \mathtt{slv}(\langle u,\ e\rangle,\overline{d}\cup\overline{d}',\tau_1\,\mu'{=}\tau)$, if $\mathsf{collapse}(\mu^{\varnothing}) = \delta^{\overline{d}'} \wedge u(\delta) = \mu'$

(S12) $\mathtt{slv}(\mathit{\Delta},\overline{d},\tau_1\,\mu_1{=}\tau_2\,\mu_2) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d}_1\cup\overline{d}_2,\gamma_1{=}\gamma_2;\tau_1{=}\tau_2)$, if $\mathsf{collapse}(\mu_1^{\overline{d}}) = \gamma_1^{\overline{d}_1}$ $\wedge\ \mathsf{collapse}(\mu_2^{\varnothing}) = \gamma_2^{\overline{d}_2}$

(S13) $\mathtt{slv}(\mathit{\Delta},\overline{d},\tau_1{=}\tau_2) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d},\mu{=}\mathtt{ar})$, if $\{\tau_1,\tau_2\} = \{\tau\,\mu,\tau_0{\rightarrow}\tau_0'\}$ $\wedge\ \mathsf{strip}(\mu) \in \mathsf{TyConName}$

equality constraint reversing

(R) $\mathtt{slv}(\mathit{\Delta},\overline{d},x{=}y) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d},y{=}x)$, if $s = \mathsf{Var}\cup\mathsf{Dependent}\cup\mathsf{App} \wedge y \in s \wedge x \notin s$,

binders

(B1) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},{\downarrow}\mathit{id}{=}x) \rightarrow \mathtt{succ}(\langle u,\ e\rangle;({\downarrow}\mathit{id} \overset{\overline{d}}{=} x))$, if $\mathit{id} \notin \mathsf{TyCon}$

(B6) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},{\downarrow}\mathit{tc}{=}\mu) \rightarrow \mathtt{succ}(\langle u,\ e\rangle;({\downarrow}\mathit{tc} \overset{\overline{d}}{=} \forall\overline{\alpha}.\,\mu'))$, if $\mu' = \mathsf{build}(u,\mu) \wedge \overline{\alpha} = \mathsf{freevars}(\mu')$

accessors

(A1) $\mathtt{slv}(\mathit{\Delta},\overline{d},{\uparrow}\mathit{id}{=}v) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d}\cup\overline{d}',v{=}x[\mathit{ren}])$,
if $\mathit{\Delta}(\mathit{id}) = (\forall\overline{v}.\,x)^{\overline{d}'} \wedge \mathsf{dom}(\mathit{ren}) = \overline{v} \wedge \mathsf{dj}(\mathsf{vars}(\langle\mathit{\Delta},v\rangle),\mathsf{ran}(\mathit{ren}))$

(A2) $\mathtt{slv}(\mathit{\Delta},\overline{d},{\uparrow}\mathit{id}{=}v) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d},v{=}x)$,
if $\mathit{\Delta}(\mathit{id}) = x \wedge \mathsf{strip}(x)$ is not of the form $\forall\overline{v}.\,x \wedge v \notin \mathsf{IdStatus}$

Figure 14.9 Constraint solving rules for type functions

at constraint solving. A constraint of the form $(\Lambda\alpha.\,\tau{=}\gamma)$ would lead to checking that $\Lambda\alpha.\,\tau$ and $\Lambda\alpha'.\,\alpha'\,\gamma$ are the same type functions because γ is considered in our system as equivalent to a type function of the form $\Lambda\alpha'.\,\alpha'\,\gamma$ (where α' is a "fresh" type variable w.r.t. a given constraint solving context). A constraint of the form $(\Lambda\alpha_1.\,\tau_1{=}\Lambda\alpha_2.\,\tau_2)$ would lead to checking that $\tau_1[\{\alpha_1\mapsto\alpha\}]$ and $\tau_2[\{\alpha_2\mapsto\alpha\}]$ can be made equal (where α is a "fresh" type variable w.r.t. a given constraint solving context).

There are two issues w.r.t. solving applications of internal type constructors to internal types where internal type constructors can be type functions, e.g., of the form $\tau_2\,(\Lambda\alpha_1.\,\tau_1)$, where dependencies are omitted for readability issues. The first issue is related to the fact that applications of type functions to internal types need eventually to be reduced. Such reductions are done by rule (S9) in Fig. 14.9. The first issue is that when an application of the form $\tau_2\,(\Lambda\alpha_1.\,\tau_1)$ is reduced at constraint solving, all the constraints on τ_1 need to have already been dealt with in order to replace all the occurrences of α_1 by τ_2 in the fully built up version of τ_1. Therefore, at constraint solving, we need to enforce that before reducing the application of a type function to an argument, all the constraints on the body of the type function have been dealt with. However we do not allow any look ahead in our constraint solver. Let us consider the two following environments, where $\gamma_1 \neq \gamma_2$, and which differ only by the swapping of the two equality constraints:

$$\text{Let}\ \ e_1\ \ \text{be}\ \ ((\alpha_1\,\gamma_1){=}(\alpha_2\,\gamma_2)\,(\Lambda\alpha'.\,\alpha));(\alpha{=}\alpha')$$
$$\text{Let}\ \ e_2\ \ \text{be}\ \ (\alpha{=}\alpha');((\alpha_1\,\gamma_1){=}(\alpha_2\,\gamma_2)\,(\Lambda\alpha'.\,\alpha))$$

When dealing with e_1, our constraint solver first deals with $((\alpha_1\,\gamma_1){=}(\alpha_2\,\gamma_2)\,(\Lambda\alpha.\,\alpha'))$ which does not lead to a type error and leads to $\alpha_2\,\gamma_2$ to be thrown away and α' to be constrained to be equal $\alpha_1\,\gamma_1$. It then deals with $\alpha{=}\alpha'$ which leads to α to also be constrained to be equal to $\alpha_1\,\gamma_1$ but which does not lead to any type error. As a matted of fact, when dealing with the first constraint of e_1 (left one) our constraint solver is not aware of the equality between α and α' and does not know if there are any more constraints on α' that have not yet been dealt with (and does not look them up). Note that solving e_2 leads to a type error. Because we believe e_1 and e_2 should have the same semantics, we need to somehow rule out environments such as e_1. Because we do not enforce our constraint solver to deal with $(\alpha{=}\alpha')$ before dealing with $((\alpha_1\,\gamma_1){=}(\alpha_2\,\gamma_2)\,(\Lambda\alpha.\,\alpha'))$, we need the initial constraint generation algorithm to generate $(\alpha{=}\alpha')$ before $((\alpha_1\,\gamma_1){=}(\alpha_2\,\gamma_2)\,(\Lambda\alpha.\,\alpha'))$. More generally, we need the initial constraint generation algorithm to generate all the constraints on μ before a constraint in which a type of the form $\tau\,\mu$ occurs.

Another solution would be to introduce another binary environment composition operator with a different semantics than the one of ";", such that unifiers generated for the right-hand-side of such an operator would not be usable for the left-hand-side. We leave the study of such a system to future work.

Equality constraints of the form (were dependencies are omitted) $\tau_1\,\delta{=}\tau$ where δ is unconstrained (see rule ($\mathsf{S10}$)) are discarded at constraint solving. We do so because δ could potentially be the type function $\Lambda\alpha.\,\tau$ where α does not occur in τ. Once again, because we discard such constraints at constraint solving, we need to require that all the constraints on δ have been generated before $\tau_1\,\delta{=}\tau$ at initial constraint generation and are dealt with before $\tau_1\,\delta{=}\tau$ at constraint solving.

Another issue w.r.t. solving applications of internal type constructors to internal types where internal type constructors can be type functions is an efficiency issue. For example, we do not wish to generate polymorphic binders of the form, e.g., $\downarrow vid{=}\forall\{\alpha\}.\,(\alpha\,\gamma_1)\,(\Lambda\alpha'.\,\alpha'\,\gamma_2)$ because this would potentially involve having to reduce the application multiple times. Therefore, because we already need our initial constraint generation algorithm to generate all the constraints on μ before a constraint in which a type of the form $\tau\,\mu$ occurs, we redefine our building function on types of the form $\tau\,\mu$ as follows (this new rule replaces the one given in Sec. 11.6):

$$\mathsf{build}(u,\tau\,\mu)=\begin{cases}\mathsf{collapse}(\tau'^{\overline{d}})[\{\alpha\mapsto\mathsf{build}(u,\tau)\}], \text{if } \mathsf{build}(u,\mu^{\varnothing})=(\Lambda\alpha.\,\tau')^{\overline{d}}\\ \mathsf{build}(u,\tau)\,\mathsf{build}(u,\gamma), \qquad\qquad \text{otherwise}\end{cases}$$

Because binders of the form $\downarrow tc{=}\kappa$ can now occur in constraint solving contexts (in e in $\langle u,\ e\rangle$), we redefine the binder forms generated at constraint solving as follows (originally defined in Sec. 11.6.6 and extended in Sec. 14.1.4):

$$\downarrow tc{=}\mu \xrightarrow{\mathsf{SolvBind}} \downarrow tc{=}\kappa$$

14.3.5 Slicing

Because we have changed our constraint generation rule for *dn*s, we need to replace the dot terms in DatName as follows:

$$\texttt{dot-e}(\overrightarrow{term}) \xrightarrow{\mathsf{DatName}} \texttt{dot-n}(\overrightarrow{term})$$

We define the new constraint generation rule for terms of the form $\texttt{dot-n}(\overrightarrow{term})$ as follows:

$$\begin{gathered}(\mathsf{G31})\, \texttt{dot-n}(\langle \overrightarrow{term}_1, \dots, \overrightarrow{term}_n \rangle) \Rightarrow \langle \delta, \alpha, \top, [e_1; \cdots; e_n] \rangle \Leftarrow \\ \overrightarrow{term}_1 \Rightarrow e_1 \wedge \cdots \wedge \overrightarrow{term}_n \Rightarrow e_n \wedge \mathsf{dja}(e_1, \dots, e_n, \delta, \alpha)\end{gathered}$$

Note that this rule is correct because our slicing algorithm (defined in Fig. 11.17) only generates dot-*dn* terms of the form $\texttt{dot-n}(\langle\rangle)$ and so no binder needs to be non-locally exported by the rule. The sequence wrapped into a dot-*dn* term is always empty when generated by our slicing algorithm because it means that it has been generated from a *dn* term of the form $\lceil tv\ \ tc \rceil^l$ and that l is sliced away (see rule (SL1) in Fig. 11.17). Given the function sl_2 on identifiers (see rule (SL9) in Fig. 11.17), we then obtain what corresponds to the dot-*dn* term $\texttt{dot-n}(\langle\rangle)$.

Our slicing algorithm does not need to be extended but we need to update the tree syntax for programs as follows:

$$\begin{aligned}\mathsf{Prod} &::= \cdots \mid \texttt{decTyp} \\ \mathsf{Dot} &::= \cdots \mid \texttt{dotN}\end{aligned}$$

We also need to modify the getDot function that associates dot markers with node kinds as follows (the function now returns a `dotN` marker and not a `dotE` marker anymore when applied to a `datname` node):

$$\mathsf{getDot}(\langle \texttt{datname}, prod \rangle) = \texttt{dotN}$$

We also need to extend the toTree function that associates trees of the form *tree* with terms of the form *term* as follows:

$$\begin{aligned}\mathsf{toTree}(\texttt{type}\ dn \overset{l}{=} ty) &= \langle \langle \texttt{dec}, \texttt{decTyp} \rangle, l, \langle \mathsf{toTree}(dn), \mathsf{toTree}(ty) \rangle \rangle \\ \mathsf{toTree}(\texttt{dot-n}(\overrightarrow{term})) &= \langle \texttt{dotN}, \mathsf{toTree}(\overrightarrow{term}) \rangle\end{aligned}$$

14.4 Non-recursive value declarations

In SML, a value declaration can either be recursive or non-recursive depending on the presence or not of the keyword `rec`. We already covered recursive value declarations (`val rec` declarations). Let us now present how to handle non-recursive value declarations. These declarations are interesting as they raise many issues such as value identifier status issues.

14.4.1 External syntax

Let us extend our external syntax with non-recursive value declarations as follows:

$$\mathsf{Dec} ::= \cdots \mid \texttt{val}\ \mathit{pat} \overset{l}{=} \mathit{exp}$$

In SML, the expression of a recursive value declaration is restricted to a fn-expression so that recursive value declarations are forced to declare functions. We do not take the restriction into consideration in this document as it does not raise any interesting issues w.r.t. type error slicing. There is no such restriction for non-recursive value declarations.

Let us provide an example of a non typable piece of code involving a non-recursive value declaration (many examples using non-recursive value declarations have already been given above, as these declarations are most useful):

```
val x = 1
val x = x 1
```

In this piece of code, x's third occurrence is bound to x's first occurrence and not to x's second occurrence. This piece of code is untypable because x's first occurrence is constrained to be an integer and x's third occurrence is constrained to be a function that takes an integer. We then obtain a type constructor clash.

Let us now present a slightly more interesting example.

```
datatype t = x
val x = 1
val x = x 1
```

The issue here is the same as for fn-expression. In our example, x's second (as well as its third and fourth) occurrence is bound to x's first occurrence. Therefore, the second declaration does not declare any identifier. We obtain two type error slices for this untypable piece of code: the first one reports a type constructor clash involving x's first and second occurrences, and the second one reports another type constructor clash involving x's first and fourth occurrences.

Let us finally wrap the second and third declarations of our last example into a structure declaration as follows:

```
datatype t = x
structure S = struct
  val x = 1
  val x = x 1
end
```

As explained above, the issue here is that the structure does not declare any identifier even though it contains declarations. This can lead to, e.g., confusing

$$\boxed{\mathsf{Declarations}}\,(\mathsf{G45})\ \mathtt{val}\ \mathit{pat} \overset{l}{=} \mathit{exp} \rightarrowtriangle (\mathit{ev}{=}\mathtt{poly}(e_2;e_1;(\alpha_1 \overset{l}{=} \alpha_2)));\mathit{ev}^l$$
$$\Leftarrow\ \mathit{pat} \rightarrowtriangle \langle \alpha_1,\ e_1\rangle \wedge \mathit{exp} \rightarrowtriangle \langle \alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \mathit{ev})$$

Figure 14.10 Constraint generation rule for non-recursive value declarations

unmatched errors.

Another interesting issue that is raised when adding non-recursive value declarations is the value polymorphism restriction which is discussed in Sec. 14.5.

14.4.2 Constraint syntax

No additional constraint term is necessary for this partial extension, but some will be required when taking into account the value polymorphism restriction (see Sec. 14.5). Our constraint solver and constraint filtering function are not changed in this section either. They will however be extended in Sec. 14.5.

14.4.3 Constraint generation

Fig. 14.10 extends our constraint generator with a rule to handle non-recursive value declarations. This rule is similar to rule (G17) defined in Fig. 11.7. Rule (G45) differs from rule (G17) by the fact that toV is not applied to e_1 and by the order in which the environments are in the generated environment. In rule (G45) for non-recursive value declarations, e_1 does not constrain e_2 so that in a declaration `val` $\mathit{pat} \overset{l}{=} \mathit{exp}$ the accessors generated for *exp* cannot refer to the binders generated for *pat*.

14.4.4 Slicing

First, we extend our tree syntax for programs as follows:

$$\mathsf{Prod} ::= \cdots \mid \mathtt{decNRec}$$

Then, we extend the toTree function as follows:

$$\mathsf{toTree}(\mathtt{val\ rec}\ \mathit{pat} \overset{l}{=} \mathit{exp}) = \langle\langle \mathtt{dec}, \mathtt{decNRec}\rangle, l, \langle \mathsf{toTree}(\mathit{pat}), \mathsf{toTree}(\mathit{exp})\rangle\rangle$$

14.5 Value polymorphism restriction

The value polymorphism restriction [146] allows one to have imperative features such as references in, e.g., SML by constraining the polymorphism of value declarations that could potentially be unsound.

We will illustrate this feature using an example given by Tofte [134] and reused (sometimes slightly modified) by many others [101, 146, 116]. First let us introduce references. The `ref` datatype and constructor are defined as follows in SML:

`datatype 'a ref = ref of 'a`. One can then create a new reference to an expression `e` as follows: `ref e`. One can access the value stored in a reference `r` as follows: `!r`. The function `!` has the following polymorphic type `'a ref -> 'a`. One can update a reference `r` as follows: `r := e` which results in `e` being stored in the reference `r`. The infix function `:=` has polymorphic type `'a ref * 'a -> unit`.

The example used by Pottier and Rémy [116] is as follows:

```
val r = ref (fn x => x)
val _ = r := fn x => x + 1
val _ = !r true
```

This piece of code declares a reference `r` to the identity function. This reference is then updated to store the successor function. Finally, the function stored in `r` is applied to `true`. It would then be unsound to generalise the type of `r` to the polymorphic type:

$$\forall\{\alpha\}.\,(\alpha\to\alpha)\,\texttt{ref}$$

because it would result in having a typable piece of code that reduces to the application of the successor function to `true`.

The value polymorphism restriction allows one to overcome this issue by restraining the body of value declarations that are allowed to be generalised. First, the expression set is partitioned into two sets: the expansive expressions and the non-expansive ones (what Wright [146] calls the syntactic values). A value declaration is not generalised if the corresponding expression is expansive. In The Definition of Standard ML [107, Sec.4.7], it is written that "the idea is that the dynamic evaluation of a non-expansive expression will neither generate an exception nor extend the domain of the memory, while the evaluation of an expansive expression might". In our restricted language, the syntax of non-expansive expressions is defined as follows:

$$\begin{array}{l} \mathit{conexp} \in \mathsf{ConExp} ::= \mathit{vid}^{l}_{\mathsf{e}} \\ \mathit{nonexp} \in \mathsf{NonExp} ::= \mathit{vid}^{l}_{\mathsf{e}} \mid \lceil \mathit{conexp}\ \mathit{nonexp} \rceil^{l} \mid \texttt{fn}\ \mathit{pat} \overset{l}{\Rightarrow} \mathit{exp} \end{array}$$

where a *conexp* has to be a datatype constructor (it can also be an exception constructor in full SML) and has to be different from the datatype constructor `ref`.

The expressions in $\mathsf{Exp} \setminus \mathsf{NonExp}$ are therefore the expansive expressions.

14.5.1 External syntax

Our external labelled syntax does not change. However, we define the functions expansive and expansiveCon which extract the dependencies responsible for an expression to be expansive as follows:

$$\boxed{\mathsf{Declarations}}\ (\mathsf{G45})\ \mathtt{val}\ \mathit{pat} \overset{l}{=} \mathit{exp} \rightarrowtriangle (\mathit{ev}{=}\mathtt{expans}(e_2;e_1;(\alpha_1 \overset{l}{=} \alpha_2), \mathsf{expansive}(\mathit{exp})));\mathit{ev}^l$$
$$\Leftarrow\ \mathit{pat} \rightarrowtriangle \langle\alpha_1,\ e_1\rangle \wedge \mathit{exp} \rightarrowtriangle \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \mathit{ev})$$

Figure 14.11 Constraint generator handling the value polymorphism restriction

$$\begin{array}{ll}
\mathsf{expansive}(\mathit{vid}^l_{\mathsf{e}}) & = \varnothing \\
\mathsf{expansive}(\mathtt{let}^l\ \mathit{dec}\ \mathtt{in}\ \mathit{exp}\ \mathtt{end}) & = \{\{l\}\} \\
\mathsf{expansive}(\mathtt{fn}\ \mathit{pat} \overset{l}{\Rightarrow} \mathit{exp}) & = \varnothing \\
\mathsf{expansive}(\lceil \mathit{exp}\ \mathit{atexp} \rceil^l) & = \{l \cup \overline{d} \mid \overline{d} \in \mathsf{expansiveCon}(\mathit{exp}) \cup \mathsf{expansive}(\mathit{atexp})\} \\
\mathsf{expansiveCon}(\mathit{vid}^l_{\mathsf{e}}) & = \{\{l, \mathit{vid}\}\} \\
\mathsf{expansiveCon}(\mathtt{let}^l\ \mathit{dec}\ \mathtt{in}\ \mathit{exp}\ \mathtt{end}) & = \{\{l\}\} \\
\mathsf{expansiveCon}(\mathtt{fn}\ \mathit{pat} \overset{l}{\Rightarrow} \mathit{exp}) & = \{\{l\}\} \\
\mathsf{expansiveCon}(\lceil \mathit{exp}\ \mathit{atexp} \rceil^l) & = \{\{l\}\}
\end{array}$$

14.5.2 Constraint syntax

We introduce new environments as follows:

$$e \in \mathsf{Env} ::= \cdots \mid \mathtt{expans}(e, \overline{\overline{d}})$$

The semantics of an environment of the form $\mathtt{expans}(e, \overline{\overline{d}})$ is that e is monomorphic if one of the set in $\overline{\overline{d}}$ is satisfied. An environment of the form $\mathtt{expans}(e, \overline{\overline{d}})$ is then a dependent $\mathtt{poly}(e)$ environment.

14.5.3 Constraint generation

Fig. 14.11 redefines rule (G45). This rule differs from the one provided in Fig 14.10 by the replacement of the poly environment by an expans environment.

Because our initial constraint generation algorithm generates these new expans forms, we have to extend the set GenEnv of initially generated environments, originally defined in Sec. 11.5.2, as follows (where pe is as redefined in Sec. 14.1.3):

$$\mathit{ge} \in \mathsf{GenEnv} ::= \cdots \mid \mathtt{expans}(\mathit{pe}, \overline{\overline{d}})$$

14.5.4 Constraint solving

Fig. 14.12 extend our constraint solver to deal with expans environments.

An expans environment can turn into a poly environment if it turns out that the corresponding declaration binds a non-expansive expression or if there is not enough information to determine whether or not the corresponding expression is expansive (rule (VPR1)). For example, the environment generated for `f`'s declaration in `val f = fn x => x` will eventually turn into a poly environment at constraint solving because the corresponding expression is a fn-expression which is non-expansive. The environment generated for `f`'s declaration in `datatype 'a t = T of 'a val f = T 1`

Value polymorphism restriction

$$\text{(VPR1) } \mathtt{slv}(\mathit{\Delta}, \overline{d}, \mathtt{expans}(e, \overline{\overline{d}})) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \mathtt{poly}(e)),$$

$$\text{if } \forall \overline{d}_0 \in \overline{\overline{d}}.\ (\overline{d}_0 = \overline{l} \cup \{\mathit{vid}\}$$

$$\wedge(\mathsf{strip}(\mathit{\Delta}[\mathit{vid}]) \notin \{\mathtt{v}, \mathtt{u}\} \vee (\mathit{\Delta}[\mathit{vid}] \text{ undefined} \wedge \mathsf{shadowsAll}(\mathit{\Delta}))))$$

$$\text{(VPR2) } \mathtt{slv}(\mathit{\Delta}, \overline{d}, \mathtt{expans}(e, \overline{\overline{d}} \cup \{\overline{l}\})) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d} \cup \overline{l}, e)$$

$$\text{(VPR3) } \mathtt{slv}(\mathit{\Delta}, \overline{d}, \mathtt{expans}(e, \overline{\overline{d}} \cup \{\overline{d}_0 \uplus \{\mathit{vid}\}\})) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d} \cup \overline{d}', e),$$

$$\text{if } (\mathsf{collapse}(\mathit{\Delta}[\mathit{vid}]) \in \{\mathtt{v}^{\overline{d}_1}, \mathtt{u}^{\overline{d}_1}\} \wedge \overline{d}' = \overline{d}_0 \cup \overline{d}_1)$$

$$\vee\ (\mathit{\Delta}[\mathit{vid}] \text{ undefined} \wedge \neg\mathsf{shadowsAll}(\mathit{\Delta}) \wedge \overline{d}' = \overline{d}_0 \cup \{\mathit{vid}\})$$

Figure 14.12 Constraint solving rules handling the value polymorphism restriction

will also eventually turn into a `poly` environment at constraint solving because the corresponding expression is the application of a datatype constructor to a non-expansive expression (special constants such as `1` are also non-expansive in SML).

Rule (VPR2) applies when dealing with the environment generated for the declaration `val f = let val g = fn x => x in g end` because let-expressions are expansive and the expansiveness does not depend on identifier statuses. The binder generated for `f` at constraint solving is then monomorphic.

Rule (VPR3) can generate value identifier dependencies if it turns out that the polymorphism of an environment depends on a value identifier not being a value variable and that this identifier is free. For example, the environment generated for `f`'s declaration in `val f = g 1` will stay monomorphic at constraint solving and will eventually be dependent on `g` being a value variable and not a datatype constructor because `g` is a free identifier and as such its status is context dependent.

Rule (VPR3) also deals with the case where the polymorphism of an environment depends on a value identifier not being a value variable and that the status of this identifier is confirmed to be a value variable. For example, the environment generated for `f`'s declaration in `val rec g = fn x => x; val f = g 1` will stay monomorphic at constraint solving and will depend on the dependencies of the status binder generated for `g`'s first occurrence (which is a value variable).

Let us now consider this example: `fn g => let val f = g 1 in (f (), f true) end`. Because `g` is not declared in the context of this fn-expression, at constraint solving, a status binder is generated associating the status `u` to `g`'s first occurrence. This binder is context dependent and depends on `g`'s status being a value variable and not a datatype constructor (see rule (B4) in Fig. 14.3). Rule (VPR3) applies and the environment generated for `f`'s declaration will stay monomorphic at constraint solving and will be dependent on `g` being a value variable and not a datatype constructor because `g`'s status in the context of `f`'s declaration is context dependent. It will also be dependent on `g`'s first occurrence itself for binding issues even though this occurrence does not help resolving the dependency on `g`'s status. A type error slice for this untypable piece of code is then as follows: `⟨..fn g => ⟨..val f = g ⟨..⟩..f ()..f true..⟩..⟩` which depends on `g` being a value variable and not a datatype constructor. Let us present why g's first occurrence is necessary using the following piece of code:

```
datatype t = h; val g = h; val u = g;
val v = fn g => let val f = g 1 in (f (), f u) end
```

If g's third occurrence was not involved in the found type error slice (similar to the one described above) then our minimiser would eventually try to minimise the following slice:

```
⟨..datatype t = h..val g = h..val u = g
 ..⟨..val f = g ⟨..⟩..f ()..f u..⟩..⟩
```

where the bindings are mixed up because in this slice g's last occurrence is bound to g's first occurrence.

Instead, the minimal type error slice computed by our TES is as follows:

```
⟨..datatype t = h..val g = h..val u = g
 ..fn g => ⟨..val f = g ⟨..⟩..f ()..f u..⟩..⟩
```

14.5.5 Constraint filtering

We update our filtering function as follows:

$$\mathsf{filt}(\mathsf{expans}(e,\overline{\overline{d}}),\overline{l}_1,\overline{l}_2) = \mathsf{expans}(\mathsf{filt}(e,\overline{l}_1,\overline{l}_2),\{\overline{d} \mid \overline{d} \in \overline{\overline{d}} \wedge \mathsf{labs}(\overline{d}) \subseteq \overline{l}_1\})$$

14.6 Type annotations

14.6.1 External syntax

First, let us extend our external syntax with type annotations as follows:

$$\begin{aligned} \mathsf{Exp} &::= \cdots \mid \mathit{exp}\,{:}^{l}\,\mathit{ty} \\ \mathsf{Pat} &::= \cdots \mid \mathit{pat}\,{:}^{l}\,\mathit{ty} \end{aligned}$$

Let us consider the following piece of code.

```
val rec g : unit -> unit = fn x => x
val u = g true
```

This piece of code is untypable because the function g is explicitly defined to be a function that takes a unit and is later applied to true. Note that there are several ways to solve the programming error. We only mention some of them below. For example, one can change the type annotation on g to be bool -> bool. One could also apply another function to true.

We define sequences of explicit type variables as follows:

$$\begin{aligned}
&\mathit{ltv} && \in \mathsf{LabTyVar} ::= \mathit{tv}_{1}^{l} \mid \mathtt{dot\text{-}d}(\overrightarrow{\mathit{term}}) \\
&\mathit{tvseq} && \in \mathsf{TyVarSeq} ::= \mathit{ltv} \mid \epsilon_{\mathsf{v}}^{l} \mid (\mathit{ltv}_1, \dots, \mathit{ltv}_n)^{l} \mid \mathtt{dot\text{-}d}(\overrightarrow{\mathit{term}})
\end{aligned}$$

Explicit type variables (tv) in type variable sequences are subscripted when occurring in type variable sequences (tv_{1}^{l}) in order to distinguish between occurrences in type variable sequences and occurrences in types.

We replace the recursive and non-recursive value declarations as follows:

$$\begin{aligned}
&\mathtt{val}\ \mathit{pat} \overset{l}{=} \mathit{exp} \xrightarrow{\mathsf{Dec}} \mathtt{val}\ \mathit{tvseq}\ \mathit{pat} \overset{l}{=} \mathit{exp} \\
&\mathtt{val\ rec}\ \mathit{pat} \overset{l}{=} \mathit{exp} \xrightarrow{\mathsf{Dec}} \mathtt{val\ rec}\ \mathit{tvseq}\ \mathit{pat} \overset{l}{=} \mathit{exp}
\end{aligned}$$

For example, the following piece of code is untypable:

(EX11)

```
val rec 'a f = fn x =>
  let val rec g : 'a -> 'a = fn x => x
  in g true
  end
```

First, `'a` is explicitly bound at the outer value declaration (`f`'s declaration). Then, before generalisation, `g`'s type is `'a -> 'a`. Because `'a` is bound to the outer value declaration, it occurs in the type environment. It is therefore not generalised when generalising `g`'s type. After generalisation, `g`'s type is then still `'a -> 'a`. Finally, when applying `g` to `true`, the non-generalised explicit type variable `'a` clashes against the type `bool`.

As usual, there as several ways of obtaining a typable piece of code. We only mention some of them below. For example, example (EX12) below (we have just removed the explicit binding of `'a` from (EX11)) is typable if it does not occur in an expression where `'a` occurs at a binding or bound occurrence because the explicit type variable `'a` is implicitly bound to the inner declaration (`g`'s declaration).

(EX12)

```
val rec f = fn x =>
  let val rec g : 'a -> 'a = fn x => x
  in g true
  end
```

In example (EX11), one could also remove the type annotation on `g` or change it into, e.g., `'b -> 'b`.

Let us present a last example:

(EX13)

```
val rec f = fn x =>
  let val rec g : 'a -> 'a = fn x => x
  in fn y : 'a => fn z : 'a => g true
  end
```

This untypable piece of code slightly differs from example (EX12). We have replaced the expression `g true` by the fn-expression `fn y : 'a => fn z : 'a => g true` in order to introduce new occurrences of the type variable `'a`. Because `'a` occurs free in `f`'s body (in the expression `fn y : 'a => fn z : 'a => g true`), it is then implicitly bound at the outer value declaration (`f`'s declaration). Now, because `'a` is bound at the outer value declaration, the occurrences of `'a` in `g`'s body are also implicitly bound at the outer value declaration and not at the inner one. We then obtain as for example (EX11) a clash between the first occurrence of the non-generalised explicit type variable `'a` and `true`'s type. As a matter of fact, we obtain two minimal type error slices. One involves `'a`'s occurrence in the pattern `y : 'a` and the other one involve `'a`'s occurrence in the pattern `z : 'a`.

14.6.2 Constraint syntax

We introduce unconfirmed type variable binders as follows:

$$\begin{aligned} \mathit{bind} &\in \mathsf{Bind} ::= \cdots \mid \downharpoonleft \mathit{tv}{=}\beta \\ e &\in \mathsf{Env} \;\; ::= \cdots \mid \mathtt{or}(e, \overline{d}) \end{aligned}$$

The difference between binders of the form $\downharpoonleft \mathit{tv}{=}\beta$ and binders of the form $\updownarrow \mathit{vid}{=}\alpha$ is that a binder of the form $\downharpoonleft \mathit{tv}{=}\beta$ cannot turn into an accessor while one of the form $\updownarrow \mathit{vid}{=}\alpha$ can as we saw in Fig. 11.10. The similarity is that both kinds of binders will look up the environment to turn into confirmed binders of the form $\downarrow \mathit{id}{=}x$. The difference between binders of the form $\downharpoonleft \mathit{tv}{=}\beta$ and binders of the form $\downarrow \mathit{id}{=}x$ is that a binder of the form $\downharpoonleft \mathit{tv}{=}\beta$ can be discarded at constraint solving while a binder of the form $\downarrow \mathit{id}{=}x$ cannot.

We need such unconfirmed type variable binders because, e.g., for example (EX11) presented above, we generate an unconfirmed binder for `'a` at the inner declaration (`g`'s declaration). In our example, this unconfirmed will obviously be discarded at constraint solving, if no constraint is filtered out, because there is already a binder generated for `'a` at the outer declaration (`f`'s declaration).

We also define environments of the form $\mathtt{or}(e, \overline{d})$. Such an environment differs from an environment of the form $e^{\overline{d}}$ by the fact that in the latest all the dependencies have to be satisfied for e to be kept at constraint filtering (we then say that e is kept "alive", or simply that it is "alive") while in an environment of the form $\mathtt{or}(e, \overline{d})$ only one of the dependencies in $\overline{d}$ has to be satisfied for e to be "alive". In an environment of the form $e^{\overline{d}}$, the set $\overline{d}$ can be seen as a conjunction of dependencies, while in an environment of the form $\mathtt{or}(e, \overline{d})$, the set $\overline{d}$ can be seen as a disjunction of dependencies. This is why we write $e^{\vee\overline{d}}$ for $\mathtt{or}(e, \overline{d})$.

For example (EX13) we generate an environment or the form $(\downharpoonleft \mathtt{'a} \overset{l}{=} \beta)^{\vee\{l_1, l_2\}}$ where l_1 is `'a`'s third occurrence's label, l_2 is `'a`'s fourth occurrence's label, and l is the label of the declaration at which `'a` is implicitly bound. Both l_1 and l_2 are "reasons" explaining why the unconfirmed binder is introduced and only one of them is necessary for the unconfirmed binder to exist.

14.6.3 Constraint generation

We extend the labtyvars function, originally defined in Sec. 14.7, to expressions and patterns as follows:

$$\begin{aligned}
&\mathsf{labtyvars}(vid_{\mathtt{e}}^{l}) &&= \varnothing\\
&\mathsf{labtyvars}(\mathtt{let}^{l}\ dec\ \mathtt{in}\ exp\ \mathtt{end}) &&= \mathsf{labtyvars}(exp)\\
&\mathsf{labtyvars}(\mathtt{fn}\ pat \overset{l}{\Rightarrow} exp) &&= \mathsf{labtyvars}(pat) \cup \mathsf{labtyvars}(exp)\\
&\mathsf{labtyvars}(\lceil exp\ atexp\rceil^{l}) &&= \mathsf{labtyvars}(exp) \cup \mathsf{labtyvars}(atexp)\\
&\mathsf{labtyvars}(exp\,{:}^{l}\,ty) &&= \mathsf{labtyvars}(exp) \cup \mathsf{labtyvars}(ty)\\
&\mathsf{labtyvars}(vid_{\mathtt{p}}^{l}) &&= \varnothing\\
&\mathsf{labtyvars}(vid^{l}\ atpat) &&= \mathsf{labtyvars}(atpat)\\
&\mathsf{labtyvars}(pat\,{:}^{l}\,ty) &&= \mathsf{labtyvars}(pat) \cup \mathsf{labtyvars}(ty)
\end{aligned}$$

This function does not extract *all* the explicit type variables occurring in an expression of a pattern. It does not extract the explicit type variables occurring in nested declarations (see case for let-expressions).

We define the function labtyvarsdec as follows:

$$\begin{aligned}
&\mathsf{labtyvarsdec}(tvseq, pat, exp) = \{tv^{\bar{l}} \mid f(tv) = \bar{l}\}\\
&\qquad \text{where } f = \Cup\{tv \mapsto \{l\} \mid\ tv \text{ does not occur in } tvseq\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ tv^{l} \in \mathsf{labtyvars}(pat) \cup \mathsf{labtyvars}(exp)\}
\end{aligned}$$

Such *tvseq*, *pat* and *exp* are meant to be those of a recursive or non-recursive value declaration.

Fig. 14.13 extends our constraint generation algorithm. Rules (G46)-(G50) are new. Rules (G17) and (G45) replace the ones respectively defined in Fig. 11.7 and Fig. 14.11. Rules (G48)-(G50) generate explicit type variable binders for type variable sequences. The generated binders are confirmed binders (of the form $\downarrow tv{=}\beta$ and not of the form $\downharpoonleft tv{=}\beta$) because type variable sequences are used in SML to explicitly bind type variables (they are not context dependent). Rules (G17) and (G45) generate unconfirmed type variable binders of the form $\downharpoonleft tv{=}\beta$ for explicit type variables that could potentially implicitly bound at value declarations. These unconfirmed binders are generated after the confirmed binders because the unconfirmed ones are dependent on the confirmed ones. This order is necessary. The order in which the unconfirmed binders are generated is not relevant because the explicit type variables are all different.

Note that instead of adding environment of the form $e^{\vee\bar{d}}$, we could have replaced the dependent environment forms by forms depending on disjunctions of conjunctions of dependencies (instead of just depending on conjunctions of dependencies). Then, instead of generating, e.g., $(\downharpoonleft tv_1 \overset{l}{=} \beta_1)^{\vee\bar{l}_1}$, we could have generated a binder of the form $\downharpoonleft tv_1 \xlongequal{\{\{l,l'\}\mid l'\in\bar{l}_1\}} \beta_1$, where at least one of $\{l, l'\}$, such that $l' \in \bar{l}_1$, has to be satisfied to the constraint represented by the dependencies to be satisfied. Because this is only needed for environment, and in order to keep the same simple dependent

$\boxed{\textbf{Expressions}}$

(G46) $exp\,{:}^{l}\,ty \mathrel{-\!\!\rhd} \langle\alpha,\ e_1;e_2;(\alpha \overset{l}{=} \alpha_1);(\alpha \overset{l}{=} \alpha_2)\rangle \;\Leftarrow\; exp \mathrel{-\!\!\rhd} \langle\alpha_1,\ e_1\rangle \wedge ty \mathrel{-\!\!\rhd} \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

$\boxed{\textbf{Patterns}}$

(G47) $pat\,{:}^{l}\,ty \mathrel{-\!\!\rhd} \langle\alpha,\ e_1;e_2;(\alpha \overset{l}{=} \alpha_1);(\alpha \overset{l}{=} \alpha_2)\rangle \;\Leftarrow\; pat \mathrel{-\!\!\rhd} \langle\alpha_1,\ e_1\rangle \wedge ty \mathrel{-\!\!\rhd} \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \alpha)$

$\boxed{\textbf{Labelled type variables } (ltv \mathrel{-\!\!\rhd} e)}$

(G48) $tv_1^{l} \mathrel{-\!\!\rhd} \downarrow tv \overset{l}{=} \beta$

$\boxed{\textbf{Type variable sequences } (tvseq \mathrel{-\!\!\rhd} e)}$

(G49) $\epsilon_{\mathtt{v}}^{l} \mathrel{-\!\!\rhd} \top$

(G50) $(ltv_1, \ldots, ltv_n)^{l} \mathrel{-\!\!\rhd} e_1; \cdots ;e_n \;\Leftarrow\; ltv_1 \mathrel{-\!\!\rhd} e_1 \wedge \cdots \wedge ltv_n \mathrel{-\!\!\rhd} e_n \wedge \mathsf{dja}(e_1, \ldots, e_n)$

$\boxed{\textbf{Declarations}}$

(G17) $\mathtt{val\ rec}\ tvseq\ pat \overset{l}{=} exp \mathrel{-\!\!\rhd} (ev{=}\mathtt{poly}(\mathtt{loc}\, e_0;e\, \mathtt{in}\, (\mathsf{toV}(e_1);e_2;(\alpha_1 \overset{l}{=} \alpha_2))));ev^{l}$
$\quad \Leftarrow\; tvseq \mathrel{-\!\!\rhd} e_0 \wedge pat \mathrel{-\!\!\rhd} \langle\alpha_1,\ e_1\rangle \wedge exp \mathrel{-\!\!\rhd} \langle\alpha_2,\ e_2\rangle$
$\quad \wedge \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n}\{tv_i^{\overline{l}_i}\}$
$\quad \wedge e = ((\downharpoonleft tv_1 \overset{l}{=} \beta_1)^{\vee \overline{l}_1}; \cdots ;(\downharpoonleft tv_n \overset{l}{=} \beta_n)^{\vee \overline{l}_n})$
$\quad \wedge \mathsf{dja}(e_0, e_1, e_2, ev, \beta_1, \ldots, \beta_n)$

(G45) $\mathtt{val}\ tvseq\ pat \overset{l}{=} exp \mathrel{-\!\!\rhd} (ev{=}\mathtt{expans}(\mathtt{loc}\, e_0;e\, \mathtt{in}\, (e_2;e_1;(\alpha_1 \overset{l}{=} \alpha_2)), \mathsf{expansive}(exp)));ev^{l}$
$\quad \Leftarrow\; tvseq \mathrel{-\!\!\rhd} e_0 \wedge pat \mathrel{-\!\!\rhd} \langle\alpha_1,\ e_1\rangle \wedge exp \mathrel{-\!\!\rhd} \langle\alpha_2,\ e_2\rangle$
$\quad \wedge \mathsf{labtyvarsdec}(tvseq, pat, exp) = \uplus_{i=1}^{n}\{tv_i^{\overline{l}_i}\}$
$\quad \wedge e = ((\downharpoonleft tv_1 \overset{l}{=} \beta_1)^{\vee \overline{l}_1}; \cdots ;(\downharpoonleft tv_n \overset{l}{=} \beta_n)^{\vee \overline{l}_n})$
$\quad \wedge \mathsf{dja}(e_0, e_1, e_2, ev, \beta_1, \ldots, \beta_n)$

Figure 14.13 Constraint generation rules for type annotations

binders

(B9) $\mathtt{slv}(\Delta, \overline{d}, \downharpoonleft tv{=}\beta) \rightarrow \mathtt{succ}(\Delta;(\downarrow tv \overset{\overline{d}}{=} \beta))$, if $\Delta(tv)$ is undefined

(B10) $\mathtt{slv}(\Delta, \overline{d}, \downharpoonleft tv{=}\beta) \rightarrow \mathtt{succ}(\Delta)$, if $\Delta(tv)$ is defined

or environments

(OR) $\mathtt{slv}(\Delta, \overline{d}, e^{\vee \{d\} \cup \overline{d}'}) \rightarrow \mathtt{slv}(\Delta, \overline{d} \cup \{d\}, e)$

Figure 14.14 Constraint solving rules to handle type annotations

form for all our kind of constraint terms, we did not adopt this solution. We leave for future work the investigation of such a system.

Because our initial constraint generation algorithm generates new forms of binders ($\downharpoonleft tv{=}\beta$), and because `poly` environment can now wrap local environments, we update LabBind and PolyEnv as follows:

$$\begin{aligned} lbind &\in \mathsf{LabBind} ::= \cdots \mid (\downharpoonleft tv \overset{l}{=} \beta)^{\vee \overline{l}} \\ pe &\in \mathsf{PolyEnv} ::= \cdots \mid \mathtt{loc}\, pe_1\, \mathtt{in}\, pe_2 \end{aligned}$$

14.6.4 Constraint solving

Because we introduced new form of binders and environments, Fig. 14.14 extends our constraint solver. Rule (B9) only picks one dependency from the dependency set labelling an environment of the form $e^{\vee \overline{d}}$ because only one of them is needed for the constraint represented by the dependency set to be satisfied. Any dependency from $\overline{d}$ can be chosen.

Expressions

$$\mathsf{toTree}(exp\,{:}^{l}\,ty) = \langle\langle\mathtt{exp}, \mathtt{expTyp}\rangle, l, \langle\mathsf{toTree}(exp), \mathsf{toTree}(ty)\rangle\rangle$$

Patterns

$$\mathsf{toTree}(pat\,{:}^{l}\,ty) = \langle\langle\mathtt{pat}, \mathtt{patTyp}\rangle, l, \langle\mathsf{toTree}(pat), \mathsf{toTree}(ty)\rangle\rangle$$

Labelled type variables

$$\mathsf{toTree}(tv_1^{l}) = \langle\langle\mathtt{labtyvar}, \mathtt{id}\rangle, l, \langle tv\rangle\rangle$$

Type variable sequences

$$\begin{aligned}\mathsf{toTree}(\epsilon_{\mathsf{v}}^{l}) &= \langle\langle\mathtt{tyvarseq}, \mathtt{tyvarseqEm}\rangle, l, \langle\rangle\rangle\\ \mathsf{toTree}((ltv_1,\dots,ltv_n)^{l}) &= \langle\langle\mathtt{tyvarseq}, \mathtt{tyvarseqSeq}\rangle, l, \mathsf{toTree}(\langle ltv_1,\dots,ltv_n\rangle)\rangle\end{aligned}$$

Figure 14.15 Extension of our conversion function from *terms* to *trees* to deal with type annotations and type variable sequences

14.6.5 Constraint filtering (Minimisation and enumeration)

We update our filtering function as follows:

$$\mathsf{filt}(e^{\vee\overline{l}}, \overline{l}_1, \overline{l}_2) = \begin{cases}\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)^{\vee\overline{l}'}, \text{if } \overline{l}' = \overline{l} \cap (\overline{l}_1 \setminus \overline{l}_2) \neq \varnothing\\ \mathsf{dum}(\mathsf{strip}(e)), & \text{if } \mathsf{dj}(\overline{l}, \overline{l}_1 \setminus \overline{l}_2) \text{ and } \neg\mathsf{dj}(\overline{l}, \overline{l}_2)\\ \odot, & \text{if } \mathsf{dj}(\overline{l}, \overline{l}_1 \cup \overline{l}_2) \text{ and } \mathsf{strip}(e) \in \mathsf{Var} \cup \mathsf{Bind}\\ \top, & \text{otherwise}\end{cases}$$

$$\mathsf{dum}(\downharpoonleft id{=}x) = (\downharpoonleft id{=}\mathsf{toDumVar}(x))$$

14.6.6 Slicing

We extend our tree syntax for programs as follows:

$$\begin{aligned}\mathsf{Class} &::= \cdots \mid \mathtt{labtyvar} \mid \mathtt{tyvarseq}\\ \mathsf{Prod} &::= \cdots \mid \mathtt{expTyp} \mid \mathtt{patTyp} \mid \mathtt{tyvarseqEm} \mid \mathtt{tyvarseqSeq}\end{aligned}$$

We extend the function getDot that associates dot markers with node kinds as follows:

$$\begin{aligned}\mathsf{getDot}(\langle\mathtt{labtyvar}, prod\rangle) &= \mathtt{dotD}\\ \mathsf{getDot}(\langle\mathtt{tyvarseq}, prod\rangle) &= \mathtt{dotD}\end{aligned}$$

Finally, Fig. 14.15 extends the function toTree that transforms *terms* into *trees*.

14.7 Signatures

This section shows how to design a type error slicer that handles some signature related features. This section deals with value, type, datatype and structure specifications. It does not deal with *include* or *sharing* specifications, and does not deal with type realisations (*where* clauses) either. Type realisations are "almost fully" supported by our implementation, we partially support *include* specifications, and we have started implementing support for *sharing* specifications.

Some kinds of errors are not handled by the system presented in this section. For example we do not handle unmatched errors: when an identifier is specified in a signature but not declared in a structure constrained by the signature. These errors are dealt with in Sec. 14.8. Another kind of error which is not dealt with in this section is when a type constructor is defined as a type function in a structure and as a datatype in the structure's signature. This kind of error is handled by our implementation but we do not provide the details in this document.

14.7.1 External syntax

First, let us extend our external syntax with signatures as follows:

$$
\begin{array}{lll}
\mathit{sigid} & \in \mathsf{SigId} & \text{(signature identifiers)}\\
\mathit{sigdec} & \in \mathsf{SigDec} & ::= \texttt{signature}\ \mathit{sigid} \overset{l}{=} \mathit{sigexp}\\
 & & \quad \mid \texttt{dot-d}(\overrightarrow{\mathit{term}})\\
\mathit{sigexp} & \in \mathsf{SigExp} & ::= \mathit{sigid}^{l} \mid \texttt{sig}^{l}\ \mathit{spec}_1 \cdots \mathit{spec}_n\ \texttt{end}\\
 & & \quad \mid \texttt{dot-s}(\overrightarrow{\mathit{term}})\\
\mathit{spec} & \in \mathsf{Spec} & ::= \texttt{val}\ \mathit{vid}\ \texttt{:}^{l}\ \mathit{ty}\\
 & & \quad \mid \texttt{type}\ \mathit{dn}^{l}\\
 & & \quad \mid \texttt{datatype}\ \mathit{dn} \overset{l}{=} \mathit{cd}\\
 & & \quad \mid \texttt{structure}\ \mathit{strid}\ \texttt{:}^{l}\ \mathit{sigexp}\\
 & & \quad \mid \texttt{dot-d}(\overrightarrow{\mathit{term}})\\
\mathit{cd} & \in \mathsf{ConDesc} & ::= \mathit{vid}^{l}_{\mathsf{c}} \mid \mathit{vid}\ \texttt{of}^{\,l}\ \mathit{ty}\\
 & & \quad \mid \texttt{dot-e}(\overrightarrow{\mathit{term}})\\
\mathit{id} & \in \mathsf{Id} & ::= \cdots \mid \mathit{sigid}\\
\mathit{strexp} & \in \mathsf{StrExp} & ::= \cdots \mid \mathit{strexp}\ \texttt{:}^{l}\ \mathit{sigexp} \mid \mathit{strexp}\ \texttt{:>}^{l}\ \mathit{sigexp}\\
\mathit{topdec} & \in \mathsf{TopDec} & ::= \mathit{strdec} \mid \mathit{sigdec}\\
\mathit{prog} & \in \mathsf{Program} & ::= \mathit{topdec}_1\texttt{;} \cdots \texttt{;}\mathit{topdec}_n
\end{array}
$$

The symbol `:>` is used for opaque constraints and `:` for translucent constraints. The structure *strexp* $\texttt{:>}^{l}$ *sigexp* is the structure *strexp* constrained by the signature *sigexp* where each of *sigexp*'s specifications has to be matched by one of *strexp*'s declarations (and similarly for *strexp* $\texttt{:}^{l}$ *sigexp*). The structure *strexp* can declare more identifiers than are specified in *sigexp*. In the structure *strexp* $\texttt{:>}^{l}$ *sigexp*, only the identifiers specified in *sigexp* can be accessed from *strexp*, i.e., only the *sigexp* part from *strexp* is visible to the outside world. The difference between *strexp* $\texttt{:>}^{l}$ *sigexp* and *strexp* $\texttt{:}^{l}$ *sigexp* is that in the first one if *sigexp* specifies a type constructor *tc* then in *strexp* $\texttt{:>}^{l}$ *sigexp* it is not constrained by its declaration in *strexp*, whereas in *strexp* $\texttt{:}^{l}$ *sigexp* the type constructor would be constrained by its declaration in *strexp*. Opaque signatures are used to abstract types from structures and are usually preferred over translucent ones for this reason.

Let us present an example involving an opaque signature:

(EX3)

```
signature s = sig val x : 'a end
structure S = struct val x = 1 end
structure T = S :> s
```

This piece of code is untypable because the type variable `'a` is more general than the type `int`. Types of declarations in structures have to be at least as general as the corresponding specifications in signatures. This kind of error will be referred as a *too general* error henceforth.

Let us now present an example illustrating the difference between opaque and translucent signatures:

(EX4)

```
signature s = sig type t val f : t -> t end
structure S = struct type t = bool val rec f = fn x => x end
structure T1 = S :> s
structure T2 = S :  s
val u1 = let open T1 in f true end
val u2 = let open T2 in f true end
```

In this piece of code, the difference between `T1` and `T2` is that `T1` is the structure `S` constrained by the signature `s` using an opaque constraint while the structure `T2` uses a translucent signature. The declaration `u2` differs from `u1` by opening the structure `T2` instead of `T1`. The application `f true` occurring in `u1` is part of an error because `f` is a function that takes a `t` as argument and not a `bool`. In `T1`, the type `t` is abstracted and is not related to `bool`. The application `f true` occurring in `u2` however, is not part of an error because `f` is there a function that takes a `bool` as argument. In `T2`, the type `t` is the `bool` type.

14.7.2 Constraint syntax

We extend our constraint system to handle signatures as follows:

$$
\begin{array}{lll}
\beta & \in \mathsf{RigidTyVar} & \text{(set of rigid type variables)} \\
\mathit{svar} & \in \mathsf{SVar} & ::= v \mid \beta \\
\rho & \in \mathsf{FRTyVar} & ::= \alpha \mid \beta \\
\mathit{sig} & \in \mathsf{SigSem} & ::= e \mid \forall\overline{\delta}.\, e \mid \langle \mathit{sig}, \overline{d} \rangle \\
\mathit{bind} & \in \mathsf{Bind} & ::= \cdots \mid \downarrow\!\mathit{sigid}{=}\mathit{sig} \\
\mathit{acc} & \in \mathsf{Accessor} & ::= \cdots \mid \uparrow\!\mathit{sigid}{=}\mathit{ev} \\
\tau & \in \mathsf{ITy} & ::= \cdots \mid \beta \\
\mu & \in \mathsf{ITyCon} & ::= \cdots \mid \mathtt{tv} \\
\mathit{subty} & \in \mathsf{SubTy} & ::= \sigma_1 \preceq_{\mathit{vid}} \sigma_2 \mid \kappa_1 \preceq_{\mathit{tc}} \kappa_2 \\
e & \in \mathsf{Env} & ::= \cdots \mid e_1{:}e_2 \mid \mathtt{ins}(e) \mid \mathit{subty}
\end{array}
$$

In this table, we introduce new type variables: the rigid type variables. These rigid type variables act as constant types but are called variables because they are allowed to be renamed and quantified over. Being considered as constant types, they are not allowed to be equal, e.g., to arrow types (they are not allowed to be vs in rules (U1)-(U6) in Fig. 11.10). Because these rigid type variables have a special status (they are not allowed in the domain of unifiers), they are not allowed in the set Var. However, we define the new variable set SVar (where "S" stands for substituable, because we allow βs to be renamed as αs do when, e.g., instantiating type schemes, where type schemes are redefined below) that contains all the variables in Var plus the rigid type variables. Type variables of the form α will now be referred as flexible type variables in contrast with rigid type variables of the form β. The set $\mathsf{FRTyVar}$ contains the flexible ("F") and rigid ("R") type variables. The terminology used to distinguish between type variables[3] is borrowed from Pottier and Rémy's implementation of their constraint system [116]. In Pottier and Rémy's implementation of their constraint system [116], a type scheme is as follows: $\forall\overline{X}.\exists\overline{Y}.[C]id_1{:}T_1\cdots id_n{:}T_n$ where $\overline{X}$ is a rigid type variable set, $\overline{Y}$ is a flexible type variable set, C is a constraint, and the T_i are types all constrained by the constraint C. Such a type scheme can bind more than one identifier. They explain that for such a type scheme to be considered consistent, the constraint $\forall\overline{X}.\exists\overline{Y}.C$ must hold [4]. They also write: "Rigid and flexible quantifiers otherwise play the same role, that is, they all end up universally quantified in the type scheme", which is why we consider two distinct sets of variables for flexible and rigid type variables and why both kinds are allowed to be universally quantified over.

Let us extend the definition of atoms, originally introduces in Sec. 11.3, as follows: let $\mathsf{atoms}(x)$ be the set of syntactic forms belonging to $\mathsf{SVar} \cup \mathsf{TyConName} \cup \mathsf{Dependency}$ and occurring in x whatever x is. Let $\mathsf{svars}(x) = \mathsf{atoms}(x) \cap \mathsf{SVar}$.

We extend the form of the explicit type variable binders and the form of type schemes as follows:

$$\downarrow tv{=}\alpha \xrightarrow{\mathsf{Bind}} \downarrow tv{=}\rho \qquad\qquad \forall\overline{\alpha}.\,\tau \xrightarrow{\mathsf{Scheme}} \forall\overline{\rho}.\,\tau$$

To allow one to instantiate our different universally quantified forms, we redefine renamings as follows:

$$\begin{aligned} \mathit{ren} \in \mathsf{Ren} = \{\mathit{ren} \mid\ & \mathit{ren} = f_1 \cup f_2 \\ & \wedge f_1 \in \mathsf{FRTyVar} \to \mathsf{ITyVar} \\ & \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{TyConVar} \\ & \wedge \mathit{ren} \text{ is injective} \\ & \wedge \mathsf{dj}(\mathsf{dom}(\mathit{ren}), \mathsf{ran}(\mathit{ren}), \mathsf{Dum})\} \end{aligned}$$

Both flexible and rigid type variables are renamed to flexible ones. So, e.g., instantiating the type scheme $\forall\{\alpha\}.\,\alpha{\to}\alpha$ or the type scheme $\forall\{\beta\}.\,\beta{\to}\beta$ both result in a type of the form $\alpha'{\to}\alpha'$.

We also extend our substitutions as follows:

$$\mathit{sub} \in \mathsf{Sub} = \{\mathit{sub} \mid \mathit{sub} = u \cup f \wedge f \in \mathsf{RigidTyVar} \to \mathsf{ITy}\}$$

[3] *Flexible* is the term usually used for existentially quantified variables and *rigid* is the term usually used for universally quantified variables.

[4] See documentation at the following location `http://www.pps.jussieu.fr/~yrg/software/mini-doc/Constraint.html`.

Therefore, $\mathsf{Ren} \subset \mathsf{Sub}$ and $\mathsf{Ren} \not\subseteq \mathsf{Unifier}$.

We extend the application of a substitution to a constraint term as follows:

$$svar[sub] = \begin{cases} x, & \text{if } sub(svar) = x \\ svar, & \text{otherwise} \end{cases}$$

Let us now define another kind of substitution called *ins* because used to deal with `ins` environments. Note that a *ins* is a *sub*: $\mathsf{Ins} \subseteq \mathsf{Sub}$. Instantiations are defined as follows:

$$ins \in \mathsf{Ins} = \{f \mid f \in \mathsf{TyConVar} \rightarrow \mathsf{TyConName} \wedge f \text{ is injective}\}$$

An environment of the form $\mathtt{ins}(e)$ is an instance of the environment e where internal type constructor variables are instantiated to internal type constructor names. Such an instantiation is performed using an *ins* as defined above.

The table above also introduces subtyping constraints of the forms $\sigma_1 \preceq_{vid} \sigma_2$ and $\kappa_1 \preceq_{tc} \kappa_2$. Checking, e.g., that σ_1 is a subtype of σ_2 (that σ_1 is at least as general as σ_2, or equivalently as written in The Definition of Standard ML [107, Sec.5.5], that σ_1 is "more polymorphic" than σ_2[5]) results in a new type scheme built from both σ_1 and σ_2. The identifier in such a constraint is used to bind the newly built type scheme at constraint solving. Therefore, a subtyping constraint of the form $\sigma_1 \preceq_{vid} \sigma_2$ is both a constraint and an environment because it constrains σ_1 to be a subtype of σ_2 and also can be responsible for the generation of a binder of the form $\downarrow vid{=}\sigma$ at constraint solving, where σ is computed from both σ_1 and σ_2. Subtyping constraints are only generated at constraint solving and not at initial constraint generation. They are generated when dealing with constraints of the form $e_1{:}e_2$ which are used to check that the validity of signature constraints on structures. When a signature constraint *sigexp* on a structure *strexp* is valid SML code, we sometimes say that *sigexp* matches *strexp*. For example, in example (EX4) the signature `s` matches the structure `S`.

Our subtyping relation departs from usual subtyping relations. Usually a type scheme σ_1 is a subtype of a type scheme σ_2 iff each function that is typed by the scheme σ_1 in a type environment can also be typed by the type scheme σ_2 in the same type environment. For example, `1` can have type `int` but cannot be associated the type $\forall\{\alpha\}.\,\alpha$. However, in our system $\mathtt{int} \preceq_{vid} \forall\{\alpha\}.\,\alpha$ is solvable ($\forall\{\alpha\}.\,\alpha \preceq_{vid} \mathtt{int}$ is also solvable). We elaborate on this below. Our definition departs from usual subtyping relations by the fact that $\forall\overline{\alpha_1}.\,\tau_1$ is a subtype of $\forall\overline{\alpha_2}.\,\tau_2$ iff $\tau_1[ren_1]$ can be made equal to $\tau_2[ren_2]$ for some renamings ren_1 and ren_2, where ren_1 renames the flexible and rigid type variables of τ_1 to "fresh" flexible type variables and where ren_2 only renames the flexible type variables of τ_2 to "fresh"

[5] Milner et al. [107] write $\sigma_1 \prec \sigma_2$ to mean that σ_2 is "more polymorphic" than σ_1. Moreover, using their notation $\sigma_1 \prec \sigma_2$ iff for all monomorphic type τ, if $\tau \prec \sigma_1$ (τ is an instance of the type scheme σ_1) then $\tau \prec \sigma_2$.

flexible variables. The renaming ren_2 does not rename rigid type variables because in type schemes, rigid type variables are used for type variables that are not allowed to be more specific whereas flexible type variables can be more specific (constrained further to be equal to type constructs). Rigid type variables give us a control on the (enforced) generality of type schemes. Therefore, the type scheme $\forall\{\beta\}.\,\beta$ cannot be more specific while $\forall\{\alpha\}.\,\alpha$ could potentially have been more specific if some constraint filtering had not occurred. In our system, `int` $\preceq_{vid} \forall\{\beta\}.\,\beta$ is not solvable but $\forall\{\beta\}.\,\beta \preceq_{vid}$ `int` is.

We associate rigid type variables with explicit type variables because of the generality imposed by the explicit type variables. Thus, allowing explicit type variables to bind rigid type variables and not only flexible ones helps us catch *too general* errors as presented above. We also add the new form `tv` to the internal type constructor set. Intuitively, a rigid type variable of the form β can turn into a flexible one but as long as it is rigid, it is considered as a constant type with which is associated the type name `tv`.

Let us illustrate why rigid type variables are vital using the following piece of code (the same as (EX3) where we replaced `'a` by `bool` in `x`'s specification):

(EX5)

```
signature s = sig val x : bool end
structure S = struct val x = 1 end
structure T = S :> s
```

Given this piece of code, our enumeration algorithm would find the type error that `x` is specified as a Boolean in `s`, which is the signature constraining `S` in `T`'s definition, and that `x` is declared as an integer in `S`. The issue is that our minimisation algorithm would eventually try to slice out the type `bool` in `x`'s specification. This would result in `x` having a type scheme of the form $\forall\{\alpha\}.\,\alpha$ in its specification. In our system, as discussed above, $\forall\{\alpha\}.\,\alpha$ and `int` are both subtypes of each other. Usually, $\forall\{\alpha\}.\,\alpha$ is considered a subtype of `int` but `int` is not considered a subtype of $\forall\{\alpha\}.\,\alpha$. Now, if we were to bind explicit type variables occurring in value specifications to flexible type variables, we would also generate a type scheme of the form $\forall\{\alpha\}.\,\alpha$ for `x`'s specification in (EX3) (instead of a type scheme of the form $\forall\{\beta\}.\,\beta$ which is currently generated by our system when no constraint is filtered out). We then would not be able to distinguish between a type scheme which is genuinely too general (in (EX3)) and a type scheme which is too general because some information has been discarded (in (EX5) where `bool` has been filtered out). In order to avoid that, explicit type variables occurring in a signature are not bound to flexible type variables but to rigid type variables.

Let rigtyvars(x) be the set of rigid type variables (in RigidTyVar) occurring in x whatever x is. Let tyconvars(x) be the set of internal type constructor variables (in TyConVar) occurring in x whatever x is.

Let the function labtyvars, which computes the set of labelled explicit type variables occurring in an explicit type, be defined as follows:

$$\begin{aligned}
\mathsf{labtyvars}(tv^{l}) &= \{tv^{l}\}\\
\mathsf{labtyvars}(ty_1 \xrightarrow{l} ty_2) &= \mathsf{labtyvars}(ty_1) \cup \mathsf{labtyvars}(ty_2)\\
\mathsf{labtyvars}(\lceil ty\ ltc \rceil^{l}) &= \mathsf{labtyvars}(ty)
\end{aligned}$$

Let the function tyvars, which computes the set of explicit type variables occurring in an explicit type, be defined as follows:

$$\mathsf{tyvars}(ty) = \{tv \mid tv^{l} \in \mathsf{labtyvars}(ty)\}$$

This function is used by rule (G35) in Fig 14.16 to generate explicit type variable binders for explicit type variables occurring in value specifications.

We extend the application of a substitution to a constraint term as follows:

$$\begin{aligned}
x_1 \preceq_{id} x_2[sub] &= x_1[sub] \preceq_{id} x_2[sub]\\
(e_1{:}e_2)[sub] &= e_1[sub]{:}e_2[sub]\\
\mathtt{ins}(e)[sub] &= \mathtt{ins}(e[sub])
\end{aligned}$$

14.7.3 Constraint generation

Fig. 14.16 presents the new constraint generation rules to handle signature related syntactic forms introduced above.

Note that rules (G32), (G33) and (G34) for signature declarations and expressions are similar to rules (G20), (G21) and (G22), defined in Fig. 11.10, for structure declarations and expressions. Rule (G32) differs from rule (G20) by the generation of the quantification over the internal type constructor variables occurring in the bound structure expression.

Rule (G35) is a simplified version of rule (G17) for recursive value declarations (defined in Fig. 11.10), where the expression is replaced by an external type and where the pattern is reduced to a single value identifier. The novelty in this rule is the binding of the explicit type variables occurring in the external type. To do so, it uses the function tyvars. For example, for the specification `val f : 'a -> 'a` we would generate a binder of the form $\downarrow$`'a`$=\beta$. The order in which the binders are generated does not matter. For example, it does not matter whether for `val f : 'a -> 'b`, `'a`'s binder or `'b`'s binder is generated first.

Rule (G36) is similar to rule (G30) defined in Fig. 14.8, but instead of binding the specified type constructor to an internal type computed from an external type, it leaves the generated internal type constructor variable unconstrained (the variable occurring in the generated binder). Such a variable might then be captured by a $\forall$ when declaring a signature, or constrained by an internal type constructor when a signature is matched against a structure during constraint solving.

Signature declarations $(\mathit{sigdec} \mathrel{-\!\triangleright} e)$

$$\text{(G32)}\ \texttt{signature}\ \mathit{sigid} \overset{l}{=} \mathit{sigexp} \mathrel{-\!\triangleright} ev'{=}(e;\downarrow\mathit{sigid} \overset{l}{=} ev);ev'^{l} \ \Leftarrow\ \mathit{sigexp} \mathrel{-\!\triangleright} \langle ev,\ e\rangle \wedge \mathsf{dja}(e, ev')$$

Signature expressions $(\mathit{sigexp} \mathrel{-\!\triangleright} \langle ev,\ e\rangle)$

$$\text{(G33)}\ \mathit{sigid}^{l} \mathrel{-\!\triangleright} \langle ev,\ \uparrow\mathit{sigid} \overset{l}{=} ev\rangle$$

$$\text{(G34)}\ \texttt{sig}^{l}\ \mathit{spec}_1 \cdots \mathit{spec}_n\ \texttt{end} \mathrel{-\!\triangleright} \langle ev,\ (ev \overset{l}{=} ev');(ev'{=}(e_1;\cdots;e_n))\rangle$$
$$\Leftarrow\ \mathit{spec}_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge \mathit{spec}_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,ev,ev')$$

Specifications $(\mathit{spec} \mathrel{-\!\triangleright} e)$

$$\text{(G35)}\ \texttt{val}\ \mathit{vid}\ \texttt{:}^{l}\ \mathit{ty} \mathrel{-\!\triangleright} (ev{=}\texttt{poly}(\texttt{loc}\downarrow tv_1 \overset{l}{=} \beta_1;\cdots;\downarrow tv_n \overset{l}{=} \beta_n\ \texttt{in}\,(e;\downarrow\mathit{vid} \overset{l}{=} \langle\alpha,\mathtt{v}\rangle)));ev^{l}$$
$$\Leftarrow\ \mathit{ty} \mathrel{-\!\triangleright} \langle\alpha,\ e\rangle \wedge \mathsf{tyvars}(\mathit{ty}) = \{tv_1,\ldots,tv_n\} \wedge \mathsf{dja}(e,ev,\beta_1,\ldots,\beta_n)$$

$$\text{(G36)}\ \texttt{type}\ \mathit{dn}^{l} \mathrel{-\!\triangleright} (ev{=}e);ev^{l} \ \Leftarrow\ \mathit{dn} \mathrel{-\!\triangleright} \langle\delta,\alpha,e,e'\rangle \wedge \mathsf{dja}(e,e',ev)$$

$$\text{(G37)}\ \texttt{structure}\ \mathit{strid}\ \texttt{:}^{l}\ \mathit{sigexp} \mathrel{-\!\triangleright} (ev'{=}(e;(\downarrow\mathit{strid} \overset{l}{=} ev)));ev'^{l} \ \Leftarrow\ \mathit{sigexp} \mathrel{-\!\triangleright} \langle ev,\ e\rangle \wedge \mathsf{dja}(e,ev')$$

$$\text{(G38)}\ \texttt{datatype}\ \mathit{dn} \overset{l}{=} \mathit{cd} \mathrel{-\!\triangleright} (ev{=}((\alpha_2 \overset{l}{=} \alpha_1\,\delta_1);e_1;\texttt{loc}\,e_1'\ \texttt{in}\,\texttt{poly}(e_2)));ev^{l}$$
$$\Leftarrow\ \mathit{dn} \mathrel{-\!\triangleright} \langle\delta_1,\alpha_1,e_1,e_1'\rangle \wedge \mathit{cd} \mathrel{-\!\triangleright} \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\gamma,ev)$$

Structure expressions

$$\text{(G39)}\ \mathit{strexp}\ \texttt{:}^{l}\ \mathit{sigexp} \mathrel{-\!\triangleright} \langle ev,\ e_2;e_1;(ev \overset{l}{=} ev_1{:}ev_2)\rangle$$
$$\Leftarrow\ \mathit{strexp} \mathrel{-\!\triangleright} \langle ev_1,\ e_1\rangle \wedge \mathit{sigexp} \mathrel{-\!\triangleright} \langle ev_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,ev)$$

$$\text{(G40)}\ \mathit{strexp}\ \texttt{:>}^{l}\ \mathit{sigexp} \mathrel{-\!\triangleright} \langle ev,\ e_2;e_1;(ev_{\texttt{dum}} \overset{l}{=} ev_1{:}ev_2);(ev \overset{l}{=} \texttt{ins}(ev_2))\rangle$$
$$\Leftarrow\ \mathit{strexp} \mathrel{-\!\triangleright} \langle ev_1,\ e_1\rangle \wedge \mathit{sigexp} \mathrel{-\!\triangleright} \langle ev_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,ev)$$

Programs $(\mathit{prog} \mathrel{-\!\triangleright} e)$

$$\text{(G41)}\ \mathit{topdec}_1\,\texttt{;}\cdots\texttt{;}\,\mathit{topdec}_n \mathrel{-\!\triangleright} e_1;\cdots;e_n \ \Leftarrow\ \mathit{topdec}_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge \mathit{topdec}_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,ev)$$

Figure 14.16 Constraint generation rules for signatures

Rule (G37) is similar to rule (G20) defined in Fig. 11.10 where, as for type specifications, the generated type constructor variables are left unconstrained. Rule (G38) is similar to rule (G18) defined in Fig. 14.8.

The constraint generation rules for constructor descriptions are the same as the ones for constructor declarations: rules (G14) and (G16) defined in Fig. 11.10.

Finally, rules (G39) and (G40) are the most interesting rules. They are the ones generating our new environments of the forms $e_1{:}e_2$. Rule (G39) generates such forms for translucent signature constraints and rule (G40) for opaque signature constraints. As opposed to rule (G39), rule (G40) for opaque signature constraints also generates $\texttt{ins}(e)$ forms. Rule (G39) generates constraints for a structure constrained by a translucent signature. The environment associated with the analysed constrained structure is computed from an environment of the form $e_1{:}e_2$. It is then obtained from both the environment generated for the structure expression and the environment generated for the signature expression. Rule (G40) generates constraints for a structure constrained by an opaque signature. The environment associated with the analysed constrained structure is not computed from an environment of the form $e_1{:}e_2$ (such an environment is still generated to check that the signature matches the structure) but from an environment of the form $\texttt{ins}(e)$. It is then obtained from the environment generated for the signature expression only.

Because our initial constraint generation algorithm generates new forms of constraints, we extend the *lbind* and *lc* forms as follows (see Sec. 11.5.2):

$$\begin{aligned} \mathit{lbind} &\in \mathsf{LabBind} ::= \cdots \mid {\downarrow}\mathit{sigid} \overset{l}{=} \mathit{ev} \\ \mathit{lc} &\in \mathsf{LabCs} \quad ::= \cdots \mid \mathit{ev} \overset{l}{=} \mathit{ev}_1{:}\mathit{ev}_2 \mid \mathit{ev} \overset{l}{=} \mathtt{ins}(\mathit{ev}') \end{aligned}$$

We also replace the initially generated external type variable binders as follows:

$$\downarrow tv \overset{l}{=} \alpha \xrightarrow{\mathsf{LabBind}} \downarrow tv \overset{l}{=} \rho$$

14.7.4 Constraint solving

First, let us extend constraint solving states and error kinds as follows:

$$\begin{aligned} \mathit{state} &\in \mathsf{State} \quad ::= \cdots \mid \mathtt{match}(\Delta, \overline{d}, e_1, e_2) \\ \mathit{ek} &\in \mathsf{ErrKind} ::= \cdots \mid \mathtt{tyVarClash} \mid \mathtt{tooGeneral}(\mu_1, \mu_2) \end{aligned}$$

Error kinds of the form $\mathtt{tooGeneral}(\mu_1, \mu_2)$ are for type errors as the one described above (*too general* errors), where a signature constrains a structure and is more general than the structure. Error kinds of the form `tyVarClash` are for type errors such that the one in the following piece of code:

```
signature s = sig val f : 'a -> 'b end
structure S = struct val rec f = fn x => x end
structure T = S :> s
```

In this piece of code, `f` is specified in the signature `s` as a function where its argument's type can differ from its body's type. In the structure `S`, the function `f` is declared as the identity function and so its argument's type has to be the same as its body's type. Finally `S` is constrained by `s`. Therefore, we report an explicit type variable clash between `'a` and `'b`. This is a special kind of *too general* errors.

We also need to extend our unifiers as follows (note that this extension also extends Sub):

$$\begin{aligned} u \in \mathsf{Unifier} = \{\textstyle\bigcup_{i=1}^{4} f_i \mid\ & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\ & \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\ & \wedge f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\ & \wedge f_4 \in \mathsf{SigSemVar} \to \mathsf{SigSem}\} \end{aligned}$$

We now allow flexible and rigid type variables to be quantified over when generating type schemes. Fig. 14.17 updates the toPoly function. The only difference with the definition in Fig. 14.2 is that the type variable set generalised over can now contain both flexible and rigid type variables.

Let us define the function scheme that computes a *for all* quantified form from a variable set, a unifier and a constraint term (either an internal type or an internal type constructor):

$$\mathsf{scheme}(u, \overline{svar}, x) = \forall \overline{svar} \cap \mathsf{svars}(x').\, x',\ \text{if } x' = \mathsf{build}(u, x)$$

$$\begin{array}{lll}
\mathsf{toPoly}(\mathit{\Delta}, {\downarrow}\mathit{vid}{=}\tau) & = \mathit{\Delta};({\downarrow}\mathit{vid} \overset{\overline{d}}{=} \forall\overline{\rho}.\,\tau'), \text{ if } \left\{\begin{array}{ll} \tau' & = \mathsf{build}(\mathit{\Delta}, \tau) \\ \overline{\rho} & = (\mathsf{vars}(\tau') \cap \mathsf{FRTyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\mathit{\Delta})) \cup \{\alpha_{\mathtt{dum}}\}) \\ \overline{d} & = \{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\mathit{\Delta}) \wedge \alpha \in \mathsf{vars}(\tau') \setminus \overline{\rho}\} \end{array}\right. & \\
\mathsf{toPoly}(\langle u,\ e\rangle, e_0^{\overline{d}}) & = \langle u',\ e;\mathsf{diff}(e, e')^{\overline{d}}\rangle, & \text{if } \mathsf{toPoly}(\langle u,\ e\rangle, e_0) = \langle u',\ e'\rangle \\
\mathsf{toPoly}(\mathit{\Delta}, e_1;e_2) & = \mathsf{toPoly}(\mathit{\Delta}', e_2), & \text{if } \mathsf{toPoly}(\mathit{\Delta}, e_1) = \mathit{\Delta}' \\
\mathsf{toPoly}(\mathit{\Delta}, e) & = \mathit{\Delta};e, & \text{if none of the above applies}
\end{array}$$

Figure 14.17 Monomorphic to polymorphic environment function generalising flexible and rigid type variables

Rule (B7) of the extension of our constraint solver defined below in Fig. 14.18, needs to build up environments to generate polymorphic forms (for signatures). We therefore need to extend the build function as follows:

$$\begin{array}{ll}
\mathsf{build}(u, {\downarrow}\mathit{id}{=}x) & = ({\downarrow}\mathit{id}{=}\mathsf{build}(u, x)) \\
\mathsf{build}(u, e_1;e_2) & = \mathsf{build}(u, e_1);\mathsf{build}(u, e_2)
\end{array}$$

Fig. 14.18 and Fig. 14.19 extend our constraint solver to deal with our new constraint terms. Fig. 14.18 presents rules to rewrite states of the form $\mathtt{slv}(\mathit{\Delta}, \overline{d}, e)$ and Fig. 14.19 presents rules to rewrite states of the form $\mathtt{match}(\mathit{\Delta}, \overline{d}, e_1, e_2)$.

The new equality constraint simplification rules (S14)-(S17) are defined to handle rigid type variables.

Rules (SM1)-(SM12) check whether a signature matches a structure. These rules are used for both translucent and opaque constraints. If $\mathtt{match}(\mathit{\Delta}, \overline{d}, e_1, e_2) \rightarrow^* \mathtt{match}(\mathit{\Delta}', \overline{d}', e_1', e_2')$ using rules (SM1)-(SM12) then $e_1 = e_1'$. Moreover, e_1 is the environment generated for a structure and e_2 is the environment generated for a signature constraining the structure.

Rules (SU1)-(SU5) handle subtyping constraints. In rule (SU1), the generated type scheme is built from τ_2 and not from τ_1. The type τ_2 is extracted from an environment generated for a signature *sigexp*. The type τ_1 is extracted from an environment generated for a structure constrained by *sigexp*. We do so in case the binding from the signature is a dummy binding. If the binding from the signature is a dummy binding then τ_2 is $\alpha_{\mathtt{dum}}$. If we were to generate a type scheme from τ_1 and not from τ_2, it could result in finding an error that involves a declaration in a structure constrained by a signature without involving the signature. Let us consider the following piece of code:

```
signature s = sig val c : bool end
structure S = struct val c = true end
structure T = S : s
val x = let open T in c () end
```

This piece of code is untypable because c is specified and declared as a Boolean and is also used as a function because it is applied to (). If we were to try to slice out c's specification, we would then generate a dummy binding for c in the environment

Some kinds of errors are not handled by the system presented in this section, although our implementation handles them. For more information please refer to the introductory paragraph of this section (Sec. 14.7).

equality simplification

(S14) $\mathtt{slv}(\mathit{\Delta},\overline{d},\tau_1{=}\tau_2) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d},\mu{=}\mathtt{tv})$, if $\{\tau_1,\tau_2\} = \{\tau\,\mu,\beta\} \wedge \mathsf{strip}(\mu) \in \mathsf{TyConName}$

(S15) $\mathtt{slv}(\mathit{\Delta},\overline{d},\tau_1{=}\tau_2) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d},\mathtt{tv}{=}\mathtt{ar})$, if $\{\tau_1,\tau_2\} = \{\tau_0{\rightarrow}\tau_0',\beta\}$

(S16) $\mathtt{slv}(\mathit{\Delta},\overline{d},\beta_1{=}\beta_2) \rightarrow \mathtt{err}(\langle\mathtt{tyVarClash},\overline{d}\rangle)$, if $\beta_1 \neq \beta_2$

(S17) $\mathtt{slv}(\mathit{\Delta},\overline{d},\mu_1{=}\mu_2) \rightarrow \mathtt{err}(\langle\mathtt{tooGeneral}(\mu_1,\mu_2),\overline{d}\rangle)$, if $\{\mu_1,\mu_2\} \in \{\{\mathtt{tv},\mathtt{ar}\},\{\mathtt{tv},\gamma\}\}$

binders

(B1) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},{\downarrow}id{=}x) \rightarrow \mathtt{succ}(\langle u,\ e\rangle;({\downarrow}id \overset{\overline{d}}{=} x))$, if $id \notin \mathsf{SigId} \cup \mathsf{TyCon}$

(B7) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},{\downarrow}sigid{=}e_1) \rightarrow \mathtt{succ}(\langle u,\ e\rangle;({\downarrow}sigid \overset{\overline{d}}{=} \forall\mathsf{tyconvars}(e_2).\,e_2))$, if $e_2 = \mathsf{build}(u,e_1)$

instantiations

(I1) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},\mathtt{ins}(e_0)) \rightarrow \mathtt{succ}(\langle u,\ e;e_1[ins]\rangle)$,
if $\mathsf{build}(u,e_0) = e_1 \wedge \mathsf{dom}(ins) = \mathsf{tyconvars}(e_1) \wedge \mathsf{dj}(\mathsf{vars}(\langle u,\ e\rangle),\mathsf{ran}(ins))$

signature constraints

(SC1) $\mathtt{slv}(\langle u,\ e\rangle,\overline{d},e_1{:}e_2) \rightarrow \mathtt{match}(\langle u,\ e\rangle,\overline{d},\mathsf{build}(u,e_1),\mathsf{build}(u,e_2))$

subtyping constraints

(SU1) $\mathtt{slv}(\mathit{\Delta},\overline{d},\sigma_1 \preceq_{vid} \sigma_2) \rightarrow \mathtt{succ}(\langle u',\ e';{\downarrow}vid \overset{\overline{d}}{=} \mathsf{scheme}(u',\overline{\rho}_1[ren_1] \cup \overline{\rho}_2[ren_2],\tau_2[ren_2])\rangle)$,
if $\forall i \in \{1,2\}.\ (\sigma_i = \forall\overline{\rho}_i.\,\tau_i \vee (\sigma_i = \tau_i \wedge \overline{\rho}_i = \varnothing \wedge \tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\} \wedge \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}),\mathsf{ran}(ren_1),\mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\mathit{\Delta},\overline{d},\tau_1[ren_1]{=}\tau_2[ren_2]) \rightarrow^* \mathtt{succ}(\langle u',\ e'\rangle)$

(SU2) $\mathtt{slv}(\mathit{\Delta},\overline{d},\sigma_1 \preceq_{vid} \sigma_2) \rightarrow \mathtt{err}(er)$,
if $\forall i \in \{1,2\}.\ (\sigma_i = \forall\overline{\rho}_i.\,\tau_i \vee (\sigma_i = \tau_i \wedge \overline{\rho}_i = \varnothing \wedge \tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\} \wedge \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}),\mathsf{ran}(ren_1),\mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\mathit{\Delta},\overline{d},\tau_1[ren_1]{=}\tau_2[ren_2]) \rightarrow^* \mathtt{err}(er)$

(SU3) $\mathtt{slv}(\mathit{\Delta},\overline{d},\kappa_1 \preceq_{tc} \kappa_2) \rightarrow \mathtt{succ}(\langle u',\ e';{\downarrow}tc \overset{\overline{d}}{=} \mathsf{scheme}(u',\overline{\alpha}_1[ren_1] \cup \overline{\alpha}_2[ren_2],\mu_2[ren_2])\rangle)$,
if $\forall i \in \{1,2\}.\ (\kappa_i = \forall\overline{\alpha}_i.\,\mu_i \wedge \mathsf{dom}(ren_i) = \overline{\alpha}_i) \wedge \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}),\mathsf{ran}(ren_1),\mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\mathit{\Delta},\overline{d},\mu_1[ren_1]{=}\mu_2[ren_2]) \rightarrow^* \mathtt{succ}(\langle u',\ e'\rangle)$

(SU4) $\mathtt{slv}(\mathit{\Delta},\overline{d},\kappa_1 \preceq_{tc} \kappa_2) \rightarrow \mathtt{err}(er)$,
if $\forall i \in \{1,2\}.\ (\kappa_i = \forall\overline{\alpha}_i.\,\mu_i \wedge \mathsf{dom}(ren_i) = \overline{\alpha}_i) \wedge \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}),\mathsf{ran}(ren_1),\mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\mathit{\Delta},\overline{d},\mu_1[ren_1]{=}\mu_2[ren_2]) \rightarrow^* \mathtt{err}(er)$

(SU5) $\mathtt{slv}(\mathit{\Delta},\overline{d},x_1 \preceq_{id} x_2) \rightarrow \mathtt{slv}(\mathit{\Delta},\overline{d} \cup \overline{d}',y_1 \preceq_{id} y_2)$,
if (x_1 is of the form $y_1^{\overline{d}'} \wedge y_2 = x_2$) $\vee$ (x_2 is of the form $y_2^{\overline{d}'} \wedge y_1 = x_1$)

Figure 14.18 Constraint solving for signature related constraints (1)

generated for the signature `s`. Now if we were to use τ_1 instead of τ_2 in rule (SU1) to build `c`'s binder in the environment generated for `T`, we would generate a binder as follows: ${\downarrow}$`c`$=\forall\varnothing.$`bool`. We would then obtain a clash with the arrow type generated for `c ()`. We would then obtain a slice as follows:

```
⟨..structure S = struct val c = true end
 ..structure T = S : ⟨..⟩
 ..⟨..open T..c ()..⟩..⟩
```

However, this is not a complete type error slice (this slice is typable) because `S` might be constrained by a signature that does not specify `c` and therefore `c`'s last occurrence would be free. As a matter of fact, `c` might be defined as a function taking a `unit` in a larger context. A complete, minimal type error slice would be as

structure/signature matching

(SM1) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \top) \rightarrow \mathtt{succ}(\mathit{\Delta})$

(SM2) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, e_1;e_2) \rightarrow \mathtt{match}(\mathit{\Delta}', \overline{d}, e, e_2)$, if $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, e_1) \rightarrow^* \mathtt{succ}(\mathit{\Delta}')$

(SM3) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, e_1;e_2) \rightarrow \mathtt{err}(er)$, if $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, e_1) \rightarrow^* \mathtt{err}(er)$

(SM4) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \downarrow vid{=}\sigma_1) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \sigma_2 \preceq_{vid} \sigma_1)$, if $e(vid) = \sigma_2$

(SM5) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \downarrow tc{=}\kappa_1) \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \kappa_2 \preceq_{tc} \kappa_1)$, if $e(tc) = \kappa_2$

(SM6) $\mathtt{match}(\langle u_1,\ e_1\rangle, \overline{d}, e, \downarrow strid{=}e_0) \rightarrow \mathtt{succ}(\langle u_2,\ e_1;e'^{\overline{d}}\rangle)$,
if $e(strid) = e'_0 \wedge \mathtt{match}(\langle u_1,\ e_1\rangle, \overline{d}, e'_0, e_0) \rightarrow^* \mathtt{succ}(\langle u_2,\ e_2\rangle)$
$\wedge\ e' = (\downarrow strid{=}\mathsf{diff}(e_1, e_2))$

(SM7) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \downarrow strid{=}e_0) \rightarrow \mathtt{err}(er)$,
if $\mathtt{match}(\mathit{\Delta}, \overline{d}, e(strid), e_0) \rightarrow^* \mathtt{err}(er)$

(SM8) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \downarrow vid{=}is_1) \rightarrow \mathtt{succ}(\mathit{\Delta};(\downarrow vid{=}is))$,
if $e[vid] = is_2 \wedge (\mathsf{solvable}(is_1 \overset{\overline{d}}{=} is_2) \vee \mathsf{strip}(is_1) = \mathtt{v}) \wedge is = \mathsf{ifNotDum}(is_1, is_2^{\overline{d}})$

(SM9) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \downarrow vid{=}is_1) \rightarrow \mathtt{err}(er)$,
if $\mathsf{strip}(is_1) \neq \mathtt{v} \wedge \mathtt{slv}(\mathit{\Delta}, \overline{d}, is_1{=}e[vid]) \rightarrow^* \mathtt{err}(er)$

(SM10) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, \downarrow id{=}x) \rightarrow \mathtt{succ}(\mathit{\Delta};(\downarrow id{=}y))$,
if $e(id)$ is undefined $\wedge\ y = \mathsf{toDumVar}(x)$

(SM11) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, ev) \rightarrow \mathtt{succ}(\mathit{\Delta};ev)$

(SM12) $\mathtt{match}(\mathit{\Delta}, \overline{d}, e, e'^{\overline{d}'}) \rightarrow \mathtt{match}(\mathit{\Delta}, \overline{d} \cup \overline{d}', e, e')$

Figure 14.19 Constraint solving for signature related constraints (2)

follows:

```
⟨..signature s = sig val c : ⟨..⟩ end
 ..structure S = struct val c = true end
 ..structure T = S : c
 ..⟨..open T..c ()..⟩..⟩
```

Note that this is not the only type error slice explaining the type error described above, another type error slice involves the signature s and not the structure S.

In rule (SU1) again, from a subtyping constraint of the form $\sigma_1 \preceq_{vid} \sigma_2$, a new type scheme σ is generated from both σ_1 and σ_2. This type scheme is then used to generate a new binder of the form $\downarrow vid{=}\sigma$. Let us explain how this new type scheme σ is generated. Let us assume that σ_1 is of the form $\forall\overline{\rho}_1.\,\tau_1$ and that σ_2 is of the form $\forall\overline{\rho}_2.\,\tau_2$. First, we generate fresh instances of τ_1 and τ_2: τ'_1 and τ'_2 respectively. The type τ'_1 is obtained from τ_1 by renaming the flexible and rigid type variables in $\overline{\rho}_1$ (flexible and rigid type variables are renamed to "fresh" flexible type variables). Because we are checking that τ_2 is not more general than τ_1 and because rigid type variables enforce the generality of type schemes, the type τ'_2 is obtained by renaming only the flexible type variables in $\overline{\rho}_2$. We then check that τ'_1 can be made equal to τ'_2. We finally generate a new type scheme σ by first building up τ'_2 to obtain τ and by then renaming (using the two renamings used to generate τ'_1 from τ_1 and τ'_2 from τ_2) the flexible and rigid type variables in $\overline{\rho}_1 \cup \overline{\rho}_2$ and by quantifying over those occurring in τ. For example, solving the following subtyping constraints:

$$\forall\{\alpha_1\}.\,\alpha_1{\rightarrow}\alpha_1 \preceq_{vid} \forall\{\alpha_2\}.\,\alpha_2$$
$$\forall\varnothing.\,\alpha_{\mathtt{dum}} \preceq_{vid} \forall\{\alpha_2\}.\,\alpha_2{\rightarrow}\alpha_2$$

result in a binder of the form $\downarrow vid{=}\forall\{\alpha\}.\,\alpha{\to}\alpha$. and solving the following subtyping constraints:

$$\begin{array}{l}\forall\{\alpha_1\}.\,\alpha_1 \preceq_{vid} \forall\{\beta\}.\,\beta{\to}\beta\\ \forall\{\alpha_1\}.\,\alpha_1{\to}\alpha_1 \preceq_{vid} \forall\{\alpha_2,\beta\}.\,\alpha_2{\to}\beta\end{array}$$

result in the binder $\downarrow vid{=}\forall\{\beta\}.\,\beta{\to}\beta$. However, solving the following subtyping constraint:

$$\forall\{\alpha_1\}.\,\alpha_1{\to}\alpha_1 \preceq_{vid} \forall\varnothing.\,\alpha_{\mathtt{dum}}$$

results in the dummy binder $\downarrow vid{=}\forall\varnothing.\,\alpha_{\mathtt{dum}}$ and solving the following subtyping constraints:

$$\begin{array}{l}\forall\{\alpha_1\}.\,\mathtt{bool}{\to}\alpha_1 \preceq_{vid} \forall\{\alpha_2\}.\,(\alpha_2{\to}\alpha_2){\to}\alpha_2\\ \forall\{\alpha_1\}.\,\mathtt{bool}{\to}\alpha_1 \preceq_{vid} \forall\{\beta,\alpha_2\}.\,\beta{\to}\alpha_2\end{array}$$

result in type errors (in type constructor clashes).

Because restricted forms of signature binders can now occur in constraint solving contexts (in e in $\langle u,\ e\rangle$), we extend the binder forms generated at constraint solving, originally defined in Sec. 11.6.6, as follows:

$$sbind \in \mathsf{SolvBind} ::= \cdots \mid {\downarrow} sigid{=}\forall\overline{\delta}.\,se$$

Because in constraint solving contexts, type variable binders can now bind flexible as well as rigid type variables, we redefine SolvBind as follows:

$$\downarrow tv{=}\alpha \xrightarrow{\mathsf{SolvBind}} \downarrow tv{=}\rho$$

14.7.5 Constraint filtering (Minimisation and enumeration)

We extend our filtering function as follows:

$$\begin{array}{ll}\mathsf{filt}(e_1{:}e_2,\overline{l}_1,\overline{l}_2) & = \mathsf{filt}(e_1,\overline{l}_1,\overline{l}_2){:}\mathsf{filt}(e_1,\overline{l}_1,\overline{l}_2)\\ \mathsf{filt}(\mathtt{ins}(e),\overline{l}_1,\overline{l}_2) & = \mathtt{ins}(\mathsf{filt}(e,\overline{l}_1,\overline{l}_2))\\ \mathsf{filt}(v,\overline{l}_1,\overline{l}_2) & = v\end{array}$$

We now need the filtering of unlabelled environment variables (we generalise the rule to all kinds of variables) because we now allow unlabelled environment variables to occur within environments of the form $e_1{:}e_2$ or $\mathtt{ins}(e)$. Note that these environments are considered shallow when initially generated (see the extension of LabCs above in Sec. 14.7.3) and are only generated as part of equality constraints. Therefore, we still follow our principle (DP7).

Note that regarding the form of the initially generated environments, our filtering function could be lazier and, e.g., we could just have: $\mathsf{filt}(e_1{:}e_2,\overline{l}_1,\overline{l}_2) = e_1{:}e_2$. We do not adopt this solution which is less robust regarding changes or extensions to the slicer.

We also extend toDumVar as follows:

$$\mathsf{toDumVar}(sig) = ev_{\mathtt{dum}}$$

Signature declarations

$$\mathsf{toTree}(\texttt{signature}\ \mathit{sigid} \overset{l}{=} \mathit{sigexp}) = \langle\langle \texttt{sigdec}, \texttt{sigdecDec}\rangle, l, \langle \mathit{sigid}, \mathsf{toTree}(\mathit{sigexp})\rangle\rangle$$

Signature expressions

$$\begin{aligned}
&\mathsf{toTree}(\mathit{sigid}^{l}) &&= \langle\langle \texttt{sigexp}, \texttt{id}\rangle, l, \langle \mathit{sigid}\rangle\rangle\\
&\mathsf{toTree}(\texttt{sig}^{l}\ \mathit{spec}_1 \cdots \mathit{spec}_n\ \texttt{end}) &&= \langle\langle \texttt{sigexp}, \texttt{sigexpSig}\rangle, l, \langle \mathsf{toTree}(\mathit{spec}_1), \ldots, \mathsf{toTree}(\mathit{spec}_n)\rangle\rangle
\end{aligned}$$

Specifications

$$\begin{aligned}
&\mathsf{toTree}(\texttt{val}\ \mathit{vid}\ \texttt{:}^{l}\ \mathit{ty}) &&= \langle\langle \texttt{spec}, \texttt{specVal}\rangle, l, \langle \mathit{vid}, \mathsf{toTree}(\mathit{ty})\rangle\rangle\\
&\mathsf{toTree}(\texttt{type}\ \mathit{dn}^{l}) &&= \langle\langle \texttt{spec}, \texttt{specTyp}\rangle, l, \langle \mathsf{toTree}(\mathit{dn})\rangle\rangle\\
&\mathsf{toTree}(\texttt{datatype}\ \mathit{dn} \overset{l}{=} \mathit{cd}) &&= \langle\langle \texttt{spec}, \texttt{specDat}\rangle, l, \langle \mathsf{toTree}(\mathit{dn}), \mathsf{toTree}(\mathit{cd})\rangle\rangle\\
&\mathsf{toTree}(\texttt{structure}\ \mathit{strid}\ \texttt{:}^{l}\ \mathit{sigexp}) &&= \langle\langle \texttt{spec}, \texttt{specStr}\rangle, l, \langle \mathit{strid}, \mathsf{toTree}(\mathit{sigexp})\rangle\rangle
\end{aligned}$$

Structure expressions

$$\begin{aligned}
&\mathsf{toTree}(\mathit{strexp}\ \texttt{:}^{l}\ \mathit{sigexp}) &&= \langle\langle \texttt{strexp}, \texttt{strexpTr}\rangle, l, \langle \mathsf{toTree}(\mathit{strexp}), \mathsf{toTree}(\mathit{sigexp})\rangle\rangle\\
&\mathsf{toTree}(\mathit{strexp}\ \texttt{:>}^{l}\ \mathit{sigexp}) &&= \langle\langle \texttt{strexp}, \texttt{strexpOp}\rangle, l, \langle \mathsf{toTree}(\mathit{strexp}), \mathsf{toTree}(\mathit{sigexp})\rangle\rangle
\end{aligned}$$

Programs

$$\mathsf{toTree}(\mathit{topdec}_1\,;\cdots;\mathit{topdec}_n) = \langle \texttt{dotD}, \langle \mathsf{toTree}(\mathit{topdec}_1), \ldots, \mathsf{toTree}(\mathit{topdec}_n)\rangle\rangle$$

Figure 14.20 Extension of toTree to deal with signatures

14.7.6 Slicing

We extend our tree syntax for programs as follows:

$$\begin{aligned}
\mathsf{Class} &::= \cdots \mid \texttt{sigdec} \mid \texttt{sigexp} \mid \texttt{spec}\\
\mathsf{Prod} &::= \cdots\\
&\quad\mid \texttt{sigdecDec}\\
&\quad\mid \texttt{sigexpSig}\\
&\quad\mid \texttt{specVal} \mid \texttt{specTyp} \mid \texttt{specDat} \mid \texttt{specStr}\\
&\quad\mid \texttt{strexpTr} \mid \texttt{strexpOp}
\end{aligned}$$

We also extend our function getDot that associates dot markers with node kinds as follows:

$$\begin{aligned}
\mathsf{getDot}(\langle \texttt{sigdec}, \mathit{prod}\rangle) &= \texttt{dotD}\\
\mathsf{getDot}(\langle \texttt{sigexp}, \mathit{prod}\rangle) &= \texttt{dotS}\\
\mathsf{getDot}(\langle \texttt{spec}, \mathit{prod}\rangle) &= \texttt{dotD}
\end{aligned}$$

Finally, Fig. 14.20 extends our function toTree that transforms a term *term* into a tree *tree*.

14.8 Reporting unmatched errors

There is a kind of error involving signatures that is not handled by the constraint solver as defined above: what we refer to as the "unmatched" errors.

Let us consider the following piece of code:

```
signature s = sig val fool : int end
structure S = struct val foo = 1 val bar = 2 end
structure T = S :> s
```

The specification `fool` from the signature `s` is not matched in the structure `S`, but `s` constrains `S` in `T`'s definition. This error could be solved in many ways, such as: (1) one could replace `fool` by `foo` in `s`, (2) one could replace `foo` by `fool` in `S`, (3) one could change `T`'s definition.

For this error we would like to report that `fool` specified in `s` is not any of `foo` or `bar` declared in `S`, but `s` constrains `S`. For that we need to be able to check that indeed `fool` is not any of `S`'s declarations.

With the system as described above, we cannot report such errors because we do not have any way of knowing whether an environment is constituted by the binders corresponding to *all* the declarations of a structure. As a matter of fact, this is not possible with the current system because of the way constraint filtering can replace environment variables and binders by $\top$.

We will now show how to extend our system to report such errors.

14.8.1 Constraint syntax

Environments are extended with a new empty and satisfied environment as follows:

$$\mathsf{Env} ::= \cdots \mid \odot$$

The meaning of the environment $\odot$ lies in between the meaning of $\top$ and the meaning of environment variables.

The difference between $\top$ and $\odot$ is that $\odot$ will be used to indicate that we filtered out an environment which has the potential to bind (either an environment variable or a binder) and not, say, an equality constraint.

The difference between $\odot$ and an environment variable is that in an environment of the form $(e;\odot)$, the environment $\odot$ does not shadow e.

14.8.2 Constraint solving

The environment $\odot$ is allowed to exist within constraint solving contexts (see Sec. 11.6.6 for the definition of SolvEnvRHS):

$$serhs \in \mathsf{SolvEnvRHS} ::= \cdots \mid \odot$$

Let us extend error kinds as follows:

$$ek \in \mathsf{ErrKind} ::= \cdots \mid \mathtt{unmatched}(id, \overline{id})$$

Fig. 14.21 extends our constraint solver with rules to handle unmatched errors. Rule (SM10) replaces the previous rule (SM10) from Fig. 14.19 and rules (SM13) and (E2) are new.

Rules (SM10) and (SM13) make use of the predicate complete (similar to shadowsAll) which is defined as follows:

Some kinds of errors are not handled by the system presented in this section, although our implementation handles them. For more information please refer to the introductory paragraph of Sec. 14.7.

structure/signature matching

(SM10) $\mathtt{match}(\varDelta, \overline{d}, e, \downarrow id{=}x) \rightarrow \mathtt{succ}(\varDelta;(\downarrow id \overset{\overline{d}}{=} \mathsf{toDumVar}(x)))$,
if $e(id)$ is undefined $\wedge \neg\mathsf{complete}(e)$

(SM13) $\mathtt{match}(\varDelta, \overline{d}, e, \downarrow id{=}x) \rightarrow \mathtt{err}(\langle \mathtt{unmatched}(id, \mathsf{getBinders}(e)), \overline{d}\rangle)$,
if $e(id)$ is undefined $\wedge\ \mathsf{complete}(e)$

(SM14) $\mathtt{match}(\varDelta, \overline{d}, e, \odot) \quad \rightarrow \mathtt{succ}(\varDelta;\odot)$

empty

(E2) $\mathtt{slv}(\varDelta, \overline{d}, \odot) \quad \rightarrow \mathtt{succ}(\varDelta;\odot)$

Figure 14.21 Constraint solving rules handling unmatched errors

$$\mathsf{complete}(e) \Leftrightarrow \begin{cases} (e \text{ of the form } \downarrow id{=}x \text{ and } x \notin \mathsf{Dum}) \\ \text{or } (e \text{ of the form } e_1;e_2 \text{ and } \forall i \in \{1,2\}.\ \mathsf{complete}(e_i)) \\ \text{or } (e \text{ of the form } e'^{\overline{d}} \text{ and } \mathsf{complete}(e')) \\ \text{or } e = \top \end{cases}$$

For example, $\mathsf{complete}(\downarrow vid{=}\sigma)$, $\neg\mathsf{complete}(\odot;\downarrow vid{=}\sigma)$, $\neg\mathsf{shadowsAll}(\odot;\downarrow vid{=}\sigma)$, $\neg\mathsf{complete}(ev;\downarrow vid{=}\sigma)$, and $\mathsf{shadowsAll}(ev;\downarrow vid{=}\sigma)$.

A “solved” environment (occurring in a constraint solving context and of the form se as defined in Fig. 11.6.6 and extended above) is said to be complete if it is not composed by an environment variable, a filtered binder or a dummy binder.

Rule (SM13) makes use of the function getBinders which gathers the identifiers bound in its argument:

$$\begin{aligned} \mathsf{getBinders}(\downarrow id{=}x) &= \{id\} \\ \mathsf{getBinders}(e_1;e_2) &= \mathsf{getBinders}(e_1) \cup \mathsf{getBinders}(e_2) \\ \mathsf{getBinders}(e^{\overline{d}}) &= \mathsf{getBinders}(e) \\ \mathsf{getBinders}(e) &= \varnothing, \text{ if none of the above applies} \end{aligned}$$

14.8.3 Constraint filtering (Minimisation and enumeration)

We add a new rule to filter $\odot$ and update the filtering of labelled environment as follows:

$$\mathsf{filt}(e^l, \overline{l}_1, \overline{l}_2) = \begin{cases} e^l, & \text{if } l \in \overline{l}_1 \setminus \overline{l}_2 \\ \mathsf{dum}(e), & \text{if } l \in \overline{l}_2 \\ \odot, & \text{if } l \notin \overline{l}_1 \cup \overline{l}_2 \text{ and } e \in \mathsf{Var} \cup \mathsf{Bind} \\ \top, & \text{otherwise} \end{cases}$$

$$\mathsf{filt}(\odot, \overline{l}_1, \overline{l}_2) = \odot$$

14.8.4 Slicing

We now need to modify our slicing algorithm. Consider the following piece of code:

```
signature s = sig val x : int val y : bool end
structure S :  s = struct val x = 1 val y = true end
structure T :> s = struct val x = 1 val y = true end
val u = let open T val z = y open S
        in fn w => (w z, w x)
        end
```

where in the fn-expression, `z`'s last occurrence is the `y` from `T` and `x`'s last occurrence comes from `S` via the structure opening. The structures `S` and `T` have the same structure body constrained by the same signature `s`, but `S` has a translucent signature while `T`'s signature is opaque.

This piece of code is untypable because `w` has a monomorphic type and is applied to `z` which is the Boolean `y` defined in `T`, and it is also applied to `x` which is the integer `x` defined in `S`.

With our current slicing algorithm, one of the type error slice we obtain would be as follows:

```
⟨..signature s = sig val x : ⟨..⟩ val y : bool end
 ..structure S :  s = struct val x = 1 end
 ..structure T :> s = ⟨..⟩
 ..⟨..open T..val z = y..open S..fn w => ⟨..w z..w x..⟩..⟩..⟩
```

which is not minimal: `s` does not match `S` because `y` is not declared in `S`.

The problem comes from our tidying of declarations in structure expressions. We therefore need to update our tidying function so that it does not discard empty dot declarations:

$$
\begin{aligned}
&\mathsf{tidy}(\langle\rangle) = \langle\rangle\\
&\mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{tree_1}\rangle, \langle \mathtt{dotD}, \overrightarrow{tree_2}\rangle\rangle @ \overrightarrow{tree})\\
&\quad = \mathsf{tidy}(\langle\langle \mathtt{dotD}, \overrightarrow{tree_1} @ \overrightarrow{tree_2}\rangle\rangle @ \overrightarrow{tree}), \text{ if } \forall tree \in \mathsf{ran}(\overrightarrow{tree_1}).\ \neg\mathsf{declares}(tree)\\
&\mathsf{tidy}(\langle tree\rangle @ \overrightarrow{tree})\\
&\quad = \langle tree\rangle @ \mathsf{tidy}(\overrightarrow{tree}), \text{ if none of the above applies}
\end{aligned}
$$

With this new tidy function, we would then obtain a slice as follows:

```
⟨..signature s = sig val x : ⟨..⟩ val y : bool end
 ..structure S :  s = struct ⟨..⟩ val x = 1 end
 ..structure T :> s = ⟨..⟩
 ..⟨..open T..val z = y..open S..fn w => ⟨..w z..w x..⟩..⟩..⟩
```

where the second occurrence of ⟨..⟩ indicates that some declarations have been sliced out in `S`'s declaration and that therefore `S` is not a "complete" structure.

$$\begin{aligned}
&(\mathsf{G24})\ \mathtt{dot\text{-}d}(\langle \mathit{term}_1,\ldots,\mathit{term}_n\rangle) \mathrel{-\!\rhd} [e_1;\cdots;e_n];\odot\\
&\qquad \Leftarrow\ \mathit{term}_1 \mathrel{-\!\rhd} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\rhd} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n)\\
&(\mathsf{G31})\ \mathtt{dot\text{-}n}(\langle \mathit{term}_1,\ldots,\mathit{term}_n\rangle) \mathrel{-\!\rhd} \langle \alpha,\alpha',\odot,[e_1;\cdots;e_n]\rangle\\
&\qquad \Leftarrow\ \mathit{term}_1 \mathrel{-\!\rhd} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\rhd} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,\alpha,\alpha')\\
&(\mathsf{G25})\ \mathtt{dot\text{-}p}(\langle \mathit{pat}_1,\ldots,\mathit{pat}_n\rangle) \mathrel{-\!\rhd} \langle \alpha,\ e_1;\cdots;e_n;\odot\rangle\\
&\qquad \Leftarrow\ \mathit{pat}_1 \mathrel{-\!\rhd} e_1 \wedge \cdots \wedge \mathit{pat}_n \mathrel{-\!\rhd} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,\alpha)\\
&(\mathsf{G42})\ \mathtt{dot\text{-}c}(\langle \mathit{term}_1,\ldots,\mathit{term}_n\rangle) \mathrel{-\!\rhd} \langle \alpha,\ [e_1;\cdots;e_n];\odot\rangle\\
&\qquad \Leftarrow\ \mathit{term}_1 \mathrel{-\!\rhd} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\rhd} e_n \wedge \mathsf{dja}(e_1,\ldots,e_n,\alpha)
\end{aligned}$$

Figure 14.22 Constraint generation rules to handle incomplete structures and signatures

We also have to replace our constraint generation rule for dot declarations, in order to generate markers of discarded binders: Fig. 14.22 redefines rule (G24) originally defined in Fig. 11.14 in Sec. 11.8.1.

However, this modification is not enough because binders are generated for *cb*s, *pat*s, and *dn*s. For example, we would like to generate a marker of discarded binder for the following declaration: `datatype 'a t =` $\langle$`..`$\rangle$.

First, let us replace the dot terms for *cb*s. We need to do so because we want to generate markers of discarded binders only for *cb* dot terms, but not for expressions and types. We replace these dot terms as follows:

$$\mathtt{dot\text{-}e}(\overrightarrow{\mathit{term}}) \xrightarrow{\mathsf{ConBind}} \mathtt{dot\text{-}c}(\overrightarrow{\mathit{term}})$$

Fig. 14.22 also redefines the constraint generation rules for the forms $\mathtt{dot\text{-}n}(\overrightarrow{\mathit{term}})$ (rule (G31)) and $\mathtt{dot\text{-}p}(\overrightarrow{\mathit{term}})$ (rule (G25)), and we introduce a new constraint generation rule for the forms $\mathtt{dot\text{-}c}(\overrightarrow{\mathit{term}})$ (rule (G42)).

We add a new dot marker to the set Dot as follows:

$$\mathsf{Dot} ::= \cdots \mid \mathtt{dotC}$$

Finally, we extend the toTree function as follows:

$$\mathsf{toTree}(\mathtt{dot\text{-}c}(\overrightarrow{\mathit{term}})) = \langle \mathtt{dotC}, \mathsf{toTree}(\overrightarrow{\mathit{term}})\rangle$$

14.9 Functors

14.9.1 External syntax

First, let us extend our external syntax with functors as follows:

$$\begin{aligned}
&\mathit{funid} &&\in \mathsf{FunId} && \text{(functor identifiers)}\\
&\mathit{strexp} &&\in \mathsf{StrExp} &&::= \cdots \mid \mathit{funid}(\mathit{strexp})^{l}\\
&\mathit{fundec} &&\in \mathsf{FunDec} &&::= \mathtt{functor}\ \mathit{funid}(\mathit{strid} : \mathit{sigexp}) \overset{l}{=} \mathit{strexp}\\
& && && \mid \mathtt{dot\text{-}d}(\overrightarrow{\mathit{term}})\\
&\mathit{topdec} &&\in \mathsf{TopDec} &&::= \cdots \mid \mathit{fundec}\\
&\mathit{id} &&\in \mathsf{Id} &&::= \cdots \mid \mathit{funid}
\end{aligned}$$

Let us consider the following piece of code:

```
                    functor F (S : sig val x : int end) =
(EX6)                 struct open S val y = x + 1 end
                    structure T = F(struct val x = true end)
```

This piece of code is untypable because `F`'s parameter is a structure that must declare an integer `x`, and `F` is applied to a structure that declares a Boolean `x`.

Therefore, for this untypable piece of code, we would like to obtain a type error slice as follows:

```
⟨..functor F (⟨..⟩ : sig val x : int end) = ⟨..⟩
 ..F(struct val x = true end)..⟩
```

Such kinds of errors are relatively easy to find and report because they just involve checking a structure against a signature and we have seen how to do that in Sec. 14.7. However some error reports involving functors are harder to find. For example, more interestingly, (assuming that `+` is the one defined in the Standard ML basis) we would also like to obtain the following slice for the same untypable piece of code ((EX6)):

```
⟨..functor F (S : sig val x : ⟨..⟩ end) =
      ⟨..open S..val ⟨..⟩ = x + ⟨..⟩..⟩
  ..F(struct val x = true end)..⟩
```

This type error slice shows that the functor `F` has a parameter `S` that specifies a value `x` that is used as an integer in `F`'s body. The functor `F` is then applied to a structure that declares `x` as being a Boolean. This means that `x`'s specification in `S`'s signature, must be at least as general as `int` and at most as general as `bool`. Therefore, we obtain a type constructor clash.

This error is more complicated to report than the first one, because it involves constraining the parameter of a functor depending on the types of the bound occurrences of the identifiers specified in the parameter's signature. In our example, it involves constraining `x`'s specification such that it has to be at least as general as the type `int` (e.g., `'a` is at least as general as `int` but `bool` is not) because of its bound occurrence which is constrained to be of type `int` via the use of `+`.

Let us now consider an even trickier example:

```
                functor F (S : sig val x : ⟨..⟩ end) = struct
                  local open S in val rec g = fn y => x end
(EX7)             val _ = (g 1) + 0
                end
                structure T = F(struct val x = true end)
```

In this incomplete piece of code, the signature of `F`'s parameter specifies a value `x` that has an entirely sliced out type. The difference with example (EX6) is that the type of x's occurrence in F's body does not allow one to constrain the type of x's specification because the context of x's occurrence in g's declaration does not constrain its type. Such a way of constraining type schemes is presented below. However, g's type depends on x's type and because of the expression (g 1) + 0, the function g must return integers. This means that x's specification has to be at least as general as the type int. As in example (EX6), because of F's application, x's specification has to be at most as general as bool. So we would like to obtain the following type error slice:

```
⟨..functor F (S : sig val x : ⟨..⟩ end) =
   ⟨..local open S in val rec g = fn ⟨..⟩ => x end
    ..(g ⟨..⟩) + ⟨..⟩..⟩
 ..F(struct val x = true end)..⟩
```

Such a type error slice is harder to obtain than the ones presented above because it involves constraining x's specification depending on its uses but also depending on the uses of the functions using x (and so on).

Let us present a final example that shows the complexity in reporting as much explanations of type errors involving functors as possible:

```
        functor F (S : sig val x : ⟨..⟩ end) = struct
          local open S in val rec g = fn y => x end
(EX8)   end
        structure T = F(struct val x = true end)
        local open T in val _ = (g 1) + 0 end
```

This example differs from example (EX7) by the fact that we took the expression (g 1) + 0 out of F's body. Now, g's occurrence in this expression does not directly refer to g's declaration in F's body but it refers to it through the application of F to struct val x = true end. Because x's specification is totally unconstrained, F's body declares the function g that can take any argument and return anything (because we have sliced out x's type in its specification). Now, because F is applied to a structure that declares x as a Boolean, the structure T declares a function g that has to return a Boolean. Finally, because the last declaration constrains g from T to be a function that returns an integer, we want to obtain the following type error slice:

```
⟨..functor F (S : sig val x : ⟨..⟩ end) =
   ⟨..local open S in val rec g = fn ⟨..⟩ => x end..⟩
 ..structure T = F(struct val x = true end)..⟩
 ..local open T in ⟨..(g ⟨..⟩) + ⟨..⟩end..⟩
```

Note that the complexity discussed above comes from, at it is often the case, dealing with incomplete information (sliced out pieces of code). It is relatively easy to report some type errors involving functors when pieces of code are complete. What we wish to accomplish in this section is designing a TES that reports as close as possible all possible explanations of a programming error involving functors (see, e.g., the two first slices provided in this section).

14.9.2 Constraint syntax

We extend our constraint syntax as follows:

ϕ	$\in \mathsf{FuncVar}$		(set of functor variables)
sv	$\in \mathsf{SchemeVar}$		(set of scheme variables)
fct	$\in \mathsf{Func}$	$::= \phi \mid e_1 \rightsquigarrow e_2 \mid \langle fct, \overline{d} \rangle$	
$fctsem$	$\in \mathsf{FuncSem}$	$::= fct \mid \forall \overline{v}.\, fct \mid \langle fctsem, \overline{d} \rangle$	
$bind$	$\in \mathsf{Bind}$	$::= \cdots \mid \downarrow\! funid{=}fctsem$	
acc	$\in \mathsf{Accessor}$	$::= \cdots \mid \uparrow\! funid{=}\phi$	
c	$\in \mathsf{EqCs}$	$::= \cdots \mid fct_1{=}fct_2$	
e	$\in \mathsf{Env}$	$::= \cdots \mid fct \cdot e \mid \mathtt{lazy}(e)$	
σ	$\in \mathsf{Scheme}$	$::= \cdots \mid \sigma_1 \capdot \sigma_2 \mid sv$	
cap	$\in \mathsf{LazyCapture}$	$::= \langle \tau, sv \rangle$	
v	$\in \mathsf{Var}$	$::= \cdots \mid \phi \mid sv$	

We extend type schemes with *intersection type schemes*. An intersection type scheme is a sequence of type schemes as follows: $\sigma_1 \capdot \cdots \capdot \sigma_n$. We only use restricted forms of intersection type schemes which are as follows: $\sigma \capdot \tau_1 \capdot \cdots \capdot \tau_n \capdot sv$, where σ is not of the form $\sigma' \capdot \sigma''$ and is called the head of the intersection type scheme, and where sv is called its tail. In such an intersection type scheme, the order of the types τ_i is not relevant but is convenient. For example, it allows one to distinguish its head and tail. It is also convenient at constraint solving to have a variable in an intersection type scheme. Moreover, in such an intersection scheme, the τ_i are meant to all be instances of the type scheme σ (the type uses of the identifier with which the intersection type scheme is associated). So for each $i \in \{1, \ldots, n\}$, we have $\sigma \preceq_{vid} \tau_i$ solvable (for some vid). Such an intersection type scheme is also called a lazy type scheme. For example, $(\forall\{\alpha\}.\, \alpha{\rightarrow}\mathtt{int}) \capdot (\mathtt{int}{\rightarrow}\mathtt{int}) \capdot (\mathtt{bool}{\rightarrow}\mathtt{int}) \capdot sv$ would be the lazy type scheme of a function of type $\forall\{\alpha\}.\, \alpha{\rightarrow}\mathtt{int}$ which is used on at least an integer and a Boolean. Such a type scheme would be derived, e.g., for `c` specified in `F`'s parameter in the following incomplete piece of code:

```
          functor F (S : sig c : ⟨..⟩ -> int end) = struct
            open S
(EX9)       val rec g = fn x => c 1
            val rec h = fn x => c true
          end
```

The `lazy` form is used to mark the parameter of a functor. It is used to have a control on which type schemes are transformed into lazy type schemes. We also introduce environments of the form $fct \cdot e$ for applications of functors to structures.

Because we introduced functor variables, we extend Dum as follows: let $\phi_{\mathtt{dum}}$ be a distinct functor variable in $\mathsf{FuncVar}$, and let $\mathsf{Dum} = \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}, \eta_{\mathtt{dum}}, \phi_{\mathtt{dum}}\}$.

We also replace the type schemes of the form $\forall\overline{\rho}.\,\tau$ as follows[6]:

$$\forall\overline{\rho}.\,\tau \xrightarrow{\mathsf{Scheme}} \forall\overline{\rho}.\,\overline{cap} \diamond \tau$$

A type scheme of the form $\forall\overline{\rho}.\,\overline{cap} \diamond \tau$ is a type scheme as defined in Sec. 14.7 (of the form $\forall\overline{\rho}.\,\tau$), augmented with a set of pairs, each composed by a internal type and a type scheme variable. Each type in this set contains at least a variable which is quantified over in the type scheme: in a type scheme of the form $\forall\overline{\rho}.\,\overline{cap} \diamond \tau$ we have $\forall\langle\tau, sv\rangle \in \overline{cap}.\ \neg\mathsf{dj}(\mathsf{vars}(\tau), \overline{\rho})$. Each pair is extracted from a lazy type scheme. Because these forms are not intuitive, let us explain why we need such forms using the following example:

(EX10)

```
functor F (S : sig val c : ⟨..⟩ end) = struct
  local open S in val rec g = fn x => x :: c end
  val _ = g true
end
```

Because `c` is used as a `list` in `F`'s body, we want to obtain a binder as follows for `c` in `F`'s parameter:

$$\downarrow\mathtt{c}{=}(\forall\{\alpha\}.\,\alpha) \Cap \sigma$$

where $\sigma = \alpha_0\,\mathtt{list} \Cap sv$ and α_0 is `x`'s type in `g`'s body. Now, instead of generating the type scheme $\forall\{\alpha_0\}.\,\alpha_0{\to}\alpha_0\,\mathtt{list}$ for `g`, we want to generate a type scheme as follows:

$$\forall\{\alpha_0\}.\,\{\langle\alpha_0\,\mathtt{list}, sv\rangle\} \diamond \alpha_0{\to}\alpha_0\,\mathtt{list}$$

so that the intersection type scheme σ can be constrained further via sv depending on the uses of `g`. In this last type scheme, the type variable α_0 in $\langle\alpha_0\,\mathtt{list}, sv\rangle$, is captured by the universal quantification of the type scheme. Then, when applying `g` to `true` we generate an instance of this type scheme. When doing so, we generate an instance $\alpha_0'{\to}\alpha_0'\,\mathtt{list}$ but we also constrain sv to be equal to $\alpha_0'\,\mathtt{list}{\Cap}sv'$ where α_0' and sv' are fresh variables. Now because `g` is applied to `true` we conclude that α_0' has to be equal to `bool`. So the intersection type scheme σ, which is the list of type uses of `c`, is eventually equal to $(\alpha_0\,\mathtt{list}) \Cap (\mathtt{bool}\,\mathtt{list}) \Cap sv'$. If the functor `F` was applied to a structure, the structure would then have to declare a `c` that can be a list of something (that has a type which is a subtype of $\alpha_0\,\mathtt{list}$ for some α_0), and more precisely, that can be a list of Boolean (that has a type which is a subtype of `bool list`). It is the case for the structures `struct val c = [] end` and `struct val c = [true] end`, but not the case for the structure `struct val c = [()] end`.

[6] Because we have already updated and extended type schemes many times above, let us recall the full definition of type schemes: $\sigma \in \mathsf{Scheme} ::= \tau \mid \forall\overline{\rho}.\,\overline{cap} \diamond \tau \mid \sigma_1 \Cap \sigma_2 \mid sv \mid \langle\sigma, \overline{d}\rangle$

In a type scheme of the form $\forall\overline{\rho}.\,\overline{cap} \diamond \tau$, the flexible type variables in $\overline{\rho}$ that also occur in $\overline{cap}$ are not definitively quantified type variables. Such type schemes occur in binders generated for functors' bodies. The quantification over such variables is conditional and the condition is resolved when applying the functor for which such a type scheme has been generated. For example, the type scheme $\forall\{\alpha_1, \alpha_2\}.\,\{\langle\alpha_2, sv\rangle\} \diamond \alpha_1{\to}\alpha_2$, can turn into $\forall\{\alpha_1\}.\,\alpha_1{\to}\mathtt{unit}$ if the type α_2 from $\langle\alpha_2, sv\rangle$ is constrained to `unit`. This can happen with the following piece of code:

```
functor F (S : sig val c : ⟨..⟩ end) = struct
  local open S in val rec g = fn x => c end
end
structure T = F(struct val c = () end)
```

This will be further illustrated below.

Because of this mechanism, these new type scheme forms cannot be subject to alpha-conversion. For example, the type scheme $\forall\{\alpha_1, \alpha_2\}.\,\{\langle\alpha_2, sv\rangle\} \diamond \alpha_1{\to}\alpha_2$ is not convertible to $\forall\{\alpha_1, \alpha_3\}.\,\{\langle\alpha_3, sv\rangle\} \diamond \alpha_1{\to}\alpha_3$. Note that this is overly restrictive because, given a type schemes of the form $\forall\overline{\rho}.\,\overline{cap} \diamond \tau$, one could safely alpha-convert the type variables in $\overline{\rho} \setminus \mathsf{vars}(\overline{cap})$.

Let $\forall\overline{\rho}.\,\tau$ stand for $\forall\overline{\rho}.\,\varnothing \diamond \tau$

Lazy type schemes of the form $\sigma_1 \mathrel{\dot\cap} \sigma_2$ are meant to be used for functors' parameters and type schemes of the form $\forall\overline{\rho}.\,\overline{cap} \diamond \tau$ where $\overline{cap} \neq \varnothing$ are meant to be used for functors' bodies. Type schemes of the form $\forall\overline{\rho}.\,\overline{cap} \diamond \tau$ where $\overline{cap} \neq \varnothing$ are not meant to be generated for signatures.

Let us now formally define the functions that extract the heads (head) and tails (tail) of intersection type schemes. The functions head and tail are defined as follows:

$$\begin{aligned}
&\mathsf{head}(\sigma_1 \mathrel{\dot\cap} \sigma_2) &&= \mathsf{head}(\sigma_1)\\
&\mathsf{head}(\sigma) &&= \sigma, \text{ if the above does not apply}\\
&\mathsf{tail}(sv, u) &&= \begin{cases} \mathsf{tail}(\sigma, u), & \text{if } u(sv) = \sigma\\ sv, & \text{otherwise}\end{cases}\\
&\mathsf{tail}(\sigma_1 \mathrel{\dot\cap} \sigma_2, u) &&= \mathsf{tail}(\sigma_2, u)\\
&\mathsf{tail}(\sigma, \langle u,\ e\rangle) &&= \mathsf{tail}(\sigma, u)
\end{aligned}$$

Note that tail is undefined if the argument is a sequence that does not end with a variable or if it is neither a variable nor an intersection type scheme.

We extend the application of a substitution to a constraint term as follows:

Functor declarations $(\mathit{fundec} \rightarrowtriangle e)$

(G43) $\texttt{functor}\ \mathit{funid}(\mathit{strid}:\mathit{sigexp}) \overset{l}{=} \mathit{strexp} \rightarrowtriangle (ev{=}(e_1;e'_1;e'_2;e'_3));ev^l$

$$\Leftarrow\ \mathit{sigexp} \rightarrowtriangle \langle ev_1,\ e_1\rangle \wedge \mathit{sigexp} \rightarrowtriangle \langle ev_2,\ e_2\rangle \wedge \mathsf{dja}(e_1, e_2, \phi, ev, ev_0, ev'_0)$$

$$\text{where } \begin{cases} e'_1 = (ev_0 \overset{l}{=} \texttt{lazy}(ev_1)) \\ e'_2 = (ev'_0 \overset{l}{=} \texttt{ins}(ev_0)) \\ e'_3 = \texttt{loc}\,{\downarrow}\mathit{strid} \overset{l}{=} ev'_0\ \texttt{in}\,(e_2;(\phi \overset{l}{=} ev'_0 \rightsquigarrow ev_2);{\downarrow}\mathit{funid} \overset{l}{=} \phi) \end{cases}$$

Structure expressions

(G44) $\mathit{funid}(\mathit{strexp})^l \rightarrowtriangle \langle ev',\ ({\uparrow}\mathit{funid} \overset{l}{=} \phi);e;(ev' \overset{l}{=} \phi \cdot ev)\rangle \Leftarrow\ \mathit{strexp} \rightarrowtriangle \langle ev,\ e\rangle \wedge \mathsf{dja}(e, ev', \phi)$

Figure 14.23 Constraint generation rules for functors

$$\begin{aligned}
(\sigma_1 \mathbin{\dot{\cap}} \sigma_2)[\mathit{sub}] &= \sigma_1[\mathit{sub}] \mathbin{\dot{\cap}} \sigma_2[\mathit{sub}] \\
(\mathit{fct} \cdot e)[\mathit{sub}] &= \mathit{fct}[\mathit{sub}] \cdot e[\mathit{sub}] \\
\texttt{lazy}(e)[\mathit{sub}] &= \texttt{lazy}(e[\mathit{sub}]) \\
(e_1 \rightsquigarrow e_2)[\mathit{sub}] &= e_1[\mathit{sub}] \rightsquigarrow e_2[\mathit{sub}] \\
(\forall\overline{\rho}.\,\overline{\mathit{cap}} \diamond \tau)[\mathit{sub}] &= \begin{cases} \forall\overline{\rho}_2 \cup \overline{\rho}_1[\mathit{sub}].\,\overline{\mathit{cap}}[\overline{\rho}_2 \trianglelefteq \mathit{sub}] \diamond \tau[\overline{\rho}_2 \trianglelefteq \mathit{sub}], \\ \quad \text{if } \overline{\rho}_1 = \overline{\rho} \cap \mathsf{svars}(\overline{\mathit{cap}}) \\ \quad \wedge\ \overline{\rho}_2 = \overline{\rho} \setminus \overline{\rho}_1 \\ \quad \wedge\ \overline{\rho}_1[\mathit{sub}] \subseteq \mathsf{SVar} \\ \quad \wedge\ \mathsf{dj}(\overline{\rho}_2, \mathsf{vars}(\overline{\rho}_2 \trianglelefteq \mathit{sub})) \\ \text{undefined, otherwise} \end{cases}
\end{aligned}$$

14.9.3 Constraint generation

Fig. 14.23 extends our constraint generator with rules to handle functor declarations and functor applications.

Let us detail what rule (G43) does. First with $ev_0{=}\texttt{lazy}(ev_1)$, we switch to a "lazy mode" to deal with the functor's parameter. With $ev'_0{=}\texttt{ins}(ev_0)$, we abstract the types specified in the signature of the functor's parameter. Then we generate two binder. A binder for the functor's parameter which is local to the functor's definition, and a binder for the functor itself.

Because our initial constraint generation algorithm generates new forms of constraints, we extend the *lbind*, *lc*, and *ge* forms as follows (see Sec. 11.5.2):

$$\begin{aligned}
\mathit{lbind} &\in \mathsf{LabBind} ::= \cdots \mid {\downarrow}\mathit{funid} \overset{l}{=} \phi \\
\mathit{lc} &\in \mathsf{LabCs} \quad ::= \cdots \mid \phi \overset{l}{=} ev_1 \rightsquigarrow ev_2 \mid ev \overset{l}{=} \phi \cdot ev' \mid ev \overset{l}{=} \texttt{lazy}(ev')
\end{aligned}$$

14.9.4 Constraint solving

First, we extend our unifiers as follows (note that this extension also extends Sub):

$$\begin{array}{lll}
\mathsf{toPoly}(\mathit{\Delta}, {\downarrow}\mathit{vid}{=}\tau) = \mathit{\Delta};({\downarrow}\mathit{vid} \overset{\overline{d}}{=} \sigma), \text{if} & \left\{\begin{array}{ll} \tau' & = \mathsf{build}(\mathit{\Delta}, \tau) \\ \overline{\rho} & = (\mathsf{vars}(\tau') \cap \mathsf{FRTyVar}) \setminus (\mathsf{vars}(\mathsf{monos}(\mathit{\Delta})) \cup \{\alpha_{\mathtt{dum}}\}) \\ \overline{d} & = \{d \mid \alpha^{\overline{d}_0 \cup \{d\}} \in \mathsf{monos}(\mathit{\Delta}) \wedge \alpha \in \mathsf{vars}(\tau') \setminus \overline{\rho}\} \\ \overline{\mathit{cap}} & = \{\langle \tau, \mathit{sv}\rangle \mid \langle \tau, \mathit{sv}\rangle \in \mathsf{inters}(\mathit{\Delta}) \wedge \neg\mathsf{dj}(\overline{\rho}, \mathsf{vars}(\tau))\} \\ \sigma & = \forall \overline{\rho}.\, \overline{\mathit{cap}} \diamond \tau' \end{array}\right. & \\
\mathsf{toPoly}(\langle u,\ e\rangle, e_0^{\overline{d}}) = \langle u',\ e;e''\rangle, & \text{if } \mathsf{toPoly}(\langle u,\ e\rangle, e_0) = \langle u',\ e'\rangle \wedge e'' = \mathsf{diff}(e, e')^{\overline{d}} & \\
\mathsf{toPoly}(\mathit{\Delta}, e_1;e_2) = \mathsf{toPoly}(\mathit{\Delta}', e_2), & \text{if } \mathit{\Delta}' = \mathsf{toPoly}(\mathit{\Delta}, e_1) & \\
\mathsf{toPoly}(\mathit{\Delta}, e) = \mathit{\Delta};e, & \text{if none of the above applies} &
\end{array}$$

Figure 14.24 Monomorphic to polymorphic environment function handling intersection type schemes

$$\begin{array}{ll}
u \in \mathsf{Unifier} = \{\bigcup_{i=1}^{6} f_i \mid & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\
& \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\
& \wedge f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\
& \wedge f_4 \in \mathsf{SigSemVar} \to \mathsf{SigSem} \\
& \wedge f_5 \in \mathsf{FuncVar} \to \mathsf{Func} \\
& \wedge f_6 \in \mathsf{SchemeVar} \to \mathsf{Scheme}\}
\end{array}$$

We extend the function build to intersection type schemes and functors as follows:

$$\begin{array}{ll}
\mathsf{build}(u, \sigma_1 \capdot \sigma_2) & = \mathsf{build}(u, \sigma_1) \capdot \mathsf{build}(u, \sigma_2) \\
\mathsf{build}(u, e_1 \leadsto e_2) & = \mathsf{build}(u, e_1) \leadsto \mathsf{build}(u, e_2)
\end{array}$$

The intersection type scheme case is used by the functions inters and rebuild defined below.

The function toLazy transforms type schemes into lazy type schemes as follows:

$$\begin{array}{ll}
{\downarrow}\mathit{vid}{=}\sigma \xrightarrow{\mathsf{toLazy}} {\downarrow}\mathit{vid}{=}\sigma \capdot \mathit{sv} & \\
{\downarrow}\mathit{strid}{=}e \xrightarrow{\mathsf{toLazy}} {\downarrow}\mathit{strid}{=}e' & \Leftrightarrow e \xrightarrow{\mathsf{toLazy}} e' \\
e_1;e_2 \xrightarrow{\mathsf{toLazy}} e_1';e_2' & \Leftrightarrow \left\{\begin{array}{l} \forall i \in \{1,2\}.\ e_i \xrightarrow{\mathsf{toLazy}} e_i' \\ \wedge\, \mathsf{dja}(\mathsf{vars}(e_1') \setminus \mathsf{vars}(e_1), \mathsf{vars}(e_2') \setminus \mathsf{vars}(e_2)) \end{array}\right. \\
e^{\overline{d}} \xrightarrow{\mathsf{toLazy}} e'^{\overline{d}} & \Leftrightarrow e \xrightarrow{\mathsf{toLazy}} e' \\
e \xrightarrow{\mathsf{toLazy}} e & \Leftrightarrow \text{if none of the above applies}
\end{array}$$

The complicated rule for environments of the form $e_1;e_2$ is to ensure that no type scheme variable is generated twice.

For example, given the functor:

```
functor F (S : sig val c : ⟨..⟩ end) = struct
  val rec g = fn x => c x
end
```

at constraint solving, we would generate the following binder for `c` in `F`'s parameter: ${\downarrow}\mathtt{c}{=}\forall\{\alpha\}.\,\alpha$. From this binder, when dealing with the constraints generated for `S`, we would then eventually generate a binder of the form: ${\downarrow}\mathtt{c}{=}(\forall\{\alpha\}.\,\alpha) \capdot \mathit{sv}$.

Fig. 14.24 redefines the function toPoly to build our new forms of type schemes. It only differs from Fig. 14.17 by the generation of $\overline{\mathit{cap}}$. It now uses the function inters which extracts the types (and their tails) from the intersection types from a given constraint solving context and which is defined as follows:

$$\mathsf{inters}(\varDelta) = \{\langle\tau, \mathsf{tail}(\sigma, \varDelta)\rangle \mid \exists vid.\ \mathsf{strip}(\varDelta(vid)) = (\sigma_1 \cap \sigma_2) \wedge \tau \cap \sigma \text{ occurs in } \mathsf{build}(\varDelta, \sigma_2)\}$$

We explain below why the constraints annotating intersection type schemes are discarded in inters's definition, i.e., why we use strip.

The way the new toPoly function works was already illustrated above with example (EX10). Still using example (EX10), let us add a word on this function now that it is formally defined. At constraint solving, when dealing with the `poly` environment generated for `g`, using inters we find that the intersection type $\alpha_0\,\texttt{list} \cap sv$ occurs in the current constraint solving context. Because α_0 occurs in this intersection type and also in the built-up monomorphic type $\alpha_0{\rightarrow}\alpha_0\,\texttt{list}$ (τ' in Fig 14.24) generated for `g`'s declaration, we finally generate the type scheme $\forall\{\alpha_0\}.\,\{\langle\alpha_0\,\texttt{list}, sv\rangle\} \diamond \alpha_0{\rightarrow}\alpha_0\,\texttt{list}$ (where $\overline{cap}$ in Fig 14.24 is then $\{\langle\alpha_0\,\texttt{list}, sv\rangle\}$) for `g` that captures the type $\alpha_0\,\texttt{list}$ from the intersection type generated for `c` (the intersection type $\alpha_0\,\texttt{list} \cap sv$).

Let us now explain why the dependencies annotating intersection type schemes are not needed in inters's definition. Let us again consider example (EX10). As explained above, at constraint solving, when dealing with the `poly` environment generated for `g`, using inters we find that the intersection type $\alpha_0\,\texttt{list} \cap sv$ occurs. In the current constraint solving context, this intersection type is labelled by l which `c`'s first occurrence label. We claim it is safe for inters to discard this label. The intuition is that the type $\alpha_0\,\texttt{list}$ by itself (and not the whole binder) is only used to constraint sv further for each of `g`'s use. Therefore, if we were to label $\alpha_0\,\texttt{list}$ with l, this label would eventually be redundant in `c`'s binder. If we were to generate $\forall\{\alpha_0\}.\,\{\langle(\alpha_0\,\texttt{list})^l, sv\rangle\} \diamond \alpha_0{\rightarrow}\alpha_0\,\texttt{list}$ (some dependencies are still omitted for clarity issues) instead of the type scheme presented above, then dealing with the constraints generated for `g true` would lead to constraining sv by an instance of $(\alpha_0\,\texttt{list})^l$. The fully built up binder associated with `c`'s first occurrence would then at this stage be of the form (where again we omit all the dependencies except l) $\downarrow\texttt{c} \overset{l}{=} (\forall\{\alpha\}.\,\alpha) \cap (\alpha_0\,\texttt{list}) \cap (\alpha_0'\,\texttt{list})^l \cap sv'$. We can observe that l's second occurrence is not needed because it occurs in `c`'s binder which already depends on l.

The function rebuild builds up the type uses gathered (in intersection type schemes) while solving the constraints generated for functors. It is defined as follows:

$$\mathsf{rebuild}(u, e_1{\rightsquigarrow}e_2) = \mathsf{build}(u, e_1){\rightsquigarrow}e_2$$

Let us consider again example (EX9). The binder generated for `c` in `F`'s parameter is as follows: $\downarrow\texttt{c}{=}(\forall\{\alpha\}.\,\alpha{\rightarrow}\texttt{int}) \cap sv$. When solving the constraints generated for `F`'s body, we also generate a unifier as follows: $\{sv \mapsto \alpha_1 \cap sv_1, sv_1 \mapsto \alpha_2 \cap sv_2\} \cup u$

such that $\mathsf{build}(u,\alpha_1) = \texttt{int}{\to}\texttt{int}$ and $\mathsf{build}(u,\alpha_2) = \texttt{bool}{\to}\texttt{int}$. When rebuilding the environment generated for `F`'s parameter once the constraints generated for its body have been solved, we obtain the following binder for `c`: $\downarrow\texttt{c}{=}(\forall\{\alpha\}.\,\alpha{\to}\texttt{int}) \mathbin{\dot{\cap}} (\texttt{int}{\to}\texttt{int}) \mathbin{\dot{\cap}} (\texttt{bool}{\to}\texttt{int}) \mathbin{\dot{\cap}} sv_2$.

We define abstractions as follows:

$$abs \in \mathsf{Abs} = \{f \mid f \in \mathsf{TyConName} \to \mathsf{TyConVar} \wedge f \text{ is injective}\}$$

In order to use our substitution notation to apply abstractions to constraint terms, we need first to extend our substitution definition.

We also extend our substitutions as follows:

$$\begin{aligned} sub \in \mathsf{Sub} = \{sub \mid\ & sub = u \cup f_1 \cup f_2 \\ & \wedge f_1 \in \mathsf{RigidTyVar} \to \mathsf{ITy} \\ & \wedge f_2 \in \mathsf{TyConName} \to \mathsf{TyConVar}\} \end{aligned}$$

Therefore, $\mathsf{Abs} \subset \mathsf{Sub}$.

We then extend the application of a substitution to a constraint term as follows:

$$\gamma[sub] = \begin{cases} \mu, \text{if } sub(\gamma) = \mu \\ \gamma, \text{otherwise} \end{cases}$$

Abstractions are used by the relation $\mathsf{abstract}$ which is itself used by rule ($\mathsf{B8}$) of the extension of our constraint solver defined below in Fig. 14.26. The relation $\mathsf{abstract}$ is used to rebuild the environment associated with the parameter of a functor and to abstract the functor over the intersection types and type constructor names defined in its parameter. The relation $\mathsf{abstract}$ is defined as follows:

$$\begin{aligned} &\langle fct, \langle u,\ e\rangle\rangle \xrightarrow{\mathsf{abstract}} \forall\overline{\alpha} \cup \mathsf{ran}(abs).\, fct_1[abs] \\ &\Leftrightarrow \begin{cases} \ \ fct_1 = \mathsf{rebuild}(u, fct) \\ \wedge\, fct_2 = \mathsf{strip}(fct_1) \\ \wedge\, \overline{\alpha} = \{\alpha \mid \tau \mathbin{\dot{\cap}} \sigma \text{ occurs in } fct_1 \wedge \alpha \in \mathsf{vars}(\tau)\} \\ \wedge\, (\text{if } fct_2 = e_1 {\rightsquigarrow} e_2 \text{ then } \overline{\gamma} = \{\gamma \mid \downarrow tc{=}\gamma \text{ occurs in } e_1\} \text{ else } \overline{\gamma} = \varnothing) \\ \wedge\, \mathsf{dom}(abs) = \overline{\gamma} \\ \wedge\, \mathsf{dja}(\mathsf{nonDums}(\langle \Delta,\ e\rangle), \mathsf{ran}(abs)) \end{cases} \end{aligned}$$

For example let us consider the following typable piece of code:

```
functor F (S : sig type t end) = struct
  local open S in datatype u = c of t end
  val rec g = fn x => c x
end
structure T = F (struct type t = int)
```

At constraint solving, when computing `F`'s binder, at first we generate the following fct (again we omit dependencies and the environment $\top$ for readability purposes):

$$(\downarrow\texttt{t}{=}\gamma){\rightsquigarrow}((\downarrow\texttt{u}{=}\forall\varnothing.\,\gamma');(\downarrow\texttt{c}{=}\forall\varnothing.\,\gamma{\to}\gamma');(\downarrow\texttt{g}{=}\forall\varnothing.\,\gamma{\to}\gamma'))$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{genLazy}(\langle u,\ e\rangle, {\downarrow}\mathit{vid}{=}\sigma) = ({\downarrow}\mathit{vid}{=}\forall(\overline{\rho}_1 \cap \mathsf{svars}(\tau')) \cup \overline{\rho}_2.\, \tau'),} \\
\multicolumn{2}{l}{\quad \text{if } \mathsf{head}(\sigma) = \forall\overline{\rho}.\, \overline{\mathit{cap}} \diamond \tau \text{ and } \overline{\rho}_1 = \overline{\rho} \setminus \mathsf{vars}(\overline{\mathit{cap}}) \text{ and } \tau' = \mathsf{build}(\overline{\rho}_1 \lhd u, \tau)} \\
\multicolumn{2}{l}{\quad \text{and } \overline{\rho}_2 = \{\rho \mid \langle \tau_0, sv_0\rangle \in \overline{\mathit{cap}} \wedge \alpha \in \mathsf{vars}(\tau_0) \cap \overline{\rho} \wedge \rho \in \mathsf{svars}(\mathsf{build}(u,\alpha))\}} \\
\mathsf{genLazy}(\mathit{\Delta}, {\downarrow}\mathit{strid}{=}e) & = ({\downarrow}\mathit{strid}{=}\mathsf{genLazy}(\mathit{\Delta}, e)) \\
\mathsf{genLazy}(\mathit{\Delta}, e_1;e_2) & = \mathsf{genLazy}(\mathit{\Delta}, e_1);\mathsf{genLazy}(\mathit{\Delta}, e_2) \\
\mathsf{genLazy}(\mathit{\Delta}, x^{\overline{d}}) & = \mathsf{genLazy}(\mathit{\Delta}, x)^{\overline{d}} \\
\mathsf{genLazy}(\mathit{\Delta}, x) & = x, \text{ if none of the above applies}
\end{array}$$

Figure 14.25 Recomputation of functors' bodies

where e is the environment generated for `F`'s body. Abstracting *fct* allows us to obtain an internal functor semantic as follows:

$$\forall\{\delta\}.\, ({\downarrow}\mathtt{t}{=}\delta) \rightsquigarrow (({\downarrow}\mathtt{u}{=}\forall\varnothing.\, \gamma');({\downarrow}\mathtt{c}{=}\forall\varnothing.\, \delta{\to}\gamma');({\downarrow}\mathtt{g}{=}\forall\varnothing.\, \delta{\to}\gamma'))$$

The function duplicate is used to duplicate intersection types when instantiating a type scheme that captures intersection types (of the form $\forall\overline{\rho}.\, \overline{\mathit{cap}}{\diamond}\tau$ where $\overline{\mathit{cap}} \neq \varnothing$):

$$\begin{array}{l}
\langle\langle u,\ e\rangle, \overline{d}, \overline{\mathit{cap}}\rangle \xrightarrow{\mathsf{duplicate}} \cup_{i=1}^{n}\{sv_i \mapsto \sigma_i\} \\
\Leftrightarrow \left\{\begin{array}{l}
\quad \uplus_{i=1}^{n}\{sv_i \mapsto \overline{\tau}_i\} = \Cup\{\mathsf{tail}(sv, u) \mapsto \{\tau^{\overline{d}}\} \mid \langle \tau, sv\rangle \in \overline{\mathit{cap}}\} \\
\wedge\, \forall i \in \{1, \ldots, n\}.\ \left\{\begin{array}{l}
\quad \overline{\tau}_i = \{\tau_1\} \uplus \cdots \uplus \{\tau_m\} \\
\wedge\, \sigma_i = \tau_1 \mathbin{\dot\cap} \cdots \mathbin{\dot\cap} \tau_m \mathbin{\dot\cap} sv'_i
\end{array}\right. \\
\wedge\, \mathsf{dja}(\mathsf{nonDums}(\langle u,\ e\rangle), sv'_1, \ldots, sv'_n)
\end{array}\right.
\end{array}$$

Let us illustrate the necessity of duplicate using example (EX7) introduced above in this section. The binder generated for `g` at constraint solving is as follows: ${\downarrow}\mathtt{g}{=}\forall\{\alpha\}.\, \{\langle\alpha_0, sv\rangle\} \diamond \alpha{\to}\alpha_0$ where α_0 is an instance of `x`'s type (from its specification) and occurs also in the intersection type scheme associated with `x` (due to `x`'s bound occurrence), and where sv is the tail of the intersection type scheme associated with `x`'s binding occurrence. Because `g` occurs in the expression `(g 1) + 0`, we instantiate this type scheme to obtain a type as follows: $\alpha'{\to}\alpha'_0$ where α' and α'_0 are fresh variables. The type α'_0 is obtained by renaming α_0 from $\langle\alpha_0, sv\rangle$. The predicate duplicate is then used to duplicate α'_0 so that the copy can be added to the intersection type associated with `x` using the intersection variable sv. Because of `(g 1) + 0`, α'_0 is further constrained to be equal to `int` and therefore, `int` occurs in the builtin version of the intersection type associated with `x`'s binding occurrence.

Fig. 14.25 defines the function genLazy. Given the application of a functor to an argument, genLazy computes new type schemes from those generated for the functor's body, which have a head of the form $\forall\overline{\rho}.\, \overline{\mathit{cap}} \diamond \tau$ depending on the types in $\overline{\mathit{cap}}$. Let us illustrate the necessity of genLazy using the following piece of code:

```
functor F (S : sig val f : ⟨..⟩ end) = struct
  local open S in val rec g = fn x => f true end
end
structure T = F(struct val f = fn x => x end)
```

At constraint solving, we eventually generate the following binder for `F` (where again dependencies and the environment $\top$ are omitted for readability purposes):

$$\downarrow\mathtt{F}{=}\forall\{\alpha_2\}.\,e_1{\leadsto}e_2 \quad\text{where}\quad \begin{cases} e_1 = (\downarrow\mathtt{f}{=}(\forall\{\alpha_0\}.\,\alpha_0)\Cap\mathtt{bool}{\rightarrow}\alpha_2\Cap sv) \\ e_2 = (\downarrow\mathtt{g}{=}\forall\{\alpha_1,\alpha_2\}.\,\{\langle\mathtt{bool}{\rightarrow}\alpha_2, sv\rangle\}\diamond\alpha_1{\rightarrow}\alpha_2) \end{cases}$$

The constraint term generated for `F`'s second occurrence is then as follows:

$$e'_1{\leadsto}e'_2 \quad\text{where}\quad \begin{cases} e'_1 = (\downarrow\mathtt{f}{=}(\forall\{\alpha_0\}.\,\alpha_0)\Cap\mathtt{bool}{\rightarrow}\alpha'_2\Cap sv) \\ e'_2 = (\downarrow\mathtt{g}{=}\forall\{\alpha_1,\alpha'_2\}.\,\{\langle\mathtt{bool}{\rightarrow}\alpha'_2, sv\rangle\}\diamond\alpha_1{\rightarrow}\alpha'_2) \end{cases}$$

Note that even though α_2 is quantified in `g`'s type scheme, it is renamed when instantiating `F`'s static semantics because it occurs (and so depends) in the intersection type associated with `f`. Such a type variable is not confirmed yet to be a quantifiable.

Because `F`'s argument is a structure that defines `f` as the identity function, the environment generated for it is as follows:

$$\downarrow\mathtt{f}{=}\forall\{\alpha\}.\,\alpha{\rightarrow}\alpha$$

When checking whether `f`'s binders from `F`'s parameter (in e'_1) and `F`'s argument match, we generate the following constraint:

$$\alpha'_2{=}\mathtt{bool}$$

by first generating an instance of $\forall\{\alpha\}.\,\alpha{\rightarrow}\alpha$ as follows: $\alpha'{\rightarrow}\alpha'$, and by constraining α' to be both equal to `bool` and α'_2 (because of $\mathtt{bool}{\rightarrow}\alpha'_2$ occurring in e'_1).

Thanks to genLazy, the environment generated at constraint solving for `T` is then as follows (generated from `g`'s binder in e'_2):

$$\downarrow\mathtt{g}{=}\forall\{\alpha_1\}.\,\alpha_1{\rightarrow}\mathtt{bool}$$

Fig. 14.26 extends our constraint solver to handle functors. Rules (B1), (A1), (A2), (SU1), and (SU2) are updated and rules (B8), (SU6), (SU7), (FP1), (FP2), (FA1), (FA2), and (FA3) are new. Rule (SU1) for subtype scheme constraints is now more complicated than in Fig. 14.18 mainly because of the computation of $\overline{cap'_1}$ as part of the generated type scheme. Let us illustrate the necessity of this computation using the following piece of code:

```
signature s = sig val g : ⟨..⟩ end
functor F (S : sig val c : ⟨..⟩ end) = struct
  open S
  structure X = struct val rec g = fn x => x :: c end
  structure T = X : s
  local open T in val u = g () end
end
```

The binder generated for `g` declared in `X` is as follows:

binders

(B1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \downarrow id{=}x) \rightarrow \mathtt{succ}(\langle u,\ e\rangle;(\downarrow id \overset{\overline{d}}{=} x))$,
if $id \notin \mathsf{FunId} \cup \mathsf{SigId} \cup \mathsf{TyCon}$

(B8) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \downarrow funid{=}fctsem) \rightarrow \mathtt{succ}(\langle u,\ e\rangle;(\downarrow funid \overset{\overline{d}}{=} fctsem'))$,
if $\langle \mathsf{build}(u, fctsem), \langle u,\ e\rangle\rangle \xrightarrow{\mathsf{abstract}} fctsem'$

accessors

(A1) $\mathtt{slv}(\varDelta, \overline{d}, \uparrow id{=}v) \rightarrow \mathtt{slv}(\varDelta, \overline{d} \cup \overline{d}', v{=}x[ren])$,
if $\varDelta(id) = (\forall \overline{svar}.\, x)^{\overline{d}'} \wedge \mathsf{dom}(ren) = \overline{svar} \wedge \mathsf{dj}(\mathsf{vars}(\langle \varDelta, v\rangle), \mathsf{ran}(ren)) \wedge id \notin \mathsf{VId}$

(A2) $\mathtt{slv}(\varDelta, \overline{d}, \uparrow id{=}v) \rightarrow \mathtt{slv}(\varDelta, \overline{d}, v{=}x)$,
if $\varDelta(id) = x \wedge id \in \mathsf{StrId} \cup \mathsf{TyVar}$

(A5) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \uparrow vid{=}\alpha) \rightarrow \mathtt{succ}(\langle u',\ e\rangle)$,
if $\varDelta(vid) = \sigma \wedge \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma \preceq_{vid} \alpha) \rightarrow^{*} \mathtt{succ}(\langle u',\ e'\rangle)$

(A6) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \uparrow vid{=}\alpha) \rightarrow \mathtt{err}(er)$,
if $\varDelta(vid) = \sigma \wedge \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma \preceq_{vid} \alpha) \rightarrow^{*} \mathtt{err}(er)$

subtyping constraints

(SU1) $\mathtt{slv}(\varDelta, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2) \rightarrow \mathtt{succ}(\langle u_1{\oplus}u_2{\oplus}u_3,\ e';\downarrow vid \overset{\overline{d}}{=} \forall\overline{\rho}.\, \overline{cap}'_1 \diamond \tau\rangle)$,
if $\sigma'_1 = \mathsf{head}(\sigma_1) \wedge \sigma'_2 = \sigma_2$
$\wedge\ \forall i \in \{1,2\}.\ (\sigma'_i = \forall\overline{\rho}_i.\, \overline{cap}_i \diamond \tau_i$ or $(\sigma'_i = \tau_i$ and $\overline{\rho}_i = \overline{cap}_i = \varnothing$ and $\tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\} \wedge \mathsf{dj}(\mathsf{vars}(\varDelta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2), \{sv'\})$
$\wedge\ \mathtt{slv}(\varDelta, \overline{d}, \tau_1[ren_1]{=}\tau_2[ren_2]) \rightarrow^{*} \mathtt{succ}(\langle u_1,\ e'\rangle)$
$\wedge\ \tau = \mathsf{build}(u_1, \tau'_2[ren_2])$
$\wedge\ \overline{\rho} = (\overline{\rho}_1[ren_1] \cup \overline{\rho}_2[ren_2]) \cap \mathsf{svars}(\tau)$
$\wedge\ \langle\langle u_1,\ e'\rangle, \overline{d}, \overline{cap}_1[ren_1]\rangle \xrightarrow{\mathsf{duplicate}} u_2 \wedge sv' \notin \mathsf{vars}(u_2)$
$\wedge$ (if $\mathsf{tail}(\sigma_1, u_1{\oplus}u_2) = sv$ then $u_3 = \{sv \mapsto \tau \Cap sv'\} \wedge \overline{cap} = \{\langle\tau, sv'\rangle\}$ else $u_3 = \overline{cap} = \varnothing$)
$\wedge\ \overline{cap}'_1 = \overline{cap} \cup \{\langle\tau'_0, sv_0\rangle \mid \langle\tau_0, sv_0\rangle \in \overline{cap}_1[ren_1] \wedge \tau'_0 = \mathsf{build}(u_1, \tau_0) \wedge \neg\mathsf{dja}(\mathsf{vars}(\tau'_0), \overline{\rho})\}$

(SU2) $\mathtt{slv}(\varDelta, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2) \rightarrow \mathtt{err}(er)$,
if $\sigma'_1 = \mathsf{head}(\sigma_1) \wedge \sigma'_2 = \sigma_2$
$\wedge\ \forall i \in \{1,2\}.\ (\sigma'_i = \forall\overline{\rho}_i.\, \overline{cap}_i \diamond \tau_i$ or $(\sigma'_i = \tau_i$ and $\overline{\rho}_i = \overline{cap}_i = \varnothing$ and $\tau_i \notin \mathsf{Dependent}))$
$\wedge\ \mathsf{dom}(ren_1) = \overline{\rho}_1 \wedge \mathsf{dom}(ren_2) = \{\alpha \mid \alpha \in \overline{\rho}_2\} \wedge \mathsf{dj}(\mathsf{vars}(\varDelta), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\wedge\ \mathtt{slv}(\varDelta, \overline{d}, \tau_1[ren_1]{=}\tau_2[ren_2]) \rightarrow^{*} \mathtt{err}(er)$

(SU6) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2 \Cap \sigma_3) \rightarrow \mathtt{slv}(\langle u',\ e\rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2)$,
if $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_3) \rightarrow^{*} \mathtt{succ}(\langle u',\ e'\rangle)$

(SU7) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_2 \Cap \sigma_3) \rightarrow \mathtt{err}(er)$,
if $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, \sigma_1 \preceq_{vid} \sigma_3) \rightarrow^{*} \mathtt{err}(er)$

functor parameters

(FP1) $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, \mathtt{lazy}(e)) \rightarrow \mathtt{succ}(\langle u_2,\ e_1;e'\rangle)$,
if $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, e) \rightarrow^{*} \mathtt{succ}(\langle u_2,\ e_2\rangle) \wedge \mathsf{diff}(e_1, e_2) \xrightarrow{\mathsf{toLazy}} e'$

(FP2) $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, \mathtt{lazy}(e)) \rightarrow \mathtt{err}(er)$,
if $\mathtt{slv}(\langle u_1,\ e_1\rangle, \overline{d}, e) \rightarrow^{*} \mathtt{err}(er)$

functor applications

(FA1) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, fct \cdot e) \rightarrow \mathtt{succ}(\varDelta';\mathsf{genLazy}(\varDelta', e_2^{\overline{d}'}))$,
if $\mathsf{build}(u, fct) = (e_1 {\rightsquigarrow} e_2)^{\overline{d}'} \wedge \mathtt{slv}(\langle u,\ e\rangle, \overline{d} \cup \overline{d}', e{:}e_1) \rightarrow^{*} \mathtt{succ}(\varDelta')$

(FA2) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, fct \cdot e) \rightarrow \mathtt{err}(er)$,
if $\mathsf{build}(u, fct) = (e_1 {\rightsquigarrow} e_2)^{\overline{d}'} \wedge \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e{:}e_1) \rightarrow^{*} \mathtt{err}(er)$

(FA3) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, fct \cdot e) \rightarrow \mathtt{succ}(\langle u,\ e\rangle)$,
if $\mathsf{strip}(\mathsf{build}(u, fct)) \in \mathsf{Var}$

Figure 14.26 Constraint solving rules for functors

$$\downarrow\mathtt{g}{=}\forall\{\alpha_0\}.\, \{\langle\alpha_0\,\mathtt{list}, sv\rangle\} \diamond \alpha_0{\rightarrow}\alpha_0\,\mathtt{list}$$

where sv is the tail of `c`'s binder and where $\alpha_0\,\mathtt{list}$ is the type generated for `c`'s bound occurrence. The type scheme generated for `g` specified in `s` is as follows: $\downarrow\mathtt{g}{=}\forall\{\alpha\}.\, \alpha$. When checking whether `g`'s specification matches `g`'s declaration (when dealing with the constraint generated for `X : s`), we generate the following binder for `g`'s declaration in `T`:

$$\downarrow\mathtt{g}{=}\forall\{\alpha_1\}.\,\{\langle\alpha_1\,\mathtt{list}, sv\rangle\}\diamond\alpha_1{\to}\alpha_1\,\mathtt{list}$$

where $\{\langle\alpha_1\,\mathtt{list}, sv\rangle\}$ is $\overline{cap}'_1$ in rule (SU1) ($\overline{cap}$ is empty). We also constrain sv to be equal $\alpha_1\,\mathtt{list}\Cap sv'$ via duplicate. Instantiating the type scheme generated for `g` in `T` leads to the further constraining of sv', and therefore to the further constraining of sv as well. For example, because `g` is applied to `()` in `u`'s body, sv' is then eventually constrained to be equal to $\mathtt{unit}\,\mathtt{list}\Cap sv''$.

Let us now consider a similar example, where `g`'s specification which was sliced in our previous example, has been replaced by a specification that respects SML syntax (we also took out the local declaration):

```
signature s = sig val g : ('a -> 'a) -> ('a -> 'a) list end
functor F (S : sig val c : ⟨..⟩ end) = struct
  open S
  structure X = struct val rec g = fn x => x :: c end
  structure T = X : s
end
```

The binder generated for `g` declared in `X` is as before:

$$\downarrow\mathtt{g}{=}\forall\{\alpha_0\}.\,\{\langle\alpha_0\,\mathtt{list}, sv\rangle\}\diamond\alpha_0{\to}\alpha_0\,\mathtt{list}$$

The binder generated for `g` specified in `s` is now as follows:

$$\downarrow\mathtt{g}{=}\forall\{\beta\}.\,(\beta{\to}\beta){\to}(\beta{\to}\beta)\,\mathtt{list}$$

When checking whether `g`'s specification matches `g`'s declaration (when dealing with the constraints generated for `S : s`), we generate the following binder for `g`'s declaration in `T`:

$$\downarrow\mathtt{g}{=}\forall\{\beta\}.\,\varnothing\diamond(\beta{\to}\beta){\to}(\beta{\to}\beta)\,\mathtt{list}$$

where $\varnothing$ is the $\overline{cap}'_1$ computed in rule (SU1). In this case we also constrain sv to be equal to $(\beta{\to}\beta)\,\mathtt{list}\Cap sv'$. The set $\overline{cap}'_1$ cannot be anything else than empty in this case because the quantified variable set contains only rigid type variables and rigid type variables cannot be constrained further. When checking that the type scheme $\forall\{\alpha_0\}.\,\{\langle\alpha_0\,\mathtt{list}, sv\rangle\}\diamond\alpha_0{\to}\alpha_0\,\mathtt{list}$ is a subtype of $\forall\{\beta\}.\,(\beta{\to}\beta){\to}(\beta{\to}\beta)\,\mathtt{list}$ we first generate instances of the two type schemes as follows: $\alpha_1{\to}\alpha_1\,\mathtt{list}$ and $(\beta{\to}\beta){\to}(\beta{\to}\beta)\,\mathtt{list}$ respectively. We then check that these two types can be made equal which leads to α_1 being constrained to be equal to $\beta{\to}\beta$. When computing $\overline{cap}'_1$, we build up $\alpha_1\,\mathtt{list}$ from $\{\langle\alpha_1\,\mathtt{list}, sv\rangle\}$ (which is a renaming of $\{\langle\alpha_0\,\mathtt{list}, sv\rangle\}$) and obtain the type $(\beta{\to}\beta)\,\mathtt{list}$ which does not contain any flexible type variable and is therefore not added to $\overline{cap}'_1$ (the condition $\neg\mathsf{dja}(\mathsf{vars}(\tau'_0),\overline{\rho})$, where $\tau'_0=(\beta{\to}\beta)\,\texttt{list}$ and $\overline{\rho}=\{\beta\}$, in rule (SU1) is false).

Finally, let us now illustrate how the different mechanisms used by our constraint solver interact to handle functor declarations and functor applications. Let us consider the following incomplete, untypable piece of code:

```
functor F (S : sig val c : ⟨..⟩ end) = struct
  local open S in val rec g = fn x => x :: c end
  val _ = g true
end
structure T = F(struct val c = [()] end)
```

We aim at obtaining the following type error slice:

```
⟨..functor F (S : sig val c : ⟨..⟩ end) =
   ⟨..local open S in val rec g = fn x => ⟨..x :: c..⟩ end
    ..g true..⟩
 ..F(struct val c = [()] end)..⟩
```

At constraint solving when solving the constraints generated for `F`'s parameter, we generate the following binder:

$$\downarrow\!\texttt{c}{=}(\forall\{\alpha_1\}.\,\alpha_1)\mathbin{\dot{\cap}} sv$$

When solving `c`'s accessor, we generate the following unifier:

$$\{sv\mapsto\alpha'_1\mathbin{\dot{\cap}} sv'\}$$

where α'_1 is an instance of `c`'s binding occurrence's type and is constrained to be equal to `c`'s bound occurrence's type. When solving the constraints generated for `g`, because α'_1 is constrained to be equal to $\alpha_2\,\texttt{list}$, we generate the following binder:

$$\downarrow\!\texttt{g}{=}\forall\{\alpha_2\}.\,\{\langle\alpha_2\,\texttt{list},sv'\rangle\}\diamond\alpha_2{\to}\alpha_2\,\texttt{list}$$

When solving the constraints generated for the last declaration in `F`'s body, because `g` is applied to the Boolean `true`, we generate an instance of `g`'s type scheme as follows (where α_2 is renamed to α'_2):

$$\alpha'_2{\to}\alpha'_2\,\texttt{list}$$

and we also generate the following unifier from $\langle\alpha_2\,\texttt{list},sv'\rangle$:

$$\{sv'\mapsto\alpha'_2\,\texttt{list}\mathbin{\dot{\cap}} sv''\}$$

where α'_2 is constrained to be equal to `bool`. Therefore, we generate the following binder for `F`:

$$\downarrow\!\texttt{F}{=}\forall\{\alpha_2\}.\,e_1{\leadsto}e_2\quad\text{where}\quad\begin{cases}e_1=(\downarrow\!\texttt{c}{=}(\forall\{\alpha_1\}.\,\alpha_1)\mathbin{\dot{\cap}}(\alpha_2\,\texttt{list})\mathbin{\dot{\cap}}(\texttt{bool}\,\texttt{list})\mathbin{\dot{\cap}} sv'')\\ e_2=(\downarrow\!\texttt{g}{=}\forall\{\alpha_2\}.\,\{\langle\alpha_2\,\texttt{list},sv'\rangle\}\diamond\alpha_2{\to}\alpha_2\,\texttt{list})\end{cases}$$

The constraint term generated for `F`'s bound occurrence is then as follows:

$$e'_1 \rightsquigarrow e'_2 \quad \text{where} \quad \begin{cases} e'_1 = (\downarrow\mathtt{c}{=}(\forall\{\alpha_1\}.\,\alpha_1) \Cap (\alpha_3\,\mathtt{list}) \Cap (\mathtt{bool}\,\mathtt{list}) \Cap \mathit{sv}'') \\ e'_2 = (\downarrow\mathtt{g}{=}\forall\{\alpha_3\}.\,\{\langle\alpha_3\,\mathtt{list}, \mathit{sv}'\rangle\} \diamond \alpha_3{\to}\alpha_3\,\mathtt{list}) \end{cases}$$

where α_2 has been renamed to α_3. The environment generated for `F`'s argument is as follows:

$$\downarrow\mathtt{c}{=}\forall\varnothing.\,\mathtt{unit}\,\mathtt{list}$$

When matching this environment against e'_1, we get a clash between `unit` and `bool` when checking that $\forall\varnothing.\,\mathtt{unit}\,\mathtt{list}$ is a subtype of $\mathtt{bool}\,\mathtt{list}$.

Because restricted forms of functor binders can now occur in constraint solving contexts (in e in $\langle u,\ e\rangle$), we extend some constraint term forms generated at constraint solving, originally defined in Sec. 11.6.6, as follows:

$$\begin{array}{lll} \mathit{sbind} & \in \mathsf{SolvBind} & ::= \cdots \mid {\downarrow}\mathit{funid}{=}\mathit{sfctsem} \\ \mathit{sfctsem} & \in \mathsf{SolvFuncSem} & ::= \mathit{sfct} \mid \forall\overline{v}.\,\mathit{sfct} \mid \langle \mathit{sfctsem}, \overline{d}\rangle \\ \mathit{sfct} & \in \mathsf{SolvFunc} & ::= \phi \mid \mathit{se}_1 {\rightsquigarrow} \mathit{se}_2 \mid \langle \mathit{sfct}, \overline{d}\rangle \end{array}$$

14.9.5 Constraint filtering (Minimisation and enumeration)

We extend our filtering algorithm as follows:

$$\begin{array}{ll} \mathsf{filt}(\mathit{fct}\cdot e, \overline{l}_1, \overline{l}_2) & = \mathsf{filt}(\mathit{fct}, \overline{l}_1, \overline{l}_2) \cdot \mathsf{filt}(e, \overline{l}_1, \overline{l}_2) \\ \mathsf{filt}(\mathtt{lazy}(e), \overline{l}_1, \overline{l}_2) & = \mathtt{lazy}(\mathsf{filt}(e, \overline{l}_1, \overline{l}_2)) \\ \mathsf{filt}(e_1 {\rightsquigarrow} e_2, \overline{l}_1, \overline{l}_2) & = \mathsf{filt}(e_1, \overline{l}_1, \overline{l}_2) {\rightsquigarrow} \mathsf{filt}(e_2, \overline{l}_1, \overline{l}_2) \\ \mathsf{toDumVar}(\mathit{fctsem}) & = \phi_{\mathtt{dum}} \end{array}$$

14.9.6 Slicing

First, we extend our tree syntax for programs as follows:

$$\begin{array}{l} \mathsf{Class} ::= \cdots \mid \mathtt{fundec} \\ \mathsf{Prod}\ ::= \cdots \mid \mathtt{fundecDec} \mid \mathtt{strexpFct} \end{array}$$

Then, Fig. 14.27 extends the toTree function. We also extend the function getDot as follows:

$$\mathsf{getDot}(\langle \mathtt{fundec}, \mathit{prod}\rangle) = \mathtt{dotD}$$

14.10 Arity clash errors

The slicer presented so far only deals with unary type constructors. Let us now present how to build a constraint mechanism and a TES that handles type constructor with unconstrained arity (unary as well as non-unary arity). Tuples are not formally presented in this document, but they can be handled using the machinery introduced in this section. Note that non-unary type constructors and tuples are both handled by our implementation.

Structure expressions
$\mathsf{toTree}(\mathit{funid}(\mathit{strexp})^{l}) = \langle\langle \mathtt{strexp}, \mathtt{strexpFct}\rangle, l, \langle \mathit{funid}, \mathsf{toTree}(\mathit{strexp})\rangle\rangle$

Functor declarations
$\mathsf{toTree}(\mathtt{functor}\ \mathit{funid}(\mathit{strid} : \mathit{sigexp}) \overset{l}{=} \mathit{strexp})$
$\quad = \langle\langle \mathtt{fundec}, \mathtt{fundecDec}\rangle, l, \langle \mathit{funid}, \mathit{strid}, \mathsf{toTree}(\mathit{sigexp}), \mathsf{toTree}(\mathit{strexp})\rangle\rangle$

Figure 14.27 Extension of our conversion function from *terms* to *trees* to deal with functors

14.10.1 External syntax

The external labelled syntax of type sequences is as follows:

$$\mathit{tyseq} \in \mathsf{TySeq} ::= \mathit{ty}^{l} \mid \epsilon^{l}_{\mathsf{t}} \mid (\mathit{ty}_1, \ldots, \mathit{ty}_n)^{l} \mid \mathtt{dot\text{-}t}(\overrightarrow{\mathit{term}})$$

We redefine atomic sequences of explicit type variables and the forms of type constructs at binding and bound positions as follows:

$$\begin{gathered} \mathit{ltv} \xrightarrow{\mathsf{TyVarSeq}} \mathit{ltv}^{l} \\ \mathit{ty}\ \mathit{tc}^{l} \xrightarrow{\mathsf{Ty}} \mathit{tyseq}\ \mathit{tc}^{l} \\ \lceil \mathit{tv}\ \mathit{tc}\rceil^{l} \xrightarrow{\mathsf{DatName}} \lceil \mathit{tvseq}\ \mathit{tc}\rceil^{l} \end{gathered}$$

An atomic type variable sequence is then labelled by two labels. The inner one is associated with the explicit type variable itself while the outer one is associated with the sequence (of length one).

Let us consider the following piece of code:

```
type ('a, 'b) t = 'a -> 'b
val rec f : int t = fn x => x
```

This piece of code is untypable because the type constructor `t` is defined as a binary type constructor and is used as an unary type constructor. As usual they are many ways of solving the programming error causing this piece of code to be untypable. We only present some of them. One could, e.g., define another type function `type 'a u = ('a, 'a) t` and to replace the type annotation `int t` by the type annotation `int u`. One could also replace the type definition `type ('a, 'b) t = 'a -> 'b` by the type definition `type 'a t = 'a -> 'a`. One could also replace the type annotation `int t` by `(int, int) t`.

We do not deal in this document with syntactic errors stemming from adding type and type variable sequences to the language. For example, `type ('a, 'a) t = 'a` is syntactically incorrect because the explicit type variable `'a` occurs twice in the type variable sequence `('a, 'a)`. Such syntactic errors are dealt with and reported using error slices by Impl-TES (see Sec. 17.1.1).

14.10.2 Constraint syntax

We introduce internal type sequences as follows:

$$
\begin{array}{lll}
\xi & \in \mathsf{ITyVarSeqVar} & \text{(type variable sequence variables)} \\
\omega & \in \mathsf{ITySeqVar} & \text{(type sequence variables)} \\
vsq & \in \mathsf{ITyVarSeq} & ::= \xi \mid \langle\!|\rho_1, \ldots, \rho_n|\!\rangle \mid \langle vsq, \overline{d}\rangle \\
sq & \in \mathsf{ITySeq} & ::= \omega \mid \langle \tau_1, \ldots, \tau_n \rangle \mid \langle sq, \overline{d}\rangle \\
c & \in \mathsf{EqCs} & ::= \cdots \mid sq_1{=}sq_2 \mid vsq_1{=}vsq_2
\end{array}
$$

We redefine internal type functions and internal type constructs as follows (App and TyFun are defined in Sec. 14.3.2 and are used in side conditions):

$$
\begin{array}{l}
\tau\,\mu \xrightarrow{\mathsf{LabTy}} sq\,\mu \\
\Lambda\alpha.\,\tau \xrightarrow{\mathsf{LabName}} \Lambda vsq.\,\tau \\
\tau\ tyf \xrightarrow{\mathsf{App}} sq\ tyf \\
\Lambda\alpha.\,\tau \xrightarrow{\mathsf{TyFun}} \Lambda vsq.\,\tau
\end{array}
$$

Note that arrow types of the form $\tau_1{\to}\tau_2$ can be encoded as follows: $\langle\tau_1,\tau_2\rangle\ \mathtt{ar}$. We do not do so because we believe the first form to be easier to read.

Let $\xi_{\mathtt{dum}}$ be a distinct variable sequence variable in ITyVarSeqVar. We extend Dum as follows: $\mathsf{Dum} = \{\alpha_{\mathtt{dum}}, ev_{\mathtt{dum}}, \delta_{\mathtt{dum}}, \eta_{\mathtt{dum}}, \phi_{\mathtt{dum}}, \xi_{\mathtt{dum}}\}$.

14.10.3 Constraint generation

Fig. 14.28 extends our constraint generation algorithm. This figure introduces three new rules to generate constraints for type sequences: (G52)-(G54). It also introduces the new rule (G51) for type variable sequences. The other rules redefine rules introduced above.

Let us consider the following type declaration: `type ('a, 'b) t = 'a -> 'b` Its labelled version is as follows: $\mathtt{type}\ \lceil(\mathtt{'a}^{l_4}_{\mathtt{1}}, \mathtt{'b}^{l_5}_{\mathtt{1}})^{l_3}\ \mathtt{t}\rceil^{l_2} \overset{l_1}{=} \mathtt{'a}^{l_7} \overset{l_6}{\to} \mathtt{'a}^{l_8}$.

Our constraint generator generates the following information for `('a, 'b) t`:

$$
\langle\alpha, \omega, e_1, e_2\rangle \\
\text{where} \begin{cases}
e_1 = (\downarrow\mathtt{t} \overset{l_2}{=} \Lambda\xi.\,\alpha) \\
e_2 = (\xi \overset{l_3}{=} \langle\!|\beta_1, \beta_2|\!\rangle ; \omega \overset{l_3}{=} \langle\alpha_1, \alpha_2\rangle ; \downarrow\mathtt{'a} \overset{l_4}{=} \beta_1 ; \alpha_1 \overset{l_4}{=} \beta_1 ; \downarrow\mathtt{'b} \overset{l_5}{=} \beta_2 ; \alpha_2 \overset{l_5}{=} \beta_2)
\end{cases}
$$

Our constraint generator generates the following information for `'a -> 'b`:

$$\langle\alpha_3, e_3\rangle \text{ where } e_3 = (\uparrow\mathtt{'a} \overset{l_7}{=} \alpha_4 ; \uparrow\mathtt{'b} \overset{l_8}{=} \alpha_5 ; (\alpha_3 \overset{l_6}{=} \alpha_4{\to}\alpha_5))$$

Finally, using rule (G30), our constraint generator generates the following environment for the entire type declaration:

$$(ev{=}((\alpha \overset{l_1}{=} \alpha_3);\mathtt{loc}\ e_2\ \mathtt{in}\,(e_3;e_1)));ev^{l_1}$$

When replacing e_1, e_2, and e_3, one obtains the following environment:

Labelled type variables $(ltv \rightarrowtriangle \langle\alpha,\ \beta,\ e\rangle)$

(G48) $tv_1^l \rightarrowtriangle \langle\alpha,\ \beta,\ \downarrow tv \overset{l}{=} \beta;\alpha \overset{l}{=} \beta\rangle$

Type variable sequences $(tvseq \rightarrowtriangle \langle\xi,\ \omega,\ e\rangle)$

(G51) $ltv^l \rightarrowtriangle \langle\xi,\ \omega,\ \xi \overset{l}{=} \llparenthesis\beta\rrparenthesis;\omega \overset{l}{=} \langle\alpha\rangle;e\rangle \ \Leftarrow\ ltv \rightarrowtriangle \langle\alpha,\ \beta,\ e\rangle$

(G49) $\epsilon_{\mathsf{v}}^l \rightarrowtriangle \langle\xi,\ \omega,\ \xi \overset{l}{=} \llparenthesis\rrparenthesis;\omega \overset{l}{=} \langle\rangle\rangle$

(G50) $(ltv_1,\ldots,ltv_n)^l \rightarrowtriangle \langle\xi,\ \omega,\ \xi \overset{l}{=} \llparenthesis\beta_1,\ldots,\beta_n\rrparenthesis;\omega \overset{l}{=} \langle\alpha_1,\ldots,\alpha_n\rangle;e_1;\cdots;e_n\rangle$
$\quad\Leftarrow\ ltv_1 \rightarrowtriangle \langle\alpha_1,\ \beta_1,\ e_1\rangle \wedge\cdots\wedge ltv_n \rightarrowtriangle \langle\alpha_n,\ \beta_n,\ e_n\rangle \wedge \mathsf{dja}(e_1,\ldots,e_n,\xi,\omega)$

Type sequences $(tyseq \rightarrowtriangle \langle\omega,\ e\rangle)$

(G52) $ty^l \rightarrowtriangle \langle\omega,\ \omega \overset{l}{=} \langle\alpha\rangle;e\rangle \ \Leftarrow\ ty \rightarrowtriangle \langle\alpha,\ e\rangle \wedge \mathsf{dja}(e,\omega)$

(G53) $\epsilon_{\mathsf{t}}^l \rightarrowtriangle \langle\omega,\ \omega \overset{l}{=} \langle\rangle\rangle$

(G54) $(ty_1,\ldots,ty_n)^l \rightarrowtriangle \langle\omega,\ \omega \overset{l}{=} \langle\alpha_1,\ldots,\alpha_n\rangle;e_1;\cdots;e_n\rangle$
$\quad\Leftarrow\ ty_1 \rightarrowtriangle \langle\alpha_1,\ e_1\rangle \wedge\cdots\wedge ty_n \rightarrowtriangle \langle\alpha_n,\ e_n\rangle \wedge \mathsf{dja}(e_1,\ldots,e_n,\omega)$

Datatype names $(dn \rightarrowtriangle \langle\alpha,\omega,e_1,e_2\rangle)$

(G13) $\lceil tvseq\ tc\rceil^l \rightarrowtriangle \langle\alpha,\omega,\downarrow tc \overset{l}{=} \Lambda\xi.\,\alpha,\ e\rangle \ \Leftarrow\ tvseq \rightarrowtriangle \langle\xi,\ \omega,\ e\rangle \wedge \mathsf{dja}(e,\alpha)$

Types

(G11) $\lceil tyseq\ ltc\rceil^l \rightarrowtriangle \langle\alpha,\ e_1;e_2;(\omega\,\delta \overset{l}{=} \alpha)\rangle \ \Leftarrow\ tyseq \rightarrowtriangle \langle\omega,\ e_1\rangle \wedge ltc \rightarrowtriangle \langle\delta,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\alpha)$

Declarations

(G17) $\texttt{val rec}\ tvseq\ pat \overset{l}{=} exp \rightarrowtriangle (ev{=}\texttt{poly}(\texttt{loc}\ e_0;e\ \texttt{in}\,(\mathsf{toV}(e_1);e_2;(\alpha_1 \overset{l}{=} \alpha_2))));ev^l$
$\quad\Leftarrow\ tvseq \rightarrowtriangle \langle\xi,\ \omega,\ e_0\rangle \wedge pat \rightarrowtriangle \langle\alpha_1,\ e_1\rangle \wedge exp \rightarrowtriangle \langle\alpha_2,\ e_2\rangle$
$\quad\wedge\mathsf{labtyvarsdec}(tvseq,pat,exp) = \uplus_{i=1}^n\{tv_i^{\bar{l}_i}\}$
$\quad\wedge e = ((\downharpoonleft tv_1 \overset{l}{=} \beta_1)^{\vee\bar{l}_1};\cdots;(\downharpoonleft tv_n \overset{l}{=} \beta_n)^{\vee\bar{l}_n})$
$\quad\wedge\mathsf{dja}(e_0,e_1,e_2,ev,\beta_1,\ldots,\beta_n)$

(G45) $\texttt{val}\ tvseq\ pat \overset{l}{=} exp \rightarrowtriangle (ev{=}\texttt{expans}(\texttt{loc}\ e_0;e\ \texttt{in}\,(e_2;e_1;(\alpha_1 \overset{l}{=} \alpha_2)),\mathsf{expansive}(exp)));ev^l$
$\quad\Leftarrow\ tvseq \rightarrowtriangle \langle\xi,\ \omega,\ e_0\rangle \wedge pat \rightarrowtriangle \langle\alpha_1,\ e_1\rangle \wedge exp \rightarrowtriangle \langle\alpha_2,\ e_2\rangle$
$\quad\wedge\mathsf{labtyvarsdec}(tvseq,pat,exp) = \uplus_{i=1}^n\{tv_i^{\bar{l}_i}\}$
$\quad\wedge e = ((\downharpoonleft tv_1 \overset{l}{=} \beta_1)^{\vee\bar{l}_1};\cdots;(\downharpoonleft tv_n \overset{l}{=} \beta_n)^{\vee\bar{l}_n})$
$\quad\wedge\mathsf{dja}(e_0,e_1,e_2,ev,\beta_1,\ldots,\beta_n)$

(G18) $\texttt{datatype}\ dn \overset{l}{=} cb \rightarrowtriangle (ev{=}((\alpha_1 \overset{l}{=} \omega_1\,\gamma);(\alpha_2 \overset{l}{=} \alpha_1);e_1;\texttt{loc}\ e_1'\ \texttt{in}\,\texttt{poly}(e_2)));ev^l$
$\quad\Leftarrow\ dn \rightarrowtriangle \langle\alpha_1,\omega_1,e_1,e_1'\rangle \wedge cb \rightarrowtriangle \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\gamma,ev)$

(G30) $\texttt{type}\ dn \overset{l}{=} ty \rightarrowtriangle (ev{=}((\alpha_1 \overset{l}{=} \alpha_2);\texttt{loc}\ e_1'\ \texttt{in}\,(e_2;e_1)));ev^l$
$\quad\Leftarrow\ dn \rightarrowtriangle \langle\alpha_1,\omega_1,e_1,e_1'\rangle \wedge ty \rightarrowtriangle \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,ev)$

Specifications

(G36) $\texttt{type}\ dn^l \rightarrowtriangle (ev{=}((\alpha \overset{l}{=} \omega\,\delta);e));ev^l \ \Leftarrow\ dn \rightarrowtriangle \langle\alpha,\omega,e,e'\rangle \wedge \mathsf{dja}(e,e',ev)$

(G38) $\texttt{datatype}\ dn \overset{l}{=} cd \rightarrowtriangle (ev{=}((\alpha_1 \overset{l}{=} \omega_1\,\delta);(\alpha_2 \overset{l}{=} \alpha_1);e_1;\texttt{loc}\ e_1'\ \texttt{in}\,\texttt{poly}(e_2)));ev^l$
$\quad\Leftarrow\ dn \rightarrowtriangle \langle\alpha_1,\omega_1,e_1,e_1'\rangle \wedge cd \rightarrowtriangle \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\gamma,ev)$

Figure 14.28 Constraint generation rules to handle type constructor with unrestricted arity

$$(ev{=}((\alpha \overset{l_1}{=} \alpha_3);\begin{array}{l}\texttt{loc}\,(\xi \overset{l_3}{=} \llparenthesis\beta_1,\beta_2\rrparenthesis;\omega \overset{l_3}{=} \langle\alpha_1,\alpha_2\rangle;\downarrow\texttt{'a} \overset{l_4}{=} \beta_1;\alpha_1 \overset{l_4}{=} \beta_1;\downarrow\texttt{'b} \overset{l_5}{=} \beta_2;\alpha_2 \overset{l_5}{=} \beta_2)\\ \texttt{in}\,((\uparrow\texttt{'a} \overset{l_7}{=} \alpha_4;\uparrow\texttt{'b} \overset{l_8}{=} \alpha_5;(\alpha_3 \overset{l_6}{=} \alpha_4{\rightarrow}\alpha_5));(\downarrow\texttt{t} \overset{l_2}{=} \Lambda\xi.\,\alpha))\end{array}));ev^{l_1}$$

Note that some constraints in this environment are not useful: $\omega \overset{l_3}{=} \langle\alpha_1,\alpha_2\rangle$, $\alpha_1 \overset{l_4}{=} \beta_1$, and $\alpha_2 \overset{l_5}{=} \beta_2$. As a matter of fact ω does not occur in any other constraint. These constraints are only useful when generating constraints for datatype declarations.

In order to illustrate this point, let us consider the following datatype declaration: `datatype ('a, 'b) t = T of 'a -> 'b`. Its labelled version is as follows: $\texttt{datatype}\ \lceil(\texttt{'a}_{\texttt{1}}^{l_4}, \texttt{'b}_{\texttt{1}}^{l_5})^{l_3}\ \texttt{t}\rceil^{l_2} \overset{l_1}{=} \texttt{T}\ \texttt{of}^{\,l_9}\ \texttt{'a}^{l_7} \xrightarrow{l_6} \texttt{'a}^{l_8}$. The same information is generated for `('a, 'b) t` and `'a -> 'b`. Our constraint generator generates the following information for `T of 'a -> 'b`:

$$\langle\alpha_6, e_4\rangle \text{ where } e_4 = e_3;\alpha_7 \overset{l_9}{=} \alpha_3{\to}\alpha_6;{\downarrow}\texttt{T} \overset{l_9}{=} \langle\alpha_7, \texttt{c}\rangle$$

Finally, using rule (G18), our constraint generator generates the following environment for the entire datatype declaration:

$$(ev{=}((\alpha \overset{l_1}{=} \omega\,\gamma);(\alpha_6 \overset{l_1}{=} \alpha);e_1;\texttt{loc}\ e_2\ \texttt{in}\,\texttt{poly}(e_4)));ev^{l_1}$$

When replacing e_1, e_2, e_3, and e_4, one obtains the following environment:

$$(ev{=}\left(\begin{array}{l}(\alpha \overset{l_1}{=} \omega\,\gamma);(\alpha_6 \overset{l_1}{=} \alpha);({\downarrow}\texttt{t} \overset{l_2}{=} \Lambda\xi.\,\alpha);\\ \texttt{loc}\,(\xi \overset{l_3}{=} \llparenthesis\beta_1, \beta_2\rrparenthesis;\omega \overset{l_3}{=} \langle\alpha_1, \alpha_2\rangle;{\downarrow}\texttt{'a} \overset{l_4}{=} \beta_1;\alpha_1 \overset{l_4}{=} \beta_1;{\downarrow}\texttt{'b} \overset{l_5}{=} \beta_2;\alpha_2 \overset{l_5}{=} \beta_2)\\ \texttt{in}\,\texttt{poly}(({\uparrow}\texttt{'a} \overset{l_7}{=} \alpha_4;{\uparrow}\texttt{'b} \overset{l_8}{=} \alpha_5;(\alpha_3 \overset{l_6}{=} \alpha_4{\to}\alpha_5));\alpha_7 \overset{l_9}{=} \alpha_3{\to}\alpha_6;{\downarrow}\texttt{T} \overset{l_9}{=} \langle\alpha_7, \texttt{c}\rangle)\end{array}\right);ev^{l_1}$$

One can see that the three constraints $\omega \overset{l_3}{=} \langle\alpha_1, \alpha_2\rangle$, $\alpha_1 \overset{l_4}{=} \beta_1$, and $\alpha_2 \overset{l_5}{=} \beta_2$ are used when dealing with datatype declarations. The variable ω occurs in the constraint $\alpha \overset{l_1}{=} \omega\,\gamma$. They are necessary to have `t`'s type depending on the labels of the explicit type variables occurring in the type variable sequence.

Note that `t`'s arity is constrained via the constraint $\xi \overset{l_3}{=} \llparenthesis\beta_1, \beta_2\rrparenthesis$.

Because of the tuples generated by the constraint generation rules (G48)-(G54), we extend the set InitGen originally defined in Sec. 11.5.1 and extended in Sec. 14.3.3 as follows:

$$cg \in \mathsf{InitGen} ::= \cdots \mid \langle\alpha, \beta, e\rangle \mid \langle\xi, \omega, e\rangle \mid \langle\omega, e\rangle$$

Also, because rule (G13) associates new forms with *dn*s, we redefine some of the forms that our initial constraint generation algorithm associates with *term*s as follows:

$$\langle\delta, \alpha, e_1, e_2\rangle \xrightarrow{\mathsf{InitGen}} \langle\alpha, \omega, e_1, e_2\rangle$$

Because our initial generation algorithm generates new forms of equality constraints, we update LabCs as follows:

$$\begin{array}{lll} shvsq \in \mathsf{ShallowITyVarSeq} & ::= \xi \mid \llparenthesis\beta_1, \dots, \beta_n\rrparenthesis \\ shseq \in \mathsf{ShallowITySeq} & ::= \omega \mid \langle\alpha_1, \dots, \alpha_n\rangle \\ lc \quad\ \in \mathsf{LabCs} & ::= \cdots \mid \xi \overset{l}{=} shvsq \mid \omega \overset{l}{=} shseq \end{array}$$

We also the initially generated type constructor binders, some shallow types, and the shallow type equality constraints as follows:

$$\begin{array}{l} {\downarrow}tc \overset{l}{=} \delta \xrightarrow{\mathsf{LabBind}} {\downarrow}tc \overset{l}{=} \Lambda\xi.\,\alpha \\ \alpha\,\delta \xrightarrow{\mathsf{ShallowITy}} \omega\,\delta \\ \alpha\,\gamma \xrightarrow{\mathsf{ShallowITy}} \omega\,\gamma \\ sit_1 \xrightarrow{\mathsf{LabCs}} sit_2 \end{array}$$

14.10.4 Constraint solving

First, let us extend error kinds as follows:

$$ek \in \mathsf{ErrKind} ::= \cdots \mid \mathtt{arity}(n_1, n_2)$$

We extend our unifiers as follows (note that this extension also extends Sub):

$$\begin{aligned} u \in \mathsf{Unifier} = \{\textstyle\bigcup_{i=1}^{8} f_i \mid\ & f_1 \in \mathsf{ITyVar} \to \mathsf{ITy} \\ & \wedge f_2 \in \mathsf{TyConVar} \to \mathsf{ITyCon} \\ & \wedge f_3 \in \mathsf{EnvVar} \to \mathsf{Env} \\ & \wedge f_4 \in \mathsf{SigSemVar} \to \mathsf{SigSem} \\ & \wedge f_5 \in \mathsf{FuncVar} \to \mathsf{Func} \\ & \wedge f_6 \in \mathsf{SchemeVar} \to \mathsf{Scheme} \\ & \wedge f_7 \in \mathsf{ITyVarSeqVar} \to \mathsf{ITyVarSeq} \\ & \wedge f_8 \in \mathsf{ITySeqVar} \to \mathsf{ITySeq}\} \end{aligned}$$

We extend the building function to internal type sequences as follows:

$$\begin{aligned} \mathsf{build}(u, \langle \tau_1, \ldots, \tau_n \rangle) &= \langle \mathsf{build}(u, \tau_1), \ldots, \mathsf{build}(u, \tau_n) \rangle \\ \mathsf{build}(u, \Lambda vsq.\, \tau) &= \Lambda \mathsf{build}(u, vsq).\, \mathsf{build}(u, \tau) \end{aligned}$$

Let the function $\mathsf{shallow}$ be defined as follows:

$$\begin{aligned} \mathsf{shallow}(\omega, \mathit{\Delta}) &= \begin{cases} \mathsf{shallow}(sq, \mathit{\Delta}), & \text{if } \mathit{\Delta}(\omega) = sq \\ \xi_{\mathtt{dum}}, & \text{otherwise} \end{cases} \\ \mathsf{shallow}(\langle \tau_1, \ldots, \tau_n \rangle, \mathit{\Delta}) &= \llangle \alpha_{\mathtt{dum}}, \ldots, \alpha_{\mathtt{dum}} \rrangle \\ \mathsf{shallow}(sq^{\overline{d}}, \mathit{\Delta}) &= \mathsf{shallow}(sq, \mathit{\Delta})^{\overline{d}} \end{aligned}$$

Fig. 14.29 extends our constraint solver. Rules (S23)-(S27), (SU8)-(SU10) are new and the other ones redefine rules introduced above.

In rule (S23), a constraint of the form $sq\,(\Lambda\xi.\,\tau_1){=}\tau$ (we omit dependencies for readability issues) leads to the constraining of ξ using a shallow version of sq which is obtained using the function $\mathsf{shallow}$. Note that at the time a type function is applied at constraint solving in our system, it is fully built up. Therefore, if the type function is of the form $\Lambda\xi.\,\tau_1$, it means that the information relative to the arguments of the type constructor for which the type function has been generated, has been sliced out. We then constrain it further using a shallow version of the type sequence to which the type function is applied to in order to catch arity errors between two bound occurrences of type constructors. We only extract a shallow version of the type sequence, which is a type variable sequence that has the same length as the type sequence. For example, `datatype 'a t = T of t -> 'a t` is untypable because, among other things, the two bound occurrences of `t` have different arities. If the constraints generated for `'a`'s first occurrence is sliced out, at constraint solving, the two bound occurrences of `t` can constrain the arity of the binding occurrence of `t` via rule (S23) which leads to an arity clash between the first bound occurrence of `t` which is nullary and the second bound occurrence of `t` which is unary.

equality simplification

(S9) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, sq\,\mu{=}\tau) \quad \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d} \cup \overline{d}_1 \cup \overline{d}_2, e),$
$\quad$ if $\mathsf{collapse}(\mu^{\varnothing}) = (\Lambda\llparenthesis\beta_1, \ldots, \beta_n\rrparenthesis^{\overline{d}_1}.\,\tau_1)^{\overline{d}_2} \wedge ren = \cup_{i=1}^{n}\{\beta_i \mapsto \alpha_i\}$
$\quad \wedge\ \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}), \mathsf{ran}(ren)) \wedge e = (sq{=}\langle\alpha_1, \ldots, \alpha_n\rangle;\mathsf{build}(\mathit{\Delta}, \tau_1)[ren]{=}\tau)$

(S23) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, sq\,\mu{=}\tau) \rightarrow \mathtt{slv}(\langle u,\ e\rangle, \overline{d} \cup \overline{d}', \xi{=}\xi'),$
$\quad$ if $\mathsf{collapse}(\mu^{\varnothing}) = (\Lambda\xi.\,\tau_1)^{\overline{d}'} \wedge \xi' = \mathsf{shallow}(sq, u)$

(S10) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, sq\,\mu{=}\tau) \rightarrow \mathtt{succ}(\langle u,\ e\rangle),$
$\quad$ if $\mathsf{strip}(\mu) = \delta \wedge \delta \notin \mathsf{dom}(u)$

(S11) $\mathtt{slv}(\langle u,\ e\rangle, \overline{d}, sq\,\mu{=}\tau) \rightarrow \mathtt{slv}(\langle u,\ e\rangle, \overline{d} \cup \overline{d}', sq\,\mu'{=}\tau),$
$\quad$ if $\mathsf{strip}(\mu) = \delta \wedge u(\delta) = \mu' \wedge \overline{d}' = \mathsf{deps}(\mu)$

(S12) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, sq\,\mu{=}sq'\,\mu') \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}_1 \cup \overline{d}_2, \gamma{=}\gamma';sq{=}sq'),$
$\quad$ if $\mathsf{collapse}(\mu^{\overline{d}}) = \gamma^{\overline{d}_1} \wedge \mathsf{collapse}(\mu^{\varnothing}) = \gamma'^{\overline{d}_2}$

(S13) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \tau_1{=}\tau_2) \quad \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \mu{=}\mathtt{ar}),$
$\quad$ if $\{\tau_1, \tau_2\} = \{sq\,\mu, \tau_0{\rightarrow}\tau_0'\} \wedge \mathsf{strip}(\mu) \in \mathsf{TyConName}$

(S14) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \tau_1{=}\tau_2) \quad \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \mu{=}\mathtt{tv}),$
$\quad$ if $\{\tau_1, \tau_2\} = \{sq\,\mu, \beta\} \wedge \mathsf{strip}(\mu) \in \mathsf{TyConName}$

(S24) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, sq{=}sq') \quad \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \tau_n{=}\tau_n';\cdots;\tau_1{=}\tau_1'),$
$\quad$ if $sq = \langle\tau_1, \ldots, \tau_n\rangle \wedge sq' = \langle\tau_1', \ldots, \tau_n'\rangle$

(S25) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, sq{=}sq') \quad \rightarrow \mathtt{err}(\langle\mathtt{arity}(n, m), \overline{d}\rangle),$
$\quad$ if $sq = \langle\tau_1, \ldots, \tau_n\rangle \wedge sq' = \langle\tau_1', \ldots, \tau_m'\rangle \wedge n \neq m$

(S26) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, vsq{=}vsq') \quad \rightarrow \mathtt{slv}(\mathit{\Delta}, \overline{d}, \rho_n{=}\rho_n';\cdots;\rho_1{=}\rho_1'),$
$\quad$ if $vsq = \llparenthesis\rho_1, \ldots, \rho_n\rrparenthesis \wedge vsq' = \llparenthesis\rho_1', \ldots, \rho_n'\rrparenthesis$

(S27) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, vsq{=}vsq') \quad \rightarrow \mathtt{err}(\langle\mathtt{arity}(n, m), \overline{d}\rangle),$
$\quad$ if $vsq = \llparenthesis\rho_1, \ldots, \rho_n\rrparenthesis \wedge vsq' = \llparenthesis\rho_1', \ldots, \rho_m'\rrparenthesis \wedge n \neq m$

subtyping constraints

(SU3) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \kappa_1 \preceq_{tc} \kappa_2) \rightarrow \mathtt{succ}(\langle u',\ e';{\downarrow}tc{=}\mathsf{scheme}(u', \overline{\alpha}_1[ren_1] \cup \overline{\alpha}_2[ren_2], \delta)\rangle),$
$\quad$ if $\kappa_1 = \forall\overline{\alpha}_1.\,\Lambda\llparenthesis\beta_1, \ldots, \beta_n\rrparenthesis^{\overline{d}_1}.\,\tau_1 \wedge \kappa_2 = \forall\overline{\alpha}_2.\,\Lambda\llparenthesis\beta_1', \ldots, \beta_n'\rrparenthesis^{\overline{d}_2}.\,(\langle\tau_1, \ldots, \tau_n\rangle^{\overline{d}_4}\,\delta)^{\overline{d}_3}$
$\quad \wedge\ \mathsf{dom}(ren_1) = \overline{\alpha}_1 \wedge \mathsf{dom}(ren_2) = \overline{\alpha}_2 \wedge \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\quad \wedge\ sub = \cup_{i=1}^{n}\{\beta_i \mapsto \tau_i[ren_2]\} \wedge \overline{d}' = \overline{d} \cup \overline{d}_1 \cup \overline{d}_2 \cup \overline{d}_3 \cup \overline{d}_4$
$\quad \wedge\ \mathtt{slv}(\mathit{\Delta}, \overline{d}', \delta{=}\Lambda\llparenthesis\beta_1', \ldots, \beta_n'\rrparenthesis.\,\tau_1[ren_1][sub]) \rightarrow^{*} \mathtt{succ}(\langle u',\ e'\rangle)$

(SU8) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \kappa_1 \preceq_{tc} \kappa_2) \rightarrow \mathtt{err}(er),$
$\quad$ if $\kappa_1 = \forall\overline{\alpha}_1.\,\Lambda\llparenthesis\beta_1, \ldots, \beta_n\rrparenthesis^{\overline{d}_1}.\,\tau_1 \wedge \kappa_2 = \forall\overline{\alpha}_2.\,\Lambda\llparenthesis\beta_1', \ldots, \beta_n'\rrparenthesis^{\overline{d}_2}.\,(\langle\tau_1, \ldots, \tau_n\rangle^{\overline{d}_4}\,\delta)^{\overline{d}_3}$
$\quad \wedge\ \mathsf{dom}(ren_1) = \overline{\alpha}_1 \wedge \mathsf{dom}(ren_2) = \overline{\alpha}_2 \wedge \mathsf{dj}(\mathsf{vars}(\mathit{\Delta}), \mathsf{ran}(ren_1), \mathsf{ran}(ren_2))$
$\quad \wedge\ sub = \cup_{i=1}^{n}\{\beta_i \mapsto \tau_i[ren_2]\} \wedge \overline{d}' = \overline{d} \cup \overline{d}_1 \cup \overline{d}_2 \cup \overline{d}_3 \cup \overline{d}_4$
$\quad \wedge\ \mathtt{slv}(\mathit{\Delta}, \overline{d}', \delta{=}\Lambda\llparenthesis\beta_1', \ldots, \beta_n'\rrparenthesis.\,\tau_1[ren_1][sub]) \rightarrow^{*} \mathtt{err}(er)$

(SU9) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \kappa_1 \preceq_{tc} \kappa_2) \rightarrow \mathtt{err}(\langle\mathtt{arity}(n, m), \overline{d}\rangle),$
$\quad$ if $\kappa_1 = \forall\overline{\alpha}_1.\,\Lambda\llparenthesis\beta_1, \ldots, \beta_n\rrparenthesis^{\overline{d}_1}.\,\tau_1 \wedge \kappa_2 = \forall\overline{\alpha}_2.\,\Lambda\llparenthesis\beta_1', \ldots, \beta_m'\rrparenthesis^{\overline{d}_2}.\,\tau_2 \wedge n \neq m$

(SU10) $\mathtt{slv}(\mathit{\Delta}, \overline{d}, \kappa_1 \preceq_{tc} \kappa_2) \rightarrow \mathtt{succ}(\mathit{\Delta};{\downarrow}tc{=}\delta_{\mathtt{dum}}),$
$\quad$ if κ_1 not of the form $\forall\overline{\alpha}_1.\,\Lambda\llparenthesis\beta_1, \ldots, \beta_n\rrparenthesis^{\overline{d}_1}.\,\tau_1$
$\quad \vee\ \kappa_2$ not of the form $\forall\overline{\alpha}_2.\,\Lambda\llparenthesis\beta_1', \ldots, \beta_m'\rrparenthesis^{\overline{d}_2}.\,(\langle\tau_1, \ldots, \tau_n\rangle^{\overline{d}_4}\,\delta)^{\overline{d}_3}$

Figure 14.29 Constraint solving rules to also handle non-unary type constructor

The complexity of the subtyping constraint rules presented in Fig. 14.29 comes partially from the fact that with non-unary type constructors, we also have to check that if a type constructor is specified in a signature constraining a structure then it has to be defined in the structure with the same arity. For example, `struct type 'a t = 'a end : sig type t end` is not typable because `t` is specified as being a unary type constructor in the signature and declared as being a nullary type constructor in the structure.

$$
\begin{array}{lll}
(\mathsf{G55})\ \mathtt{dot\text{-}v}(\overrightarrow{\mathit{term}}) \mathrel{-\!\triangleright} \langle \xi, \omega, [e_1;\cdots;e_n]\rangle & \Leftarrow & \mathit{term}_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\dots,e_n,\xi,\omega) \\
(\mathsf{G56})\ \mathtt{dot\text{-}l}(\overrightarrow{\mathit{term}}) \mathrel{-\!\triangleright} \langle \alpha, \beta, [e_1;\cdots;e_n]\rangle & \Leftarrow & \mathit{term}_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\dots,e_n,\alpha,\beta) \\
(\mathsf{G57})\ \mathtt{dot\text{-}t}(\overrightarrow{\mathit{term}}) \mathrel{-\!\triangleright} \langle \omega, [e_1;\cdots;e_n]\rangle & \Leftarrow & \mathit{term}_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\dots,e_n,\omega) \\
(\mathsf{G31})\ \mathtt{dot\text{-}n}(\overrightarrow{\mathit{term}}) \mathrel{-\!\triangleright} \langle \alpha, \omega, \odot, [e_1;\cdots;e_n]\rangle & \Leftarrow & \mathit{term}_1 \mathrel{-\!\triangleright} e_1 \wedge \cdots \wedge \mathit{term}_n \mathrel{-\!\triangleright} e_n \wedge \mathsf{dja}(e_1,\dots,e_n,\alpha,\omega)
\end{array}
$$

Figure 14.30 Constraint generation rules to handle incomplete sequences

14.10.5 Slicing

Because we have changed our constraint generation rules for type variable sequences and labelled type variables, we need to replace some dot terms as follows:

$$
\begin{array}{l}
\mathtt{dot\text{-}d}(\overrightarrow{\mathit{term}}) \xrightarrow{\mathsf{TyVarSeq}} \mathtt{dot\text{-}v}(\overrightarrow{\mathit{term}}) \\
\mathtt{dot\text{-}d}(\overrightarrow{\mathit{term}}) \xrightarrow{\mathsf{LabTyVar}} \mathtt{dot\text{-}l}(\overrightarrow{\mathit{term}})
\end{array}
$$

Fig. 14.30 defines new constraint generation rules for our new dot terms as follows and redefines the one for dot *dn*s.

Because the environments generated for type variable sequences are always used in local environment (of the form $\mathtt{loc}\, e_1\, \mathtt{in}\, e_2$) we do not need to generate any $\odot$ environment in rules (G55) and (G56).

We extend our tree syntax for programs as follows:

$$
\begin{array}{lll}
\mathsf{Class} & ::= & \cdots \mid \mathtt{tyseq} \\
\mathsf{Prod} & ::= & \cdots \\
 & & \mid \mathtt{tyvarseqSgl} \mid \mathtt{tyvarseqEm} \mid \mathtt{tyvarseqSeq} \\
 & & \mid \mathtt{tyseqSgl} \mid \mathtt{tyseqEm} \mid \mathtt{tyseqSeq} \\
\mathsf{Dot} & ::= & \cdots \mid \mathtt{dotV} \mid \mathtt{dotT}
\end{array}
$$

We also extend the function getDot that associates dot markers with node kinds as follows:

$$\mathsf{getDot}(\langle \mathtt{tyseq}, \mathit{prod}\rangle) = \mathtt{dotT}$$

We also redefine this function on `tyvarseq` nodes as follows:

$$\mathsf{getDot}(\langle \mathtt{tyvarseq}, \mathit{prod}\rangle) = \mathtt{dotV}$$

Finally, Fig. 14.31 extends the function toTree that transforms *term*s into *tree*s.

Non-unary type constructors raise interesting slicing and highlighting issues. Let us consider the following piece of code: `type 'a t = int val x : t`. This piece of code is untypable because `t` is defined as being unary and is used as a nullary type constructor. The type error slice that report this error would then be as follows: ⟨..`type` ⟨..⟩ `t` = ⟨..⟩..`t`..⟩ because, among other things, the explicit type variables `'a` is not part of the error. The obvious problem with this slice is that ⟨..⟩ in ⟨..⟩ `t` can be a sliced out type variable sequence of length zero which means that this slice has to be typable. First, note that this issue does not arise in our labelled syntax because ⟨..⟩ in ⟨..⟩ `t` is in fact the type variable sequence $\mathtt{dot\text{-}l}(\varnothing)^l$ which is

Type variable sequences	$\mathsf{toTree}(ltv^{l})$	$= \langle\langle \mathtt{tyvarseq}, \mathtt{tyvarseqSgl}\rangle, l, \langle \mathsf{toTree}(ltv)\rangle\rangle$
Type sequences	$\mathsf{toTree}(ty^{l})$	$= \langle\langle \mathtt{tyseq}, \mathtt{tyseqSgl}\rangle, l, \langle ty\rangle\rangle$
	$\mathsf{toTree}(\epsilon_{\mathtt{t}}^{l})$	$= \langle\langle \mathtt{tyseq}, \mathtt{tyseqEm}\rangle, l, \langle\rangle\rangle$
	$\mathsf{toTree}((ty_1, \dots, ty_n)^{l})$	$= \langle\langle \mathtt{tyseq}, \mathtt{tyseqSeq}\rangle, l, \mathsf{toTree}(\langle ty_1, \dots, ty_n\rangle)\rangle$
Types	$\mathsf{toTree}(tyseq\ tc^{l})$	$= \langle\langle \mathtt{ty}, \mathtt{tyCon}\rangle, l, \langle \mathsf{toTree}(tyseq), tc\rangle\rangle$
Datatype names	$\mathsf{toTree}(\lceil tvseq\ tc\rceil^{l})$	$= \langle\langle \mathtt{datname}, \mathtt{datnameCon}\rangle, l, \langle tvseq, tc\rangle\rangle$
Dot terms	$\mathsf{toTree}(\mathtt{dot\text{-}v}(\overrightarrow{term}))$	$= \langle \mathtt{dotV}, \mathsf{toTree}(\overrightarrow{term})\rangle$
	$\mathsf{toTree}(\mathtt{dot\text{-}t}(\overrightarrow{term}))$	$= \langle \mathtt{dotT}, \mathsf{toTree}(\overrightarrow{term})\rangle$

Figure 14.31 Extension of our conversion function from *terms* to *trees* to handle type and type variable sequences

different from the sliced out empty type variable sequence dot-v(∅). The problem comes from the fact that there is no explicit syntax representing a unary sequence in SML. To solve this issue, we add special parentheses in our slice language, in addition to ⟨ and ⟩. We print $\mathtt{dot\text{-}l}(\varnothing)^{l}$ as follows: ⟦⟨..⟩⟧ which is then different from ⟨..⟩ which is an entirely sliced out sequence. Finally, the slice reporting the error described above is then as follows: ⟨..type ⟦⟨..⟩⟧ t = ⟨..⟩..t..⟩. This error is highlighted as follows: type 'a t = int val x : t. The box around 'a indicates that t's first occurrence is unary and that 'a itself is not part of the reported error. The highlighted empty space preceding t's second occurrence indicates that this occurrence of t is nullary. The extra parentheses ⟦ and ⟧ are also used to display type sequences of the form $\mathtt{dot\text{-}e}(\langle\rangle)^{l}$.

Let us consider a similar example which only differs from the previous example by the removal of the white space between the colon and t: type 'a t = int val x = 1 :t. As above, this piece of code is untypable because t is defined as being unary and is used as a nullary type constructor. The issue is that now when highlighting this type error in the code, we cannot anymore highlight the white space before t's second occurrence because there is no such space. We therefore have to come up with a convention to highlight such errors. A possibility is to put a box around the type constructor itself when the fact that it is a nullary type constructor is part of the reported error. We would then obtain the following highlighting: type 'a t = int val x :t.

Finally, let us present another issue raised by non-unary type constructors using the following untypable datatype declaration: datatype 'a t = T of ('a, 'a) t t. Because a datatype declaration is recursive, t's two last occurrences are bound to t's first occurrence. Now, t's second occurrence is a binary type constructor while t's third occurrence is unary. Therefore we report the following error: ⟨..datatype ⟨..⟩ t = ⟨..(⟨..⟩, ⟨..⟩) t t..⟩..⟩. The highlighting of this error in the original code is as follows: datatype 'a t = T of ('a, 'a) t t. Note that in this case, a portion of the code is highlighted inside the box. Whether or not t's first occurrence has to be part of the report is disputable. For example, the system presented so far does not report any error for type 'a u = T of ('a, 'a) t t where t is free even though there is no way of completing this piece of code with a declaration of `t` such that the piece of code would be typable. Given this piece of code, we should then report an arity type error. We do not present in this document how to report such errors and how to report `⟨..(⟨..⟩, ⟨..⟩) t t..⟩` instead of the slice presented above but our implementation report such errors. Informally, reporting such errors in our implementation involves the generation at constraint solving of special binders of free, or bound by dummy binders, type constructors.

Chapter 15

Extensions for better error handling

15.1 Merged minimal type error slices

We have found cases needing the display of many minimal errors at once. The combination of at least two minimal type error slices is called a merged type error slice. We present in this section two cases for which our TES report merged type error slice: for record field name clashes and for unmatched specifications. Note that our TES does not merge minimal type error slices but directly generates merged type error slices.

15.1.1 Records

One important case is in record field name clashes where, e.g., the highlighting `val {foo,bar} = {fool=0,bar=1}` reports two minimal errors at once: that `fool` is not in the set {`foo`, `bar`} and `foo` is not in the set {`fool`, `bar`}. This merged error is preferable over the minimal errors because of the explosion in the number of minimal slices. Green highlights the fields that are common to different minimal slices. For merged slices minimality is understood as follows: retain a single blue/purple field name in one of the two clashing records and all field names in the other.

15.1.2 Signatures

With the constraint solver as defined above, our TES would report two minimal *unmatched* type error slices for the following piece of code:

```
structure S = struct val (fool, barr, x, y) = (1, 2, 3, 4) end
signature s = sig val foo : int val bar : int val x : int end
structure T = S :> s
```

One of the type errors is that the specification `foo` in `s` is not matched in the structure `S` (that declares `fool`, `barr`, `x` and `y`), but `s` constrains `S` in `T`. The other error is similar but concerns the specification `bar`.

This is another typical example where finding and reporting merged minimal error slices would be useful. For the example above, instead of the two reports described above, we would prefer a highlighting that would looks like:

```
structure S = struct val (fool, barr, x, y) = (1, 2, 3) end
signature s = sig val foo : int val bar : int val x : int end
structure T = S :> s
```

This highlighting shows that `foo` and `bar` are not matched in the structure `S`, but also suppose that `x` might not be the matching for `foo` or `bar` as `x` is specified in the signature `s`. Note that `x` is still reported because we cannot know if `x` in the structure `S` is definitely not the matching of, e.g., `foo` in the signature `s`.

We could obtain this slice by altering the part of our constraint solver defined in Fig. 14.18, Fig. 14.19, and Fig. 14.21.

First, we want unmatched error kinds to be as follows instead (we replace the previous form by this new one):

$$ek \in \mathsf{ErrKind} ::= \cdots \mid \mathtt{unmatched}(\overline{id}_1, \overline{id}_2, \overline{id}_3)$$

For the highlighting presented above, the generated error kind would then be $\mathtt{unmatched}(\overline{id}_1, \overline{id}_2, \overline{id}_3)$, where $\overline{id}_1$ is the set of identifiers highlighted in purple (the identifiers specified in `s` that are not declared in `S`), $\overline{id}_2$ is the set of identifiers highlighted in blue (the identifiers declared in `S` that are not specified in `s`) and $\overline{id}_3$ is the set of identifiers highlighted in green (the identifiers both specified in `s` and declared in `S`).

Then, when checking if a signature matches a structure, in order to gather (1) the identifiers that are specified in the signature but not declared in the structure, (2) the identifiers that are declared in the structure but not specified in the signature, and (3) the identifiers that are both specified in the signature and declared in the structure, we extend our "match" states as follows:

$$\begin{array}{lll}\Theta & \in \mathsf{Unmatched} & ::= \langle \overline{id}_1, \overline{id}_2 \rangle \\ \mathit{state} & \in \mathsf{State} & ::= \cdots \mid \mathtt{match}(\Delta, \overline{d}, \Theta, e_1, e_2) \mid \mathtt{succ}(\Delta, \Theta)\end{array}$$

In order to update Θs, we define the two functions addI and addO (where "I" stands for "in" and "O" stands for "out") as follows: $\mathsf{addI}(\langle \overline{id}_1, \overline{id}_2 \rangle, id) = \langle \overline{id}_1, \overline{id}_2 \cup \{id\} \rangle$ and $\mathsf{addO}(\langle \overline{id}_1, \overline{id}_2 \rangle, id) = \langle \overline{id}_1 \cup \{id\}, \overline{id}_2 \rangle$. The function addI is used when an identifier has been checked to be both specified in a signature *sigexp* and declared in a structure which is constrained by the signature *sigexp*. The function addO is used when an identifier has been checked to be declared in a structure *strexp* but not in a signature that constrain the structure *strexp*.

Finally, Fig. 15.1 updates the rules defined in Fig. 14.18, Fig. 14.19, and Fig. 14.21 to handle the reporting of merged unmatched errors. Rule (SC1) is updated and we add two new rules for signature constraints: (SC2) and (SC3). Rules (SC2) and (SM17) are new and replace rule (SM13).

The difference between this new algorithm and the one presented in Fig. 14.18, Fig. 14.19, and Fig. 14.21, is that when checking that a signature matches a structure, this new algorithm gathers the identifiers that are both specified in the signature and declared in the structure (rules (SM4), (SM5), and (SM6)) and also gathers the identifier that are not matched in the structure (rule (SM10)). If there exists such an identifier, it means that there is an unmatched error. We then wait to check the matching of the entire signature against the structure to finally report all such unmatched identifiers in a single error report (rules (SC2) and (SM17)).

Note that such type error reports for unmatched errors are still imperfect. For example, the highlighting above does not show that {`fool`, `barr`, `x`, `y`} is precisely the set of identifiers declared in the structure `S`. Similarly, the highlighting does not show that {`foo`, `bar`, `x`} is precisely the set of identifiers specified in the signature `s`. Note that this is made precise in our type error slices because in `S`, e.g., no declaration is entirely sliced out and replaced by ⟨..⟩. We could then consider the following convention when highlighting a type error: if all the identifiers declared in a structure or specified in a signature are involved in the reported error and this information is necessary for the error to occur then we highlight the blank spaces (if any) preceding the corresponding `val`, `type`, `datatype` and `structure` keywords.

We would then obtain the following highlighting which is a bit more informative than the one presented above:

```
structure S = struct val (fool, barr, x, y) = (1, 2, 3) end
signature s = sig val foo : int val bar : int val x : int end
structure T = S :> s
```

It is important to find conventions as intuitive as possible because the issue with such conventions is that they have to be known by the user for highlightings to be understandable.

15.2 End points

Some error reports involve what we call end points. It the case for clash errors such as type constructor clashes. The two end points of a type constructor clash error are the two program locations responsible for the generation of two distinct type constructors that are constrained to be equal at constraint solving. More generally, the end points of a clash error are the program locations responsible for the generation of two distinct constraint terms that are constrained to be equal

Some kinds of errors are not handled by the system presented in this section, although our implementation handles them. For more information please refer to the introductory paragraph of Sec. 14.7.

signature constraints

$$
\begin{array}{lll}
(\mathsf{SC1}) & \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1{:}e_2) \rightarrow \mathtt{succ}(\varDelta'), & \text{if } \mathsf{build}(u, e_1) = e'_1 \wedge \mathsf{build}(u, e_2) = e'_2 \\
 & & \wedge\ \mathtt{match}(\langle u,\ e\rangle, \overline{d}, \langle\varnothing,\varnothing\rangle, e'_1, e'_2) \rightarrow^* \mathtt{succ}(\varDelta', \Theta) \\
 & & \wedge\ (\Theta = \langle\varnothing, \overline{id}_2\rangle \vee \neg\mathsf{complete}(e'_1{;}e'_2)) \\
(\mathsf{SC2}) & \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1{:}e_2) \rightarrow \mathtt{err}(\langle ek, \overline{d}\rangle), & \text{if } \mathsf{build}(u, e_1) = e'_1 \wedge \mathsf{build}(u, e_2) = e'_2 \\
 & & \wedge\ \mathtt{match}(\langle u,\ e\rangle, \overline{d}, \langle\varnothing,\varnothing\rangle, e'_1, e'_2) \rightarrow^* \mathtt{succ}(\varDelta', \Theta) \\
 & & \wedge\ \Theta = \langle\overline{id}_1, \overline{id}_2\rangle \wedge \overline{id}_1 \neq \varnothing \wedge \mathsf{complete}(e'_1{;}e'_2) \\
 & & \wedge\ ek = \mathtt{unmatched}(\overline{id}_1, \mathsf{getBinders}(e'_1) \setminus \overline{id}_2, \overline{id}_2) \\
(\mathsf{SC3}) & \mathtt{slv}(\langle u,\ e\rangle, \overline{d}, e_1{:}e_2) \rightarrow \mathtt{err}(er), & \text{if } \mathsf{build}(u, e_1) = e'_1 \wedge \mathsf{build}(u, e_2) = e'_2 \\
 & & \wedge\ \mathtt{match}(\langle u,\ e\rangle, \overline{d}, \langle\varnothing,\varnothing\rangle, e'_1, e'_2) \rightarrow^* \mathtt{err}(er)
\end{array}
$$

structure/signature matching

$$
\begin{array}{lll}
(\mathsf{SM1}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, \top) & \rightarrow \mathtt{succ}(\varDelta, \Theta) \\
(\mathsf{SM2}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, e_1{;}e_2) & \rightarrow \mathtt{match}(\varDelta', \overline{d}, \Theta', e, e_2), \\
 & \quad \text{if } \mathtt{match}(\varDelta, \overline{d}, \Theta, e, e_1) \rightarrow^* \mathtt{succ}(\varDelta', \Theta') & \\
(\mathsf{SM3}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, e_1{;}e_2) & \rightarrow \mathtt{err}(er), \\
 & \quad \text{if } \mathtt{match}(\varDelta, \overline{d}, \Theta, e, e_1) \rightarrow^* \mathtt{err}(er) & \\
(\mathsf{SM4}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}vid{=}\sigma_1) & \rightarrow \mathtt{succ}(\varDelta', \mathsf{addI}(\Theta, vid)), \\
 & \quad \text{if } e(vid) = \sigma_2 \wedge \mathtt{slv}(\varDelta, \overline{d}, \sigma_2 \preceq_{vid} \sigma_1) \rightarrow^* \mathtt{succ}(\varDelta') & \\
(\mathsf{SM15}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}vid{=}\sigma_1) & \rightarrow \mathtt{err}(er), \\
 & \quad \text{if } e(vid) = \sigma_2 \wedge \mathtt{slv}(\varDelta, \overline{d}, \sigma_2 \preceq_{vid} \sigma_1) \rightarrow^* \mathtt{err}(er) & \\
(\mathsf{SM5}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}tc{=}\kappa_1) & \rightarrow \mathtt{succ}(\varDelta', \mathsf{addI}(\Theta, tc)), \\
 & \quad \text{if } e(tc) = \kappa_2 \wedge \mathtt{slv}(\varDelta, \overline{d}, \kappa_2 \preceq_{tc} \kappa_1) \rightarrow^* \mathtt{succ}(\varDelta') & \\
(\mathsf{SM16}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}tc{=}\kappa_1) & \rightarrow \mathtt{err}(er), \\
 & \quad \text{if } e(tc) = \kappa_2 \wedge \mathtt{slv}(\varDelta, \overline{d}, \kappa_2 \preceq_{tc} \kappa_1) \rightarrow^* \mathtt{err}(er) & \\
(\mathsf{SM6}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}strid{=}e_0) & \rightarrow \mathtt{succ}(\varDelta', \mathsf{addI}(\Theta, strid)), \\
 & \quad \text{if } e(strid) = e'_0 \wedge \varDelta = \langle u_1,\ e_1\rangle & \\
 & \quad \wedge\ \mathtt{match}(\varDelta, \overline{d}, \langle\varnothing,\varnothing\rangle, e'_0, e_0) \rightarrow^* \mathtt{succ}(\langle u_2,\ e_2\rangle, \langle\overline{id}_1, \overline{id}_2\rangle) & \\
 & \quad \wedge\ (\overline{id}_1 = \varnothing \vee \neg\mathsf{complete}(e'_0{;}e_0)) \wedge \varDelta' = \langle u_2,\ e_1{;}({\downarrow}strid \overset{\overline{d}}{=} \mathsf{diff}(e_1, e_2))\rangle & \\
(\mathsf{SM17}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}strid{=}e_0) & \rightarrow \mathtt{err}(\langle ek, \overline{d}\rangle), \\
 & \quad \text{if } e(strid) = e'_0 \wedge \mathtt{match}(\varDelta, \overline{d}, \langle\varnothing,\varnothing\rangle, e'_0, e_0) \rightarrow^* \mathtt{succ}(\varDelta', \langle\overline{id}_1, \overline{id}_2\rangle) \wedge \overline{id}_1 \neq \varnothing & \\
 & \quad \wedge\ \mathsf{complete}(e'_0{;}e_0) \wedge ek = \mathtt{unmatched}(\overline{id}_1, \mathsf{getBinders}(e'_0) \setminus \overline{id}_2, \overline{id}_2) & \\
(\mathsf{SM7}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}strid{=}e_0) & \rightarrow \mathtt{err}(er), \\
 & \quad \text{if } \mathtt{match}(\varDelta, \overline{d}, \langle\varnothing,\varnothing\rangle, e(strid), e_0) \rightarrow^* \mathtt{err}(er) & \\
(\mathsf{SM8}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}vid{=}is_1) & \rightarrow \mathtt{succ}(\varDelta{;}({\downarrow}vid{=}is), \Theta), \\
 & \quad \text{if } e[vid] = is_2 \wedge (\mathsf{solvable}(is_1 \overset{\overline{d}}{=} is_2) \vee \mathsf{strip}(is_1) = \mathtt{v}) \wedge is = \mathsf{ifNotDum}(is_1, is_2^{\overline{d}}) & \\
(\mathsf{SM9}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}vid{=}is_1) & \rightarrow \mathtt{err}(er), \\
 & \quad \text{if } \mathsf{strip}(is_1) \neq \mathtt{v} \wedge \mathtt{slv}(\varDelta, \overline{d}, is_1{=}e[vid]) \rightarrow^* \mathtt{err}(er) & \\
(\mathsf{SM10}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, {\downarrow}id{=}x) & \rightarrow \mathtt{succ}(\varDelta{;}({\downarrow}id{=}y), \Theta'), \\
 & \quad \text{if } e(id) \text{ is undefined} \wedge y = \mathsf{toDumVar}(x) \wedge \Theta' = \mathsf{addO}(\Theta, id) & \\
(\mathsf{SM11}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, ev) & \rightarrow \mathtt{succ}(\varDelta{;}ev, \Theta) \\
(\mathsf{SM12}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, e'^{\overline{d}'}) & \rightarrow \mathtt{match}(\varDelta, \overline{d} \cup \overline{d}', \Theta, e, e') \\
(\mathsf{SM14}) & \mathtt{match}(\varDelta, \overline{d}, \Theta, e, \odot) & \rightarrow \mathtt{succ}(\varDelta{;}\odot, \Theta)
\end{array}
$$

Figure 15.1 Constraint solving to handle merged unmatched errors

during constraint solving.

The end points of a minimal type error clash are notable program locations because they are the sources of conflicting types and because as such they allow us to derive the kind of the error and therefore they allow us to produce a verbose type error message.

For example the end points of the type constructor clash in `fn x => (x 1, x true)` are the locations of `1` and `true`. As discussed above, we use different colours to highlight end points. The type error report for this error is composed by, among other things, the following highlighting:

```
fn x => (x 1, x true)
```

and the following verbose message:

```
Type constructor clash between int and bool
```

This report does not involve the Standard ML basis but a builtin basis where `1` can only have the type `int` (from the initial static basis [107, Appendix C]). When checked against the Standard ML basis where `1` is overloaded to several different `int` types, one obtains the following message:

```
Constant 1 overloaded to the overloading class Int not including bool
```

The overloading class `Int` is a set of `int` types that contains the type `int` from the initial static basis (See Sec. 18.3 for more details on overloading).

An unmatched error can be regarded as a clash error between two sets of identifiers. For example, in

```
signature s = sig val y : int end
structure S = struct val x = 1 end :> s
```

the set $\{\mathtt{y}\}$ should be included in the set $\{\mathtt{x}\}$. The end points of the unmatched error in this piece of code are the locations of `x` and `y`.

In order to keep track of end points, changes in our constraint system are required. Let us informally present how $\mathsf{Impl\text{-}TES}$ handles end points. We only informally present how to handle end points because formally presenting this feature of our TES the way we have implemented it would require updating most of the machinery presented so far.

First, we annotate the type constructor names in the internal type constructor set as follows: we replace the γs in ITyCon by terms of the form $\langle \gamma, l \rangle$. We do the same for $\mathtt{ar}$ and replace it by $\langle \mathtt{ar}, l \rangle$. That is to say, We define the following set:

$$\widetilde{\gamma} \in \mathsf{LabTyConName} ::= \langle \gamma, l \rangle \mid \langle \mathtt{ar}, l \rangle$$

and replace the type constructor names in ITyCon as follows:

$$\gamma \xrightarrow{\mathsf{ITyCon}} \widetilde{\gamma}$$

As part of an informal presentation on how to handle end points, this figure only updates few constraint generation rules. Not all the rules that need to be updated are redefined in this figure.

$$\begin{array}{l}(\mathsf{G18})\ \mathtt{datatype}\ dn \overset{l}{=} cb \mathrel{-\!\!\triangleright} (ev{=}((\alpha_1 \overset{l}{=} \omega_1\, \langle\gamma, l\rangle);(\alpha_2 \overset{l}{=} \alpha_1);e_1;\mathtt{loc}\ e_1'\ \mathtt{in}\,\mathtt{poly}(e_2)));ev^l \\ \qquad \Leftarrow\ dn \mathrel{-\!\!\triangleright} \langle\alpha_1,\omega_1,e_1,e_1'\rangle \wedge cb \mathrel{-\!\!\triangleright} \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\gamma,ev) \\ (\mathsf{G3})\ \lceil exp\ \ atexp\rceil^l \mathrel{-\!\!\triangleright} \langle\alpha,\ e_1;e_2;(\alpha_1 \overset{l}{=} \alpha_2 \overset{l}{\rightarrow} \alpha)\rangle\ \Leftarrow\ exp \mathrel{-\!\!\triangleright} \langle\alpha_1,\ e_1\rangle \wedge atexp \mathrel{-\!\!\triangleright} \langle\alpha_2,\ e_2\rangle \wedge \mathsf{dja}(e_1,e_2,\alpha)\end{array}$$

Figure 15.2 Redefinition of some constraint generation rules to handle end points

As part of an informal presentation on how to handle end points, this figure only updates few constraint solving rules. Not all the rules that need to be updated are redefined in this figure.

$$\begin{array}{l}(\mathsf{S12})\ \mathtt{slv}(\varDelta,\overline{d},sq\,\mu{=}\tau) \rightarrow \mathtt{slv}(\varDelta,\overline{d}_1 \cup \overline{d}_2,\widetilde{\gamma}{=}\widetilde{\gamma}';sq{=}sq'), \\ \qquad \text{if } \tau = sq'\,\mu' \wedge \mathsf{collapse}(\mu^{\overline{d}}) = \widetilde{\gamma}^{\overline{d}_1} \wedge \mathsf{collapse}(\mu'^{\varnothing}) = \widetilde{\gamma}'^{\overline{d}_2} \\ (\mathsf{S13})\ \mathtt{slv}(\varDelta,\overline{d},\tau_1{=}\tau_2)\ \ \rightarrow \mathtt{slv}(\varDelta,\overline{d},\mu{=}\langle\mathtt{ar},l\rangle), \\ \qquad \text{if } \{\tau_1,\tau_2\} = \{sq\,\mu,\tau_0 \overset{l}{\rightarrow} \tau_0'\} \wedge \mathsf{strip}(\mu) \in \mathsf{LabTyConName} \\ (\mathsf{S6})\ \ \mathtt{slv}(\varDelta,\overline{d},\mu_1{=}\mu_2) \rightarrow \mathtt{err}(\langle\mathtt{tyConsClash}(\mu_1,\mu_2),\overline{d}\rangle), \\ \qquad \text{if } \{\mu_1,\mu_2\} \in \{\{\langle\gamma,l_1\rangle,\langle\gamma',l_2\rangle\},\{\langle\gamma,l_1\rangle,\langle\mathtt{ar},l_2\rangle\}\} \wedge \gamma \neq \gamma'\end{array}$$

Figure 15.3 Redefining of some constraint solving rules to handle end points

We also remove ar from ITyCon. We label arrow types as follows:

$$\tau_1{\rightarrow}\tau_2 \xrightarrow{\mathsf{ITy}} \tau_1 \overset{l}{\rightarrow} \tau_2$$

At constraint generation, instead of generating γ's, we generate constraint terms of the form $\langle\gamma, l\rangle$ where l is the label annotating the labelled external syntactic form responsible for γ's generation. For example, we would replace rule (G18) defined in Fig. 14.28 by the one defined in Fig. 15.2. The new rule only differs from the old one by the replacement of the generated γ by $\langle\gamma, l\rangle$. We also need to update each rule introducing a type of the form $\tau_1{\rightarrow}\tau_2$. For example, we need to replace rule (G3) defined in Fig. 11.7 by the one defined in Fig. 15.2. The new rule only differs from the old one by the replacement of $\alpha_1{\rightarrow}\alpha_2$ by $\alpha_1 \overset{l}{\rightarrow} \alpha_2$. Note that Fig. 15.2 only presents a few changes that need to be made to our initial constraint generation algorithm. Not all the necessary changes are presented in this figure.

We also have to update some constraint solving rules. For example, we replace rule (S12) defined in Fig. 14.29 by the one defined in Fig. 15.3. The only difference with the old rule is that TyConName has been replaced by LabTyConName. Another example is rule (S13) which is originally defined in Fig. 14.29 and which is updated in Fig. 15.3. The only difference with the old rule is that ar is replaced by $\langle\mathtt{ar}, l\rangle$, $\tau_0{\rightarrow}\tau_0'$ is replaced by $\tau_0 \overset{l}{\rightarrow} \tau_0'$ and TyConName has been replaced by LabTyConName. Yet another example is rule (S6) which is originally defined in Fig. 11.10 and which is updated in Fig. 15.3. In the new rule, l_1 and l_2 are the two end points of a type constructor clash. Note that Fig. 15.3 only presents a few changes that need to be made to our constraint solver. Not all the necessary changes are presented in this figure.

Instead of changing the syntax of internal types and internal type constructors, it could be interesting to investigate the handling of end points defined as dependencies as follows:

$$d \in \mathsf{Dependency} ::= \cdots \mid \mathsf{e}(l)$$

We leave this investigation for future work.

Chapter 16

Some of TES' properties

16.1 Compositionality

16.1.1 Status of the compositionality of our TES

The TES originally defined by Haack and Wells [57] allowed a compositional analysis. Their constraint generation algorithm was accumulating the types of identifiers at bound occurrences in an environment using intersection types. When dealing with a polymorphic declaration of an identifier *id*, their constraint generation was duplicating the constraints generated for the declaration as many times as there were types associated with *id* in the environment generated for its scope. This approach led to a combinatorial explosion in the number of generated constraints. To solve this combinatorial explosion, we switched to another approach to polymorphic declarations. Bindings are now solved at constraint solving. At constraint solving our TES forces the solving of the constraints generated for a polymorphic declaration before using it. Constrained types are simplified into types. We then only have to duplicate the type of a polymorphic declaration and not all the constraints initially generated for it. This idea was initially based on other works such as the ones by, e.g., Gustavsson and Svenningsson [55] or Pottier and Rémy [116].

Because of this change in our system we have lost the compositionality of our analysis. As a matter of fact, because we force the solving of the constraints generated for a polymorphic declaration before using it, if the declaration refers to a free identifier, once the type of the polymorphic declaration is generated from the constraints, this type is then independent from the free identifier's type. For example, when solving e, the environment generated for `val rec f = fn x => z`, because `z` occurs free, `f`'s type is of the form (where dependencies have been omitted for readability reasons): $\forall\{\alpha_1, \alpha_2\}.\, \alpha_1 \to \alpha_2$ where α_1 is `x`'s type and α_2 is a type constrained to be equal to `z`'s type. This type scheme does not depend on `z`. If `f`'s declaration is placed in a larger context containing the declaration `val z = ()`, to be able to recompute `f`'s type in this larger context we need to solve the environment generated for `val z = ()` and then solve once again e. We cannot reuse any information previously computed while solving e the first time.

However, the compositionality of our initial constraint generation algorithm is not affected by this change. It remained compositional thanks to our system of binders and accessors. Our constraint generation algorithm is not based on environments that accumulate the types of identifiers at bound occurrences. For an identifier at a bound occurrence, we generate an accessor as part of the generated environment. When dealing with an identifier *id* at binding occurrence we do not generate constraints relating the type of *id* to its bound occurrences. We do not compute bindings at initial constraint generation but for such an identifier we generate a binder as part of the generated environment. We therefore delay the solving of bindings to be dealt with at constraint solving instead.

These binders and accessors are especially necessary to obtain a compositional initial constraint generation algorithm while handling features such as `open` declarations and dealing with SML identifier statuses. When dealing with an `open` declaration and when the opened structure identifier is free, we are facing the fact that the structure might be in the scope of identifiers that it re-declares. Without binders and accessors, at constraint generation, one can then choose to either (1) shadow all the identifiers in which the `open` declaration is in the scope of, or (2) shadow none of them, or (3) solve the structure opening. None of these solutions would allow one to design a compositional constraint generation algorithm. A compositional constraint generation algorithm must allow the structure declaration to be analysed after analysing declarations which open it. Solutions (1) and (2) are not suitable because it might turn out that the structure only partially shadows the declared identifiers in which the `open` declaration is in the scope of. Solution (3) would require having the opened structure already analysed by the constraint generation algorithm when dealing with its opening. Also, solution (3) would not allow one to separate the constraint generation phase from the constraint solving phase and would not allow "faithful" representations of pieces of code in a constraint language. In our system, when dealing with an `open` declaration, we generate an accessor referring to the opened structure identifier and then export the environment declared by the structure via an environment variable.

Let us now discuss the handling of SML identifier statuses. When dealing with an identifier *vid* in a pattern that is not a recursive function (`f` is a recursive function in `val rec f = fn x => x`, but it is not in `val f = fn x => x`) the status of *vid* is resolved by looking at its context. If *vid* is declared as a recursive function in its context then *vid* is forced to be a value variable and not a datatype constructor and if *vid* is declared as a datatype constructor in its context then *vid* is forced to be a datatype constructor and not a value variable. If *vid* is neither declared as a value variable nor as a datatype constructor or if *vid* is free in its context then *vid* could either be a value variable or a datatype constructor. If the analysed piece of code is complete then it means that *vid* is a value variable but if the piece of code is incomplete we cannot resolve the status of *vid*. In our system if we cannot resolve the status of an identifier *vid* then it is considered as a dependent value variable (dependent on *vid*'s status). At constraint generation we therefore generate unconfirmed binders (see Sec. 14.1) which allow us to delay the resolution of identifier status to be dealt with at constraint solving. Making this decision at initial constraint generation would not allow our initial constraint generation algorithm to be compositional.

Because accessors and binders allow us to delay the resolution of bindings to be dealt with at constraint solving rather than at constraint generation, we can therefore obtain a compositional initial constraint generation algorithm. However, because constraint solving requires the context of an environment e to be solved before solving e, it is therefore not compositional.

16.1.2 Future work on compositionality

Unfortunately, our initial constraint generator is not compositional anymore once fixity declarations are added to the language. Fixity declarations influence the parsing of a piece of code. We do not have a good solution to handle fixity declarations in a compositional way. Therefore, our TES deals with fixity at parsing time. We leave the study of a compositional constraint generation algorithm in the presence of fixity declarations for future work.

Finally, we believe that the intersection type machinery introduced to handle functors in Sec. 14.9 could be used to partially recover the compositionality of constraint solving. For example, let us consider the declaration `val rec f = fn x => z`. Informally, instead of discarding `z`'s accessor, we could imagine generating an accessor of the form (we omit dependencies and $\top$ for readability purposes) $\uparrow\mathtt{z}{=}\alpha \Cap sv$ which would be stored in the constraint solving context from the state in which the constraint solver is when dealing with `z`'s accessor. We would also generate a binder of the form $\downarrow\mathtt{f}{=}\forall\{\alpha,\alpha'\}.\,\{\langle\alpha,sv\rangle\}\diamond\alpha'{\to}\alpha$ for `f`. If, e.g., `val u = if f () then 1 else 0` was in the scope of `f`'s declaration, we would then constrain sv to be equal to $\mathtt{bool} \Cap sv'$. For the sequence of the two declarations, we would then generate an environment of the form $(\uparrow\mathtt{z}{=}\alpha \Cap \mathtt{bool} \Cap sv');(\downarrow\mathtt{f}{=}\forall\{\alpha,\alpha'\}.\,\{\langle\alpha,sv\rangle\} \diamond \alpha'{\to}\alpha)$. If these two declarations were in the scope of `val z = ()`, where `z` has type `unit`, we would then obtain a type error clash when constraining `unit` to be a subtype of `bool`. If instead these two declarations were in the scope of `val z = true`, where `z` has type `bool`, we would constrain further `f`'s binder to be $\downarrow\mathtt{f}{=}\forall\{\alpha'\}.\,\alpha'{\to}\mathtt{bool}$ by constraining α to be equal to `bool`. However, we believe that such a solution would be inefficient. Let us also sketch the implications of such a system in the presence of `open` declarations. Let us consider the following sequence of declarations:

`val z = (); open S; val rec f = fn x => z`. Instead of simply discarding `S`'s binder we would then store it in the constraint solving context from the state in which the constraint solver is when dealing with `S`'s accessor. We would then generate the following environment $(\downarrow\mathtt{z}{=}\forall\varnothing.\,\mathtt{unit});(\uparrow\mathtt{S}{=}ev);ev;(\uparrow\mathtt{z}{=}\alpha\capdot sv);(\downarrow\mathtt{f}{=}\forall\{\alpha,\alpha'\}.\,\{\langle\alpha,sv\rangle\}\diamond\alpha'{\to}\alpha)$. It becomes then unclear what to do when also dealing with, among other things, signatures. We also leave the investigation of such a system for future work.

16.2 Satisfiability of Yang et al.'s criteria

Yang, Wells, Trinder and Michaelson [149] provide a list of criteria for good type error reports. We will now informally present how our type error reports meet these criteria.

First, let us point out that in TES a type error report is composed by a type error slice, a highlighting, a verbose explanation of the kind of the error, and a set of identifier statuses context dependencies.

Correct. For the same reasons as listed in Sec. 11.9, we have not formally proved that, given a piece of code, our initial constraint generation algorithm generates unsolvable constraints if and only if the piece of code does not have a static semantics in SML. We however strongly believe this result to be true.

Moreover, every SML compiler already contains a type inference algorithm ensuring only type safe code is compiled. Standard software engineering techniques, like our database of 550 regression tests (typable and untypable pieces of SML), are much more cost effective for ensuring high quality error slices. This database is used to check the empirical correctness of our algorithms.

Note that we do not plan on building another SML compiler but instead we would like to obtain an interface where the errors reported by TES would be preferred over the ones of any SML compiler. This interface could regularly run our TES while programmers are implementing (e.g., every time programmers stop typing for a certain amount of time). If a type error was discovered by our TES it would then be reported to the user, otherwise we would rely on a SML compiler chosen by the user to find errors that our TES does not find (this would be considered as a bug of our TES once our implementation finished) and to compile the code. We leave the building of such an interface for future work.

Precise. We have not proved the minimality result stated in Sec. 11.9 but we strongly believe that our TES only reports minimal errors. We believe that our type error slices are minimal and that therefore they are precise because they do not involve portions of code not participating in the reported errors.

Succinct. Our verbose explanations are succinct. For example, for `() ()`, we would report the type error slice $\langle$`..() `$\langle$`..`$\rangle$`..`$\rangle$. We would also report a verbose, clear and brief message explaining that the error is a type constructor clash between the type unit and the functional type.

A-mechanical. TES does not report any internal constraint term computed while searching for type errors.

Source-based. We consider the main components of type error reports in TES to be the highlightings. Our highlightings directly present type errors in the user code and therefore are source-based. A type error slice however is based on the user code, where portions not participating to the reported error are omitted. The omissions are made explicit thanks to dots and extra parentheses. Note that a type error slice is therefore not strictly speaking source-based because it involves extra symbols. However, type error slices are mainly in our reports to formally define type errors and to make explicit the scoping of identifiers in the highlightings.

Unbiased. TES is unbiased thanks to its enumeration algorithm which is designed to find all minimal unsatisfiable portions of a constraint/environment. Moreover, by default no location is presented in our system as being more important than others. End points are highlighted using different colours because they are used among other things to generate our verbose error messages. They are by no means more important than the other locations. Note the use of "by default" above. Even though we do not believe that any location in a type error slice should be more important than the other ones, we also believe that this could be relaxed depending on users' preferences. For example, one could prefer looking at the non signature related portions of a highlighting and therefore would prefer having the signature related portions of a type error highlighted with a lighter colour. This has not been implemented or investigated yet.

Comprehensive. Thanks to both our highlightings and our type error slices, given a type error report, the user does not need to look at any other portion that is not involved in the report. Moreover, our type error slices are unambiguous. In our type error slices, bindings of identifiers are made explicit thanks to our extra dots and parentheses.

Chapter 17

Implementation discussion

17.1 Other implemented features

17.1.1 Syntax errors

As mentioned in Sec. 14.1 and Sec. 14.10, our implementation also reports some context-sensitive and context-insensitive syntactic errors. Let us present some examples.

We have already mentioned in Sec. 14.1 that our TES reports that `x` occurring twice in the pattern in `fn (x, x) => x` is an error only if `x` has value variable status. This is a context-sensitive syntactic error that depends on the `x`'s status. We report the following highlighting `fn (x, x) => x` where `fn` and `=>` are highlighted to show that the highlighted `x`'s occur in a pattern.

We also report various context-insensitive multi-occurrence syntax errors. For example, in Sec. 14.10, we mentioned that `type ('a, 'a) t = 'a` is syntactically incorrect because the explicit type variable `'a` occurs twice in the type variable sequence `('a, 'a)`. We report the following highlighting `type ('a, 'a) t = 'a` where `type` and `=` are highlighted to show that the highlighted `'a`'s occur in the type variable sequence of a type declaration. The datatype declaration `datatype t = T | T` is also syntactically incorrect because it declares twice the datatype constructor `T`. We report the following highlighting `datatype t = T | T`. Also, `datatype t = V and u = V` which declares, among other things, two datatypes `t` and `u` is syntactically incorrect because `V` is declared as a datatype constructor of both `t` and `u` in the same datatype declaration. We report the following highlighting `datatype t = V and u = V`. We report many other cases of multi-occurrence syntax errors that we do not discuss in this document.

Let us present another kind of context-insensitive syntactic error. The datatype specification `datatype ('a,'b)t = T of 'a -> 'c` is syntactically incorrect because the type variable `'c` does not occur in the type variable sequence `('a,'b)`. We report the following highlighting `datatype ('a,'b)t = T of 'a -> 'c`. We also explain in the report that a type variable is unbound in the declaration.

Let us present a last example. As mentioned in Sec. 11.2, recursive declarations' bodies must be fn-expressions. For example, `val rec f = ()` is not syntactically correct because `()` is not a fn-expression. We report the following highlighting `val rec f = ()`.

17.1.2 Datatype replications

A datatype replications in SML is of the form `datatype t = datatype u`. For example if `u` is defined in the context as follows: `datatype u = U | V`, then the datatype replication will have the effect to splice `U`'s constructors into the current environment.

Datatype replications are handled similarly to `open` declarations in Impl-TES. In our implementation we also associate environments with external type constructors. For example, for `datatype u = U | V`, we generate a binder for `u` that carries `u`'s type but it also binds an environment which is the environment generated for its constructors (`U` and `V` in our example). Then we deal with the datatype declaration by generating an accessor that does not access to `u`'s internal type but that access to the environment associated with `u`.

17.1.3 Exceptions

When adding exceptions, one has to consider another identifier status: exception constructors. Let us present some interesting issues raised by exceptions. First, let us consider the following typable piece of code:

```
exception ex of int;
exception fx = ex;
val x = fn () => raise fx 0;
```

The exception constructor `ex` is unary. But the arity of `fx` cannot be inferred by just looking at the declaration `exception fx = ex`. The arity of `fx` depends on `ex`'s arity. That is why when dealing with exceptions we need more that the dummy status variable $\eta_{\mathtt{dum}}$. When generating constraints for the declaration `exception fx = ex`, we associate a status variable with the exception `fx`, which we constrain to be equal to `ex`'s status, which is obtained via an accessor.

There is another issue raised when dealing with such declarations. The issue is that given `exception fx = ex`, we do not need to know `ex`'s arity to known that `fx` is an exception constructor. We therefore consider extra raw statuses. We have a nullary exception raw status `e0` and a unary exception raw status `e1` (as we have `d` and `c` for datatype constructors), but we also have an extra exception raw status `e` for when the arity of an exception constructor is unknown. For our example, at initial constraint generation we constrain `fx`'s status to be equal to `ex`'s status and we also constrain it to be equal to e. This constraining is made such that the constraint on fx's status with ex's status will predominate the constraint on fx's status with the raw status e.

17.1.4 Long identifiers

Long identifiers are used to access identifiers defined in structures. Let us consider the following simple typable SML program:

(EX14)

```
structure S = struct
  val a = 1
  val f = fn x => x + 1
  structure T = struct val b = f a end
end
val x = S.T.b + 1
```

The main point of this example is that S.T.b is a long identifier that allows one to access the identifier b defined in T, itself defined in S. A long identifier is a sequence (possibly empty) of structure identifiers, each of them followed by a dot, followed by an identifier. In order to handle long identifiers in Impl-TES we use long accessors where instead of an identifier one can have a labelled long identifier.

One can then obtain *unmatched* errors involving long identifiers. For example, if one replaces S.T.b by S.T.v in example (EX14) one obtains the following highlighting:

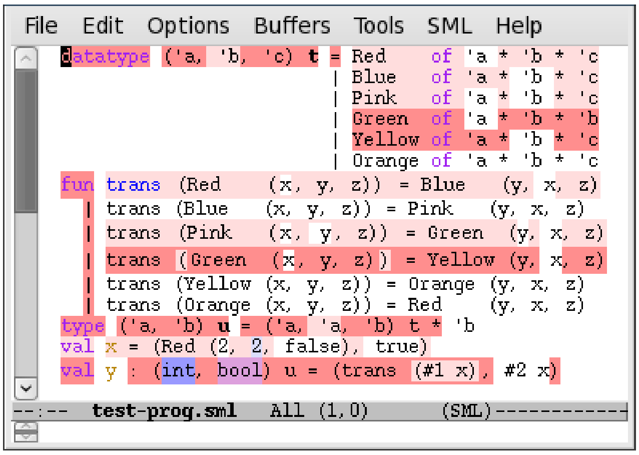
```
structure S = struct
  val a = 1
  val f = fn x => x + 1
  structure T = struct val b = f a end
end
val x = S.T.v + 1
```

We have not fully finished implementing support for long identifiers, but we plan in reporting the two following slices for the following variant of example (EX14):

```
structure S = struct
  val a = 1
  val f = fn x => x + 1
  structure T = struct val b = f a end
end
val x = S.Y.b + 1
```

```
structure S = struct
  val a = 1
  val f = fn x => x + 1
  structure T = struct val b = f a end
end
val x = S.Y.b + 1
```

This example differs from example (EX14) by the replacement of S.T.b by S.Y.b. The first highlighting shows that S.Y tries to access Y in S and that S does not declare S. The second highlighting shows that S.Y. tries to access the structure Y in S and that S does not declare any structure called Y.



File Edit Options Buffers Tools SML Help

```
datatype ('a, 'b, 'c) t = Red    of 'a * 'b * 'c
                        | Blue   of 'a * 'b * 'c
                        | Pink   of 'a * 'b * 'c
                        | Green  of 'a * 'b * 'b
                        | Yellow of 'a * 'b * 'c
                        | Orange of 'a * 'b * 'c
fun trans (Red    (x, y, z)) = Blue   (y, x, z)
  | trans (Blue   (x, y, z)) = Pink   (y, x, z)
  | trans (Pink   (x, y, z)) = Green  (y, x, z)
  | trans (Green  (x, y, z)) = Yellow (y, x, z)
  | trans (Yellow (x, y, z)) = Orange (y, x, z)
  | trans (Orange (x, y, z)) = Red    (y, x, z)
type ('a, 'b) u = ('a, 'a, 'b) t * 'b
val x = (Red (2, 2, false), true)
val y : (int, bool) u = (trans (#1 x), #2 x)
```

--:-- test-prog.sml All (1,0) (SML)-----------

Figure 17.1 Highlighting of a SML type error in Emacs

17.2 Performance

Our implementation is currently usable for small projects (a few thousand lines) and is steadily improving. Our latest TES is 10 to 100 times faster in many cases than before we switched to using our constraint/environments. Our previous TES version was already enormously faster than HW-TES (the original TES by Haack and Wells) due to avoiding duplication of polymorphic types. We believe that more careful use of data structures and algorithms will allow much better performance.

Minimisation and enumeration are expensive. The expense of minimisation is handled by (1) reporting partially minimised slices to the user interface while minimisation continues in the background, and (2) designing the constraint solving system to avoid including unneeded parts of the program in slices whenever possible (which means each iteration of minimisation does less work, as explained in Sec.11.7.6). The expense of enumeration is handled by reporting slices to the user interface as they are produced while continuing enumeration in the background. Wolfram's result [145] shows there will be an exponential number of minimal type error slices in the worst case, so we merely aim to quickly present a few of them.

17.3 User interface

An Emacs interface (and a preliminary one for Vim) highlights slices in the edited source code. There is also a terminal command-line interface. Fig. 17.1 presents a screenshot of the type error presented in Sec. 10.4.2 highlighted in Emacs. The light pink corresponds to slices other than the focused one. Other such screenshots are provided in Ch. 13.

17.4 The Standard ML basis library

Our examples have used operators like `::` and `+`. For now, we allow one to define the Standard ML basis in a file, and we provide a file declaring a portion of the basis. For the future, we have begun implementing a way to use library types extracted from a running instance of SML/NJ.

Chapter 18

Future work

18.1 Examples exhibiting the desire for even more type error reports

We have found some cases of incomplete pieces of code that we do not believe could be made typable by completing them. We present some of them in this section. Not reporting such errors prevents TES from reporting all minimal type error slices in the presence of incomplete pieces of code.

18.1.1 An example involving structures and signatures

We do not believe that the following incomplete piece of code could be made typable:

```
signature S = sig val f : ⟨..⟩ end
structure U = struct val f = true end : S
structure V = struct val f = () end : S
```

As a matter of fact, whatever ⟨..⟩ is replaced by, the piece of code would always be untypable. Finding such errors is complicated because, e.g., the following piece can be made typable by replacing ⟨..⟩ by `t`:

```
signature S = sig type t val f : ⟨..⟩ end
structure U = struct type t = bool val f = true end : S
structure V = struct type t = unit val f = () end : S
```

18.1.2 An example involving datatype constructors

Let us consider this other example in which `C` is free:

```
val _ = fn (C _) =>
           C () ()
```

The first occurrence of c forces c to be a unary (datatype or exception) constructor. The second occurrence of c forces c to take two arguments. In SML, datatype and exception constructors can take one argument at most. Therefore, we believe that there is no declaration of c that would make the piece of code typable. Currently our TES does not complain. We believe we could generate an error by generating at constraint solving a binder for c when dealing with the accessor generating for c's first occurrence. This binder would force c to have an arrow type. Note that this piece of code is incomplete in the sense that c is constrained to be a datatype constructor and there is no declaration of c as such.

18.1.3 An example involving type annotations

Let us now consider the following piece of code in which u is free:

```
datatype 'a t = T of 'a t
fun f x = T (x :  u)
```

We believe that there is no declaration of u that would make this piece of code typable. If u was defined as a datatype then it would have to be different from t because u's definition would have to precede t's definition. If u was defined as a type function then because it does not take any argument it would have to be a nullary type function. It then would have to be equal to a type that does not mention any type variable and therefore it would have to be equal to a type construct where the type constructor is different from t because u's definition would have to precede t's definition. We have not yet investigated the report of such errors.

18.2 Missing features

Some of SML's features are not yet handled by Impl-TES or by Form-TES. We do not yet deal with type and signature sharing, equality types, and flexible records.

Impl-TES handles non-flexible records but we have not started investigating flexible records. However, because we allow programmers to use flexible records in pieces of code (we parse them), Impl-TES handle them in a way that cannot cause false errors to be found. We have started implementing support for type sharing but it is currently at an early stage (we only catch a few errors involving type sharing specifications). We believe that the handling of equality types will require the introduction of another kind of rigid type variables (equality rigid type variables). We believe that the handling of these features will not require fundamental extensions to our constraint system. The handling of these features is left for future work.

Also, we have not yet implemented or formalised support for overloading resolution as specified in The Definition of Standard ML [107, Appendix E]: "Every overloaded constant and value identifier has among its types a *default type*, which is assigned to it, when the surrounding text does not resolve the overloading. For this purpose, the surrounding text is no larger than the smallest enclosing structure-level declaration; an implementation may require that a smaller context determines the type.” We currently do not do anything when “the surrounding text does not resolve the overloading”. For example, Impl-TES considers the following piece of code to be typable:

```
structure S = struct fun f x y = x + y end
open S
val x1 = f 1 2
val x2 = f 1.1 2.2
```

In SML the operator `+` is overloaded to the overloading class `Int` described in Sec. 18.3. If one follows The Definition of Standard ML, because the surrounding text of `+` in `f`’s definition does not resolve the overloading of `+`, it results that when dealing with `S` the function `f` is forced to be a function from `int` to `int` because the type `int` is the default type of the overloading class `Int`. (Note that implementations are allowed to resolve the overloading of `+` when inferring `f`’s type.) Therefore, `x1` is fine but `x2`’s body should be involved in a type error.

Overloading is further discussed in Sec. 18.3.

18.3 Overloading

18.3.1 Status of TES’ handling of overloading

Impl-TES partially handles overloaded operators and constants. We also allow the user to overload operators and to define overloading classes thanks to overloading declarations. These declarations are useful to define the Standard ML basis (Impl-TES uses a basis file containing most of the declarations from the Standard ML basis). There are however some issues stemming from the handling of overloading. One issue is that we feel that we do not currently do a good job at reporting type error slices involving overloaded operators or constants. Usually such errors involve many types from many structures from the Standard ML basis, and these tend to cloud type error slices. Let us first informally present our overloading declarations. We will then illustrate the issue mentioned above. Overloaded operators and constants are overloaded over overloading classes. An overloading class is the union of a number of type constructors. For example the overloading class `Int` is a type constructor set containing at least `int`. Similarly are defined the overloading classes `Real`, `Word`, `String`, and `Char` (See The Definition of Standard ML [107, Appendix E]). These overloading classes are called *basic*. On top of the basic overloading classes are defined the *composite* overloading classes which combine the basic overloading classes. For example the overloading class `RealInt` is defined as `Real`∪`Int`. Note that `Int` can contain (and usually does) other type constructors. In SML/NJ, it also contains, e.g., the type `Int.int` which is in SML/NJ the same as the type `int` (which is the `int` type at top-level), and also contains `Int32.int` which is in SML/NJ different from the type `int`. In SML/NJ, the overloading class `Int` contains many other type constructors. In Impl-TES, overloading classes can be defined using `overload` declarations which follow the following labelled syntax:

$$\begin{array}{lll} \mathit{ovcid} & \in \mathsf{OverloadingClassId} & \text{(overloading classes identifiers)} \\ \mathit{ovcitem} & \in \mathsf{OverloadingItem} & ::= \mathtt{in}^{l}\ \mathit{tc} \mid \mathit{ovcid}^{l} \\ \mathit{ovcseq} & \in \mathsf{OverloadingSeq} & ::= (\mathit{ovcitem}_1, \dots, \mathit{ovcitem}_n)^{l} \\ \mathit{dec} & \in \mathsf{Dec} & ::= \cdots \mid \mathtt{overload}\ \mathit{ovcid}^{l}\ \mathit{ovcseq} \end{array}$$

For example, in the basis file provided with the implementation of Impl-TES, the overloading class `Int` is defined as follows:

```
overload Int (int, Int.int, Int31.int, Int32.int,
              Position.int, IntInf.int, LargeInt.int)
```

We then use other kinds of overloading declarations to overload operators. These declarations follow the following labelled syntax:

$$\mathit{dec} \in \mathsf{Dec} ::= \cdots \mid \mathtt{overload}\ \mathit{vid}\ :^{l}\ \mathit{ty}\ \mathtt{with}\ \mathit{tv}\ \mathtt{in}\ \mathit{ovcseq}$$

For example, in our basis file, `+` is overloaded as follows:

```
overload + : 'a * 'a -> 'a with 'a in (in Int, in Word, in Real)
```

18.3.2 An issue in handling overloading

Let us now consider the following erroneous piece of code: `val x = 1 + true`. Because `true` is of type `bool` which is not a type in any of the overloading classes `Int`, `Word` or `Real` then one obtains a type error. The type error slice reporting this error needs to contain ⟨..⟩ `+ true`, but it also need to contain `+`'s definition and also all the types on which `+` is overloaded. In this case it involves reporting portions of many structures from the basis. All the reported information tend to cloud the main point of the error which is that `true` is not any of the types on which `+` is overloaded. Therefore, even though our error reports are correct, we believe we need to develop a way to "fold" such errors. The same arguments applies for overloaded constants. This is left for future work.

18.4 Tracking programming errors using TES

Even though type error slices are already of a great help on their own, we believe we could improve our error reports by proposing guidance to users to navigate through error slices. Let us consider the type error slice presented in Fig. 10.2 in Sec. 10.4.2. Sec. 10.4.2 contains some text describing a way of reading the presented type error slice, depending on the bindings in the slice. We would like to automate this in the future. We would like to make data flow information, computed at constraint solving, available to users. It is however not evident that such guidance on how to read type error slices would be useful for every error kind. We believe it would for at least type constructor clashes and circularity errors.

18.5 Combining TES with suggestions to repair type errors

We see that our work can enable work for suggesting fixes, because it can correctly calculate the portion of a program that is involved in a type error, while excluding the uninvolved portion. This would allow fix suggestions to correctly consider all the spots which need to be considered to find the right place for the fix. In the absence of information equivalent to a correct type error slice, automated fix suggestion will inevitably sometimes suggest fixes at the wrong places. We believe it could be interesting to study the combination of our work with other approaches to error reporting, e.g., by Hage and Heeren [59] or by Lerner et al. [99].

18.6 Proving the correctness of TES

Once Form-TES will be close enough to Full-TES and stable enough, we would like to prove its correctness (i.e., given a piece of code, it finds all and only the minimal errors of the given piece of code if and only if the piece of code is untypable). This would require proving the correctness of the different components of TES, i.e., of constraint generation, constraint solving, minimisation, enumeration, and slicing.

Appendix A

Proofs of Part I

A.1 From a semantic proof to a syntactic one (Ch. 4)

A.1.1 Saturation, variable, abstraction properties (Sec. 4.1)

Proof of Lemma 4.1.2. 1. If $r = \beta\eta$, the proof is by induction on the length of the reduction $M \rightarrow^*_{\beta\eta} N$.

- If $M = N$ then $M[x := P] = N[x := P]$. We prove that $N[x := P] \rightarrow^*_{\beta\eta} N[x := Q]$ by induction on the structure of N.
  - Let $N \in \mathsf{Var}$. If $N = x$ then $N[x := P] = P \rightarrow^*_{\beta\eta} Q = N[x := Q]$, else $N[x := P] = N = N[x := Q]$.
  - Let $N = \lambda y.N'$. By IH, $N[x := P] = \lambda y.N'[x := P] \rightarrow^*_{\beta\eta} \lambda y.N'[x := Q] = N[x := Q]$ such that $y \notin \mathsf{fv}(PQx)$.
  - Let $N = N_1 N_2$. By IH, $N[x := P] = N_1[x := P]N_2[x := P] \rightarrow^*_{\beta\eta} N_1[x := Q]N_2[x := Q] = N[x := Q]$.
- Let $M \rightarrow^*_{\beta\eta} M' \rightarrow_{\beta\eta} N$. By IH, $M[x := P] \rightarrow^*_{\beta\eta} M'[x := Q]$. We prove that $M'[x := Q] \rightarrow_{\beta\eta} N[x := Q]$ by induction on the structure of M'.
  - Let $M' \in \mathsf{Var}$ then nothing to prove since M' does not reduce.
  - Let $M' = \lambda y.M'_1$.
    - ∗ Either $N = \lambda y.M'_2$ such that $M'_1 \rightarrow_{\beta\eta} M'_2$. By IH, $M'_1[x := Q] \rightarrow_{\beta\eta} M'_2[x := Q]$. So $M'[x := Q] = \lambda y.M'_1[x := Q] \rightarrow_{\beta\eta} \lambda y.M'_2[x := Q] = N[x := Q]$ such that $y \notin \mathsf{fv}(Qx)$.
    - ∗ Or $M'_1 = Ny$ such that $y \notin \mathsf{fv}(N)$. So $M'[x := Q] = \lambda y.N[x := Q]y \rightarrow_{\eta} N[x := Q]$ such that $y \notin \mathsf{fv}(Qx)$.
  - Let $M' = M_1 M_2$.

    - Either $N = M_1'M_2$ such that $M_1 \to_{\beta\eta} M_1'$. By IH, $M_1[x := Q] \to_{\beta\eta} M_1'[x := Q]$. So $M'[x := Q] = M_1[x := Q]M_2[x := Q] \to_{\beta\eta} M_1'[x := Q]M_2[x := Q] = N[x := Q]$.
    - Or $N = M_1M_2'$ such that $M_2 \to_{\beta\eta} M_2'$. By IH, $M_2[x := Q] \to_{\beta\eta} M_2'[x := Q]$, so $M'[x := Q] = M_1[x := Q]M_2[x := Q] \to_{\beta\eta} M_1[x := Q]M_2'[x := Q] = N[x := Q]$.
    - Or $M_1 = \lambda y.M_1'$ and $N = M_1'[y := M_2]$. So, $M'[x := Q] = (\lambda y.M_1'[x := Q])M_2[x := Q] \to_\beta M_1'[x := Q][y := M_2[x := Q]] = N[x := Q]$ by the well known substitution lemma and such that $y \notin \mathsf{fv}(Qx)$.

If $r = \beta$, the proof is by induction on the length of the reduction $M \to_\beta^* N$.

- If $M = N$ then $M[x := P] = N[x := P]$. We prove that $N[x := P] \to_\beta^* N[x := Q]$ by induction on the structure of N.
  - Let $N \in \mathsf{Var}$. If $N = x$ then $N[x := P] = P \to_\beta^* Q = N[x := Q]$, else $N[x := P] = N = N[x := Q]$.
  - Let $N = \lambda y.N'$. By IH, $N[x := P] = \lambda y.N'[x := P] \to_\beta^* \lambda y.N'[x := Q] = N[x := Q]$ such that $y \notin \mathsf{fv}(PQx)$.
  - Let $N = N_1N_2$. By IH, $N[x := P] = N_1[x := P]N_2[x := P] \to_\beta^* N_1[x := Q]N_2[x := Q] = N[x := Q]$.
- Let $M \to_\beta^* M' \to_\beta N$. By IH, $M[x := P] \to_\beta^* M'[x := Q]$. We prove that $M'[x := Q] \to_\beta N[x := Q]$ by induction on the structure of M'.
  - Let $M' \in \mathsf{Var}$ then nothing to prove since M' does not reduce.
  - Let $M' = \lambda y.M_1'$. Then $N = \lambda y.M_2'$ such that $M_1' \to_\beta M_2'$. By IH, $M_1'[x := Q] \to_\beta M_2'[x := Q]$, so $M'[x := Q] = \lambda y.M_1'[x := Q] \to_\beta \lambda y.M_2'[x := Q] = N[x := Q]$ such that $y \notin \mathsf{fv}(Qx)$.
  - Let $M' = M_1M_2$.
    - Either $N = M_1'M_2$ such that $M_1 \to_\beta M_1'$. By IH, $M_1[x := Q] \to_\beta M_1'[x := Q]$, so $M'[x := Q] = M_1[x := Q]M_2[x := Q] \to_\beta M_1'[x := Q]M_2[x := Q] = N[x := Q]$.
    - Or $N = M_1M_2'$ such that $M_2 \to_\beta M_2'$. By IH, $M_2[x := Q] \to_\beta M_2'[x := Q]$, so $M'[x := Q] = M_1[x := Q]M_2[x := Q] \to_\beta M_1[x := Q]M_2'[x := Q] = N[x := Q]$.
    - Or $M_1 = \lambda y.M_1'$ and $N = M_1'[y := M_2]$. So, $M'[x := Q] = (\lambda y.M_1'[x := Q])M_2[x := Q] \to_\beta M_1'[x := Q][y := M_2[x := Q]] = N[x := Q]$ by the well known substitution lemma and such that $y \notin \mathsf{fv}(Qx)$.

2. We prove this lemma by induction on the structure of M.

- Let $M \in \mathsf{Var}$ then either $M = x$ and so $\mathsf{fv}(M[x := N]) = \mathsf{fv}(N) = \mathsf{fv}((\lambda x.M)N)$. Or $M \neq x$ and so $\mathsf{fv}(M[x := N]) = \mathsf{fv}(M) \subseteq \mathsf{fv}(M) \cup \mathsf{fv}(N) = \mathsf{fv}((\lambda x.M)N)$.
- Let $M = \lambda y.P$ then $\mathsf{fv}(M[x := N]) = \mathsf{fv}(\lambda y.P[x := N]) = \mathsf{fv}(P[x := N]) \setminus \{y\} \subseteq^{IH} \mathsf{fv}((\lambda x.P)N) \setminus \{y\} = \mathsf{fv}((\lambda x.M)N)$ such that $y \notin \mathsf{fv}(Nx)$.
- let $M = P_1P_2$ then $\mathsf{fv}(M[x := N]) = \mathsf{fv}(P_1[x := N]) \cup \mathsf{fv}(P_2[x := N]) \subseteq^{IH} \mathsf{fv}((\lambda x.P_1)N) \cup \mathsf{fv}((\lambda x.P_2)N) = \mathsf{fv}((\lambda x.M)N)$.

3. We prove this lemma by induction on the length of the reduction $M \to^*_{\beta\eta} N$.

   - Let $M = N$ then $\mathsf{fv}(M) = \mathsf{fv}(N)$.
   - Let $M \to^*_{\beta\eta} M' \to_{\beta\eta} N$. By IH, $\mathsf{fv}(M') \subseteq \mathsf{fv}(M)$. We prove that $\mathsf{fv}(N) \subseteq \mathsf{fv}(M')$ by induction on the structure of M'.
     - Let $M' \in \mathsf{Var}$ then nothing to prove since M' does not reduce.
     - Let $M' = \lambda x.P$.
       - Either $N = \lambda x.Q$ such that $P \to_{\beta\eta} Q$. By IH, $\mathsf{fv}(Q) \subseteq \mathsf{fv}(P)$. So $\mathsf{fv}(N) \subseteq \mathsf{fv}(M')$.
       - Or $P = Nx$ such that $x \notin \mathsf{fv}(N)$. So $\mathsf{fv}(N) = \mathsf{fv}(M')$.
     - Let $M' = P_1P_2$.
       - Either $N = P_1'P_2$ such that $P_1 \to_{\beta\eta} P_1'$. By IH, $\mathsf{fv}(P_1') \subseteq \mathsf{fv}(P_1)$, so $\mathsf{fv}(N) \subseteq \mathsf{fv}(M')$.
       - Or $N = P_1P_2'$ such that $P_2 \to_{\beta\eta} P_2'$. By IH, $\mathsf{fv}(P_2') \subseteq \mathsf{fv}(P_2)$, so $\mathsf{fv}(N) \subseteq \mathsf{fv}(M')$.
       - Or $P_1 = \lambda x.P_0$ and $N = P_0[x := P_2]$. By Lemma 4.1.2.2, $\mathsf{fv}(N) \subseteq \mathsf{fv}(M')$.

   A corollary of this result is that if $M \to^*_{\beta} N$ then $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$.

4 By induction on the length of the reduction $\lambda x.M \to^*_{\beta\eta} N$.

   - Let $\lambda x.M = N$ then it is done.
   - Let $\lambda x.M \to^*_{\beta\eta} P \to_{\beta\eta} N$. By IH:
     - Either $P = \lambda x.Q$ such that $M \to^*_{\beta\eta} Q$. Then, by compatibility:
       - Either $Q = Nx$ such that $x \notin \mathsf{fv}(N)$. So it is done since $M \to^*_{\beta\eta} Nx$.
       - Or $N = \lambda x.M'$ such that $Q \to_{\beta\eta} M'$. So it is done since $M \to^*_{\beta\eta} M'$.
     - Or $M \to^*_{\beta\eta} Px$ such that $x \notin \mathsf{fv}(P)$. So $M \to^*_{\beta\eta} Nx$ and it is done since by Lemma 4.1.2.3, $x \notin \mathsf{fv}(N)$.

5 By induction on the length of the reduction $Mx \to^*_{\beta\eta} N$.

- Let $N = Mx$ then it is done.
- Let $Mx \to^*_{\beta\eta} P \to_{\beta\eta} N$. Then by IH, $M \to^*_{\beta\eta} Q$ (by Lemma 4.1.2.3, $x \notin \mathsf{fv}(Q)$) and:
  - Either $P = Qx$. Then, by compatibility:
    - * Either $N = Q'x$ such that $Q \to_{\beta\eta} Q'$. So it is done since $M \to^*_{\beta\eta} Q'$.
    - * Or $Q = \lambda y.Q'$ and $N = Q'[y := x]$. So $M \to^*_{\beta\eta} \lambda y.Q' = \lambda x.N$.
  - Or $Q = \lambda x.P$. So it is done since $M \to^*_{\beta\eta} Q = \lambda x.P \to_{\beta\eta} \lambda x.N$.

6. (a) If $k = 0$ then $P = Q$ is a direct r-reduct of Q, absurd.

   (b) Assume $k = 1$, we prove $P = M[x := N]$ by case on r.

   - Let $r = \beta$. The proof is by case on $Q = (\lambda x.M)N \to_\beta P$.
     - If $(\lambda x.M)N \to_\beta M[x := N]$ then we are done.
     - If $(\lambda x.M)N \to_\beta (\lambda x.M')N = P$ such that $M \to_\beta M'$ then P is a direct β-reduct of $(\lambda x.M)N$, absurd.
     - If $(\lambda x.M)N \to_\beta (\lambda x.M)N' = P$ such that $N \to_\beta N'$ then P is a direct β-reduct of $(\lambda x.M)N$, absurd.
   - Let $r = \beta\eta$. The proof is by case on $Q = (\lambda x.M)N \to_{\beta\eta} P$.
     - If $(\lambda x.M)N \to_\beta M[x := N]$, then we are done.
     - If $\lambda x.M \to_{\beta\eta} R$ and $P = RN$ then:
       - * Either $R = \lambda x.M'$ such that $M \to_{\beta\eta} M'$. So P is a direct $\beta\eta$-reduct of $(\lambda x.M)N$, absurd.
       - * Or $M = Rx$ and $x \notin FV(R)$. Hence, $P = RN = M[x := N]$ and we are done.
     - If $N \to_{\beta\eta} N'$ and $P = (\lambda x.M)N'$ then P is a direct $\beta\eta$-reduct of $(\lambda x.M)N$, absurd.

   (c) We prove the statement by induction on $k \geq 1$.

   - If $k = 1$ then it is done since by (b) $P = M[x := N]$.
   - Else, let $k \geq 1$ and $Q = (\lambda x.M)N \to^k_r R \to_r P$.
     - If R is a direct r-reduct of Q, then $R = (\lambda x.M')N'$, such that $M \to^*_r M'$ and $N \to^*_r N'$. Since P is not a direct r-reduct of Q, P is not a direct r-reduct of R. Hence by (b), $P = M'[x := N']$.
     - Else, by IH, there exists a direct r-reduct $(\lambda x.M')N'$ of Q such that $M'[x := N'] \to^*_r R \to_r P$.

7. If P is a direct r-reduct of $(\lambda x.M)N$ then $P = (\lambda x.M')N'$ such that $M \rightarrow_r^* M'$ and $N \rightarrow_r^* N'$. So $P \rightarrow_r M'[x := N']$ and $M[x := N] \rightarrow_r^* M'[x := N']$, by Lemma 4.1.2.1. If P is not a direct r-reduct of $(\lambda x.M)N$ then by Lemma 4.1.2.6, there exists a direct r-reduct, $(\lambda x.M')N'$ of $(\lambda x.M)N$ such that $M \rightarrow_r^* M'$, $N \rightarrow_r^* N'$ and $M'[x := N'] \rightarrow_r^* P$. Finally, by Lemma 4.1.2.1, $M[x := N] \rightarrow_r^* M'[x := N'] \rightarrow_r^* P$, .

8.a) Let $n \geq 0$, $M[x := N] \in \mathsf{CR}^r$, $(\lambda x.M)N \rightarrow_r^* M_1$ and $(\lambda x.M)N \rightarrow_r^* M_2$. By Lemma 4.1.2.7, there exist M_1' and M_2' such that $M_1 \rightarrow_r^* M_1'$, $M[x := N] \rightarrow_r^* M_1'$, $M_2 \rightarrow_r^* M_2'$ and $M[x := N] \rightarrow_r^* M_2'$. Then we conclude using $M[x := N] \in \mathsf{CR}^r$.

8.b) Let $n \geq 0$ and for all $i \in \{1, \dots, n\}$, $M_i \in \mathsf{CR}^r$. First we prove that if $xM_1 \cdots M_n \rightarrow_r^* N$ then $N = xM_1' \cdots M_n'$ such that for all $i \in \{1, \dots, n\}$, $M_i \rightarrow_r^* M_i'$. We prove the result by induction on the length of the reduction $xM_1 \cdots M_n \rightarrow_r^* N$.

- Let $xM_1 \cdots M_n = N$ then it is done
- Let $xM_1 \cdots M_n \rightarrow_r^* N' \rightarrow_r N$. By IH, $N' = xM_1' \cdots M_n'$ such that for all $i \in \{1, \dots, n\}$, $M_i \rightarrow_r^* M_i'$. We prove the result by induction on n.
  - Let $n = 0$ then it is done since x does not reduce by $\rightarrow_r$.
  - Let $n = m + 1$ such that $m \geq 0$. By compatibility:
    - Either $N = PM_n'$ such that $xM_1' \cdots M_m' \rightarrow_r P$ Then by IH $P = xM_1'' \cdots M_m''$ such that for all $i \in \{1, \dots, m\}$, $M_i' \rightarrow_r^* M_i''$. So it is done.
    - Or $N = xM_1' \cdots M_m' M_n''$ such that $M_n' \rightarrow_r M_n''$ then it is done.

8.c) Case β: Let $\lambda x.M \rightarrow_\beta^* P_1$ and $\lambda x.M \rightarrow_\beta^* P_2$ then $P_1 = \lambda x.M_1$ and $P_2 = \lambda x.M_2$ such that $M \rightarrow_\beta^* M_1$ and $M \rightarrow_\beta^* M_2$. By hypothesis, there exists M_3 such that $M_1 \rightarrow_\beta^* M_3$ and $M_2 \rightarrow_\beta^* M_3$. So $P_1 \rightarrow_\beta^* \lambda x.M_3$ and $P_2 \rightarrow_\beta^* \lambda x.M_3$.

Case $\beta\eta$: Let $\lambda x.M \rightarrow_{\beta\eta}^* P_1$ and $\lambda x.M \rightarrow_{\beta\eta}^* P_2$. By Lemma 4.1.2.4:

- Either $P_1 = \lambda x.Q_1$ such that $M \rightarrow_{\beta\eta}^* Q_1$ and $P_2 = \lambda x.Q_2$ such that $M \rightarrow_{\beta\eta}^* Q_2$. So by hypothesis there exists Q_3 such that $Q_1 \rightarrow_{\beta\eta}^* Q_3$ and $Q_2 \rightarrow_{\beta\eta}^* Q_3$, hence, $P_1 \rightarrow_{\beta\eta}^* \lambda x.Q_3$ and $P_2 \rightarrow_{\beta\eta}^* \lambda x.Q_3$.
- Or $P_1 = \lambda x.Q_1$ such that $M \rightarrow_{\beta\eta}^* Q_1$ and $M \rightarrow_{\beta\eta}^* P_2 x$ such that $x \notin \mathsf{fv}(P_2)$. By hypothesis there exists Q_3 such that $Q_1 \rightarrow_{\beta\eta}^* Q_3$ and $P_2 x \rightarrow_{\beta\eta}^* Q_3$. So, by Lemma 4.1.2.5 $P_2 \rightarrow_{\beta\eta}^* Q_2$ (by Lemma 4.1.2.3, $x \notin \mathsf{fv}(Q_2)$) and:
  - Either $Q_3 = Q_2 x$. So $P_1 = \lambda x.Q_1 \rightarrow_{\beta\eta}^* \lambda x.Q_3 = \lambda x.Q_2 x \rightarrow_\eta Q_2$.
  - Or $Q_2 = \lambda x.Q_3$. So it is done since $P_1 = \lambda x.Q_1 \rightarrow_{\beta\eta}^* \lambda x.Q_3$.

- Or $M \to_{\beta\eta}^* P_1x$ such that $x \notin \mathsf{fv}(P_1)$ and $P_2 = \lambda x.Q_2$ such that $M \to_{\beta\eta}^* Q_2$. This case is similar to the previous one.
- Or $M \to_{\beta\eta}^* P_1x$ such that $x \notin \mathsf{fv}(P_1)$ and $M \to_{\beta\eta}^* P_2x$ such that $x \notin \mathsf{fv}(P_2)$. By hypothesis, there exists Q_3 such that $P_1x \to_{\beta\eta}^* Q_3$ and $P_2x \to_{\beta\eta}^* Q_3$. By Lemma 4.1.2.5, $P_1 \to_{\beta\eta}^* Q_1$ and $P_2 \to_{\beta\eta}^* Q_2$. By Lemma 4.1.2.3, $x \notin \mathsf{fv}(Q_1) \cup \mathsf{fv}(Q_2)$. Therefore:
  - Either $Q_3 = Q_1x$ and $Q_3 = Q_2x$ so $Q_1 = Q_2$.
  - Or $Q_3 = Q_1x$ and $Q_2 = \lambda x.Q_3$ so $Q_2 \to_\eta Q_1$.
  - Or $Q_1 = \lambda x.Q_3$ and $Q_3 = Q_2x$ so $Q_1 \to_\eta Q_2$.
  - Or $Q_1 = \lambda x.Q_3$ and $Q_2 = \lambda x.Q_3$ so $Q_1 = Q_2$.

□

A.1.2 Pseudo Development Definitions (Sec 4.2)

Proof of Lemma 4.2.7. 1 By induction on the structure of M.

- Let $M = x$ then $\Psi_c(M) = M$.
- Let $M = \lambda x.N$. Let $x \neq c$. By IH, $\Psi_c(N) \to_c^* N$. Then, $\Psi_c(M) = \lambda x.\Psi_c(N) \to_c^* \lambda x.N = M$.
- Let $M = M_1M_2$. By IH, for $i \in \{1,2\}$, $\Psi_c(M_i) \to_c^* M_i$.
  - If M_1 is a λ-abstraction, then $\Psi_c(M) = \Psi_c(M_1)\Psi_c(M_2) \to_c^* M_1M_2 = M$.
  - Else $\Psi_c(M) = c\Psi_c(M_1)\Psi_c(M_2) \to_c \Psi_c(M_1)\Psi_c(M_2) \to_c^* M_1M_2 = M$.

2 By induction on the length of the reduction $M \to_c^* N$. The basic case ($M = N$) is trivial. Let us prove the induction case. Let $M \to_c M' \to_c^* N$. By IH, $\mathsf{fv}(M') \setminus \{c\} = \mathsf{fv}(N) \setminus \{c\}$. We prove that $\mathsf{fv}(M) \setminus \{c\} = \mathsf{fv}(M') \setminus \{c\}$ by induction on the size of the derivation of $M \to_c M'$ and then by case on the last rule of the derivation.

- Let $M = cM' \to_c M'$ then it is done.
- Let $M = \lambda x.P \to_c \lambda x.P' = M'$ such that $P \to_c P'$ then it is done by IH.
- Let $M = PQ \to_c P'Q = M'$ such that $P \to_c P'$ then it is done by IH.
- Let $M = PQ \to_c PQ' = M'$ such that $Q \to_c Q'$ then it is done by IH.

3 Corollary of Lemma 4.2.7.1 and Lemma 4.2.7.2.

4 Let $M \in \Lambda_c^{\beta\eta}$. We prove by induction on the structure of M that $M \notin \mathsf{A}_c$.

- Let $M \in \mathsf{Var}_c$ then $M \notin \mathsf{A}_c$.

- Let $M = \lambda x.M_1$ then $M \notin \mathsf{A}_c$.
- Let $M = (\lambda x.M_1)M_2$ then because $\lambda x.M_1 \notin \mathsf{A}_c$ then $M \notin \mathsf{A}_c$.
- Let $M = cM_1M_2$. By IH, $M_2 \notin \mathsf{A}_c$ so $M \notin \mathsf{A}_c$.
- Let $M = cM_1$. By IH, $M_1 \notin \mathsf{A}_c$ so $M \notin \mathsf{A}_c$.

5 We prove this lemma by induction on the structure of d.

- Let $d = c$ then $cM \rightarrow_c M$.
- Let $d = d_1d_2$ then by IH, $d = d_1d_2 \rightarrow_c^* d_2$ and again by IH, $d_2M \rightarrow_c^* M$, so by compatibility $dM \rightarrow_c^* M$.

6 $\Rightarrow$) We prove this lemma by induction on the length of the reduction $M \rightarrow_c^* c$.

- Let $M = c$ then it is done.
- Let $M \rightarrow_c^* M' \rightarrow_c c$. We prove the lemma by induction on the length of the derivation of $M' \rightarrow_c c$ and then by case on the last rule.
    - Let $M' = cc \rightarrow_c c$ then $M' \in \mathsf{A}_c$ and by IH, $M \in \mathsf{A}_c$.
    - Let $M' = \lambda x.M_1 \rightarrow_c \lambda x.M_2 = c$ such that $M_1 \rightarrow_c M_2$, then it is done because by case on c, $c \neq \lambda x.M_2$.
    - Let $M' = M_1M_2 \rightarrow_c M_1'M_2 = c$ such that $M_1 \rightarrow_c M_1'$. By case on d, $M_1', M_2 \in \mathsf{A}_c$, so by IH, $M_1 \in \mathsf{A}_c$. Hence, $M' \in \mathsf{A}_c$ and by IH, $M \in \mathsf{A}_c$.
    - Let $M' = M_1M_2 \rightarrow_c M_1M_2' = c$ such that $M_2 \rightarrow_c M_2'$. By case on d, $M_1, M_2' \in \mathsf{A}_c$ so by IH, $M_2 \in \mathsf{A}_c$. Hence $M' \in \mathsf{A}_c$ and by IH, $M \in \mathsf{A}_c$.

$\Leftarrow$) We prove this lemma by induction on the reduction $c \rightarrow_c^* N$.

- Let $c = N$ then it is done.
- Let $c \rightarrow_c^* N' \rightarrow_c N$. By IH, $N' \in \mathsf{A}_c$. We prove that $N \in \mathsf{A}_c$ by induction on the size of the derivation of $N' \rightarrow_c N$ and then by case on the last rule.
    - Let $N' = cN \rightarrow_c N$ then $N \in \mathsf{A}_c$.
    - Let $N' = \lambda x.P \rightarrow_c \lambda x.P' = N$ such that $P \rightarrow_c P'$ then it is done because by case on N', $N' \neq \lambda x.P$.
    - Let $N' = PQ \rightarrow_c P'Q = N$ such that $P \rightarrow_c P'$. Then $P, Q \in \mathsf{A}_c$, by IH $P' \in \mathsf{A}_c$, so $N \in \mathsf{A}_c$.
    - Let $N' = PQ \rightarrow_c PQ' = N$ such that $Q \rightarrow_c Q'$. Then $P, Q \in \mathsf{A}_c$, by IH $Q' \in \mathsf{A}_c$, so $N \in \mathsf{A}_c$.

7 We prove this lemma by induction on the length of the reduction $M \rightarrow_c^* N$. The basic case is trivial. Let us prove the induction case. Let $M \rightarrow_c M' \rightarrow_c^* N$. We prove the lemma by induction on the structure of M.

- Let $M = x$ then it is done since $M \to_c M'$ is wrong.
- Let $M = \lambda x.M_1$ then by compatibility $M' = \lambda x.M_1'$ such that $M_1 \to_c M_1'$. By IH, $N = \lambda x.N_1$ such that $M_1' \to_c^* N_1$. Hence, $M_1 \to_c^* N_1$.
- Let $M = M_1M_2$. By compatibility:
  - Either $M' = M_1'M_2$ such that $M_1 \to_c M_1'$. By IH, either $M_1' \in \mathsf{A}_c$ and $M_2 \to_c^* N$ or $N = N_1N_2$ and $M_1' \to_c^* N_1$ and $M_2 \to_c^* N_2$. In the first case, by Lemma 4.2.7.6, $M_1 \in \mathsf{A}_c$. In the second case, $M_1 \to_c^* N_1$.
  - Or $M' = M_1M_2'$ such that $M_2 \to_c^* M_2'$. By IH, either $M_1 \in \mathsf{A}_c$ and $M_2' \to_c^* N$ or $N = N_1N_2$ and $M_1 \to_c^* N_1$ and $M_2' \to_c^* N_2$. In the first case, $M_2 \to_c^* N$. In the second case, $M_2 \to_c^* N_2$.
  - Or $M_1 = c$ and $M = cM_2 \to_c M_2 = M'$. Then it is done because $M = cM_2 \to_c M_2 = M' \to_c^* N$.

8 We prove this lemma by induction on the structure of M.

- Let $M = y$. By Lemma 4.2.7.7, $M' = y$. If $y = x$ then $M[x := N] = N \to_c^* N' = M'[x := N']$. Else $y \neq x$ and $M[x := N] = M = M' = M'[x := N']$.
- Let $M = \lambda y.M_1$. Let $y \notin \mathsf{fv}(N) \cup \mathsf{fv}(N') \cup \{x\}$. Then by Lemma 4.2.7.7, $M' = \lambda y.M_1'$ such that $M_1 \to_c^* M_1'$. Hence, by IH, $M[x := N] = \lambda y.M_1[x := N] \to_c^* \lambda y.M_1'[x := N'] = M'[x := N']$.
- Let $M = M_1M_2$. By Lemma 4.2.7.7, either $M_1 \in \mathsf{A}_c$ and $M_2 \to_c^* M'$ or $M' = M_1'M_2'$ and $M_1 \to_c^* M_1'$ and $M_2 \to_c^* M_2'$.
  - If $M_1 \in \mathsf{A}_c$ and $M_2 \to_c^* M'$ then by IH and Lemma 4.2.7.5, $M[x := N] = (M_1M_2)[x := N] = M_1(M_2[x := N]) \to_c^* M_2[x := N] \to_c^* M'[x := N']$.
  - If $M' = M_1'M_2'$ and $M_1 \to_c^* M_1'$ and $M_2 \to_c^* M_2'$ then by IH, $M[x := N] = (M_1M_2)[x := N] = M_1[x := N]M_2[x := N] \to_c^* M_1'[x := N']M_2'[x := N'] = M'[x := N']$.

9 We prove this lemma by induction on the length of the reduction $M \to_c^* N$. The basic case is trivial. Let $M \to_c M' \to_c^* N$. We prove that $M \to_c M'$ is false by first proving that if $M \to_c M'$ then $c \in \mathsf{fv}(M)$ by induction on the size of the derivation $M \to_c M'$ and then by case on the last rule of the derivation:

- Let $M = cM' \to_c M'$ then $c \in \mathsf{fv}(M)$.
- Let $M = \lambda x.M_1 \to_c \lambda x.M_1' = M'$ such that $M_1 \to_c M_1'$. Let $x \neq c$. By IH, $c \in \mathsf{fv}(M_1)$, hence $c \in \mathsf{fv}(M)$.
- Let $M = M_1M_1 \to_c M_1'M_2 = M'$ such that $M_1 \to_c M_1'$. By IH, $c \in \mathsf{fv}(M_1) \subseteq \mathsf{fv}(M)$.

- Let $M = M_1M_2 \to_c M_1M_2'$ such that $M_2 \to_c M_2'$. By IH, $c \in \mathsf{fv}(M_2) \subseteq \mathsf{fv}(M)$.

10 We prove this lemma by induction on the structure of M.

- Let $M = x$ then by Lemma 4.2.7.7 it is done because $M = P = N$.
- Let $M = \lambda x.M'$. Let $x \neq c$. By Lemma 4.2.7.7, $N = \lambda x.N'$ and $P = \lambda x.P'$ such that $M' \to_c^* N'$ and $M' \to_c^* N'$. By IH, $P' \to_c^* N'$, hence $P \to_c^* N$.
- Let $M = M_1M_2$. By Lemma 4.2.7.7:
  - Either $M_2 \to_c^* P$, $M_2 \to_c^* N$ and $M_1 \in \mathsf{A}_c$. By IH, $P \to_c^* N$.
  - Or $M_2 \to_c^* P$, $M_1 \in \mathsf{A}_c$, $N = N_1N_2$, $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. By Lemma 4.2.7.6, $N_1 \in \mathsf{A}_c$, so $c \in \mathsf{fv}(N_1) \subseteq \mathsf{fv}(N)$. We get a contradiction.
  - Or $P = P_1P_2$, $M_1 \to_c^* P_1$, $M_2 \to_c^* P_2$, $M_1 \in \mathsf{A}_c$ and $M_2 \to_c^* N$. By IH, $P_2 \to_c^* N$. By Lemma 4.2.7.6, $P_1 \in \mathsf{A}_c$. By Lemma 4.2.7.5, $P \to_c^* P_2 \to_c^* N$.
  - Or $P = P_1P_2$, $N = N_1N_2$, $M_1 \to_c^* P_1$, $M_1 \to_c^* N_1$, $M_2 \to_c^* P_2$, $M_2 \to_c^* N_2$. By IH, $P_1 \to_c^* N_1$ and $P_2 \to_c^* N_2$. Hence, $P \to_c^* N$.

□

A.1.3 A simple Church-Rosser proof for β-reduction (Sec. 4.3)

Proof of Lemma 4.3.1. We prove the result by induction on the structure of M:

- Let $M = x \in \mathsf{Var}_c$ and $M \in s$ then $x[x := M] = M \in s$.

- Let $M = \lambda x.N$. Let $\mathsf{fv}(N) \setminus \{c, x\} = \{x_1, \ldots, x_n\}$ and $M_1, \ldots, M_n \in s$. Let $x \notin \mathsf{fv}(M_1) \cup \cdots \cup \mathsf{fv}(M_n)$. Because $s \in \mathsf{VAR}$ then $x \in s$. By IH, $N[x_1 := M_1, \ldots, x_n := M_n] \in s$. Because $s \in \mathsf{ABS}$ then $(\lambda x.N)[x_1 := M_1, \ldots, x_n := M_n]s \in s$.

- Let $M = cPQ$. Let $\mathsf{fv}(P) \setminus \{c\} = \{x_1, \ldots, x_n\} \uplus \{x_1', \ldots, x_{n_1}'\}$, $\mathsf{fv}(Q) \setminus \{c\} = \{x_1, \ldots, x_n\} \uplus \{x_1'', \ldots, x_{n_2}''\}$, $\mathsf{dj}(\{x_1', \ldots, x_{n_1}'\}, \{x_1'', \ldots, x_{n_2}''\})$ and $M_1, \ldots, M_n, M_1', \ldots, M_{n_1}', M_1'', \ldots, M_{n_2}'' \in s$. By IH, $P[x_1 := M_1, \ldots, x_n := M_n, x_1' := M_1', \ldots, x_{n_1}' := M_{n_1}']$, $Q[x_1 := M_1, \ldots, x_n := M_n, x_1'' := M_1'', \ldots, x_{n_2}'' := M_{n_2}''] \in s$. Because $s \in \mathsf{VAR}$ then $(cPQ)[x_1 := M_1, \ldots, x_n := M_n, x_1' := M_1', \ldots, x_{n_1}' := M_{n_1}', x_1'' := M_1'', \ldots, x_{n_2}'' := M_{n_2}''] \in s$.

- Let $M = (\lambda x.P)Q$. Let $\mathsf{fv}(P) \setminus \{c, x\} = \{x_1, \ldots, x_n\} \uplus \{x_1', \ldots, x_{n_1}'\}$, $\mathsf{fv}(Q) \setminus \{c\} = \{x_1, \ldots, x_n\} \uplus \{x_1'', \ldots, x_{n_2}''\}$ and $M_1, \ldots, M_n, M_1', \ldots, M_{n_1}', M_1'', \ldots, M_{n_2}'' \in s$ and $\mathsf{dj}(\{x_1', \ldots, x_{n_1}'\}, \{x_1'', \ldots, x_{n_2}''\})$. Let $x \notin \mathsf{fv}(M_1) \cup \cdots \cup \mathsf{fv}(M_n) \cup \mathsf{fv}(M_1') \cup$

$\cdots \cup \mathsf{fv}(M'_{n_1}) \cup \mathsf{fv}(M''_1) \cup \cdots \cup \mathsf{fv}(M''_{n_2})$. By IH, $Q' = Q[x_1 := M_1, \ldots, x_n := M_n, x'_1 := M'_1, \ldots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \ldots, x''_{n_2} := M''_{n2}] \in s$. By IH, $P[x_1 := M_1, \ldots, x_n := M_n, x'_1 := M'_1, \ldots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \ldots, x''_{n_2} := M''_{n_2}, x := Q'] \in s$. Because $s \in \mathsf{SAT}$, $((\lambda x.P)Q)[x_1 := M_1, \ldots, x_n := M_n, x'_1 := M'_1, \ldots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \ldots, x''_{n_2} := M''_{n_2}] \in s$. □

Proof of Lemma 4.3.3. By induction on the structure of M.

- Let $M \in \mathsf{Var}$, so $\Psi_c(M) = M \in \mathsf{Var}_c$, since $M \neq c$.
- Let $M = \lambda x.N$. Let $x \neq c$. By IH, $\Psi_c(N) \in \Lambda^\beta_c$, so $\Psi_c(M) = \lambda x.\Psi_c(N) \in \Lambda^\beta_c$.
- Let $M = PQ$.
  - If $P = \lambda x.N$ such that $x \neq c$ then $\Psi_c(M) = (\lambda x.\Psi_c(N))\Psi_c(Q)$. By IH, $\Psi_c(N), \Psi_c(Q) \in \Lambda^\beta_c$, so $\Psi_c(M) \in \Lambda^\beta_c$.
  - Else $\Psi_c(M) = c\Psi_c(P)\Psi_c(Q)$. By IH, $\Psi_c(P), \Psi_c(Q) \in \Lambda^\beta_c$, so $\Psi_c(M) \in \Lambda^\beta_c$.

□

Proof of Lemma 4.3.4.

1 By induction on the structure of M.

- Let $M \in \mathsf{Var}_c$. Either $M = x$, then $M[x := N] = N \in \Lambda^\beta_c$. Or, $M \neq x$ and so $M[x := N] = M \in \Lambda^\beta_c$.
- Let $M = \lambda y.P$ such that $y \in \mathsf{Var}_c$ and $P \in \Lambda^\beta_c$. By IH, $P[x := N] \in \Lambda^\beta_c$. Then, $M[x := N] = \lambda y.P[x := N] \in \Lambda^\beta_c$ such that $y \notin \mathsf{fv}(N) \cup \{x\}$.
- Let $M = (\lambda y.P)Q$ such that $y \in \mathsf{Var}_c$ and $P, Q \in \Lambda^\beta_c$. By IH, $P[x := N], Q[x := N] \in \Lambda^\beta_c$. Then, $M[x := N] = (\lambda y.P[x := N])Q[x := N] \in \Lambda^\beta_c$ such that $y \notin \mathsf{fv}(N) \cup \{x\}$.
- Let $M = cPQ$ such that $P, Q \in \Lambda^\beta_c$. By IH, $P[x := N], Q[x := N] \in \Lambda^\beta_c$. Then, $M[x := N] = cP[x := N]Q[x := N] \in \Lambda^\beta_c$.

2 We prove the lemma by induction on the length of the derivation $M \to^*_\beta N$.

- let $M = N$ then it is done.
- Let $M \to^*_\beta M' \to_\beta N$. By IH, $M' \in \Lambda^\beta_c$. We prove that $N \in \Lambda^\beta_c$ by induction on the structure of M'.
  - Let $M' \in \mathsf{Var}_c$ then it is done because M' does not reduce.
  - Let $M' = \lambda x.P$ such that $x \in \mathsf{Var}_c$ and $P \in \Lambda^\beta_c$, so by compatibility $N = \lambda x.P'$ such that $P \to_\beta P'$. By IH, $P' \in \Lambda^\beta_c$ so $N \in \Lambda^\beta_c$.
  - Let $M' = (\lambda x.P)Q$ such that $x \in \mathsf{Var}_c$ and $P, Q \in \Lambda^\beta_c$. By compatibility:

        - Either $N = (\lambda x.P')Q$ such that $P \to_\beta P'$. By IH, $P' \in \Lambda_c^\beta$ so $N \in \Lambda_c^\beta$.
        - Or $N = (\lambda x.P)Q'$ such that $Q \to_\beta Q'$. By IH, $Q' \in \Lambda_c^\beta$ so $N \in \Lambda_c^\beta$.
        - Or $N = P[x := Q]$, so by Lemma 4.3.4.1, $N \in \Lambda_c^\beta$
    - Let $M' = cPQ$ such that $P, Q \in \Lambda_c^\beta$. By compatibility:
        - Either $N = cP'Q$ such that $P \to_\beta P'$. By IH, $P' \in \Lambda_c^\beta$ so $N \in \Lambda_c^\beta$.
        - Or $N = cPQ'$ such that $Q \to_\beta Q'$. By IH, $Q' \in \Lambda_c^\beta$ so $N \in \Lambda_c^\beta$.

3 We prove this lemma by induction on the structure of M.

- Let $M \in \mathsf{Var}_c$ then it is done because by Lemma 4.2.7.7, $N = M$ and $\Psi_c(N) = M$.
- Let $M = \lambda x.M'$. By Lemma 4.2.7.7, $N = \lambda x.N'$ such that $M' \to_c^* N'$. By IH, $M' \to_c^* \Psi_c(N')$. Hence, $M \to_c^* \lambda x.\Psi_c(N') = N$.
- Let $M = (\lambda x.M_1)M_2$. By Lemma 4.2.7.7, $N = (\lambda x.N_1)N_2$ such that $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. By IH, $M_1 \to_c^* \Psi_c(N_1)$ and $M_2\Psi_c(N_2)$, so $M \to_c^* (\lambda x.\Psi_c(N_1))\Psi_c(N_2) = \Psi_c(N)$.
- Let $M = cM_1M_2$. By Lemma 4.2.7.7 and Lemma 4.2.7.4:
    - Either $N = N_1N_2$ such that $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. By IH, $M_1 \to_c^* \Psi_c(N_1)$ and $M_2 \to_c^* \Psi_c(N_2)$. If N_1 is a λ-abstraction then $M \to_c^* c\Psi_c(N_1)\Psi_c(N_2) \to_c \Psi_c(N_1)\Psi_c(N_2) = \Psi_c(N)$ else $M \to_c^* c\Psi_c(N_1)\Psi_c(N_2) = \Psi_c(N)$.
    - Or $N = cN_1N_2$ such that $M_2 \to_c^* N_1$ and $M_2 \to_c^* N_2$. We obtain a contradiction because by IH, $c \notin \mathsf{fv}(N)$.

4 We prove this lemma by induction on the structure of M.

- Let $M \in \mathsf{Var}_c$ then it is done with $N = M$.
- Let $M = \lambda x.M'$. By IH there exists N' such that $c \notin \mathsf{fv}(N')$ and $M' \to_c^* N'$. So, $M \to_c^* \lambda x.N' = N$ and $c \notin \mathsf{fv}(N)$.
- Let $M = (\lambda x.M_1)M_2$. By IH, there exists N_1, N_2 such that $c \notin \mathsf{fv}(N_1) \cup \mathsf{fv}(N_2)$, $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. So, $M \to_c^* (\lambda x.N_1)N_2 = N$ and $c \notin \mathsf{fv}(N)$.
- Let $M = cM_1M_2$. By IH, there exists N_1, N_2 such that $c \notin \mathsf{fv}(N_1) \cup \mathsf{fv}(N_2)$, $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. So, $M \to_c^* cN_1N_2 \to_c N_1N_2 = N$ and $c \notin \mathsf{fv}(N)$.

□

Proof of Lemma 4.3.5.

1 By induction on the structure of M_1.

- Let $M_1 \in \mathsf{Var}_c$ then it is done because M_1 does not reduce.
- Let $M_1 = \lambda x.P_1$ such that $P_1 \in \Lambda_c^\beta$ and $x \in \mathsf{Var}_c$, then by Lemma 4.2.7.7, $M_2 = \lambda x.P_2$ such that $P_1 \rightarrow_c^* P_2$ and by compatibility $N_1 = \lambda x.Q_1$ such that $P_1 \rightarrow_\beta Q_1$. By IH, there exists Q_2 such that $P_2 \rightarrow_\beta Q_2$ and $Q_1 \rightarrow_c^* Q_2$. So it is done with $N_2 = \lambda x.Q_2$.
- let $M_1 = (\lambda x.P_1)Q_1$ such that $P_1, Q_1 \in \Lambda_c^\beta$ and $x \in \mathsf{Var}_c$ then by Lemma 4.2.7.7, $M_2 = (\lambda x.P_2)Q_2$ such that $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By compatibility:
  - Either $N_1 = (\lambda x.P_1')Q_1$ such that $P_1 \rightarrow_\beta P_1'$. By IH, there exist P_2' such that $P_2 \rightarrow_\beta P_2'$ and $P_1' \rightarrow_c^* P_2'$. So it is done with $N_2 = (\lambda x.P_2')Q_2$.
  - Or $N_1 = (\lambda x.P_1)Q_1'$ such that $Q_1 \rightarrow_\beta Q_1'$. By IH, there exists Q_2' such that $Q_2 \rightarrow_\beta Q_2'$ and $Q_1' \rightarrow_c^* Q_2'$. So it is done with $N_2 = (\lambda x.P_2)Q_2'$.
  - Or $N_1 = P_1[x := Q_1]$. By Lemma 4.2.7.8, it is done with $N_2 = P_2[x := Q_2]$.
- Let $M_1 = cP_1Q_1$ such that $P_1, Q_1 \in \Lambda_c^\beta$. By Lemmas 4.2.7.7 and 4.2.7.4:
  - Either $M_2 = cP_2Q_2$ such that $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By compatibility:
    * Either $N_1 = cP_1'Q_1$ such that $P_1 \rightarrow_\beta P_1'$. By IH, there exists P_2' such that $P_2 \rightarrow_\beta P_2'$ and $P_1' \rightarrow_c^* P_2'$. So it is done with $N_2 = cP_2'Q_2$.
    * Or $N_1 = cP_1Q_1'$ such that $Q_1 \rightarrow_\beta Q_1'$. By IH, there exists Q_2' such that $Q_2 \rightarrow_\beta Q_2'$ and $Q_1' \rightarrow_c^* Q_2'$. So it is done with $N_2 = cP_2Q_2'$.
  - Or $M_2 = P_2Q_2$ such that $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By compatibility:
    * Either $N_1 = cP_1'Q_1$ such that $P_1 \rightarrow_\beta P_1'$. By IH, there exists P_2' such that $P_2 \rightarrow_\beta P_2'$ and $P_1' \rightarrow_c^* P_2'$. So it is done with $N_2 = P_2'Q_2$.
    * Or $N_1 = cP_1Q_1'$ such that $Q_1 \rightarrow_\beta Q_1'$. By IH, there exists Q_2' such that $Q_2 \rightarrow_\beta Q_2'$ and $Q_1' \rightarrow_c^* Q_2'$. So it is done with $N_2 = P_2Q_2'$.

2 By induction on the length of the reduction $M_1 \rightarrow_\beta^* N_1$ using Lemma 4.3.5.1.

□

Proof of Lemma 4.3.6.

⇒) Let $M \rightarrow_\beta^* N$. Let c be a variable such that $c \notin \mathsf{fv}(M)$. By Lemma 4.1.2.3, $c \notin \mathsf{fv}(N)$. We prove that $M \rightarrow_1^* N$ by induction on the size of the reduction $M \rightarrow_\beta^* N$.

- If $M = N$, then it is done since $M \to_1^* N$.
- If $M \to_\beta^* M' \to_\beta N$. By Lemma 4.1.2.3, $c \notin \mathsf{fv}(M')$. By IH, $M \to_1^* M'$. We prove that $M' \to_1 N$ by induction on the structure of M'.
  - Let $M' \in \mathsf{Var}$ then it is done because M' does not reduce.
  - Let $M' = \lambda x.P$ such that $x \neq c$, then by compatibility $N = \lambda x.P'$ and $P \to_\beta P'$. By IH, $P \to_1 P'$. By definition, $\Psi_c(P) \to_\beta^* Q$ and $Q \to_c^* P'$. So $\Psi_c(\lambda x.P) = \lambda x.\Psi_c(P) \to_\beta^* \lambda x.Q$ and $\lambda x.Q \to_c^* \lambda x.P' = N$. Hence, $M' \to_1 N$.
  - Let $M' = PQ$.

    (a) If $P = \lambda x.P_1$ such that $x \neq c$ then by compatibility:

    * Either $N = (\lambda x.P_2)Q$ such that $P_1 \to_\beta P_2$. By IH, $P_1 \to_1 P_2$. By definition, $\Psi_c(P_1) \to_\beta^* P_1'$ and $P_1' \to_c^* P_2$. So, $\Psi_c(M') = (\lambda x.\Psi_c(P_1))\Psi_c(Q) \to_\beta^* (\lambda x.P_1')\Psi_c(Q)$ and by Lemma 4.2.7.1, $(\lambda x.P_1')\Psi_c(Q) \to_c^* (\lambda x.P_2)Q = N$. Hence, $M' \to_1 N$.
    * Or $N = (\lambda x.P_1)Q_1$ such that $Q \to_\beta Q_1$. By IH, $Q \to_1 Q_1$. By definition, $\Psi_c(Q) \to_\beta^* Q_2$ and $Q_2 \to_c^* Q_1$. So, $\Psi_c(M') = (\lambda x.\Psi_c(P_1))\Psi_c(Q) \to_\beta^* (\lambda x.\Psi_c(P_1))Q_2$ and by Lemma 4.2.7.1, $(\lambda x.\Psi_c(P_1))Q_2 \to_c^* (\lambda x.P_1)Q_1 = N$. Hence, $M' \to_1 N$.
    * Or $N = P_1[x := Q]$. So, $\Psi_c(M') = (\lambda x.\Psi_c(P_1))\Psi_c(Q) \to_\beta \Psi_c(P_1)[x := \Psi_c(Q)]$ and by Lemma 4.2.7.1 and Lemma 4.2.7.8 $\Psi_c(P_1)[x := \Psi_c(Q)] \to_c^* P_1[x := Q]$. Hence, $M' \to_1 N$.

    (b) Else, by compatibility:

    * Either $N = P'Q$ such that $P \to_\beta P'$. By IH, $P \to_1 P'$. By definition, $\Psi_c(P) \to_\beta^* P_1$ and $P_1 \to_c^* P'$. So, $\Psi_c(M') = c\Psi_c(P)\Psi_c(Q) \to_\beta^* cP_1\Psi_c(Q)$ and by Lemma 4.2.7.1 $cP_1\Psi_c(Q) \to_c^* cP'Q \to_c P'Q = N$. So $M' \to_1 N$.
    * Or $N = PQ'$ such that $Q \to_\beta Q'$. By IH, $Q \to_1 Q'$. By definition, $\Psi_c(Q) \to_\beta^* Q_1$ and $Q_1 \to_c^* Q'$. So, $\Psi_c(M') = c\Psi_c(P)\Psi_c(Q) \to_\beta^* c\Psi_c(P)Q_1$ and by Lemma 4.2.7.1 $c\Psi_c(P)Q_1 \to_c^* cPQ' \to_c PQ' = N$. So $M' \to_1 N$.

$\Leftarrow$) Let $M \to_1^* N$. We prove that $M \to_\beta^* N$ by induction on the size of the derivation $M \to_1^* N$.

- Let $M = N$, then it is done since $M \to_\beta^* N$.
- Let $M \to_1^* M' \to_1 N$. By IH, $M \to_\beta^* M'$. Because $M' \to_1 N$ then by definition there exists P such that $\Psi_c(M') \to_\beta^* P$ and $P \to_c^* N$ and $c \notin \mathsf{fv}(M') \cup \mathsf{fv}(N)$. By Lemma 4.3.3, $\Psi_c(M') \in \Lambda_c^\beta$. By Lemma 4.2.7.1, $\Psi_c(M') \to_c^* M'$. By Lemma 4.3.5.2, there exists Q such that $P \to_c^* Q$ and

$M' \to_\beta^* Q$. By Lemma 4.1.2.3, $c \notin \mathsf{fv}(Q)$. By Lemma 4.3.4.2, $P \in \Lambda_c^\beta$. By Lemma 4.2.7.10, $Q \to_c^* N$. By Lemma 4.2.7.9, $Q = N$. Hence $M' \to_\beta^* N$.

□

Proof of Lemma 4.3.7.

1 By definition, there exist P_1, P_2 such that $\Psi_c(M) \to_\beta^* P_1$, $\Psi_c(M) \to_\beta^* P_2$, $P_1 \to_c^* M_1$, $P_2 \to_c^* M_2$ and $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(M_1) \cup \mathsf{fv}(M_2)$. By Lemma 4.3.3, $\Psi_c(M) \in \Lambda_c^\beta$. So by Corollary 4.3.2, there exists P_3 such that $P_1 \to_\beta^* P_3$ and $P_2 \to_\beta^* P_3$. By Lemma 4.3.4.2, $P_1, P_2, P_3 \in \Lambda_c^\beta$. By Lemma 4.3.4.4, there exists M_3 such that $P_3 \to_c^* M_3$ and $c \notin \mathsf{fv}(M_3)$. By Lemma 4.3.4.3, $P_1 \to_c^* \Psi_c(M_1)$ and $P_2 \to_c^* \Psi_c(M_2)$. By Lemma 4.3.5.2, there exist Q_1, Q_2 such that $P_3 \to_c^* Q_1$, $P_3 \to_c^* Q_2$, $\Psi_c(M_1) \to_\beta^* Q_1$ and $\Psi_c(M_2) \to_\beta^* Q_2$. By Lemma 4.2.7.10, $Q_1 \to_c^* M_3$ and $Q_2 \to_c^* M_3$. So $M_1 \to_1 M_3$ and $M_2 \to_1 M_3$.

2 By Lemma 4.3.7.1 □

A.1.4 A simple Church-Rosser proof for $\beta\eta$-reduction (Sec. 4.4)

Proof of Lemma 4.4.1. We prove the result by induction on the structure of M:

- Let $M = x \in \mathsf{Var}_c$ and $M \in s$ then $x[x := M] = M \in s$.

- Let $M = \lambda x.N$. Let $\mathsf{fv}(N) \setminus \{c, x\} = \{x_1, \dots, x_n\}$ and $M_1, \dots, M_n \in s$. Let $x \notin \mathsf{fv}(M_1) \cup \dots \cup \mathsf{fv}(M_n)$. Because $s \in \mathsf{VAR}$ then $x \in s$. By IH, $N[x_1 := M_1, \dots, x_n := M_n] \in s$. Because $s \in \mathsf{ABS}$ then $(\lambda x.N)[x_1 := M_1, \dots, x_n := M_n] \in s$.

- Let $M = cPQ$. Let $\mathsf{fv}(P) \setminus \{c\} = \{x_1, \dots, x_n, x'_1, \dots, x'_{n_1}\}$, $\mathsf{fv}(Q) \setminus \{c\} = \{x_1, \dots, x_n, x''_1, \dots, x''_{n_2}\}$, $\{x'_1, \dots, x'_{n_1}\} \cap \{x''_1, \dots, x''_{n_2}\} = \varnothing$ and $M_1, \dots, M_n, M'_1, \dots, M'_{n_1}, M''_1, \dots, M''_{n_2} \in s$. By IH, $P[x_1 := M_1, \dots, x_n := M_n, x'_1 := M'_1, \dots, x'_{n_1} := M'_{n_1}]$, $Q[x_1 := M_1, \dots, x_n := M_n, x''_1 := M''_1, \dots, x''_{n_2} := M''_{n_2}] \in s$. Because $s \in \mathsf{VAR}$ then $(cPQ)[x_1 := M_1, \dots, x_n := M_n, x'_1 := M'_1, \dots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \dots, x''_{n_2} := M''_{n_2}] \in s$.

- Let $M = (\lambda x.P)Q$. Let $\mathsf{fv}(P) \setminus \{c, x\} = \{x_1, \dots, x_n, x'_1, \dots, x'_{n_1}\}$, $\mathsf{fv}(Q) \setminus \{c\} = \{x_1, \dots, x_n, x''_1, \dots, x''_{n_2}\}$ and $M_1, \dots, M_n, M'_1, \dots, M'_{n_1}, M''_1, \dots, M''_{n_2} \in s$ and $\{x'_1, \dots, x'_{n_1}\} \cap \{x''_1, \dots, x''_{n_2}\} = \varnothing$. Let $x \notin \mathsf{fv}(M_1) \cup \dots \cup \mathsf{fv}(M_n) \cup \mathsf{fv}(M'_1) \cup \dots \cup \mathsf{fv}(M'_{n_1}) \cup \mathsf{fv}(M''_1) \cup \dots \cup \mathsf{fv}(M''_{n_2})$. By IH, $Q' = Q[x_1 := M_1, \dots, x_n := M_n, x'_1 := M'_1, \dots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \dots, x''_{n_2} := M''_{n_2}] \in s$. By IH, $P[x_1 := M_1, \dots, x_n := M_n, x'_1 := M'_1, \dots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \dots, x''_{n_2} := M''_{n_2}, x := Q'] \in s$. Because $s \in \mathsf{SAT}$, $((\lambda x.P)Q)[x_1 := M_1, \dots, x_n := M_n, x'_1 := M'_1, \dots, x'_{n_1} := M'_{n_1}, x''_1 := M''_1, \dots, x''_{n_2} := M''_{n_2}] \in s$.

- Let $M = cP$. Let $\mathsf{fv}(P) \setminus \{c\} = \{x_1, \ldots, x_n\}$ and $M_1, \ldots, M_n \in s$. By IH, $P[x_1 := M_1, \ldots, x_n := M_n] \in s$. Because $s \in \mathsf{VAR}$ then $c(P[x_1 := M_1, \ldots, x_n := M_n]) = (cP)[x_1 := M_1, \ldots, x_n := M_n] \in s$. □

Proof of Lemma 4.4.4.

1 By induction on the structure of M.

- Let $M \in \mathsf{Var}_c$. If $M = x$ then $M[x := N] = N \in \Lambda_c^{\beta\eta}$. Else $M[x := N] = M \in \Lambda_c^{\beta\eta}$.
- Let $M = \lambda y.P$ such that $y \in \mathsf{Var}_c$ and $P \in \Lambda_c^{\beta\eta}$. Let $y \notin \mathsf{fv}(N) \cup \{x\}$. By IH, $P[x := N] \in \Lambda_c^{\beta\eta}$. Then, $M[x := N] = \lambda y.P[x := N] \in \Lambda_c^{\beta\eta}$.
- Let $M = (\lambda y.P)Q$ such that $y \in \mathsf{Var}_c$ and $P, Q \in \Lambda_c^{\beta\eta}$. By IH, $P[x := N], Q[x := N] \in \Lambda_c^{\beta\eta}$. Then, $M[x := N] = (\lambda y.P[x := N])Q[x := N] \in \Lambda_c^{\beta\eta}$, such that $y \notin \mathsf{fv}(N) \cup \{x\}$.
- Let $M = cPQ$ such that $P, Q \in \Lambda_c^{\beta\eta}$. By IH, $P[x := N], Q[x := N] \in \Lambda_c^{\beta\eta}$. Then, $M[x := N] = cP[x := N]Q[x := N] \in \Lambda_c^{\beta\eta}$.
- Let $M = cP$ such that $P \in \Lambda_c^{\beta\eta}$. By IH, $P[x := N] \in \Lambda_c^{\beta\eta}$. Then, $M[x := N] = c(P[x := N]) \in \Lambda_c^{\beta\eta}$.

2 We prove the lemma by induction on the length of the derivation $M \to_{\beta\eta}^* N$.

- Let $M = N$ then it is done.
- Let $M \to_{\beta\eta}^* M' \to_{\beta\eta} N$. By IH, $M' \in \Lambda_c^{\beta\eta}$. We prove that $N \in \Lambda_c^{\beta\eta}$ by induction on the structure of M'.
  - Let $M' \in \mathsf{Var}_c$ then it is done because M' does not reduce.
  - Let $M' = \lambda x.P$ such that $x \in \mathsf{Var}_c$ and $P \in \Lambda_c^{\beta\eta}$. By compatibility:
    - Either $N = \lambda x.P'$ such that $P \to_{\beta\eta} P'$. By IH, $P' \in \Lambda_c^{\beta\eta}$ so $N \in \Lambda_c^{\beta\eta}$.
    - Or $P = Nx$ such that $x \notin \mathsf{fv}(N)$. Because $P \in \Lambda_c^{\beta\eta}$, by case on P, either $N = cN'$ such that $N' \in \Lambda_c^{\beta\eta}$, so $N = cN' \in \Lambda_c^{\beta\eta}$. Or $N = \lambda y.N'$ such that $y \in \mathsf{Var}_c$ and $N' \in \Lambda_c^{\beta\eta}$, so $N = \lambda y.N' \in \Lambda_c^{\beta\eta}$.
  - Let $M' = (\lambda x.P)Q$ such that $x \in \mathsf{Var}_c$ and $P, Q \in \Lambda_c^{\beta\eta}$. By compatibility:
    - Either $N = (\lambda x.P')Q$ such that $P \to_{\beta\eta} P'$. By IH, $P' \in \Lambda_c^{\beta\eta}$ so $N \in \Lambda_c^{\beta\eta}$.
    - Or $N = P'Q$ and $P = P'x$ such that $x \notin \mathsf{fv}(P')$. Because $P \in \Lambda_c^{\beta\eta}$, either $P' = cP''$ such that $P'' \in \Lambda_c^{\beta\eta}$, and so we obtain $N = cP''Q \in \Lambda_c^{\beta\eta}$. Or $P' = \lambda y.P''$ such that $P'' \in \Lambda_c^{\beta\eta}$ and $y \in \mathsf{Var}_c$, and so we obtain $N = (\lambda y.P'')Q \in \Lambda_c^{\beta\eta}$.

- - - $*$ Or $N = (\lambda x.P)Q'$ such that $Q \to_{\beta\eta} Q'$. By IH, $Q' \in \Lambda_c^{\beta\eta}$ so $N \in \Lambda_c^{\beta\eta}$.
    - $*$ Or $N = P[x := Q]$. So, by Lemma 4.4.4.1, $N \in \Lambda_c^{\beta\eta}$.
  - Let $M' = cPQ$ such that $P, Q \in \Lambda_c^{\beta\eta}$. By compatibility:
    - $*$ Either $N = cP'Q$ such that $P \to_{\beta\eta} P'$. By IH, $P' \in \Lambda_c^{\beta\eta}$ so $N \in \Lambda_c^{\beta\eta}$.
    - $*$ Or $N = cPQ'$ such that $Q \to_{\beta\eta} Q'$. By IH, $Q' \in \Lambda_c^{\beta\eta}$ so $N \in \Lambda_c^{\beta\eta}$.
  - Let $M' = cP$ such that $P \in \Lambda_c^{\beta\eta}$, so by compatibility $N = cP'$ such that $P \to_{\beta\eta} P'$. By IH, $P' \in \Lambda_c^{\beta\eta}$ so $N \in \Lambda_c^{\beta\eta}$.

3 We prove this lemma by induction on the structure of M.

- Let $M \in \mathsf{Var}_c$ then it is done because by Lemma 4.2.7.7, $N = M$ and $\Psi_c(N) = M$.
- Let $M = \lambda x.M'$. By Lemma 4.2.7.7, $N = \lambda x.N'$ such that $M' \to_c^* N'$. By IH, $M' \to_c^* \Psi_c(N')$. Hence, $M \to_c^* \lambda x.\Psi_c(N') = N$.
- Let $M = (\lambda x.M_1)M_2$. By Lemma 4.2.7.7, $N = (\lambda x.N_1)N_2$ such that $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. By IH, $M_1 \to_c^* \Psi_c(N_1)$ and $M_2\Psi_c(N_2)$, so $M \to_c^* (\lambda x.\Psi_c(N_1))\Psi_c(N_2) = \Psi_c(N)$.
- Let $M = cM_1M_2$. By Lemma 4.2.7.7 and Lemma 4.2.7.4:
  - Either $N = N_1N_2$ such that $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. By IH, $M_1 \to_c^* \Psi_c(N_1)$ and $M_2 \to_c^* \Psi_c(N_2)$. If N_1 is a λ-abstraction then $M \to_c^* c\Psi_c(N_1)\Psi_c(N_2) \to_c \Psi_c(N_1)\Psi_c(N_2) = \Psi_c(N)$ else $M \to_c^* c\Psi_c(N_1)\Psi_c(N_2) = \Psi_c(N)$.
  - Or $N = cN_1N_2$ such that $M_2 \to_c^* N_1$ and $M_2 \to_c^* N_2$. We obtain a contradiction because by IH, $c \notin \mathsf{fv}(N)$.
- Let $M = cM'$. By Lemma 4.2.7.7:
  - Either $M' \to_c^* N$. By IH, $M' \to_c^* \Psi_c(N)$, so $M \to_c M' \to_c^* \Psi_c(N)$.
  - Or $N = cN'$ and $M' \to_c^* N'$. We obtain a contradiction because by IH, $c \notin \mathsf{fv}(N)$.

4 We prove this lemma by induction on the structure of M.

- Let $M \in \mathsf{Var}_c$ then it is done with $N = M$.
- Let $M = \lambda x.M'$. By IH there exists N' such that $c \notin \mathsf{fv}(N')$ and $M' \to_c^* N'$. So, $M \to_c^* \lambda x.N' = N$ and $c \notin \mathsf{fv}(N)$.
- Let $M = (\lambda x.M_1)M_2$. By IH, there exists N_1, N_2 such that $c \notin \mathsf{fv}(N_1) \cup \mathsf{fv}(N_2)$, $M_1 \to_c^* N_1$ and $M_2 \to_c^* N_2$. So, $M \to_c^* (\lambda x.N_1)N_2 = N$ and $c \notin \mathsf{fv}(N)$.

- Let $M = cM_1M_2$. By IH, there exists N_1, N_2 such that $c \notin \mathsf{fv}(N_1) \cup \mathsf{fv}(N_2)$, $M_1 \rightarrow_c^* N_1$ and $M_2 \rightarrow_c^* N_2$. So, $M \rightarrow_c^* cN_1N_2 \rightarrow_c N_1N_2 = N$ and $c \notin \mathsf{fv}(N)$.
- Let $M = cM'$. By IH, there exists N such that $c \notin \mathsf{fv}(N)$ and $M' \rightarrow_c^* N$. So, $M \rightarrow_c M' \rightarrow_c^* N$.

□

Proof of Lemma 4.4.5.

1 We prove this lemma by induction on the structure of M_1.

- Let $M_1 \in \mathsf{Var}_c$, then it is done because M_1 does not reduce.
- Let $M_1 = \lambda x.P_1$ such that $x \in \mathsf{Var}_c$ and $P_1 \in \Lambda_c^{\beta\eta}$. By Lemma 4.2.7.7, $M_2 = \lambda x.P_2$ such that $P_1 \rightarrow_c^* P_2$. By compatibility:
  - Either $N_1 = \lambda x.P_1'$ such that $P_1 \rightarrow_{\beta\eta} P_1'$. By IH, there exits P_2' such that $P_2 \rightarrow_{\beta\eta} P_2'$ and $P_1' \rightarrow_c^* P_2'$. So it is done with $N_2 = \lambda x.P_2'$.
  - Or $P_1 = N_1x$ such that $x \notin \mathsf{fv}(N_1)$. Because $P_1 \in \Lambda_c^{\beta\eta}$ then by case on P_1, $N_1 \in \Lambda_c^{\beta\eta}$ By Lemmas 4.2.7.7 and 4.2.7.4, $P_2 = N_1'x$ and $N_1 \rightarrow_c^* N_1'$. By Lemma 4.2.7.2, $x \notin \mathsf{fv}(N_1')$. So $M_2 = \lambda x.N_1'x \rightarrow_\eta N_1' = N_2$.
- Let $M_1 = (\lambda x.P_1)Q_1$ such that $x \in \mathsf{Var}_c$ and $P_1, Q_1 \in \Lambda_c^{\beta\eta}$. Therefore, by Lemma 4.2.7.7, $M_2 = (\lambda x.P_2)Q_2$ such that $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By compatibility:
  - Either, $N_1 = P_1[x := Q_1]$. We have, $M_2 \rightarrow_\beta P_2[x := Q_2] = N_2$ and by Lemma 4.2.7.8, $N_1 \rightarrow_c^* N_2$.
  - Or, $N_1 = (\lambda x.P_1')Q_1$ such that $P_1 \rightarrow_{\beta\eta} P_1'$. By IH, there exists P_2' such that $P_2 \rightarrow_{\beta\eta} P_2'$ and $P_1' \rightarrow_c^* P_2'$. So, $M_2 = (\lambda x.P_2)Q_2 \rightarrow_{\beta\eta} (\lambda x.P_2')Q_2 = N_2$ and $N_1 \rightarrow_c^* N_2$.
  - Or $P_1 = R_1x$ such that $x \notin \mathsf{fv}(R_1)$ and $N_1 = R_1Q_1$. Because $P_1 \in \Lambda_c^{\beta\eta}$ then by case on P_1, $R_1 \in \Lambda_c^{\beta\eta}$. By Lemmas 4.2.7.7 and 4.2.7.4, $P_2 = R_1'x$ and $R_1 \rightarrow_c^* R_1'$. By Lemma 4.2.7.2, $x \notin \mathsf{fv}(R_1')$. So $M_2 = (\lambda x.R_1'x)Q_2 \rightarrow_\eta R_1'Q_2 = N_2$ and $N_1 = R_1Q_1 \rightarrow_c^* N_2$.
  - Or, $N_1 = (\lambda x.P_1)Q_1'$ such that $Q_1 \rightarrow_{\beta\eta} Q_1'$. By IH, there exist Q_2' such that $Q_2 \rightarrow_{\beta\eta} Q_2'$ and $Q_1' \rightarrow_c^* Q_2'$. So, $M_2 = (\lambda x.P_2)Q_2 \rightarrow_{\beta\eta} (\lambda x.P_2)Q_2' = N_2$ and $N_1 \rightarrow_c^* N_2$.
- Let $M_1 = cP_1Q_1$ such that $P_1, Q_1 \in \Lambda_c^{\beta\eta}$. By compatibility:
  - Either $N_1 = cP_1'Q_1$ such that $P_1 \rightarrow_{\beta\eta} P_1'$. By Lemmas 4.2.7.7 and 4.2.7.4:

      - ∗ Either $M_2 = P_2Q_2$ such $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By IH, there exists P'_2 such that $P'_1 \rightarrow_c^* P'_2$ and $P_2 \rightarrow_{\beta\eta} P'_2$. So it is done with $N_2 = P'_2Q_2$.
      - ∗ Or $M_2 = cP_2Q_2$ such that $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By IH, there exists P'_2 such that $P'_1 \rightarrow_c^* P'_2$ and $P_2 \rightarrow_{\beta\eta} P'_2$. So it is done with $N_2 = cP'_2Q_2$.
    - Or $N_1 = cP_1Q'_1$ such that $Q_1 \rightarrow_{\beta\eta} Q'_1$. By Lemma 4.2.7.7 and Lemma 4.2.7.4:
      - ∗ Either $M_2 = P_2Q_2$ such $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By IH, there exists Q'_2 such that $Q'_1 \rightarrow_c^* Q'_2$ and $Q_2 \rightarrow_{\beta\eta} Q'_2$. So it is done with $N_2 = P_2Q'_2$.
      - ∗ Or $M_2 = cP_2Q_2$ such that $P_1 \rightarrow_c^* P_2$ and $Q_1 \rightarrow_c^* Q_2$. By IH, there exists Q'_2 such that $Q'_1 \rightarrow_c^* Q'_2$ and $Q_2 \rightarrow_{\beta\eta} Q'_2$. So it is done with $N_2 = cP_2Q'_2$.
  - • Let $M_1 = cP_1$ such that $P_1 \in \Lambda_c^{\beta\eta}$. Then by compatibility $N_1 = cP'_1$ such that $P_1 \rightarrow_{\beta\eta} P'_1$. By Lemma 4.2.7.7:
    - Either $M_2 = P_2$ and $P_1 \rightarrow_c^* P_2$. By IH, there exists P'_2 such that $P_2 \rightarrow_{\beta\eta} P'_2$ and $P'_1 \rightarrow_c^* P'_2$. So it is done with $N_2 = P'_2$.
    - Or $M_2 = cP_2$ and $P_1 \rightarrow_c^* P_2$. By IH, there exists P'_2 such that $P_2 \rightarrow_{\beta\eta} P'_2$ and $P'_1 \rightarrow_c^* P'_2$. So it is done with $N_2 = cP'_2$.

2 Easy by Lemma 4.4.5.1. □

Proof of Lemma 4.4.6.

⇒) Let $M \rightarrow_{\beta\eta}^* N$. Let c be a variable such that $c \notin \mathsf{fv}(M)$. By Lemma 4.1.2.3, $c \notin \mathsf{fv}(N)$. We prove that $M \rightarrow_2^* N$ by induction on the size of the reduction $M \rightarrow_{\beta\eta}^* N$.

  - ▼ If $M = N$, then it is done since $M \rightarrow_2^* N$.
  - ▼ If $M \rightarrow_{\beta\eta}^* M' \rightarrow_{\beta\eta} N$. By Lemma 4.1.2.3, $c \notin \mathsf{fv}(M')$. By IH, $M \rightarrow_2^* M'$. We prove that $M' \rightarrow_2 N$ by induction on the structure of M'.
    - • Let $M' \in \mathsf{Var}$. It is done because M' does not reduce.
    - • Let $M' = \lambda x.P$ such that $x \neq c$. By compatibility:
      - Either $N = \lambda x.P'$ such that $P \rightarrow_{\beta\eta} P'$. By IH, $P \rightarrow_2 P'$. By definition there exists Q such that $\Psi_c(P) \rightarrow_{\beta\eta}^* Q$ and $Q \rightarrow_c^* P'$. Then $\Psi_c(M') = \lambda x.\Psi_c(P) \rightarrow_{\beta\eta}^* \lambda x.Q$ and $\lambda x.Q \rightarrow_c^* \lambda x.P'$. Hence, $M' \rightarrow_2 N$.
      - Or $P = Nx$ such that $x \notin \mathsf{fv}(N)$. By Lemma 4.2.7.3, $x \notin \mathsf{fv}(\Psi_c(N))$.

- - - * If N is a λ-abstraction then we have $\Psi_c(M') = \lambda x.\Psi_c(P) = \lambda x.\Psi_c(N)x \rightarrow_\eta \Psi_c(N)$, and by Lemma 4.2.7.1, $\Psi_c(N) \rightarrow_c^* N$. Hence, $M' \rightarrow_2 N$.
    - * Else, $\Psi_c(M') = \lambda x.\Psi_c(P) = \lambda x.c\Psi_c(N)x \rightarrow_\eta c\Psi_c(N)$ and by Lemma 4.2.7.1, $c\Psi_c(N) \rightarrow_c \Psi_c(N) \rightarrow_c^* N$. Hence, $M' \rightarrow_2 N$.
- • Let $M' = PQ$.
  - – If $P = \lambda x.P_1$, such that $x \neq c$ then $M' = (\lambda x.P_1)Q$ and by compatibility:
    - * Either $N = (\lambda x.P_2)Q$ and $P_1 \rightarrow_{\beta\eta} P_2$. By IH, $P_1 \rightarrow_2 P_2$. By definition there exists P_1' such that $\Psi_c(P_1) \rightarrow_{\beta\eta}^* P_1'$ and $P_1' \rightarrow_c^* P_2$. So, $\Psi_c(M') = (\lambda x.\Psi_c(P_1))\Psi_c(Q) \rightarrow_{\beta\eta}^* (\lambda x.P_1')\Psi_c(Q)$ and by Lemma 4.2.7.1, $(\lambda x.P_1')\Psi_c(Q) \rightarrow_c^* (\lambda x.P_2)Q = N$. Hence, $M' \rightarrow_2 N$.
    - * Or, $N = P_0Q$ and $P_1 = P_0x$ such that $x \notin \mathsf{fv}(P_0)$. By Lemma 4.2.7.3, $x \notin \mathsf{fv}(\Psi_c(P_0))$. If P_0 is a λ-abstraction then $\Psi_c(M') = (\lambda x.\Psi_c(P_0)x)\Psi_c(Q) \rightarrow_\eta \Psi_c(P_0)\Psi_c(Q) = \Psi_c(N)$. Else, $\Psi_c(M') = (\lambda x.c\Psi_c(P_0)x)\Psi_c(Q) \rightarrow_\eta c\Psi_c(P_0)\Psi_c(Q) = \Psi_c(N)$. In both cases by Lemma 4.2.7.1, $\Psi_c(N) \rightarrow_c^* N$, and so, $M' \rightarrow_2 N$.
    - * Or $N = (\lambda x.P_1)Q_1$ such that $Q \rightarrow_{\beta\eta} Q_1$. By IH, $Q \rightarrow_2 Q_1$. By definition there exists Q_2 such that $\Psi_c(Q) \rightarrow_{\beta\eta}^* Q_2$ and $Q_2 \rightarrow_c^* Q_1$. So, $\Psi_c(M') = (\lambda x.\Psi_c(P_1))\Psi_c(Q) \rightarrow_{\beta\eta}^* (\lambda x.\Psi_c(P_1))Q_2$ and by Lemma 4.2.7.1, $(\lambda x.\Psi_c(P_1))Q_2 \rightarrow_c^* (\lambda x.P_1)Q_1 = N$. Hence, $M' \rightarrow_2 N$.
    - * Or $N = P_1[x := Q]$. So, $\Psi_c(M') = (\lambda x.\Psi_c(P_1))\Psi_c(Q) \rightarrow_\beta \Psi_c(P_1)[x := \Psi_c(Q)]$ and by Lemma 4.2.7.1 and Lemma 4.2.7.8, $\Psi_c(P_1)[x := \Psi_c(Q)] \rightarrow_c^* P_1[x := Q]$. Hence, $M' \rightarrow_1 N$.
  - – Else,
    - * Either $N = P'Q$ such that $P \rightarrow_{\beta\eta} P'$. By IH, $P \rightarrow_2 P'$. By definition, there exists P_1 such that $\Psi_c(P) \rightarrow_{\beta\eta}^* P_1$ and $P_1 \rightarrow_c^* P'$. So, $\Psi_c(M') = c\Psi_c(P)\Psi_c(Q) \rightarrow_{\beta\eta}^* cP_1\Psi_c(Q)$ and by Lemma 4.2.7.1, $cP_1\Psi_c(Q) \rightarrow_c^* cP'Q \rightarrow_c N$. So $M' \rightarrow_2 N$.
    - * Or $N = PQ'$ such that $Q \rightarrow_{\beta\eta} Q'$. By IH, $Q \rightarrow_2 Q'$. By definition, there exists Q_1 such that $\Psi_c(Q) \rightarrow_{\beta\eta}^* Q_1$ and $Q_1 \rightarrow_c^* Q'$. Therefore, $\Psi_c(M') = c\Psi_c(P)\Psi_c(Q) \rightarrow_\beta^* c\Psi_c(P)Q_1$ and by Lemma 4.2.7.1, $c\Psi_c(P)Q_1 \rightarrow_c^* cPQ' \rightarrow_c N$. So $M' \rightarrow_2 N$.

$\Leftarrow$) Let $M \rightarrow_2^* N$. We prove that $M \rightarrow_{\beta\eta}^* N$ by induction on the size of the derivation $M \rightarrow_2^* N$.

- Let $M = N$, then it is done because $M \rightarrow^*_{\beta\eta} N$.
- Let $M \rightarrow^*_2 M' \rightarrow_2 N$. By IH, $M \rightarrow^*_{\beta\eta} M'$. Because $M' \rightarrow_2 N$ then by definition there exists P such that $\Psi_c(M') \rightarrow^*_{\beta\eta} P$ and $P \rightarrow^*_c N$ and $c \notin \mathsf{fv}(M') \cup \mathsf{fv}(N)$. By Lemma 4.4.3, $\Psi_c(M') \in \Lambda^{\beta\eta}_c$. By Lemma 4.2.7.1, $\Psi_c(M') \rightarrow^*_c M'$. By Lemma 4.4.5.2, there exists Q such that $P \rightarrow^*_c Q$ and $M' \rightarrow^*_{\beta\eta} Q$. By Lemma 4.1.2.3, $c \notin \mathsf{fv}(Q)$. By Lemma 4.4.4.2, $P \in \Lambda^{\beta\eta}_c$. By Lemma 4.2.7.10, $Q \rightarrow^*_c N$. By Lemma 4.2.7.9, $Q = N$. Hence $M' \rightarrow^*_{\beta\eta} N$.

□

Proof of Lemma 4.4.7.

1 By definition, there exist P_1, P_2 such that $\Psi_c(M) \rightarrow^*_{\beta\eta} P_1$, $\Psi_c(M) \rightarrow^*_{\beta\eta} P_2$, $P_1 \rightarrow^*_c M_1$, $P_2 \rightarrow^*_c M_2$ and $c \notin \mathsf{fv}(M) \cup \mathsf{fv}(M_1) \cup \mathsf{fv}(M_2)$. By Lemma 4.4.3, $\Psi_c(M) \in \Lambda^{\beta\eta}_c$. So by Corollary 4.4.2, there exists P_3 such that $P_1 \rightarrow^*_{\beta\eta} P_3$ and $P_2 \rightarrow^*_{\beta\eta} P_3$. By Lemma 4.4.4.2, $P_1, P_2, P_3 \in \Lambda^{\beta\eta}_c$. By Lemma 4.4.4.4, there exists M_3 such that $P_3 \rightarrow^*_c M_3$ and $c \notin \mathsf{fv}(M_3)$. By Lemma 4.4.4.3, $P_1 \rightarrow^*_c \Psi_c(M_1)$ and $P_2 \rightarrow^*_c \Psi_c(M_2)$. By Lemma 4.4.5.2, there exist Q_1, Q_2 such that $P_3 \rightarrow^*_c Q_1$, $P_3 \rightarrow^*_c Q_2$, $\Psi_c(M_1) \rightarrow^*_{\beta\eta} Q_1$ and $\Psi_c(M_2) \rightarrow^*_{\beta\eta} Q_2$. By Lemma 4.2.7.10, $Q_1 \rightarrow^*_c M_3$ and $Q_2 \rightarrow^*_c M_3$. So $M_1 \rightarrow_2 M_3$ and $M_2 \rightarrow_2 M_3$.

2 Easy by Lemma 4.4.7.1. □

A.2 Comparisons and conclusions (Sec. 5)

Proof of Lemma 5.3.2. 2 Let $M \Rightarrow_\beta N$. We prove that $M \rightarrow_1 N$ by induction on the size of the derivation of $M \Rightarrow_\beta N$ and then by case on the last rule of the derivation.

- Let $M \Rightarrow_\beta M = N$ then it is done because by Lemma 4.2.7.1, $\Psi_c(M) \rightarrow^*_c M$.
- Let $M = \lambda x.P \Rightarrow_\beta \lambda x.P' = N$ such that $P \Rightarrow_\beta P'$. Let $x \neq c$. Then $c \notin \mathsf{fv}(P) \cup \mathsf{fv}(P')$. By IH, $P \rightarrow_1 P'$. By definition, there exists Q where $\Psi_c(P) \rightarrow^*_\beta Q \rightarrow^*_c P'$. So $\Psi_c(M) = \lambda x.\Psi_c(P) \rightarrow^*_\beta \lambda x.Q \rightarrow^*_c \lambda x.P' = N$. Hence $M \rightarrow_1 N$.
- Let $M = PQ \Rightarrow_\beta P'Q' = N$ such that $P \Rightarrow_\beta P'$ and $Q \Rightarrow_\beta Q'$. Then $c \notin \mathsf{fv}(P) \cup \mathsf{fv}(P') \cup \mathsf{fv}(Q) \cup \mathsf{fv}(Q')$. By IH, $P \rightarrow_1 P'$ and $Q \rightarrow_1 Q'$. By definition, where P'' and Q'' such that $\Psi_c(P) \rightarrow^*_\beta P'' \rightarrow^*_c P'$ and $\Psi_c(Q) \rightarrow^*_\beta Q'' \rightarrow^*_c Q'$.

  - If P is a λ-abstraction then $\Psi_c(M) = \Psi_c(P)\Psi_c(Q) \rightarrow^*_\beta P''Q'' \rightarrow^*_c P'Q' = N$. So $M \rightarrow_1 N$.
  - Else $\Psi_c(M) = c\Psi_c(P)\Psi_c(Q) \rightarrow^*_\beta cP''Q'' \rightarrow^*_c P'Q' = N$. So $M \rightarrow_1 N$.

- Let $M = (\lambda x.P)Q \Rightarrow_\beta P'[x := Q'] = N$ such that $P \Rightarrow_\beta P'$ and $Q \Rightarrow_\beta Q'$. Let $x \neq c$. Then $c \notin \mathsf{fv}(P) \cup \mathsf{fv}(Q)$. By Lemma 5.3.2.1, $c \notin \mathsf{fv}(P') \cup \mathsf{fv}(Q')$. By IH, $P \rightarrow_1 P'$ and $Q \rightarrow_1 Q'$. By definition, there exist P'' and Q'' such that $\Psi_c(P) \rightarrow^*_\beta P'' \rightarrow^*_c P'$ and $\Psi_c(Q) \rightarrow^*_\beta Q'' \rightarrow^*_c Q'$. So $\Psi_c(M) = (\lambda x.\Psi_c(P))\Psi_c(Q) \rightarrow^*_\beta (\lambda x.P'')Q'' \rightarrow_\beta P''[x := Q'']$ and by Lemma 4.2.7.8 $P''[x := Q''] \rightarrow^*_c P'[x := Q'] = N$. So $M \rightarrow_1 N$.

3. Let $M \Rightarrow_{\beta\eta} N$. We prove that $M \rightarrow_2 N$ by induction on the size of the derivation of $M \Rightarrow_{\beta\eta} N$ and then by case on the last rule of the derivation.

   - Let $M \Rightarrow_{\beta\eta} M = N$ then it is done because by Lemma 4.2.7.1, $\Psi_c(M) \rightarrow^*_c M$.
   - Let $M = \lambda x.P \Rightarrow_{\beta\eta} \lambda x.P' = N$ such that $P \Rightarrow_{\beta\eta} P'$. Let $x \neq c$. Then $c \notin \mathsf{fv}(P) \cup \mathsf{fv}(P')$. By IH, $P \rightarrow_2 P'$. By definition, there exists Q such that $\Psi_c(P) \rightarrow^*_{\beta\eta} Q$ and $Q \rightarrow^*_c P'$. So $\Psi_c(M) = \lambda x.\Psi_c(P) \rightarrow^*_{\beta\eta} \lambda x.Q$ and $\lambda x.Q \rightarrow^*_c \lambda x.P' = N$. So $M \rightarrow_2 N$.
   - Let $M = PQ \Rightarrow_{\beta\eta} P'Q' = N$ such that $P \Rightarrow_{\beta\eta} P'$ and $Q \Rightarrow_{\beta\eta} Q'$. Then $c \notin \mathsf{fv}(P) \cup \mathsf{fv}(P') \cup \mathsf{fv}(Q) \cup \mathsf{fv}(Q')$. By IH, $P \rightarrow_2 P'$ and $Q \rightarrow_2 Q'$. By definition, there exist P'' and Q'' such that $\Psi_c(P) \rightarrow^*_{\beta\eta} P''$, $\Psi_c(Q) \rightarrow^*_{\beta\eta} Q''$, $P'' \rightarrow^*_c P'$ and $Q'' \rightarrow^*_c Q'$.
     - If P is a λ-abstraction then $\Psi_c(M) = \Psi_c(P)\Psi_c(Q) \rightarrow^*_{\beta\eta} P''Q''$ and $P''Q'' \rightarrow^*_c P'Q' = N$. So $M \rightarrow_2 N$.
     - Else $\Psi_c(M) = c\Psi_c(P)\Psi_c(Q) \rightarrow^*_{\beta\eta} cP''Q''$ and $cP''Q'' \rightarrow_c P''Q'' \rightarrow^*_c P'Q' = N$. So $M \rightarrow_2 N$.
   - Let $M = (\lambda x.P)Q \Rightarrow_{\beta\eta} P'[x := Q'] = N$ such that $P \Rightarrow_{\beta\eta} P'$ and $Q \Rightarrow_{\beta\eta} Q'$. Let $x \neq c$. Then $c \notin \mathsf{fv}(P) \cup \mathsf{fv}(Q)$. By Lemma 5.3.2.1, $c \notin \mathsf{fv}(P') \cup \mathsf{fv}(Q')$. By IH, $P \rightarrow_2 P'$ and $Q \rightarrow_2 Q'$. By definition, there exist P'' and Q'' such that $\Psi_c(P) \rightarrow^*_{\beta\eta} P''$, $\Psi_c(Q) \rightarrow^*_{\beta\eta} Q''$, $P'' \rightarrow^*_c P'$ and $Q'' \rightarrow^*_c Q'$. So $\Psi_c(M) = (\lambda x.\Psi_c(P))\Psi_c(Q) \rightarrow^*_{\beta\eta} (\lambda x.P'')Q'' \rightarrow_\beta P''[x := Q'']$ and by Lemma 4.2.7.8 $P''[x := Q''] \rightarrow^*_c P'[x := Q'] = N$. So $M \rightarrow_2 N$.
   - Let $M = \lambda x.Px \Rightarrow_{\beta\eta} N$ such that $P \Rightarrow_{\beta\eta} N$ and $x \notin \mathsf{fv}(P)$. Then $c \notin \mathsf{fv}(P)$. Let $x \neq c$. By IH, $P \rightarrow_2 N$. By definition, there exists Q such that $\Psi_c(P) \rightarrow^*_{\beta\eta} Q$ and $Q \rightarrow^*_c N$. By Lemma 4.2.7.3, $x \notin \mathsf{fv}(\Psi_c(P))$.
     - If P is a λ-abstraction then $\Psi_c(M) = \lambda x.\Psi_c(P)x \rightarrow_\eta \Psi_c(P) \rightarrow^*_{\beta\eta} Q$ and $Q \rightarrow^*_c N$. So $M \rightarrow_2 N$.

- Else $\Psi_c(M) = \lambda x.c\Psi_c(P)x \rightarrow_\eta c\Psi_c(P) \rightarrow^*_{\beta\eta} cQ$ and $cQ \rightarrow_c Q \rightarrow^*_c N$. So $M \rightarrow_2 N$.

□

Appendix B

Proofs of Part II

B.1 The $\lambda I^{\mathbb{N}}$ and $\lambda^{\mathcal{L}_{\mathbb{N}}}$ calculi and associated type systems (Ch. 7)

B.1.1 The syntax of the indexed λ-calculi (Sec. 7.1)

Proof of Lemma 7.1.2. We want to prove that on $\mathcal{L}_{\mathbb{N}}$, $\preceq$ is reflexive, transitive, and antisymmetric. Let us prove that $\preceq$ is reflexive w.r.t. $\mathcal{L}_{\mathbb{N}}$. Let $L \in \mathcal{L}_{\mathbb{N}}$. By definition $L \preceq L$ because $L = L :: \oslash$. Let us prove that $\preceq$ is transitive. Let $L_1 \preceq L_2$ and $L_2 \preceq L_3$. By definition there exist L_4 and L_5 such that $L_2 = L_1 :: L_4$ and $L_3 = L_2 :: L_5$. Therefore $L_3 = (L_1 :: L_4) :: L_5 = L_1 :: (L_4 :: L_5)$ (it is also easy to check that $\preceq$ is associative). Let us prove that $\preceq$ is antisymmetric. Assume $L_1 \preceq L_2$ and $L_2 \preceq L_1$. By definition there exist L_3 and L_4 such that $L_2 = L_1 :: L_3$ and $L_1 = L_2 :: L_4$. Therefore $L_1 = L_1 :: L_3 :: L_4$. Which means that $L_3 = L_4 = \oslash$. □

Proof of Lemma 7.1.6. ⇒) By definition. ⇐) Each of 1. and 2. is by cases on the derivation $\lambda x^n.M \in \mathbb{M}$ respectively $M_1 M_2 \in \mathbb{M}$. □

Lemma B.1.1. *Let $i \in \{1, 2, 3\}$.*

1. *On $\mathcal{M}_i$, $\diamond$ is reflexive and symmetric but not transitive.*
2. (a) *Let $M, (N_1 N_2) \in \mathcal{M}_i$. We have $M \diamond \{N_1, N_2\}$ iff $M \diamond (N_1 N_2)$.*

   (b) *Let $M, \lambda x^I.N \in \mathcal{M}_i$ such that $\forall I'.\ x^{I'} \notin \mathsf{fv}(M)$. We have $M \diamond N$ iff $M \diamond (\lambda x^I.N)$.*

   (c) *Let $M, N[(x_i^{I_i} := N_i)_p] \in \mathcal{M}_i$ and $\overline{M} = \{N\} \cup \{N_i \mid i \in \{1, \ldots, p\}\} \subset \mathcal{M}_i$. If $M \diamond \overline{M}$ then $M \diamond N[(x_i^{I_i} := N_i)_p]$.*
3. *Let $M_1[(x_i^{I_i} := N_i)_p], M_2[(x_i^{I_i} := N_i)_p] \in \mathcal{M}_i$ and $\overline{M} = \{M_1, M_2\} \cup \{N_i \mid i \in \{1, \ldots, p\}\}$. If $\diamond\overline{M}$ then $M_1[(x_i^{I_i} := N_i)_p] \diamond M_2[(x_i^{I_i} := N_i)_p]$.*

4. *Let $M \in \mathcal{M}_i$ and $\{I_1, \dots, I_n\} = \{I \mid x^I$ occurs in $M\}$. If $i \in \{1, 2\}$ then $\mathsf{deg}(M) = \mathsf{min}(I_1, \dots, I_n)$. If $i = 3$ then $\forall i \in \{1, \dots, n\}$. $\mathsf{deg}(M) \preceq I_i$.*

5. *Let $\overline{M} = \{M\} \cup \{N_i \mid 1 \leq i \leq p\} \subset \mathcal{M}_i$. We have:*

   (a) ($\diamond\overline{M}$ and $\forall j \in \{1, \dots, p\}$. $\mathsf{deg}(N_j) = I_j$) iff $M[(x_i^{I_i} := N_i)_p] \in \mathcal{M}_i$.

   (b) If $\diamond\overline{M}$ and $\forall j \in \{1, \dots, p\}$. $\mathsf{deg}(N_j) = I_j$, then $\mathsf{deg}(M[(x_i^{I_i} := N_i)_p]) = \mathsf{deg}(M)$.

6. *Let $M, N, P \in \mathcal{M}_i$. If $\diamond\{M, N, P\}$, $\mathsf{deg}(N) = I$, $\mathsf{deg}(P) = J$ and $x^I \notin \mathsf{fv}(P) \cup \{y^J\}$ then $M[x^I := N][y^J := P] = M[y^J := P][x^I := N[y^J := P]]$.*

7. *Let $M, N, P \in \mathcal{M}_i$. If $M \diamond P$ and $\mathsf{fv}(M) = \mathsf{fv}(N)$ then $N \diamond P$.*

8. *Let $i \in \{1, 2\}$ and $M, N \in \mathcal{M}_i$ where $\mathsf{deg}(N) = n$ and $x^n \in \mathsf{fv}(M)$. We have: $M[x^n := N] \in \mathbb{M}$ iff $M, N \in \mathbb{M}$ and $M \diamond N$.* □

Proof of Lemma B.1.1.

1. For reflexivity, we show by induction on $M \in \mathcal{M}_i$ that if $x^I, x^J \in \mathsf{fv}(M)$ then $I = J$. Symmetry is by definition of $\diamond$. For failure of transitivity take z^1, y^2 and z^2 for the case $i \in \{1, 2\}$ and $z^{\oslash}$, $y^{(1)}$ and $z^{(1)}$ for the case $i = 3$.

2. 2a. Let $M, (N_1N_2) \in \mathcal{M}_i$. Let $M \diamond \{N_1, N_2\}$. Assume $x^{I_1} \in \mathsf{fv}(M)$ and $x^{I_2} \in \mathsf{fv}(N_1N_2)$. Then $x^{I_2} \in \mathsf{fv}(N_1)$ or $x^{I_2} \in \mathsf{fv}(N_2)$. In either case, by hypothesis and definition of $\diamond$, $I_1 = I_2$. Therefore $M \diamond N_1N_2$. Let $M \diamond N_1N_2$. Assume $x^{I_1} \in \mathsf{fv}(M)$ and $x^{I_2} \in \mathsf{fv}(N_1)$. Then by definition of $\diamond$, $I_1 = I_2$. Assume $x^{I_1} \in \mathsf{fv}(M)$ and $x^{I_2} \in \mathsf{fv}(N_2)$ then by definition of $\diamond$, $I_1 = I_2$. Therefore $M \diamond \{N_1, N_2\}$.

   2b. Let $M, \lambda x^I.N \in \mathcal{M}_i$ such that $\forall I'$. $x^{I'} \notin \mathsf{fv}(M)$. Let $M \diamond N$. Assume $y^{I_1} \in \mathsf{fv}(M)$ and $y^{I_2} \in \mathsf{fv}(\lambda x^I.N)$. Then $y^{I_2} \in \mathsf{fv}(N) \setminus \{x^I\} \subseteq \mathsf{fv}(N)$. By definition of $\diamond$, $I_1 = I_2$. Therefore $M \diamond \lambda x^I.N$. Let $M \diamond \lambda x^I.N$. Assume $y^{I_1} \in \mathsf{fv}(M)$ and $y^{I_2} \in \mathsf{fv}(N)$. Because $\forall I'$. $x^{I'} \notin \mathsf{fv}(M)$ and $y^{I_1} \in \mathsf{fv}(M)$ then $x \neq y$. Therefore $y^{I_2} \in \mathsf{fv}(\lambda x^I.N)$. By hypothesis and definition of $\diamond$, $I_1 = I_2$. Therefore $M \diamond N$.

   2c. Let $M, N[(x_i^{I_i} := N_i)_p] \in \mathcal{M}_i$, $\overline{M} = \{N\} \cup \{N_i \mid i \in \{1, \dots, p\}\} \subset \mathcal{M}_i$, and $M \diamond \overline{M}$. Assume $y^{I_1} \in \mathsf{fv}(M)$ and $y^{I_2} \in \mathsf{fv}(N[(x_i^{I_i} := N_i)_p])$. Therefore $y^{I_2} \in \mathsf{fv}(N)$ or $y^{I_2} \in \mathsf{fv}(N_i)$ for a $i \in \{1, \dots, p\}$. In either case, by hypothesis and definition of $\diamond$, $I_1 = I_2$. Therefore $M \diamond N[(x_i^{I_i} := N_i)_p]$.

3. By 2c, $M_1 \diamond M_2[(x_i^{I_i} := N_i)_p]$ and $N_j \diamond M_2[(x_i^{I_i} := N_i)_p]$ $\forall\, 1 \leq j \leq p$, and, by 2c again and by 1, $M_1[(x_i^{I_i} := N_i)_p] \diamond M_2[(x_i^{I_i} := N_i)_p]$.

4. By induction on M.

5. Direction $\Leftarrow$) of 5a. is by definition of substitution because substitution is only defined on such conditions.

   We prove direction $\Rightarrow$) of 5a. and 5b. by induction on M. Let $i \in \{1, 2\}$.

   - Let $M = y^I$. If there exists $j \in \{1, \dots, p\}$ such that $y^I = x^{I_j}$ then $M[(x_i^{I_i} := N_i)_p] = N_j \in \mathcal{M}_i$. Also $\mathsf{deg}(M[(x_i^{I_i} := N_i)_p]) = \mathsf{deg}(N_j) = I_j = I = \mathsf{deg}(M)$. If there is no $j \in \{1, \dots, p\}$ such that $y^I = x^{I_j}$ then $M[(x_i^{I_i} := N_i)_p] = M \in \mathcal{M}_i$. Also $\mathsf{deg}(M[(x_i^{I_i} := N_i)_p]) = \mathsf{deg}(M)$.
   - Let $M = \lambda y^I.M_1$ such that $y^I \in \mathsf{fv}(M_1)$ and $\forall I'.\ \forall j \in \{1, \dots, p\}.\ y^{I'} \notin \mathsf{fv}(N_j) \cup \{x_j^{I_j}\}$. By 2b., $\diamond\{M_1\} \cup \{N_j \mid j \in \{1, \dots, p\}\}$. By IH, $M_1[(x_i^{I_i} := N_i)_p] \in \mathcal{M}_2$ and $\mathsf{deg}(M_1[(x_i^{I_i} := N_i)_p]) = \mathsf{deg}(M_1)$. Therefore, $M[(x_i^{I_i} := N_i)_p] = \lambda y^I.M_1[(x_i^{I_i} := N_i)_p] \in \mathcal{M}_2$ because $y^I \in \mathsf{fv}(M_1[(x_i^{I_i} := N_i)_p])$. Also, $\mathsf{deg}(M[(x_i^{I_i} := N_i)_p]) = \mathsf{deg}(M_1[(x_i^{I_i} := N_i)_p]) = \mathsf{deg}(M_1) = \mathsf{deg}(M)$.
   - Let $M = M_1 M_2$ such that $M_1 \diamond M_2$. By 2a., $\diamond\{M_1, M_2\} \cup \{N_j \mid j \in \{1, \dots, p\}\}$. Let $P_1 = M_1[(x_i^{I_i} := N_i)_p]$ and $P_2 = M_2[(x_i^{I_i} := N_i)_p]$. By IH, $P_1 \in \mathcal{M}_2$, $P_2 \in \mathcal{M}_2$, $\mathsf{deg}(P_1) = \mathsf{deg}(M_1)$, and $\mathsf{deg}(P_2) = \mathsf{deg}(M_2)$. By 3., $P_1 \diamond P_2$. Therefore, $M[(x_i^{I_i} := N_i)_p] = P_1 P_2 \in \mathcal{M}_2$. Finally, one obtains $\mathsf{deg}(M[(x_i^{I_i} := N_i)_p]) = \mathsf{min}(P_1, P_2) = \mathsf{min}(\mathsf{deg}(M_1), \mathsf{deg}(M_2)) = \mathsf{deg}(M)$.

   The proof for $i = 3$ is similar

6. By induction on M using 2c. and 5a.

7. If $x^I \in \mathsf{fv}(N) = \mathsf{fv}(M)$ and $x^J \in \mathsf{fv}(P)$ then since $M \diamond P$, $I = J$.

8. By induction on M.

   - By definition of substitution, $x^n[x^n := N] \in \mathbb{M}$ iff $x^n, N \in \mathbb{M}$ and $x^n \diamond N$.
   - Let $M = \lambda y^m.M'$ such that $\forall m'.\ y^{m'} \notin \mathsf{fv}(N) \cup \{x^n\}$. Then $(\lambda y^m.M')[x^n := N] \in \mathbb{M} \Rightarrow \lambda y^m.M'[x^n := N] \in \mathbb{M}$ and $y^m \in \mathsf{fv}(M') \setminus \mathsf{fv}(N)$ (since $\lambda y^m.M' \in \mathcal{M}_1$) $\Rightarrow^{\text{Lemma 7.1.6}} M'[x^n := N] \in \mathbb{M}$, $y^m \in \mathsf{fv}(M'[x^n := N])$ and $y^m \in \mathsf{fv}(M') \setminus \mathsf{fv}(N) \Leftrightarrow^{\text{by IH}} M', N \in \mathbb{M}$, $M' \diamond N$, $y^m \in \mathsf{fv}(M'[x^n := N])$ and $y^m \in \mathsf{fv}(M') \setminus \mathsf{fv}(N) \Leftrightarrow^{\text{by 2b and Lemma 7.1.6}} \lambda y^m.M', N \in \mathbb{M}$ and $\lambda y^m.M' \diamond N$.
   - Let $M = M_1 M_2$. Note that $M_1 \diamond M_2$. Then $(M_1 M_2)[x^n := N] \in \mathbb{M} \Leftrightarrow M_1[x^n := N] M_2[x^n := N] \in \mathbb{M}$ and $\diamond\{M_1, M_2, N\}$ (because $(M_1 M_2)[x^n := N] \in \mathcal{M}_i$) $\Leftrightarrow^{\text{by 5b and Lemma 7.1.6}} M_1[x^n := N], M_2[x^n := N] \in \mathbb{M}$, $M_1[x^n := N] \diamond M_2[x^n := N]$, $\diamond\{M_1, M_2, N\}$ and $\mathsf{deg}(M_1) = \mathsf{deg}(M_1[x^n := N]) \leq \mathsf{deg}(M_2[x^n := N]) = \mathsf{deg}(M_2) \Leftrightarrow^{\text{by IH}} M_1, M_2, N \in \mathbb{M}$, $\diamond\{M_1, M_2, N\}$ and $\mathsf{deg}(M_1) \leq \mathsf{deg}(M_2) \Leftrightarrow^{\text{by 2a and Lemma 7.1.6}} M_1 M_2, N \in \mathbb{M}$ and $(M_1 M_2) \diamond N$.

□

Proof of Theorem 7.1.11. We only prove 2. Let $M \in \mathcal{M}_2$. First we prove that if $M \rightarrowtriangle_\beta N$ then $\mathsf{fv}(M) = \mathsf{fv}(N)$, $\mathsf{deg}(M) = \mathsf{deg}(N)$, and $M \in \mathbb{M}$ iff $N \in \mathbb{M}$. We prove this result by induction on the derivation $M \rightarrowtriangle_\beta N$ and the by case on the last rule of the derivation. We only prove the case $M = (\lambda x^n.M_1)M_2$ and $N = M_1[n := M_2]$ such that $\forall m.\ x^m \in \mathsf{fv}(M_2)$ and $\mathsf{deg}(M_2) = n$ (derivation of $M \rightarrowtriangle_\beta N$ is of length 1). Because $M \in \mathcal{M}_2$ then $x^n \in \mathsf{fv}(M_1)$ and $(\lambda x^n.M_1) \diamond M_2$. One obtains that $\mathsf{fv}(M) = (\mathsf{fv}(M_1) \setminus \{x^n\}) \cup \mathsf{fv}(M_2) = \mathsf{fv}(N)$ because $x^n \in \mathsf{fv}(M_1)$. Also $\mathsf{deg}(M) = \mathsf{min}(\mathsf{deg}(\lambda x^n.M_1), \mathsf{deg}(M_2)) = \mathsf{min}(\mathsf{deg}(M_1), n)$. By Lemma B.1.1.4, because $x^n \in \mathsf{fv}(M_1)$ and $\mathsf{deg}(x^n) = n$ then $\mathsf{deg}(M_1) \leq n = \mathsf{deg}(M_2)$. By Lemma B.1.1.2b, $M_1 \diamond M_2$. Therefore $\mathsf{deg}(M) = \mathsf{deg}(M_1)$ and by Lemma B.1.1.5b, $\mathsf{deg}(N) = \mathsf{deg}(M_1) = \mathsf{deg}(M)$. Let us now prove that $M \in \mathbb{M} \Leftrightarrow N \in \mathbb{M}$. This result is easily obtained using Lemma B.1.1.8. □

Lemma B.1.2. *Let* $i \in \{1, 2, 3\}$, $\twoheadrightarrow \in \{\rightarrowtriangle, \rightarrowtriangle^*\}$, $r \in \{\beta, \beta\eta, h\}$, $p \geq 0$ *and* $M, N, P, N_1, \ldots, N_p \in$ $\mathcal{M}_i$.

1. *If* $M \twoheadrightarrow_r N$, $P \twoheadrightarrow_r Q$, *and* $M \diamond P$ *then* $N \diamond Q$.

2. *If* $M \twoheadrightarrow_r N$, $M \diamond P$, *and* $\mathsf{deg}(P) = I$ *then* $M[x^I := P] \twoheadrightarrow_r N[x^I := P]$.

3. *If* $N \twoheadrightarrow_r P$, $M \diamond N$, *and* $\mathsf{deg}(N) = I$ *then* $M[x^I := N] \twoheadrightarrow_r^* M[x^I := P]$.

4. *If* $M \rightarrowtriangle_r^* N$, $P \rightarrowtriangle_r^* P'$, $M \diamond P$, *and* $\mathsf{deg}(P) = I$ *then* $M[x^I := P] \rightarrowtriangle_r^* N[x^I := P']$. □

Proof of Lemma B.1.2.

1. The result is obtained because by Lemma 7.1.11, $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$ and $\mathsf{fv}(Q) \subseteq \mathsf{fv}(P)$.

2. Note that, by Lemma 1, $N \diamond P$. Case $\rightarrowtriangle_r$ is by induction on M using Lemmas B.1.1.5b and B.1.1.6. Case $\rightarrowtriangle_r^*$ is by induction on the length of $M \rightarrowtriangle_r^* N$ using the result for case $\rightarrowtriangle_r$.

3. Note that, by Lemma 1, $M \diamond P$ and by Lemma 7.1.11, $\mathsf{deg}(P) = \mathsf{deg}(N) = I$. Case $\rightarrowtriangle_r$ is by induction on M. Case $\rightarrowtriangle_r^*$ is by induction on the length of $M \rightarrowtriangle_r^* N$ using the result for case $\rightarrowtriangle_r$.

4. Use 2. and 3. □

The next lemma shows that the lifting of a term to higher or lower degrees, is a well behaved operation with respect to all that matters (free variables, reduction, joinability, substitution, etc.).

Lemma B.1.3. *Let* $p \geq 0$, $i \in \{1, 2\}$ *and* $M, N, N_1, N_2, \ldots, N_p \in \mathcal{M}_i$.

1. *(a)* $\mathsf{deg}(M^+) = \mathsf{deg}(M) + 1$, $(M^+)^- = M$ *and* $x^n \in \mathsf{fv}(M^+)$ *iff* $x^{n-1} \in \mathsf{fv}(M)$.

   (b) If $\mathsf{deg}(M) > 0$ *then* $M^- \in \mathcal{M}_i$, $\mathsf{deg}(M^-) = \mathsf{deg}(M) - 1$, $(M^-)^+ = M$ *and* $(x^n \in \mathsf{fv}(M^-) \Leftrightarrow x^{n+1} \in \mathsf{fv}(M))$.

   (c) Let $\overline{M} \subset \mathcal{M}_i$. *Then,*

      i. $\diamond\overline{M}$ *iff* $\diamond\overline{M}^+$.

      ii. If $\mathsf{deg}(\overline{M}) > 0$ *then* $\diamond\overline{M}$ *iff* $\diamond\overline{M}^-$.

      iii. $M \in \overline{M}^+$ *iff* $(M^- \in \overline{M}$ *and* $\mathsf{deg}(M) > 0)$.

   (d) $M \in \mathbb{M}$ *iff* $M^+ \in \mathbb{M} \cap \mathcal{M}_i$.

   (e) If $\mathsf{deg}(M) > 0$ *then* $M \in \mathbb{M}$ *iff* $M^- \in \mathbb{M}$.

2. *Let* $\overline{M} = \{M\} \cup \{N_i \mid i \in \{1, \dots, p\}\} \subset \mathcal{M}_i$. *If* $\diamond\overline{M}$ *then* $(M[(x_i^{n_i} := N_i)_p])^+ = M^+[(x_i^{n_i+1} := N_i{}^+)_p]$.

3. *If* $\mathsf{deg}(M), \mathsf{deg}(N) > 0$, *and* $M \diamond N$ *then* $(M[x^{n+1} := N])^- = M^-[x^n := N^-]$. □

Proof of Lemma B.1.3.

1. 1a. and 1b. are by induction on M. For 1(c)i. use 1a. For 1(c)ii. use 1b. As to 1(c)iii., if $M \in \overline{M}^+$ then $M = P^+$ where $P \in \overline{M}$ and by 1a., $\mathsf{deg}(M) = \mathsf{deg}(P) + 1 > 0$ and $M^- = (P^+)^- = P$. Hence, $M^- \in \overline{M}$ and $\mathsf{deg}(M) > 0$. On the other hand, if $M^- \in \overline{M}$ and $\mathsf{deg}(M) > 0$ then by 1b., $M = P^+$ and $(M^-)^+ = M \in \overline{M}^+$. 1d. is by induction on M using 1a., 1(c)i. and Lemma 7.1.6. Finally, for 1e., by 1b. and 1d., $M = (M^-)^+ \in \mathbb{M} \Leftrightarrow M^- \in \mathbb{M}$.

2. By induction on M (by 1(c)i. and Lemma B.1.1.5, we have $M[(x_i^{n_i} := N_i)_p] \in \mathcal{M}_i$ and $M^+[(x_i^{n_i+1} := N_i{}^+)_p] \in \mathcal{M}_i$).

3. By induction on M (by 1(c)ii. and Lemma B.1.1.5, we have $M[x^{n+1} := N] \in \mathcal{M}_i$ and $M^-[x^n := N^-] \in \mathcal{M}_i$).

□

Lemma B.1.4. *Let* $r \in \{\eta, \beta\eta\}$, $\twoheadrightarrow \in \{\rightarrowtriangle, \rightarrowtriangle^*\}$, $p \geq 0$, $i \in \{1, 2\}$ *and* $M, N \in \mathcal{M}_i$.

1. *If* $M \twoheadrightarrow_r N$ *then* $M^+ \twoheadrightarrow_r N^+$.

2. *If* $\mathsf{deg}(M) > 0$ *and* $M \twoheadrightarrow_r N$ *then* $M^- \twoheadrightarrow_r N^-$.

3. *If* $M \twoheadrightarrow_r N^+$ *then* $M^- \twoheadrightarrow_r N$.

4. *If* $M^+ \twoheadrightarrow_r N$ *then* $M \twoheadrightarrow_r N^-$. □

Proof of Lemma B.1.4.

1. The case $r \in \{\eta\}$ and $\to = \rightarrowtriangle$ is by induction on $M \rightarrowtriangle_r N$ using Lemma B.1.5, for case $\rightarrowtriangle_{\beta\eta}$ use the results for $\rightarrowtriangle_\beta$ (Lemma B.1.5) and $\rightarrowtriangle_\eta$, case $\rightarrowtriangle^*_r$ is by induction on the length of $M \rightarrowtriangle^*_r N$ using the result for case $\rightarrowtriangle_r$.

2. Similar to 1.

3. By Lemma 7.1.11.2, Lemma B.1.5 and 2 above, $M^- \to N$.

4. Similar to 3. □

Lemma B.1.5. *Let* $\to \in \{\rightarrowtriangle_\beta, \rightarrowtriangle_\eta, \rightarrowtriangle_{\beta\eta}, \rightarrowtriangle_h, \rightarrowtriangle^*_\beta, \rightarrowtriangle^*_\eta, \rightarrowtriangle^*_{\beta\eta}, \rightarrowtriangle^*_h\}$*,* $i \geq 0$*,* $p \geq 0$ *and* $M, N, N_1, \ldots, N_p \in \mathcal{M}_3$*. We have:*

1. $M^{+i} \in \mathcal{M}_3$ *and* $\mathsf{deg}(M^{+i}) = i :: \mathsf{deg}(M)$ *and* x^K *occurs in* M^{+i} *iff* $K = i :: L$ *and* x^L *occurs in* M*.*

2. $M \diamond N$ *iff* $M^{+i} \diamond N^{+i}$*.*

3. *Let* $\overline{M} \subseteq \mathcal{M}_3$ *then* $\diamond\overline{M}$ *iff* $\diamond\overline{M}^{+i}$*.*

4. $(M^{+i})^{-i} = M$*.*

5. *If* $\diamond\{M\} \cup \{N_j \mid j \in \{1, \ldots, p\}\}$ *and* $\forall j \in \{1, \ldots, p\}.\ \mathsf{deg}(N_j) = L_j$ *then* $(M[(x_j^{L_j} := N_j)_p])^{+i} = M^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]$*.*

6. *If* $M \to N$ *then* $M^{+i} \to N^{+i}$*.*

7. *If* $\mathsf{deg}(M) = i :: L$ *then:*
   - *(a)* $M = P^{+i}$ *for some* $P \in \mathcal{M}_3$*,* $\mathsf{deg}(M^{-i}) = L$ *and* $(M^{-i})^{+i} = M$*.*
   - *(b) If* $\forall j \in \{1, \ldots, p\}.\ \mathsf{deg}(N_j) = i :: K_j$ *and* $\diamond\{M\} \cup \{N_j \mid j \in \{1, \ldots, p\}\}$ *then* $(M[(x_j^{i::K_j} := N_j)_p])^{-i} = M^{-i}[(x_j^{K_j} := N_j^{-i})_p]$*.*
   - *(c) If* $M \to N$ *then* $M^{-i} \to N^{-i}$*.*

8. *If* $M \to N^{+i}$ *then there is* $P \in \mathcal{M}_3$ *such that* $M = P^{+i}$ *and* $P \to N$*.*

9. *If* $M^{+i} \to N$ *then there is* $P \in \mathcal{M}_3$ *such that* $N = P^{+i}$ *and* $M \to P$*.* □

Proof of Lemma B.1.5.

1. We only prove the lemma by induction on M:
   - If $M = x^L$ then $M^{+i} = x^{i::L} \in \mathcal{M}_3$ and $\mathsf{deg}(x^{i::L}) = i :: L = i :: \mathsf{deg}(x^L)$.

- If $M = \lambda x^L.M_1$ then $M_1 \in \mathcal{M}_3$, $L \succeq \mathsf{deg}(M_1)$ and $M^{+i} = \lambda x^{i::L}.M_1^{+i}$. By IH, $M_1^{+i} \in \mathcal{M}_3$ and $\mathsf{deg}(M_1^{+i}) = i :: \mathsf{deg}(M_1)$ and x^K occurs in M_1^{+i} iff $K = i :: K'$ and $y^{K'}$ occurs in M_1. So $i :: L \succeq i :: \mathsf{deg}(M_1) = \mathsf{deg}(M_1^{+i})$. Hence, $\lambda x^{i::L}.M_1^{+i} \in \mathcal{M}_3$. Moreover, $\mathsf{deg}(M^{+i}) = \mathsf{deg}(M_1^{+i}) = i :: \mathsf{deg}(M_1) = i :: \mathsf{deg}(M)$. If y^K occurs in M^{+i} then either $y^K = x^{i::L}$, so it is done because x^L occurs in M. Or y^K occurs in M_1^{+i}. By IH, $K = i :: K'$ and $y^{K'}$ occurs in M_1. So $y^{K'}$ occurs in M. If y^K occurs in M then either $y^K = x^L$ and then $y^{i::K}$ occurs in M^{+i}. Or y^K occurs in M_1. Then by IH, $y^{i::K}$ occurs in M_1^{+i}. So, $y^{i::K}$ occurs in M^{+i}.

- If $M = M_1M_2$ then $M_1, M_2 \in \mathcal{M}_3$, $\mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$, $M_1 \diamond M_2$ and $M^{+i} = M_1^{+i}M_2^{+i}$. By IH, $M_1^{+i}, M_2^{+i} \in \mathcal{M}_3$, $\mathsf{deg}(M_1^{+i}) = i :: \mathsf{deg}(M_1)$, $\mathsf{deg}(M_2^{+i}) = i :: \mathsf{deg}(M_2)$, y^K occurs in M_1^{+i} iff $K = i :: K'$ and $y^{K'}$ occurs in M_1, and y^K occurs in M_2^{+i} iff $K = i :: K'$ and $y^{K'}$ occurs in M_2. Let $x^L \in \mathsf{fv}(M_1^{+i})$ and $x^K \in \mathsf{fv}(M_2^{+i})$ then, using IH, $L = i :: L'$, $K = i :: K'$, $x^{L'}$ occurs in M_1 and $x^{K'}$ occurs in M_2. Using $M_1 \diamond M_2$, we obtain $L' = K'$, so $L = K$. Hence, $M_1^{+i} \diamond M_2^{+i}$. Because $\mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$ then $\mathsf{deg}(M_1^{+i}) = i :: \mathsf{deg}(M_1) \preceq i :: \mathsf{deg}(M_2) = \mathsf{deg}(M_2^{+i})$. So, $M^{+i} \in \mathcal{M}_3$. Moreover, $\mathsf{deg}(M^{+1}) = \mathsf{deg}(M_1^{+i}) = i :: \mathsf{deg}(M_1) = i :: \mathsf{deg}(M)$. If x^L occurs in M^{+i} then either x^L occurs in M_1^{+i} and using IH, $L = i :: L'$ and $x^{L'}$ occurs in M_1, so $x^{L'}$ occurs in M. Or x^L occurs in M_2^{+i} and using IH, $L = i :: L'$ and $x^{L'}$ occurs in M_2, so $x^{L'}$ occurs in M. If x^L occurs in M then either x^L occurs in M_1 so by IH $x^{i::L}$ occurs in M_1^{+i}, hence $x^{i::L}$ occurs in M^{+i}. Or x^L occurs in M_2 so by IH $x^{i::L}$ occurs in M_2^{+i}, hence $x^{i::L}$ occurs in M^{+i}.

2. Assume $M \diamond N$. Let $x^L \in \mathsf{fv}(M^{+i})$ and $x^K \in \mathsf{fv}(N^{+i})$ then by Lemma B.1.5.1, $L = i :: L'$, $K = i :: K'$, $x^{L'} \in \mathsf{fv}(M)$ and $x^{K'} \in \mathsf{fv}(N)$. Using $M \diamond N$ we obtain $K' = L'$ and so $K = L$.

   Assume $M^{+i} \diamond N^{+i}$. Let $x^L \in \mathsf{fv}(M)$ and $x^K \in \mathsf{fv}(N)$ then by Lemma B.1.5.1, $x^{i::L} \in \mathsf{fv}(M^{+i})$ and $x^{i::K} \in \mathsf{fv}(N^{+i})$. Using $M^{+i} \diamond N^{+i}$ we obtain $i :: K = i :: L$ and so $K = L$.

3. Let $\overline{M} \subseteq \mathcal{M}_3$.

   Assume $\diamond\overline{M}$. Let $M, N \in \overline{M}^{+i}$. Then by definition, $M = P^{+i}$ and $N = Q^{+i}$ such that $P, Q \in \overline{M}$. Because by hypothesis $P \diamond Q$ then by Lemma B.1.5.2, $M \diamond N$.

   Assume $\diamond\overline{M}^{+i}$. Let $M, N \in \overline{M}$ then $M^{+i}, N^{+i} \in \overline{M}^{+i}$. Because by hypothesis $M^{+i} \diamond N^{+i}$ then by Lemma B.1.5.2, $M \diamond N$.

4. By Lemma B.1.5.1, $M^{+i} \in \mathcal{M}_3$ and $\mathsf{deg}(M^{+i}) = i :: \mathsf{deg}(M)$. We prove the lemma by induction on M.

- Let $M = x^L$ then $M^{+i} = x^{i::L}$ and $(M^{+i})^{-i} = x^L$.
- Let $M = \lambda x^L.M_1$ such that $M_1 \in \mathcal{M}_3$ and $L \succeq \mathsf{deg}(M_1)$. Then, $(M^{+i})^{-i} = (\lambda x^{i::L}.M_1^{+i})^{-i} = \lambda x^L.(M_1^{+i})^{-i} =^{\mathrm{IH}} \lambda x^L.M_1$.
- Let $M = M_1M_2$ such that $M_1, M_2 \in \mathcal{M}_3$, $M_1 \diamond M_2$ and $\mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$. Then, $(M^{+i})^{-i} = (M_1^{+i}M_2^{+i})^{-i} = (M_1^{+i})^{-i}(M_2^{+i})^{-i} =^{\mathrm{IH}} M_1M_2$.

5. By 3, $\diamond\{M^{+i}\} \cup \{N_j^{+i} \mid j \in \{1,\dots,p\}\}$. By 1. and Lemma B.1.1.5a, $M[(x_j^{L_j} := N_j)_p]$ and $M^{+i}[(x_j^{i::L_j} := N_j^{+i})_p] \in \mathcal{M}_3$. By induction on M:

   - Let $M = y^K$. If $\forall j \in \{1,\dots,p\}.\ y^K \neq x_j^{L_j}$ then $y^K[(x_j^{L_j} := N_j)_p] = y^K$. Hence $(y^K[(x_j^{L_j} := N_j)_p])^{+i} = y^{i::K} = y^{i::K}[(x_j^{i::L_j} := N_j^{+i})_p]$. If $\exists j \in \{1,\dots,p\}.\ y^K = x_j^{L_j}$ then $y^K[(x_j^{L_j} := N_j)_p] = N_j$. Hence $(y^K[(x_j^{L_j} := N_j)_p])^{+i} = N_j^{+i} = y^{i::K}[(x_j^{i::L_j} := N_j^{+i})_p]$.
   - Let $M = \lambda y^K.M_1$ such that $\forall K'.\ \forall j \in \{1,\dots,p\}.\ y^{K'} \notin \mathsf{fv}(N_j) \cup \{x_j^{L_j}\}$. Then $M[(x_j^{L_j} := N_j)_p] = \lambda y^K.M[(x_j^{L_j} := N_j)_p]$. By Lemma B.1.1.2b, $\diamond\{M_1\} \cup \{N_j \mid j \in \{1,\dots,p\}\}$, and by IH, $(M_1[(x_j^{L_j} := N_j)_p])^{+i} = M_1^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]$. Hence, $(M[(x_j^{L_j} := N_j)_p])^{+i} = \lambda y^{i::K}.(M_1[(x_j^{L_j} := N_j)_p])^{+i} = \lambda y^{i::K}.M_1^{+i}[(x_j^{i::L_j} := N_j^{+i})_p] = (\lambda y^K.M_1)^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]$.
   - Let $M = M_1M_2$. $M[(x_j^{L_j} := N_j)_p] = M_1[(x_j^{L_j} := N_j)_p]M_2[(x_j^{L_j} := N_j)_p]$. By Lemma B.1.1.2a, $\diamond\{M_1\} \cup \{N_j \mid j \in \{1,\dots,p\}\}$ and $\diamond\{M_2\} \cup \{N_j \mid j \in \{1,\dots,p\}\}$. By IH, $(M_1[(x_j^{L_j} := N_j)_p])^{+i} = M_1^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]$ and $(M_2[(x_j^{L_j} := N_j)_p])^{+i} = M_2^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]$. Hence $(M[(x_j^{L_j} := N_j)_p])^{+i} = (M_1[(x_j^{L_j} := N_j)_p])^{+i}(M_2[(x_j^{L_j} := N_j)_p])^{+i} = M_1^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]M_2^{+i}[(x_j^{i::L_j} := N_j^{+i})_p] = M^{+i}[(x_j^{i::L_j} := N_j^{+i})_p]$.

6. By Lemma B.1.5.1, if $M, N \in \mathcal{M}_3$ then $M^{+i}, N^{+i} \in \mathcal{M}_3$.

   - Let $\rightarrow$ be $\rightarrowtriangle_\beta$. By induction on $M \rightarrowtriangle_\beta N$.
     - Let $M = (\lambda x^L.M_1)M_2 \rightarrowtriangle_\beta M_1[x^L := M_2] = N$ where $\mathsf{deg}(M_2) = L$. By Lemma B.1.5.1, $\mathsf{deg}(M_2^{+i}) = i :: L$. Therefore $M^{+i} = (\lambda x^{i::L}.M_1^{+i})M_2^{+i} \rightarrowtriangle_\beta M_1^{+i}[x^{i::L} := M_2^{+i}] = (M_1[x^L := M_2])^{+i}$.
     - Let $M = \lambda x^L.M_1 \rightarrowtriangle_\beta \lambda x^L.N_1 = N$ such that $M_1 \rightarrowtriangle_\beta N_1$. By IH, $M_1^{+i} \rightarrowtriangle_\beta N_1^{+i}$, hence $M^{+i} = \lambda x^{i::L}.M_1^{+i} \rightarrowtriangle_\beta \lambda x^{i::L}.N_1^{+i} = N^{+i}$.
     - Let $M = M_1M_2 \rightarrowtriangle_\beta N_1M_2 = N$ such that $M_1 \rightarrowtriangle_\beta N_1$. By IH, $M_1^{+i} \rightarrowtriangle_\beta N_1^{+i}$, hence $M^{+i} = M_1^{+i}M_2^{+i} \rightarrowtriangle_\beta N_1^{+i}M_2^{+i} = N^{+i}$.
     - Let $M = M_1M_2 \rightarrowtriangle_\beta M_1N_2 = N$ such that $M_2 \rightarrowtriangle_\beta N_2$. By IH, $M_2^{+i} \rightarrowtriangle_\beta N_2^{+i}$, hence $M^{+i} = M_1^{+i}M_2^{+i} \rightarrowtriangle_\beta N_1^{+i}M_2^{+i} = N^{+i}$.
   - Let $\rightarrow$ be $\rightarrowtriangle_\beta^*$. By induction on $\rightarrowtriangle_\beta^*$ using $\rightarrowtriangle_\beta$.
   - Let $\rightarrow$ be $\rightarrowtriangle_\eta$. We only do the base case. The inductive cases are as for $\rightarrowtriangle_\beta$. Let $M = \lambda x^L.Nx^L \rightarrowtriangle_\eta N$ where $x^L \notin \mathsf{fv}(N)$. By Lemma B.1.5.1, $x^{i::L} \notin \mathsf{fv}(N^{+i})$ Then $M^{+i} = \lambda x^{i::L}.N^{+i}x^{i::L} \rightarrowtriangle_\eta N^{+i}$.

- Let $\to$ be $\rightarrowtriangle_{\eta}^{*}$. By induction on $\rightarrowtriangle_{\eta}^{*}$ using $\rightarrowtriangle_{\eta}$.
- Let $\to$ be $\rightarrowtriangle_{\beta\eta}$, $\rightarrowtriangle_{\beta\eta}$, $\rightarrowtriangle_{h}$ or $\rightarrowtriangle_{h}^{*}$. By the previous items.

7. (a) By induction on M:

   - Let $M = y^{i::L}$ then $y^L \in \mathcal{M}_3$ and $\mathsf{deg}((y^{i::L})^{-i}) = \mathsf{deg}(y^L) = L$ and $((y^{i::L})^{-i})^{+i} = y^{i::L}$.
   - Let $M = \lambda y^K.M_1$ such that $M_1 \in \mathcal{M}_3$ and $K \succeq \mathsf{deg}(M_1)$. Because $\mathsf{deg}(M_1) = \mathsf{deg}(M) = i :: L$, by IH, $M_1 = P^{+i}$ for some $P \in \mathcal{M}_3$, $\mathsf{deg}(M_1^{-i}) = L$ and $(M_1^{-i})^{+i} = M_1$. Because $K \succeq i :: L$ then $K = i :: L :: K'$ for some K'. Let $Q = \lambda y^{L::K'}.P$. By Lemma B.1.5.4, $P = (P^{+i})^{-i} = M_1^{-i}$ then $\mathsf{deg}(P) = L$. Because $L \preceq L :: K'$ then $Q \in \mathcal{M}_3$ and $Q^{+i} = M$. Moreover, using Lemma B.1.5.4, $\mathsf{deg}(M^{-i}) = \mathsf{deg}(Q) = \mathsf{deg}(P) = L$ and $(M^{-i})^{+i} = P^{+i} = M$.
   - Let $M = M_1 M_2$ such that $M_1, M_2 \in \mathcal{M}_3$, $M_1 \diamond M_2$ and $\mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$. Then $\mathsf{deg}(M) = \mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$, so $\mathsf{deg}(M_2) = i :: L :: L'$ for some L'. By IH $M_1 = P_1^{+i}$ for some $P_1 \in \mathcal{M}_3$, $\mathsf{deg}(M_1^{-i}) = L$ and $(M_1^{-i})^{+i} = M_1$. Again by IH, $M_2 = P_2^{+i}$ for some $P_2 \in \mathcal{M}_3$, $\mathsf{deg}(M_2^{-i}) = L :: L'$ and $(M_2^{-i})^{+i} = M_2$. If $y^{K_1} \in \mathsf{fv}(P_1)$ and $y^{K_2} \in \mathsf{fv}(P_2)$ then by Lemma B.1.5.1, $K_1' = i :: K_1$, $K_2' = i :: K_2$, $x^{K_1'} \in \mathsf{fv}(M_1)$ and $x^{K_2'} \in \mathsf{fv}(M_2)$. Thus $K_1' = K_2'$, so $K_1 = K_2$ and $P_1 \diamond P_2$. Because $\mathsf{deg}(P_1) = \mathsf{deg}(M_1^{-i}) = L \preceq L :: L' = \mathsf{deg}(M_2^{-i}) = \mathsf{deg}(P_2)$ then $Q = P_1 P_2 \in \mathcal{M}_3$ and $Q^{+i} = (P_1 P_2)^{+i} = P_1^{+i} P_2^{+i} = M$. Moreover, by Lemma B.1.5.4 $\mathsf{deg}(M^{-i}) = \mathsf{deg}(Q) = \mathsf{deg}(P_1) = L$ and $(M^{-i})^{+i} = Q^{+i} = M$.

   (b) By the previous item, there exist $M', N_1', \ldots, N_n' \in \mathcal{M}_3$ such that $M = M'^{+i}$ and $\forall j \in \{1, \ldots, p\}.\ N_j = N_j'^{+i}$. By Lemma B.1.5.3, $\diamond\{M'\} \cup \{N_j' \mid j \in \{1, \ldots, p\}\}$. By Lemma B.1.5.4, $M^{-i} = M'$ and $\forall j \in \{1, \ldots, p\}.\ N_j^{-i} = N_j'$. So, $\diamond\{M^{-i}\} \cup \{N_j^{-i} \mid j \in \{1, \ldots, p\}\}$. By Lemma B.1.1.5a, $M[(x_j^{i::K_j} := N_j)_p], M^{-i}[(x_j^{K_j} := N_j^{-i})_p] \in \mathcal{M}_3$ and $\mathsf{deg}(M[(x_j^{i::K_j} := N_j)_p]) = \mathsf{deg}(M) = i :: L$. We prove the result by induction on M:

   - Let $M = y^{i::L}$. If $(\forall j \in \{1, \ldots, p\}.\ y^{i::L} \neq x_j^{i::K_j})$ then $y^{i::L}[(x_j^{i::K_j} := N_j)_p] = y^{i::L}$. Hence $(y^{i::L}[(x_j^{i::K_j} := N_j)_p])^{-i} = y^L = y^L[(x_j^{K_j} := N_j^{-i})_p]$. If $\exists 1 \leq j \leq p, y^{i::L} = x_j^{i::K_j}$ then $y^{i::L}[(x_j^{i::K_j} := N_j)_p] = N_j$. Hence $(y^{i::L}[(x_j^{i::K_j} := N_j)_p])^{-i} = N_j^{-i} = y^L[(x_j^{K_j} := N_j^{-i})_p]$.
   - Let $M = \lambda y^K.M_1$ such that $M_1 \in \mathcal{M}_3$, $K \succeq \mathsf{deg}(M_1)$, and $\forall K'.\ \forall j \in \{1, \ldots, p\}.\ y^{K'} \notin \mathsf{fv}(N_j) \cup \{x_j^{i::K_j}\}$. Then, $M[(x_j^{i::K_j} := N_j)_p] = \lambda y^K.M_1[(x_j^{i::K_j} := N_j)_p]$. By Lemma B.1.1.2b, $\diamond\{M_1\} \cup \{N_j \mid j \in \{1, \ldots, p\}\}$. By definition $\mathsf{deg}(M) = \mathsf{deg}(M_1)$. By IH, $(M_1[(x_j^{i::K_j} :=$

$N_j)_p])^{-i} = M_1^{-i}[(x_j^{K_j} := N_j^{-i})_p]$. Because $\mathsf{deg}(M_1) = i :: L \preceq K$ then $K = i :: L :: K'$ for some K'. Hence, $(M[(x_j^{i::K_j} := N_j)_p])^{-i} = \lambda y^{L::K'}.(M_1[(x_j^{i::K_j} := N_j)_p])^{-i} = \lambda y^{L::K'}.M_1^{-i}[(x_j^{K_j} := N_j^{-i})_p] = (\lambda y^K.M_1)^{-i}[(x_j^{K_j} := N_j^{-i})_p]$.

- Let $M = M_1M_2$ such that $M_1, M_2 \in \mathcal{M}_3$, $M_1 \diamond M_2$ and $\mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$. Let $P_1 = M_1[(x_j^{i::K_j} := N_j)_p]$ and $P_2 = M_2[(x_j^{i::K_j} := N_j)_p]$. Then, $M[(x_j^{i::K_j} := N_j)_p] = P_1P_2$. By Lemma B.1.1.2a, $\diamond\{M_1\} \cup \{N_j \mid j \in \{1, \dots, p\}\}$ and $\diamond\{M_2\} \cup \{N_j \mid j \in \{1, \dots, p\}\}$. By definition $\mathsf{deg}(M) = \mathsf{deg}(M_1) \preceq \mathsf{deg}(M_2)$. Therefore $\mathsf{deg}(M_2) = i :: L :: L'$ for some L'. By IH, $P_1^{-i} = M_1^{-i}[(x_j^{K_j} := N_j^{-i})_p]$ and $P_2^{-i} = M_2^{-i}[(x_j^{K_j} := N_j^{-i})_p]$. Finally, $(M[(x_j^{i::K_j} := N_j)_p])^{-i} = P_1^{-i}P_2^{-i} = M_1^{-i}[(x_j^{K_j} := N_j^{-i})_p]M_2^{-i}[(x_j^{K_j} := N_j^{-i})_p] = M^{-i}[(x_j^{K_j} := N_j^{-i})_p]$.

(c) Using Lemma B.1.5.4, Lemma 7.1.11 and the first item, we prove that $M^{-i}, N^{-i} \in \mathcal{M}_3$.

- Let $\to$ be $\rightarrowtriangle_\beta$. By induction on $M \rightarrowtriangle_\beta N$.
  - Let $M = (\lambda x^K.M_1)M_2 \rightarrowtriangle_\beta M_1[x^K := M_2] = N$ where $\mathsf{deg}(M_2) = K$. Because $M \in \mathcal{M}_3$ then $M_1 \in \mathcal{M}_3$. Because $i :: L = \mathsf{deg}(M) = \mathsf{deg}(M_1) \preceq K$ then $K = i :: L :: K'$. By Lemma B.1.5.7, $\mathsf{deg}(M_2^{-i}) = L :: K'$. Hence, $M^{-i} = (\lambda x^{L::K'}.M_1^{-i})M_2^{-i} \rightarrowtriangle_\beta M_1^{-i}[x^{L::K'} := M_2^{-i}] = (M_1[x^K := M_2])^{-i}$.
  - Let $M = \lambda x^K.M_1 \rightarrowtriangle_\beta \lambda x^K.N_1 = N$ such that $M_1 \rightarrowtriangle_\beta N_1$. Because $M \in \mathcal{M}_3$, $M_1 \in \mathcal{M}_3$ and $K \succeq \mathsf{deg}(M_1)$. By definition $\mathsf{deg}(M) = \mathsf{deg}(M_1)$. Because $i :: L = \mathsf{deg}(M_1) \preceq K$, $K = i :: L :: K'$ for some K'. By IH, $M_1^{-i} \rightarrowtriangle_\beta N_1^{-i}$, hence $M^{-i} = \lambda x^{L::K'}.M_1^{-i} \rightarrowtriangle_\beta \lambda x^{L::K'}.N_1^{-i} = N^{-i}$.
  - Let $M = M_1M_2 \rightarrowtriangle_\beta N_1M_2 = N$ such that $M_1 \rightarrowtriangle_\beta N_1$. Because $M \in \mathcal{M}_3$ then $M_1 \in \mathcal{M}_3$. By definition $\mathsf{deg}(M) = \mathsf{deg}(M_1) = i :: L$. By IH, $M_1^{-i} \rightarrowtriangle_\beta N_1^{-i}$, hence $M^{-i} = M_1^{-i}M_2^{-i} \rightarrowtriangle_\beta N_1^{-i}M_2^{-i} = N^{-i}$.
  - Let $M = M_1M_2 \rightarrowtriangle_\beta M_1N_2 = N$ such that $M_2 \rightarrowtriangle_\beta N_2$. Because $M \in \mathcal{M}_3$ then $M_2 \in \mathcal{M}_3$. By definition $\mathsf{deg}(M_2) \succeq \mathsf{deg}(M_1) = \mathsf{deg}(M) = i :: L$. So $\mathsf{deg}(M_2) = i :: L :: L'$ for some L'. By IH, $M_2^{-i} \rightarrowtriangle_\beta N_2^{-i}$, hence $M^{-i} = M_1^{-i}M_2^{-i} \rightarrowtriangle_\beta N_1^{-i}M_2^{-i} = N^{-i}$.
- Let $\to$ be $\rightarrowtriangle_\beta^*$. By induction on $\rightarrowtriangle_\beta^*$. using $\rightarrowtriangle_\beta$.
- Let $\to$ be $\rightarrowtriangle_\eta$. We only do the base case. The inductive cases are as for $\rightarrowtriangle_\beta$. Let $M = \lambda x^K.Nx^K \rightarrowtriangle_\eta N$ where $x^K \notin \mathsf{fv}(N)$. Because $i :: L = \mathsf{deg}(M) = \mathsf{deg}(N) \preceq K$ then $K = i :: L :: K'$ for some K'. By Lemma B.1.5.7, $N = N'^{+i}$ for some $N' \in \mathcal{M}_3$. By Lemma B.1.5.7, $N' = N^{-i}$. By Lemma B.1.5.1, $x^{L::K'} \notin \mathsf{fv}(N^{-i})$. Then $M^{-i} =$

$\lambda x^{L::K'}.N^{-i}x^{L::K'} \rightarrowtriangle_{\eta} N^{-i}$.

- Let $\rightarrow$ be $\rightarrowtriangle_{\eta}^{*}$. By induction on $\rightarrowtriangle_{\eta}^{*}$ using $\rightarrowtriangle_{\eta}$.
- Let $\rightarrow$ be $\rightarrowtriangle_{\beta\eta}$, $\rightarrowtriangle_{\beta\eta}$, $\rightarrowtriangle_{h}$ or $\rightarrowtriangle_{h}^{*}$. By the previous items.

8. By 1., $\mathsf{deg}(N^{+i}) = i :: \mathsf{deg}(N)$. By Lemma 7.1.11, $\mathsf{deg}(M) = \mathsf{deg}(N^{+i})$. By 7., $M = M'^{+i}$ such that $M' \in \mathcal{M}_3$. By 4., $M' = (M'^{+i})^{-i} = M^{-i}$. By 7., $M^{-i} \rightarrow (N^{+i})^{-i}$. By 4., $(N^{+i})^{-i} = N$.

9. By 1., $\mathsf{deg}(M^{+i}) = i :: \mathsf{deg}(M)$. By Lemma 7.1.11, $\mathsf{deg}(M^{+i}) = \mathsf{deg}(N)$. By 7., $N = N'^{+i}$ such that $N' \in \mathcal{M}_3$. By 4., $M = (M^{+i})^{-i}$ By 7., $(M^{+i})^{-i} \rightarrow N^{-i}$. By 4., $N^{-i} = (N'^{+i})^{-i} = N'$.

□

B.1.2 Confluence of $\rightarrowtriangle_{\beta}^{*}$ and $\rightarrowtriangle_{\beta\eta}^{*}$

In this section we establish the confluence of $\rightarrowtriangle_{\beta}^{*}$ and $\rightarrowtriangle_{\beta\eta}^{*}$ using the standard parallel reduction method.

Definition B.1.6. Let $r \in \{\beta, \beta\eta\}$. We define the binary relation $\overset{\rho_r}{\to}$ on $\mathcal{M}_i$, where $i \in \{1, 2, 3\}$, by:

(PR1) $M \overset{\rho_r}{\to} M$

(PR2) If $M \overset{\rho_r}{\to} M'$ and $\lambda x^I.M, \lambda x^I.M' \in \mathcal{M}_i$ then $\lambda x^I.M \overset{\rho_r}{\to} \lambda x^I.M'$.

(PR3) If $M \overset{\rho_r}{\to} M'$, $N \overset{\rho_r}{\to} N'$ and $MN, M'N' \in \mathcal{M}_i$ then $MN \overset{\rho_r}{\to} M'N'$

(PR4) If $M \overset{\rho_r}{\to} M'$, $N \overset{\rho_r}{\to} N'$ and $(\lambda x^I.M)N, M'[x^I := N'] \in \mathcal{M}_i$ then $(\lambda x^I.M)N \overset{\rho_r}{\to} M'[x^I := N']$

(PR5) If $M \overset{\rho_{\beta\eta}}{\to} M'$, $x^I \notin \mathsf{fv}(M)$ and $\lambda x^I.Mx^I \in \mathcal{M}_i$ then $\lambda x^I.Mx^I \overset{\rho_{\beta\eta}}{\to} M'$

We denote the transitive closure of $\overset{\rho_r}{\to}$ by $\overset{\rho_r}{\twoheadrightarrow}$. When $M \overset{\rho_r}{\to} N$ (resp. $M \overset{\rho_r}{\twoheadrightarrow} N$), we can also write $N \overset{\rho_r}{\leftarrow} M$ (resp. $N \overset{\rho_r}{\twoheadleftarrow} M$). If $\mathit{rel}, \mathit{rel}' \in \{\overset{\rho_r}{\to}, \overset{\rho_r}{\twoheadrightarrow}, \overset{\rho_r}{\leftarrow}, \overset{\rho_r}{\twoheadleftarrow}\}$, we write $M_1\ \mathit{rel}\ M_2\ \mathit{rel}'\ M_3$ instead of $M_1\ \mathit{rel}\ M_2$ and $M_2\ \mathit{rel}'\ M_3$. □

We now prove the relation between $\rightarrowtriangle_{r}$ for $r \in \{\beta, \beta\eta\}$ and $\overset{\rho_r}{\to}$.

Lemma B.1.7. *Let $r \in \{\beta, \beta\eta\}$, $i \in \{1, 2, 3\}$ and $M \in \mathcal{M}_i$.*

1. *If $M \rightarrowtriangle_{r} M'$ then $M \overset{\rho_r}{\to} M'$.*

2. *If $M \overset{\rho_r}{\to} M'$ then $M' \in \mathcal{M}_i$, $M \rightarrowtriangle_{r}^{*} M'$, $\mathsf{fv}(M') \subseteq \mathsf{fv}(M)$, $\mathsf{deg}(M) = \mathsf{deg}(M')$ and if $i \in \{1, 2\}$, $\mathsf{fv}(M') = \mathsf{fv}(M)$.*

3. *If $M \overset{\rho_r}{\to} M'$, $N \overset{\rho_r}{\to} N'$ and $M \diamond N$ then $M' \diamond N'$.* □

Proof of Lemma B.1.7.

1. By induction on the derivation of $M \twoheadrightarrow_r M'$ and then by case on the last rule used in the derivation. We prove the case where $M = (\lambda x^I.M_1)M_2 \twoheadrightarrow_\beta M_1[x^I := M_2] = M'$. such that $\mathsf{deg}(M_2) = I$ and $\forall I'.\ x^{I'} \notin \mathsf{fv}(M_2)$. By definition $M \in \mathcal{M}_i$ and $M_1, M_2 \in \mathcal{M}_i$. By Lemma B.1.1.1 and Lemma B.1.1.2, $M_1 \diamond M_2$. By Lemma B.1.1.5a, $M' \in \mathcal{M}_i$. Using rules $(\mathsf{PR1})$ and $(\mathsf{PR4})$

2. By induction on the derivation of $M \xrightarrow{\rho_r} M'$ using Lemmas 7.1.11 and B.1.2.4.

3. $M' \diamond N'$ since by 2., $\mathsf{fv}(M') \subseteq \mathsf{fv}(M)$ and $\mathsf{fv}(N') \subseteq \mathsf{fv}(N')$ and $M \diamond N$. □

Lemma B.1.8. *Let $r \in \{\beta, \beta\eta\}$, $i \in \{1, 2, 3\}$, $M, N \in \mathcal{M}_i$, $N \xrightarrow{\rho_r} N'$, $\mathsf{deg}(N) = I$, and $M \diamond N$. We have:*

1. *$M[x^I := N] \xrightarrow{\rho_r} M[x^I := N']$.*

2. *If $M \xrightarrow{\rho_r} M'$ then $M[x^I := N] \xrightarrow{\rho_r} M'[x^I := N']$.* □

Proof of Lemma B.1.8. By Lemma B.1.7.2, $\mathsf{deg}(N') = \mathsf{deg}(N) = I$ and $\mathsf{fv}(N') \subseteq \mathsf{fv}(N)$, and by Lemma B.1.7.3, $M \diamond N'$.

1. By Lemma B.1.1.5a, $M[x^I := N], M[x^I := N'] \in \mathcal{M}_i$.

   Let $i \in \{1, 2\}$. By induction on M:

   - Let $M = y^n$. If $x^I = y^n$ then $M[x^I := N] = N \xrightarrow{\rho_r} N' = M[x^I := N']$. If $x^I \neq y^n$ then $M[x^I := N] = M \xrightarrow{\rho_r} M = M[x^I := N']$.
   - Let $M = \lambda y^n.M_1$ such that $y^n \in \mathsf{fv}(M_1)$ and $\forall m.\ y^m \notin \mathsf{fv}(N)$. By Lemma B.1.1.2b, $M_1 \diamond N$. By IH, $M_1[x^I := N] \xrightarrow{\rho_r} M_1[x^I := N']$. Hence, $M[x^I := N] = \lambda y^n.M_1[x^I := N] \xrightarrow{\rho_r} \lambda y^n.M_1[x^I := N'] = M[x^I := N']$
   - Let $M = M_1M_2$ such that $M_1 \diamond M_2$. By Lemma B.1.1.2a, $\{M_1, M_2\} \diamond N$. By IH $M_1[x^I := N] \xrightarrow{\rho_r} M_1[x^I := N']$ and $M_2[x^I := N] \xrightarrow{\rho_r} M_2[x^I := N']$. Hence, $M[x^I := N] = M_1[x^I := N]M_2[x^I := N] \xrightarrow{\rho_r} M_1[x^I := N']M_2[x^I := N'] = M[x^I := N']$

   The proof for $i = 3$ is similar.

2. By Lemma B.1.7.3, $M' \diamond N'$. By induction on $M \xrightarrow{\rho_r} M'$ using 1., Lemmas B.1.1.2, B.1.1.3, B.1.1.5a, and B.1.7.3. We only consider one interesting case where $(\lambda y^J.M_1)M_2 \xrightarrow{\rho_\beta} M_1'[y^J := M_2']$, $M_1 \xrightarrow{\rho_\beta} M_1'$, $M_2 \xrightarrow{\rho_\beta} M_2'$, $(\lambda y^J.M_1)M_2, M_1'[y^J := M_2'] \in \mathcal{M}_i$, and $\forall J'.\ y^{J'} \notin \mathsf{fv}(N) \cup \{x^I\} \cup \mathsf{fv}(M_2)$. Because $(\lambda y^J.M_1)M_2 \in \mathcal{M}_i$, by definition, $M_1, M_2 \in \mathcal{M}_i$. By Lemma B.1.7.2, $M_1', M_2' \in \mathcal{M}_i$. By Lemma B.1.1.5a, $M_1' \diamond M_2'$ and $\mathsf{deg}(M_2') = J$. By Lemma B.1.1.2, $M_1 \diamond N$ and $M_2 \diamond N$. By Lemma B.1.7.3, $M_1' \diamond N$ and $M_2' \diamond N$. By Lemma B.1.7.3,

$M'_1 \diamond N'$ and $M'_2 \diamond N'$. By Lemma B.1.7.2, $\mathsf{deg}(N') = I$. By Lemma B.1.1.5a, $M_1[x^I := N], M_2[x^I := N], M'_1[x^I := N'], M'_2[x^I := N'] \in \mathcal{M}_i$. By Lemma B.1.1.2, $M_1 \diamond M_2$. By Lemma B.1.1.3. $M_1[x^I := N] \diamond M_2[x^I := N]$ and $M'_1[x^I := N'] \diamond M'_2[x^I := N']$. By Lemma B.1.1.5b, $\mathsf{deg}(M_1[x^I := N]) = \mathsf{deg}(M_1)$, $\mathsf{deg}(M_2[x^I := N]) = \mathsf{deg}(M_2)$, and $\mathsf{deg}(M'_2[x^I := N']) = \mathsf{deg}(M'_2) = J$. By Lemma B.1.1.5a, $M'_1[x^I := N'][y^J := M'_2[x^I := N']] \in \mathcal{M}_i$. Therefore $\lambda y^J.M_1[x^I := N] \in \mathcal{M}_i$ By Lemma B.1.1.2, $(\lambda y^J.M_1[x^I := N]) \diamond M_2[x^I := N]$. Therefore $(\lambda y^J.M_1[x^I := N])M_2[x^I := N] \in \mathcal{M}_i$. By Lemma B.1.1.6, $M'_1[x^I := N'][y^J := M'_2[x^I := N']] = M'_1[y^J := M'_2][x^I := N']$. Hence, $(\lambda y^J.M_1[x^I := N])M_2[x^I := N] \overset{\rho_\beta}{\to} M'_1[x^I := N'][y^J := M'_2[x^I := N']]$ and so, $((\lambda y^J.M_1)M_2)[x^I := N] \overset{\rho_\beta}{\to} M'_1[y^J := M'_2][x^I := N']$. □

Lemma B.1.9. *Let $r \in \{\beta, \beta\eta\}$, $i \in \{1, 2, 3\}$ and $M \in \mathcal{M}_i$.*

1. *If $M = x^I \overset{\rho_r}{\to} N$ then $N = x^I$.*
2. *If $M = \lambda x^I.P \overset{\rho_\beta}{\to} N$ then $N = \lambda x^I.P'$ where $P \overset{\rho_\beta}{\to} P'$.*
3. *If $M = \lambda x^I.P \overset{\rho_{\beta\eta}}{\to} N$ then one of the following holds:*
   - *$N = \lambda x^I.P'$ where $P \overset{\rho_{\beta\eta}}{\to} P'$.*
   - *$P = P'x^I$ where $x^I \notin \mathsf{fv}(P')$ and $P' \overset{\rho_{\beta\eta}}{\to} N$.*
4. *If $M = PQ \overset{\rho_r}{\to} N$ then one of the following holds:*
   - *$N = P'Q'$, $P \overset{\rho_r}{\to} P'$, $Q \overset{\rho_r}{\to} Q'$, $P \diamond Q$, and $P' \diamond Q'$.*
   - *$P = \lambda x^I.P'$, $N = P''[x^I := Q']$, $\mathsf{deg}(Q) = \mathsf{deg}(Q') = I$, $P' \overset{\rho_r}{\to} P''$, $Q \overset{\rho_r}{\to} Q'$, $P' \diamond Q$ and $P'' \diamond Q'$.*

□

Proof of Lemma B.1.9. 1. By induction on the derivation of $x^I \overset{\rho_r}{\to} N$.
2. By induction on the derivation of $\lambda x^I.P \overset{\rho_\beta}{\to} N$ using Lemma B.1.7.2.
3. By induction on the derivation of $\lambda x^I.P \overset{\rho_{\beta\eta}}{\to} N$ using Lemma B.1.7.2.
4. By induction on the derivation of $PQ \overset{\rho_r}{\to} N$ using Lemma B.1.7.2 and B.1.7.3. □

Lemma B.1.10. *Let $r \in \{\beta, \beta\eta\}$, $i \in \{1, 2, 3\}$ and $M, M_1, M_2 \in \mathcal{M}_i$.*

1. *If $M_2 \overset{\rho_r}{\leftarrow} M \overset{\rho_r}{\to} M_1$ then there exists $M' \in \mathcal{M}_i$ such that $M_2 \overset{\rho_r}{\to} M' \overset{\rho_r}{\leftarrow} M_1$.*
2. *If $M_2 \overset{\rho_r}{\twoheadleftarrow} M \overset{\rho_r}{\twoheadrightarrow} M_1$ then there exits $M' \in \mathcal{M}_i$ such that $M_2 \overset{\rho_r}{\twoheadrightarrow} M' \overset{\rho_r}{\twoheadleftarrow} M_1$.* □

Proof of Lemma B.1.10. 1. Both cases ($r = \beta$ and $r = \beta\eta$) are by induction on M. We only do the $\beta\eta$ case making discriminate use of Lemma B.1.9.

- If $M = x^I$, by Lemma B.1.9, $M_1 = M_2 = x^I$. Take $M' = x^I$.

- If $N_2P_2 \overset{\rho\beta\eta}{\leftarrow} NP \overset{\rho\beta\eta}{\rightarrow} N_1P_1$ where $N_2 \overset{\rho\beta\eta}{\leftarrow} N \overset{\rho\beta\eta}{\rightarrow} N_1$ and $P_2 \overset{\rho\beta\eta}{\leftarrow} P \overset{\rho\beta\eta}{\rightarrow} P_1$. Then, by IH, $\exists N', P' \in \mathcal{M}_i$ such that $N_2 \overset{\rho\beta\eta}{\rightarrow} N' \overset{\rho\beta\eta}{\leftarrow} N_1$ and $P_2 \overset{\rho\beta\eta}{\rightarrow} P' \overset{\rho\beta\eta}{\leftarrow} P_1$. By definition, $N_1 \diamond P_1$. By Lemma B.1.7.2, $\mathsf{deg}(N_1) = \mathsf{deg}(N')$ and $\mathsf{deg}(P_1) = \mathsf{deg}(P')$. By Lemma B.1.7.3, $N' \diamond P'$. If $i \in \{1, 2\}$ then $N'P' \in \mathcal{M}_i$. If $i = 3$ then $\mathsf{deg}(N_1) \preceq \mathsf{deg}(P_1)$, so $\mathsf{deg}(N') \preceq \mathsf{deg}(P')$ and $N'P' \in \mathcal{M}_i$. Hence, $N_2P_2 \overset{\rho\beta\eta}{\rightarrow} N'P' \overset{\rho\beta\eta}{\leftarrow} N_1P_1$.

- If $P_1[x^I := Q_1] \overset{\rho\beta\eta}{\leftarrow} (\lambda x^I.P)Q \overset{\rho\beta\eta}{\rightarrow} P_2[x^I := Q_2]$ where $P_1 \overset{\rho\beta\eta}{\leftarrow} P \overset{\rho\beta\eta}{\rightarrow} P_2$ and $Q_1 \overset{\rho\beta\eta}{\leftarrow} Q \overset{\rho\beta\eta}{\rightarrow} Q_2$. Then, by IH, $\exists P', Q' \in \mathcal{M}_i$ such that $P_1 \overset{\rho\beta\eta}{\rightarrow} P' \overset{\rho\beta\eta}{\leftarrow} P_2$ and $Q_1 \overset{\rho\beta\eta}{\rightarrow} Q' \overset{\rho\beta\eta}{\leftarrow} Q_2$. By Lemma B.1.1.5a, $\mathsf{deg}(Q_1) = \mathsf{deg}(Q_2) = I$, $P_1 \diamond Q_1$ and $P_2 \diamond Q_2$. Hence, by Lemma B.1.8.2, $P_1[x^I := Q_1] \overset{\rho\beta\eta}{\rightarrow} P'[x^I := Q'] \overset{\rho\beta\eta}{\leftarrow} P_2[x^I := Q_2]$.

- If $(\lambda x^I.P_1)Q_1 \overset{\rho\beta\eta}{\leftarrow} (\lambda x^I.P)Q \overset{\rho\beta\eta}{\rightarrow} P_2[x^I := Q_2]$ where $P \overset{\rho\beta\eta}{\rightarrow} P_1$, $P \overset{\rho\beta\eta}{\rightarrow} P_2$, $Q_1 \overset{\rho\beta\eta}{\leftarrow} Q \overset{\rho\beta\eta}{\rightarrow} Q_2$ and $\forall I'.\ x^{I'} \notin \mathsf{fv}(Q)$. By IH, $\exists P', Q' \in \mathcal{M}_i$ such that $P_1 \overset{\rho\beta\eta}{\rightarrow} P' \overset{\rho\beta\eta}{\leftarrow} P_2$ and $Q_1 \overset{\rho\beta\eta}{\rightarrow} Q' \overset{\rho\beta\eta}{\leftarrow} Q_2$. By Lemma B.1.1.1 and Lemma B.1.1.2b, $P \diamond Q$. By Lemma B.1.7.3, $P' \diamond Q'$. By Lemma B.1.1.5a, $\mathsf{deg}(Q_2) = I$ and $P_2 \diamond Q_2$. By Lemma B.1.7.2, $\mathsf{deg}(Q') = I$. By Lemma B.1.1.5a, $P'[x^I := Q'] \in \mathcal{M}_i$. Hence, $(\lambda x^n.P_1)Q_1 \overset{\rho\beta\eta}{\rightarrow} P'[x^n := Q']$ and by Lemma B.1.8.2, $P_2[x^n := Q_2] \overset{\rho\beta\eta}{\rightarrow} P'[x^n := Q']$.

- If $P_1Q_1 \overset{\rho\beta\eta}{\leftarrow} (\lambda x^I.Px^I)Q \overset{\rho\beta\eta}{\rightarrow} P_2[x^I := Q_2]$ where $P \overset{\rho\beta\eta}{\rightarrow} P_1$, $Px^I \overset{\rho\beta\eta}{\rightarrow} P_2$, $Q_1 \overset{\rho\beta\eta}{\leftarrow} Q \overset{\rho\beta\eta}{\rightarrow} Q_2$, and $\forall I'.\ x^{I'} \notin \mathsf{fv}(Q) \cup \mathsf{fv}(P)$. By Lemma B.1.1.5a, $\mathsf{deg}(Q_2) = I$. By Lemma B.1.7.2, $\mathsf{deg}(Q_1) = I$. By Lemma B.1.1.1 and Lemma B.1.1.2, $\diamond\{P, x^I, Q\}$. By Lemma B.1.7.3, $\diamond\{P_1, x^I, Q_1\}$. By Lemma B.1.7.2, $\mathsf{deg}(P) = \mathsf{deg}(P_1)$ and $x^I \notin \mathsf{fv}(P_1)$. If $i \in \{1, 2\}$ then $P_1x^I \in \mathcal{M}_i$. If $i = 3$ then $\mathsf{deg}(P) \preceq I$, so $\mathsf{deg}(P_1) \preceq I$ and $Px^I \in \mathcal{M}_i$. Hence $Px^I \diamond Q$ and by Lemma B.1.1.5a, $P_1Q_1 = (P_1x^I)[x^I := Q_1] \in \mathcal{M}_i$. Moreover, $Px^I \overset{\rho\beta\eta}{\rightarrow} P_1x^I$ and we conclude as in the third item.

- If $\lambda x^I.N_2 \overset{\rho\beta\eta}{\leftarrow} \lambda x^I.N \overset{\rho\beta\eta}{\rightarrow} \lambda x^I.N_1$ where $N_2 \overset{\rho\beta\eta}{\leftarrow} N \overset{\rho\beta\eta}{\rightarrow} N_1$. By IH, there is $N' \in \mathcal{M}_i$ such that $N_2 \overset{\rho\beta\eta}{\rightarrow} N' \overset{\rho\beta\eta}{\leftarrow} N_1$. If $i \in \{1, 2\}$ then $x^I \in \mathsf{fv}(N_1)$, so by Lemma B.1.7.2, $x^I \in \mathsf{fv}(N)$, hence $\lambda x^I.N' \in \mathcal{M}_i$. If $i = 3$ then by Lemma B.1.7.2, $I \succeq \mathsf{deg}(N_1) = \mathsf{deg}(N')$, so $\lambda x^I.N' \in \mathcal{M}_i$. Hence, $\lambda x^n.N_2 \overset{\rho\beta\eta}{\rightarrow} \lambda x^n.N' \overset{\rho\beta\eta}{\leftarrow} \lambda x^n.N_1$.

- If $M_1 \overset{\rho\beta\eta}{\leftarrow} \lambda x^I.Px^I \overset{\rho\beta\eta}{\rightarrow} M_2$ where $M_1 \overset{\rho\beta\eta}{\leftarrow} P \overset{\rho\beta\eta}{\rightarrow} M_2$. By IH, there is $M' \in \mathcal{M}_i$ such that $M_2 \overset{\rho\beta\eta}{\rightarrow} M' \overset{\rho\beta\eta}{\leftarrow} M_1$.

- If $M_1 \overset{\rho\beta\eta}{\leftarrow} \lambda x^I.Px^I \overset{\rho\beta\eta}{\rightarrow} \lambda x^I.P'$, where $P \overset{\rho\beta\eta}{\rightarrow} M_1$, $Px^I \overset{\rho\beta\eta}{\rightarrow} P'$ and $x^I \notin \mathsf{fv}(P)$. By the $\diamond$ property, for all J, $x^J \notin \mathsf{fv}(P)$. By Lemma B.1.9:

- Either $P' = P''x^I$ and $P \overset{\rho_{\beta\eta}}{\to} P''$. By IH, there is $M' \in \mathcal{M}_i$ such that $P'' \overset{\rho_{\beta\eta}}{\to} M' \overset{\rho_{\beta\eta}}{\leftarrow} M_1$. By Lemma B.1.7.2, $x^I \not\in \mathsf{fv}(P'')$ and $\mathsf{deg}(P'') \leq n$. Hence, $M_2 = \lambda x^I.P''x^I \overset{\rho_{\beta\eta}}{\to} M' \overset{\rho_{\beta\eta}}{\leftarrow} M_1$.
- Or $P = \lambda y^I.P''$ and $P' = P'''[y^I := x^I]$ such that $P'' \overset{\rho_{\beta\eta}}{\to} P'''$ and where $x \neq y$. If $i \in \{1,2\}$ then $y^I \in \mathsf{fv}(P'')$, so by Lemma B.1.7.2, $y^I \in \mathsf{fv}(P''')$ and $\lambda y^I.M''' \in \mathcal{M}_i$. If $i = 3$ then by Lemma B.1.7.2, $\mathsf{deg}(P''') = \mathsf{deg}(P'') \preceq I$ and for all J, $x^J \not\in \mathsf{fv}(P''')$. So $\lambda y^I.M''' \in \mathcal{M}_i$. Hence, $P = \lambda y^I.P'' \overset{\rho_{\beta\eta}}{\to} \lambda y^I.P'''$. Moreover, $\lambda x^I.P' = \lambda x^I.P'''[y^I := x^I] = \lambda y^I.P'''$. We conclude using as in the sixth item.

2. First show by induction on $M \overset{\rho_r}{\twoheadrightarrow} M_1$ (and using 1.) that if $M_2 \overset{\rho_r}{\twoheadleftarrow} M \overset{\rho_r}{\twoheadrightarrow} M_1$ then there is $M' \in \mathcal{M}_i$ such that $M_2 \overset{\rho_r}{\twoheadrightarrow} M' \overset{\rho_r}{\twoheadleftarrow} M_1$. Then use this to show 2. by induction on $M \overset{\rho_r}{\twoheadrightarrow} M_2$. □

Proof of Theorem 7.1.13.

1. By Lemma B.1.10.2, $\overset{\rho_r}{\twoheadrightarrow}$ is confluent. By Lemma B.1.7.1 and B.1.7.2, $M \overset{\rho_r}{\twoheadrightarrow} N$ iff $M \rightarrowtriangle^*_r N$. Then $\rightarrowtriangle^*_r$ is confluent.
2. $\Leftarrow$) is by definition of $\simeq_\beta$. $\Rightarrow$) is by induction on $M_1 \simeq_\beta M_2$ using 1. □

B.1.3 The types of the indexed calculi (Sec. 7.2)

Proof of Lemma 7.2.3. 1. The $\Rightarrow$) directions are by definition, and the $\Leftarrow$) directions are by induction on the derivations of $U \to T \in \mathsf{GITy}$ for 1a., of $U \sqcap V \in \mathsf{GITy}$ for 1b., and of $eU \in \mathsf{GITy}$ for 1c.

2. 2a. By induction on T.

   2b. By induction on U.

   * Let $U = U_1 \sqcap U_2$ such that $U_1, U_2 \in \mathsf{ITy}_2$. Because $\sqcap$ is commutative, let $\mathsf{deg}(U_1) = n$ and $\mathsf{deg}(U_2) = n'$ such that $n' \geq n$. By IH, $U_1 = \sqcap_{i=1}^{m}\vec{e}_{j(1:n),i}V_i$ and $U_2 = \sqcap_{i=m+1}^{m+m'}\vec{e}_{j(1:n'),i}V_i$ such that $m, m' \geq 1$, $\exists i \in \{1,\dots,m\}.\ V_i \in \mathsf{Ty}_2$, and $\exists i \in \{m+1,\dots,m'\}.\ V_i \in \mathsf{Ty}_2$. Let $\forall i \in \{1,\dots,m\}.\ V'_i = V_i$. Let $\forall i \in \{m{+}1,\dots,m{+}m'\}.\ V'_i = \vec{e}_{j(n+1:n'),i}V_i$. Therefore $U_1 \sqcap U_2 = \sqcap_{i=1}^{m+m'}\vec{e}_{j(1:n),i}V'_i$ and $m + m' \geq 1$.
   * Let $U = eU_1$ such that $U_1 \in \mathsf{ITy}_2$. Then $\mathsf{deg}(U) = n = n' + 1 = \mathsf{deg}(U_1) + 1$ By IH, $U_1 = \sqcap_{i=1}^{m}\vec{e}_{j(1:n'),i}V_i$ such that $m \geq 1$ and $\exists i \in \{1,\dots,m\}.\ V_i \in \mathsf{Ty}_2$. Therefore $U = \sqcap_{i=1}^{m}e\vec{e}_{j(1:n'),i}V_i$.
   * The case $U \in \mathsf{Ty}_2$ is trivial.

   2c. By induction on U.

- $*$ Let $U = U_1 \sqcap U_2$ then by 1b., $U_1, U_2 \in \mathsf{GITy}$ and $\mathsf{deg}(U_1) = \mathsf{deg}(U_2)$. By IH, $U_1 = \sqcap_{i=1}^{m} \vec{e}_{j(1:n),i} V_i$ and $U_2 = \sqcap_{i=m+1}^{m+m'} \vec{e}_{j(1:n),i} V_i$ such that $m, m' \geq 1$ and $\forall i \in \{1, \ldots, m'\}$. $V_i \in \mathsf{Ty}_2 \cap \mathsf{GITy}$. Therefore $U_1 \sqcap U_2 = \sqcap_{i=1}^{m+m'} \vec{e}_{j(1:n),i} V_i$.
- $*$ Let $U = eU_1$ then by 1c., $U_1 \in \mathsf{GITy}$. Also $\mathsf{deg}(U) = n = n' + 1 = \mathsf{deg}(U_1) + 1$ By IH, $U_1 = \sqcap_{i=1}^{m} \vec{e}_{j(1:n'),i} V_i$ such that $m \geq 1$ and $\forall i \in \{1, \ldots, m\}$. $V_i \in \mathsf{Ty}_2 \cap \mathsf{GITy}$. Therefore $U = \sqcap_{i=1}^{m} e\vec{e}_{j(1:n'),i} V_i$.
- $*$ The cases $U = U_1 \rightarrow T$ and $U = a$ are trivial.

2d. $\Leftarrow$) By 1. $\Rightarrow$) By 2., $\mathsf{deg}(U) \geq 0 = \mathsf{deg}(T)$. Hence, by 1., $U \rightarrow T \in \mathsf{GITy}$. □

B.1.4 The type systems $\vdash_1$ and $\vdash_2$ for $\lambda I^{\mathbb{N}}$ and $\vdash_3$ for $\lambda^{\mathcal{L}_{\mathbb{N}}}$ (Sec. 7.3)

Proof of Lemma 7.3.4. 1. By induction on the derivation $\Gamma \sqsubseteq \Gamma'$ and then by case on the last rule of the derivation.

- Let $\Gamma = \Gamma'$ using rule (ref) then use rule $(\sqsubseteq_{\mathsf{c}})$.
- Let $\Gamma \sqsubseteq \Gamma'$ be derived from $\Gamma \sqsubseteq \Gamma''$ and $\Gamma'' \sqsubseteq \Gamma'$ using rule (tr). By IH, $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ and $\Gamma, (x^I : U) \sqsubseteq \Gamma'', (x^I : U')$. Therefore $x^I \notin \mathsf{dom}(\Gamma'')$. Again by IH, $\mathsf{dom}(\Gamma'') = \mathsf{dom}(\Gamma')$ and $\Gamma'', (x^I : U') \sqsubseteq \Gamma', (x^I : U')$. Therefore, using rule (tr), $\Gamma, (x^I : U) \sqsubseteq \Gamma', (x^I : U')$. Also, $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$.
- Let $\Gamma = \Gamma_1, (y^{I'} : U_1) \sqsubseteq \Gamma_1, (y^{I'} : U_2) = \Gamma'$ be derived from $U_1 \sqsubseteq U_2$ and $y^{I'} \notin \mathsf{dom}(\Gamma_1)$ using rule $(\sqsubseteq_{\mathsf{c}})$. Therefore $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$ Using rule $(\sqsubseteq_{\mathsf{c}})$, $\Gamma, (x^I : U) = \Gamma_1, (y^{I'} : U_1), (x^I : U) \sqsubseteq \Gamma_1, (y^{I'} : U_1), (x^I : U')$. Using rule $(\sqsubseteq_{\mathsf{c}})$ again, $\Gamma_1, (y^{I'} : U_1), (x^I : U') \sqsubseteq \Gamma_1, (y^{I'} : U_2), (x^I : U') = \Gamma', (x^I : U')$. Therefore using rule (tr), $\Gamma \sqsubseteq \Gamma'$.

2. We prove the direction $\Rightarrow$) by induction on the size of the derivation $\Gamma \sqsubseteq \Gamma'$ and then by case on the last rule of the derivation.

- Let $\Gamma = \Gamma'$ using rule (ref) then we are done because $\Gamma = (x_i^{I_i} : U_i)_n$ and by rule (ref), $\forall i \in \{1, \ldots, n\}$. $U_i \sqsubseteq U_i$.
- Let $\Gamma \sqsubseteq \Gamma'$ be derived from $\Gamma \sqsubseteq \Gamma''$ and $\Gamma'' \sqsubseteq \Gamma'$ using rule (tr). By IH, $\Gamma = (x_i^{I_i} : U_i)_n$, $\Gamma'' = (x_i^{I_i} : U_i'')_n$, and $\forall i \in \{1, \ldots, n\}$. $U_i \sqsubseteq U_i''$. By IH again $\Gamma'' = (x_i^{I_i} : U_i'')_n$, $\Gamma' = (x_i^{I_i} : U_i')_n$, and $\forall i \in \{1, \ldots, n\}$. $U_i'' \sqsubseteq U_i'$. Therefore, using rule (tr), $\forall i \in \{1, \ldots, n\}$. $U_i \sqsubseteq U_i'$.
- Let $\Gamma, (x^I : U_1) \sqsubseteq \Gamma, (x^I : U_2)$ be derived from $U_1 \sqsubseteq U_2$ and $x^I \notin \mathsf{dom}(\Gamma)$ using rule $(\sqsubseteq_{\mathsf{c}})$ and we are done.

We prove the direction $\Leftarrow$) by induction on n. If $n = 0$ then it is done. Let $\Gamma = \Gamma_1, (x^{I_n} : U_n)$, $\Gamma' = \Gamma'_1, (x^{I_n} : U_n)$ and $\forall i \in \{1, \ldots, n\}$. $U_i \sqsubseteq U'_i$, such that $\Gamma_1 = (x_i^{I_i} : U_i)_m$ and $\Gamma'_1 = (x_i^{I_i} : U'_i)_m$. By IH, $\Gamma_1 \sqsubseteq \Gamma'_1$. By 1., $\Gamma \sqsubseteq \Gamma'$.

3. First we prove the direction $\Rightarrow$) by induction on the derivation of $\Gamma \vdash_j U \sqsubseteq \Gamma' \vdash_j U'$ and the by case on the last rule of the derivation.

   - Let $\Gamma \vdash_j U = \Gamma' \vdash_j U'$ using rule (ref) then it is done because $\Gamma = \Gamma'$ and $U = U'$ and by rule (ref), $\Gamma \sqsubseteq \Gamma$ and $U \sqsubseteq U$.
   - Let $\Gamma \vdash_j U \sqsubseteq \Gamma' \vdash_j U'$ be derived from $\Gamma \vdash_j U \sqsubseteq \Gamma'' \vdash_j U''$ and $\Gamma'' \vdash_j U'' \sqsubseteq \Gamma' \vdash_j U'$ using rule (tr). By IH, $\Gamma \sqsubseteq \Gamma''$, $\Gamma'' \sqsubseteq \Gamma'$, $U \sqsubseteq U''$, and $U'' \sqsubseteq U'$. Therefore using rule (tr), $\Gamma \sqsubseteq \Gamma'$ and $U \sqsubseteq U'$.
   - Let $\Gamma \vdash_j U \sqsubseteq \Gamma' \vdash_j U'$ using rule $(\sqsubseteq_{\langle\rangle})$ then we are done using the premises.

   The direction $\Leftarrow$) is obtained using rule $(\sqsubseteq_{\langle\rangle})$.

4. We prove this result by induction on the derivation of $U_1 \sqsubseteq U_2$ and then by case on the last rule of the derivation.

   - Case (ref) is trivial.
   - Let $U_1 \sqsubseteq U_2$ be derived from $U_1 \sqsubseteq U$ and $U \sqsubseteq U_2$ using rule (tr). By IH, $\mathsf{deg}(U_1) = \mathsf{deg}(U) = \mathsf{deg}(U_2)$ and ($U_1 \in \mathsf{GITy}$ iff $U \in \mathsf{GITy}$ iff $U_2 \in \mathsf{GITy}$).
   - Let $U_1 = U_2 \sqcap U \sqsubseteq U_2$ be derived from $\mathsf{deg}(U_2) = \mathsf{deg}(U)$ (and $U \in \mathsf{GITy}$ in ITy_2) using rule $(\sqcap_{\mathsf{E}})$. Then $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = \mathsf{deg}(U)$. Let $j = 2$. Using Lemma 7.2.3.1b, $U_1 \in \mathsf{GITy}$ iff $U_2 \in \mathsf{GITy}$.
   - Let $U_1 = U'_1 \sqcap U''_1 \sqsubseteq U'_2 \sqcap U''_2 = U_2$ be derived from $U'_1 \sqsubseteq U'_2$ and $U''_1 \sqsubseteq U''_2$ (and $\mathsf{deg}(U'_1) = \mathsf{deg}(U''_1)$ in ITy_3) using rule $(\sqcap)$. By IH, $\mathsf{deg}(U'_1) = \mathsf{deg}(U'_2)$, $\mathsf{deg}(U''_1) = \mathsf{deg}(U''_2)$, $U'_1 \in \mathsf{GITy}$ iff $U'_2 \in \mathsf{GITy}$, and $U''_1 \in \mathsf{GITy}$ iff $U''_2 \in \mathsf{GITy}$. In ITy_2, $\mathsf{deg}(U_1) = \mathsf{min}(\mathsf{deg}(U'_1), \mathsf{deg}(U''_1)) = \mathsf{min}(\mathsf{deg}(U'_2), \mathsf{deg}(U''_2)) = \mathsf{deg}(U_2)$. Also, using Lemma 7.2.3.1b, we prove $U_1 \in \mathsf{GITy}$ iff $U_2 \in \mathsf{GITy}$. In ITy_3, $\mathsf{deg}(U'_2) = \mathsf{deg}(U'_1) = \mathsf{deg}(U''_1) = \mathsf{deg}(U''_2)$ and $\mathsf{deg}(U_1) = \mathsf{deg}(U'_1) = \mathsf{deg}(U'_2) = \mathsf{deg}(U_2)$.
   - Let $U_1 = U'_1 \rightarrow T_1 \sqsubseteq U'_2 \rightarrow T_2 = U_2$ be derived from $U'_2 \sqsubseteq U'_1$ and $T_1 \sqsubseteq T_2$ using rule $(\rightarrow)$. By IH, $\mathsf{deg}(U'_1) = \mathsf{deg}(U'_2)$, $\mathsf{deg}(T_1) = \mathsf{deg}(T_2)$, $U'_1 \in \mathsf{GITy}$ iff $U'_2 \in \mathsf{GITy}$, and $T_1 \in \mathsf{GITy}$ iff $T_2 \in \mathsf{GITy}$. In ITy_2, $\mathsf{deg}(U_1) = \mathsf{min}(\mathsf{deg}(U'_1), \mathsf{deg}(T_1)) = \mathsf{min}(\mathsf{deg}(U'_2), \mathsf{deg}(T_2)) = \mathsf{deg}(U_2)$. Also, using Lemma 7.2.3.1a, we prove $U_1 \in \mathsf{GITy}$ iff $U_2 \in \mathsf{GITy}$. In ITy_3, $\mathsf{deg}(U_1) = \oslash = \mathsf{deg}(U_2)$.
   - Let $U_1 = eU'_1 \sqsubseteq eU'_2 = U_2$ be derived from $U'_1 \sqsubseteq U'_2$ using rule $(\sqsubseteq_{\mathsf{exp}})$. By IH, $\mathsf{deg}(U'_1) = \mathsf{deg}(U'_2)$ and $U'_1 \in \mathsf{GITy}$ iff $U'_2 \in \mathsf{GITy}$. In ITy_2, $\mathsf{deg}(U_1) =$

$\mathsf{deg}(U_1') + 1 = \mathsf{deg}(U_2') + 1 = \mathsf{deg}(U_2)$. Also using Lemma 7.2.3.1c, we prove $U_1 \in \mathsf{GITy}$ iff $U_2 \in \mathsf{GITy}$. In ITy_3, $\mathsf{deg}(U_1) = i :: \mathsf{deg}(U_1') = i :: \mathsf{deg}(U_2') = \mathsf{deg}(U_2)$.

5. We prove this result by induction on the derivation of $\Gamma_1 \sqsubseteq \Gamma_2$ and then by case on the last rule of the derivation.

   - Case (ref) is trivial.
   - Let $\Gamma_1 \sqsubseteq \Gamma_2$ be derived from $\Gamma_1 \sqsubseteq \Gamma$ and $\Gamma \sqsubseteq \Gamma_2$ using rule (tr). By IH, $\mathsf{deg}(\Gamma_1) = \mathsf{deg}(\Gamma) = \mathsf{deg}(\Gamma_2)$.
   - Let $\Gamma_1 = \Gamma, (x^I : U_1) \sqsubseteq \Gamma, (x^I : U_2) = \Gamma_2$ such that $x^I \notin \mathsf{fv}(\Gamma)$ be derived from $U_1 \sqsubseteq U_2$ using rule $(\sqsubseteq_{\mathsf{c}})$. We conclude using 5.

6. This result is proved by a simple induction on a derivation of the form $\Psi_1 \sqsubseteq \Psi_2$ and then by case on the last rule used in the derivation.

   The most interesting case is in ITy_3, if $U_1 = U_1' \sqcap U_1'' \sqsubseteq U_2' \sqcap U_2'' = U_2$ derived from $U_1' \sqsubseteq U_2'$, $U_1'' \sqsubseteq U_2''$, and $\mathsf{deg}(U_1') = \mathsf{deg}(U_1'')$ using rule $(\sqcap)$. To prove that $U_2' \sqcap U_2'' \in \mathsf{ITy}_3$ we need to prove that $\mathsf{deg}(U_2') = \mathsf{deg}(U_2'')$. This is obtained using 4.

7. We prove this result by induction on the derivation of $\Gamma_1 \sqsubseteq \Gamma_2$ and then by case on the last rule of the derivation.

   - If $\Gamma_1 = \Gamma_2$ is derived using rule (ref) then we are done.
   - Let $\Gamma_1 \sqsubseteq \Gamma_2$ be derived from $\Gamma_1 \sqsubseteq \Gamma$ and $\Gamma \sqsubseteq \Gamma_2$ using rule (tr). By IH, $\Gamma_1 \in \mathsf{GTyEnv} \Leftrightarrow \Gamma \in \mathsf{GTyEnv} \Leftrightarrow \Gamma_2 \in \mathsf{GTyEnv}$.
   - Let $\Gamma_1 = \Gamma, (y^n : U_1) \sqsubseteq \Gamma, (y^n : U_2) = \Gamma_2$ such that $y^n \notin \mathsf{dom}(\Gamma)$ be derived from $U_1 \sqsubseteq U_2$ using rule $(\sqsubseteq_{\mathsf{c}})$. If $\Gamma_1 \in \mathsf{GTyEnv}$ then $\Gamma \in \mathsf{GTyEnv}$ and $U_1 \in \mathsf{GITy}$. By 4., $U_2 \in \mathsf{GITy}$ and therefore $\Gamma_2 \in \mathsf{GTyEnv}$. This other direction is similar. □

Lemma B.1.11. *In the relevant context (*ITy_2*,* Ty_2*,* TyEnv_2 *or* Typing_2*), we have:*

1. *If $U \sqsubseteq V \sqcap a$ then $U = U' \sqcap a$.*
2. *Let $U_1 \sqsubseteq U_2$.*

   (a) *If $U_2 \in \mathsf{GITy}$ and $\mathsf{deg}(U_2) = n$ then $U_1 = \sqcap_{i=1}^{m} \vec{e}_{j(1:n),i} T_i$ and $U_2 = \sqcap_{i=1}^{m'} \vec{e'}_{j(1:n),i} T_i'$, such that $m, m' \geq 1$, $\forall i \in \{1, \ldots, m\}$. $T_i \in \mathsf{Ty}_2$, $\forall i \in \{1, \ldots, m'\}$. $T_i' \in \mathsf{Ty}_2$ and $\forall i \in \{1, \ldots, m'\}$. $\exists k \in \{1, \ldots, m\}$. $\vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T_k \sqsubseteq T_i'$.*

(b) *Let* $U_1 = \sqcap_{i=1}^{m}\vec{e}_{j(1:n_i),i}(V_i{\to}T_i)$ *and* $U_2 = \sqcap_{i=1}^{p}\vec{e'}_{j(1:m_i),i}(V'_i{\to}T'_i)$*. If* $U_1 \in \mathsf{GITy}$ *and* $\mathsf{deg}(U_1) = n$ *then* $\forall i \in \{1,\dots,m\}.\ \forall k \in \{1,\dots,p\}.\ n_i = m_k = n$ *and* $\forall k \in \{1,\dots,p\}.\ \exists i \in \{1,\dots,m\}.\ \vec{e}_{j(1:n),i} = \vec{e'}_{j(1:n),k} \wedge V'_k \sqsubseteq V_i \wedge T_i \sqsubseteq T'_k$*.*

3. *If* $eU \sqsubseteq V$ *then* $V = eU'$ *where* $U \sqsubseteq U'$*.*

4. *If* $U{\to}T \sqsubseteq V$ *and* $U{\to}T \in \mathsf{GITy}$ *then* $V = \sqcap_{i=1}^{p}(U_i{\to}T_i)$ *where* $p \geq 1$ *and* $\forall i \in \{1,\dots,p\}.\ U_i \sqsubseteq U \wedge T \sqsubseteq T_i$*.*

5. *If* $\sqcap_{i=1}^{m}\vec{e}_{j(1:n_i),i}(V_i{\to}T_i) \sqsubseteq V$ *where* $V \in \mathsf{GITy}$*,* $\mathsf{deg}(V) = n$ *and* $m \geq 1$ *then* $\forall i \in \{1,\dots,m\}.\ n_i = n$ *and* $V = \sqcap_{i=1}^{p}\vec{e'}_{j(1:n),i}(V'_i{\to}T'_i)$ *where* $p \geq 1$ *and* $\forall i \in \{1,\dots,p\}.\ \exists k \in \{1,\dots,m\}.\ \vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge V'_i \sqsubseteq V_k \wedge T_k \sqsubseteq T'_i$*.*

6. *If* $\Psi_1 \sqsubseteq \Psi_2$ *then* $\mathsf{deg}(\Psi_1) = \mathsf{deg}(\Psi_2)$ *and* Ψ_1 *is good iff* Ψ_2 *is good.*

7. *If* $U \sqsubseteq U'_1 \sqcap U'_2$ *then* $U = U_1 \sqcap U_2$ *where* $U_1 \sqsubseteq U'_1$ *and* $U_2 \sqsubseteq U'_2$*.*

8. *If* $\Gamma \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2$ *then* $\Gamma = \Gamma_1 \sqcap \Gamma_2$ *where* $\Gamma_1 \sqsubseteq \Gamma'_1$ *and* $\Gamma_2 \sqsubseteq \Gamma'_2$*.* □

Proof of Lemma B.1.11.

1. By induction on $U \sqsubseteq V \sqcap a$.

2. By induction on the derivation of $U_1 \sqsubseteq U_2$ using Lemmas 7.2.3.

   2a. By induction on the derivation of $U_1 \sqsubseteq U_2$ and then by case on the last rule of the derivation.

   ∗ Case (ref). The result is trivial using Lemma 7.2.3.2c.

   ∗ Case (tr). There exists U_3 such that $U_1 \sqsubseteq U_3$ and $U_3 \sqsubseteq U_2$. By Lemma 7.3.4.4, $U_1, U_3 \in \mathsf{GITy}$ and $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = \mathsf{deg}(U_3) = n$. By IH, $U_3 = \sqcap_{i=1}^{m_3}\vec{e''}_{j(1:n),i}T''_i$, $U_2 = \sqcap_{i=1}^{m_2}\vec{e'}_{j(1:n),i}T'_i$, where $m_2, m_3 \geq 1$, $\forall i \in \{1,\dots,m_3\}.\ T''_i \in \mathsf{Ty}_2$, $\forall i \in \{1,\dots,m_2\}.\ T'_i \in \mathsf{Ty}_2$ and $\forall i \in \{1,\dots,m_2\}.\ \exists k \in \{1,\dots,m_3\}.\ \vec{e''}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T''_k \sqsubseteq T'_i$. By IH again, $U_1 = \sqcap_{i=1}^{m_1}\vec{e}_{j(1:n),i}T_i$ where $m_1 \geq 1$, $\forall i \in \{1,\dots,m_1\}.\ T_i \in \mathsf{Ty}_2$ and $\forall i \in \{1,\dots,m_3\}.\ \exists k \in \{1,\dots,m_1\}.\ \vec{e}_{j(1:n),k} = \vec{e''}_{j(1:n),i} \wedge T_k \sqsubseteq T''_i$. Therefore $\forall i \in \{1,\dots,m_2\}.\ \exists k \in \{1,\dots,m_1\}.\ \vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T_k \sqsubseteq T'_i$ using rule (tr).

   ∗ Case ($\sqcap_{\mathsf{E}}$). There exists $U_3 \in \mathsf{GITy} \cap \mathsf{ITy}_2$ such that $U_1 = U_2 \sqcap U_3$ and $\mathsf{deg}(U_3) = \mathsf{deg}(U_2)$. Therefore, by Lemma 7.2.3.2c. $U_2 = \sqcap_{i=1}^{m}\vec{e}_{j(1:n),i}T_i$ such that $m \geq 1$ and $\forall i \in \{1,\dots,m\}.\ T_i \in \mathsf{Ty}_2$ and $U_3 = \sqcap_{i=m+1}^{m+m'}\vec{e}_{j(1:n),i}T_i$ such that $m' \geq 1$ and $\forall i \in \{m+1,\dots,m+m'\}.\ T_i \in \mathsf{Ty}_2$. Finally, we have $U_1 = U_2 \sqcap U_3 = \sqcap_{i=1}^{m+m'}\vec{e}_{j(1:n),i}T_i$ such that $m + m' \geq 1$ and $\forall i \in \{1,\dots,m+m'\}.\ T_i \in \mathsf{Ty}_2$, and trivially we have that $\forall i \in \{1,\ldots,m\}.\ \exists k \in \{1,\ldots,m+m'\}.\ \vec{e}_{j(1:n),k} = \vec{e}_{j(1:n),i} \wedge T_k \sqsubseteq T_i$ by picking $k = i$ for each i.

- $*$ Case ($\sqcap$). Then, $U_1 = U_1' \sqcap U_1''$, $U_2 = U_2' \sqcap U_2''$, $U_1' \sqsubseteq U_2'$, and $U_1'' \sqsubseteq U_2''$. By Lemma 7.2.3.2c, $U_2 = \sqcap_{i=1}^{m} \vec{e'}_{j(1:n),i} T_i'$ such that $m \geq 1$ and $\forall i \in \{1,\ldots,m\}.\ T_i' \in \mathsf{Ty}_2$. By Lemma 7.2.3.1b and Lemma 7.3.4.4, $U_1, U_2', U_2'', U_1', U_1'' \in \mathsf{GITy}$ and $\mathsf{deg}(U_2) = \mathsf{deg}(U_1) = \mathsf{deg}(U_2') = \mathsf{deg}(U_2'') = \mathsf{deg}(U_1') = \mathsf{deg}(U_1'') = n$. Because $\sqcap$ is commutative, let us choose that $m = m_1 + m_2$, $U_2' = \sqcap_{i=1}^{m_1} \vec{e'}_{j(1:n),i} T_i'$, and $U_2'' = \sqcap_{i=m_1+1}^{m_1+m_2} \vec{e'}_{j(1:n),i} T_i'$. We have that $m_1, m_2 \geq 1$. By IH, we obtain $U_1' = \sqcap_{i=1}^{m_1'} \vec{e}_{j(1:n),i} T_i$ and $U_1'' = \sqcap_{i=m_1'+1}^{m_1'+m_2'} \vec{e}_{j(1:n),i} T_i$ such that $m_1', m_2' \geq 1$, $\forall i \in \{1,\ldots,m_1'+m_2'\}.\ T_i \in \mathsf{Ty}_2$, $\forall i \in \{1,\ldots,m_1\}.\ \exists k \in \{1,\ldots,m_1'\}.\ \vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T_k \sqsubseteq T_i'$ and $\forall i \in \{m_1+1,\ldots,m_1+m_2\}.\ \exists k \in \{m_1'+1,\ldots,m_1'+m_2'\}.\ \vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T_k \sqsubseteq T_i'$. Therefore $U_1 = U_1' \sqcap U_1'' = \sqcap_{i=1}^{m_1'+m_2'} \vec{e}_{j(1:n),i} T_i$. Finally, one obtains that $\forall i \in \{1,\ldots,m_1+m_2\}.\ \exists k \in \{1,\ldots,m_1'+m_2'\}.\ \vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T_k \sqsubseteq T_i'$.
- $*$ Case ($\to$) is trivial.
- $*$ Case ($\sqsubseteq_{\mathsf{exp}}$). There exists U_1' and U_2' such that $U_1 = eU_1'$, $U_2 = eU_2'$ and $U_1' \sqsubseteq U_2'$. By Lemma 7.2.3.1c, $U_2' \in \mathsf{GITy}$. Also, $\mathsf{deg}(U_2) = n = n' + 1$ where $\mathsf{deg}(U_2') = n'$. By IH, we obtain $U_1' = \sqcap_{i=1}^{m} \vec{e}_{j(1:n'),i} T_i$, $U_2' = \sqcap_{i=1}^{m'} \vec{e'}_{j(1:n'),i} T_i'$, such that $m, m' \geq 1$, $\forall i \in \{1,\ldots,m\}.\ T_i \in \mathsf{Ty}_2$, $\forall i \in \{1,\ldots,m'\}.\ T_i' \in \mathsf{Ty}_2$, and also $\forall i \in \{1,\ldots,m'\}.\ \exists k \in \{1,\ldots,m\}.\ \vec{e}_{j(1:n'),k} = \vec{e'}_{j(1:n'),i} \wedge T_k \sqsubseteq T_i'$. Therefore, $U_1 = eU_1' = \sqcap_{i=1}^{m} e\vec{e}_{j(1:n'),i} T_i$, $U_2' = \sqcap_{i=1}^{m'} e\vec{e'}_{j(1:n'),i} T_i'$, and $\forall i \in \{1,\ldots,m'\}.\ \exists k \in \{1,\ldots,m\}.\ e\vec{e}_{j(1:n'),k} = e\vec{e'}_{j(1:n'),i} \wedge T_k \sqsubseteq T_i'$.

2b. We do case (tr): $\dfrac{\sqcap_{i=1}^{m} \vec{e}_{j(1:n_i),i}(V_i \to T_i) \sqsubseteq V \quad V \sqsubseteq \sqcap_{i=1}^{p} \vec{e'}_{j(1:m_i),i}(V_i' \to T_i')}{\sqcap_{i=1}^{m} \vec{e}_{j(1:n_i),i}(V_i \to T_i) \sqsubseteq \sqcap_{i=1}^{p} \vec{e'}_{j(1:m_i),i}(V_i' \to T_i')}$.

By Lemma 7.3.4.4, $V \in \mathsf{GITy}$ and $\mathsf{deg}(V) = n$. By 2a., we have $\forall i \in \{1,\ldots,m\}.\ n_i = n$ and $V = \sqcap_{i=1}^{q} \vec{e''}_{j(1:n),i} T_i''$ where $q \geq 1$, $\forall i \in \{1,\ldots,q\}.\ T_i'' \in \mathsf{Ty}_2$, and $\forall i \in \{1,\ldots,q\}.\ \exists k \in \{1,\ldots,m\}.\ \vec{e''}_{j(1:n),i} = \vec{e}_{j(1:n),k} \wedge V_k \to T_k \sqsubseteq T_i''$. If $T_i'' = a$ then, by 1., $V_i \to T_i = V' \sqcap a$. Absurd. Hence, $\forall i \in \{1,\ldots,q\}.\ T_i'' = W_i \to T_i'''$ and $V = \sqcap_{i=1}^{q} \vec{e''}_{j(1:n),i}(W_i \to T_i''')$. By IH, $\forall k \in \{1,\ldots,q\}.\ \exists i \in \{1,\ldots,m\}.\ \vec{e}_{j(1:n),i} = \vec{e''}_{j(1:n),k} \wedge W_k \sqsubseteq V_i \wedge T_i \sqsubseteq T_k'''$. Again by IH, $\forall i \in \{1,\ldots,p\}.\ m_j = m$ and $\forall k \in \{1,\ldots,p\}.\ \exists i \in \{1,\ldots,q\}.\ \vec{e''}_{j(1:n),i} = \vec{e'}_{j(1:n),k} \wedge V_k' \sqsubseteq W_i \wedge T_i''' \sqsubseteq T_k'$. Hence, $\forall k \in \{1,\ldots,p\}.\ \exists i \in \{1,\ldots,m\}.\ \vec{e'}_{j(1:n),k} = \vec{e}_{j(1:n),i} \wedge V_k' \sqsubseteq V_i \wedge T_i \sqsubseteq T_k'$.

3. By induction on $eU \sqsubseteq V$.

4. By 2a., $V = \sqcap_{i=1}^{p} T_i'$ where $p \geq 1$ and $\forall i \in \{1,\ldots,p\}.\ U \to T \sqsubseteq T_i'$. If $T_i' = a$ then, by 1., $U \to T = U' \sqcap a$. Absurd. Hence, $T_i' = U_i \to T_i$. Hence, by 2b.,

$\forall i \in \{1, \dots, p\}.\ U_i \sqsubseteq U \wedge T \sqsubseteq T_i$.

5. By 2a., $\forall i \in \{1, \dots, m\}.\ n_i = n$ and $V = \sqcap_{i=1}^{p} \vec{e'}_{j(1:n),i} T''_i$ where $p \geq 1$ and $\forall i \in \{1, \dots, p\}.\ \exists k \in \{1, \dots, m\}.\ \vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge V_k \to T_k \sqsubseteq T''_i$. Let $i \in \{1, \dots, p\}$. If $T''_i = a$ then, by 1., $V_k \to T_k = U' \sqcap a$. Absurd. Hence, $T''_i = V'_i \to T'_i$. Finally, By 4., $V'_i \sqsubseteq V_k$ and $T_{j_i} \sqsubseteq T'_i$.

6. Using previous items and Lemmas 7.3.4.4 and 7.3.4.7.

7. By induction on $U \sqsubseteq U'_1 \sqcap U'_2$.

   - Case (ref): Let $\overline{U'_1 \sqcap U'_2 \sqsubseteq U'_1 \sqcap U'_2}$.

     By rule (ref), $U'_1 \sqsubseteq U'_1$ and $U'_2 \sqsubseteq U'_2$.
   - Case (tr): Let $\dfrac{U \sqsubseteq U'' \quad U'' \sqsubseteq U'_1 \sqcap U'_2}{U \sqsubseteq U'_1 \sqcap U'_2}$.

     By IH, $U'' = U''_1 \sqcap U''_2$ such that $U''_1 \sqsubseteq U'_1$ and $U''_2 \sqsubseteq U'_2$. Again by IH, $U = U_1 \sqcap U_2$ such that $U_1 \sqsubseteq U''_1$ and $U_2 \sqsubseteq U''_2$. So by rule (tr), $U_1 \sqsubseteq U'_1$ and $U_2 \sqsubseteq U'_2$.
   - Case $(\sqcap_{\mathsf{E}})$: Let $\dfrac{U \in \mathsf{GITy} \quad \mathsf{deg}(U'_1 \sqcap U'_2) = \mathsf{deg}(U)}{(U'_1 \sqcap U'_2) \sqcap U \sqsubseteq U'_1 \sqcap U'_2}$.

     By rule (ref), $U'_1 \sqsubseteq U'_1$ and $U'_2 \sqsubseteq U'_2$. Moreover:

     * If $\mathsf{deg}(U) = \mathsf{deg}(U'_1 \sqcap U'_2) = \mathsf{deg}(U'_1)$ then by rule $(\sqcap_{\mathsf{E}})$, $U'_1 \sqcap U \sqsubseteq U'_1$. We are done.
     * If $\mathsf{deg}(U) = \mathsf{deg}(U'_1 \sqcap U'_2) = \mathsf{deg}(U'_2)$ then by rule $(\sqcap_{\mathsf{E}})$, $U'_2 \sqcap U \sqsubseteq U'_2$. We are done.
   - Case $(\sqcap)$: Let $\dfrac{U_1 \sqsubseteq U'_1 \quad U_2 \sqsubseteq U'_2}{U_1 \sqcap U_2 \sqsubseteq U'_1 \sqcap U'_2}$.

     Then we are done.
   - Case $(\sqsubseteq_{\mathsf{exp}})$: Let $\dfrac{U \sqsubseteq U'_1 \sqcap U'_2}{eU \sqsubseteq eU'_1 \sqcap eU'_2}$.

     By IH, $U = U_1 \sqcap U_2$ such that $U_1 \sqsubseteq U'_1$ and $U_2 \sqsubseteq U'_2$. So, $eU = eU_1 \sqcap eU_2$ and by rule $(\sqsubseteq_{\mathsf{exp}})$, $eU_1 \sqsubseteq eU'_1$ and $eU_2 \sqsubseteq eU'_2$.

8. By induction on $\Gamma \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2$.

   - Case (ref): Let $\overline{\Gamma'_1 \sqcap \Gamma'_2 \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2}$.

     By rule (ref), $\Gamma'_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$.
   - Case (tr): Let $\dfrac{\Gamma \sqsubseteq \Gamma'' \quad \Gamma'' \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2}{\Gamma \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2}$.

     By IH, $\Gamma'' = \Gamma''_1 \sqcap \Gamma''_2$ such that $\Gamma''_1 \sqsubseteq \Gamma'_1$ and $\Gamma''_2 \sqsubseteq \Gamma'_2$. Again by IH, $\Gamma = \Gamma_1 \sqcap \Gamma_2$ such that $\Gamma_1 \sqsubseteq \Gamma''_1$ and $\Gamma_2 \sqsubseteq \Gamma''_2$. So by rule (tr), $\Gamma_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$.

- Case ($\sqsubseteq_{\mathsf{c}}$): Let $\dfrac{U_1 \sqsubseteq U_2}{\Gamma, (y^n : U_1) \sqsubseteq \Gamma, (y^n : U_2)}$ where $\Gamma, (y^n : U_2) = \Gamma'_1 \sqcap \Gamma'_2$.
    - $*$ If $\Gamma'_1 = \Gamma''_1, (y^n : U'_2)$ and $\Gamma'_2 = \Gamma''_2, (y^n : U''_2)$ such that $U_2 = U'_2 \sqcap U''_2$ then by 7, $U_1 = U'_1 \sqcap U''_1$ such that $U'_1 \sqsubseteq U'_2$ and $U''_1 \sqsubseteq U''_2$. Hence $\Gamma = \Gamma''_1 \sqcap \Gamma''_2$ and $\Gamma, (y^n : U_1) = \Gamma_1 \sqcap \Gamma_2$ where $\Gamma_1 = \Gamma''_1, (y^n : U'_1)$ and $\Gamma_2 = \Gamma''_2, (y^n : U''_1)$ such that $\Gamma_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$ by rule ($\sqsubseteq_{\mathsf{c}}$).
    - $*$ If $y^n \notin \mathsf{dom}(\Gamma'_1)$ then $\Gamma = \Gamma'_1 \sqcap \Gamma''_2$ where $\Gamma''_2, (y^n : U_2) = \Gamma'_2$. Hence, $\Gamma, (y^n : U_1) = \Gamma'_1 \sqcap \Gamma_2$ where $\Gamma_2 = \Gamma''_2, (y^n : U_1)$. By rules ($\mathsf{ref}$) and ($\sqsubseteq_{\mathsf{c}}$), $\Gamma'_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$.
    - $*$ If $y^n \notin \mathsf{dom}(\Gamma'_2)$ then similar to the above case.

□

Lemma B.1.12. *In the relevant context (*ITy_3, Ty_3, TyEnv_3 *or* Typing_3*), we have:*

1. *If* $T \in \mathsf{Ty}_3$ *then* $\mathsf{deg}(T) = \oslash$.
2. *Let* $U \in \mathsf{ITy}_3$. *If* $\mathsf{deg}(U) = L = (n_i)_m$ *then* $U = \omega^L$ *or* $U = \vec{\mathsf{e}}_L \sqcap_{i=1}^p T_i$ *where* $p \geq 1$ *and* $\forall i \in \{1, \ldots, p\}.\ T_i \in \mathsf{Ty}_3$.
3. *Let* $U_1, U_2 \in \mathsf{ITy}_3$ *and* $U_1 \sqsubseteq U_2$.
    (a) *If* $U_1 = \omega^K$ *then* $U_2 = \omega^K$.
    (b) *If* $U_1 = \vec{\mathsf{e}}_K U$ *then* $U_2 = \vec{\mathsf{e}}_K U'$ *and* $U \sqsubseteq U'$.
    (c) *If* $U_2 = \vec{\mathsf{e}}_K U$ *then* $U_1 = \vec{\mathsf{e}}_K U'$ *and* $U \sqsubseteq U'$.
    (d) *If* $U_1 = \sqcap_{i=1}^p \vec{\mathsf{e}}_K (U_i \to T_i)$ *where* $p \geq 1$ *then* $U_2 = \omega^K$ *or* $U_2 = \sqcap_{j=1}^q \vec{\mathsf{e}}_K (U'_j \to T'_j)$ *where* $q \geq 1$ *and* $\forall j \in \{1, \ldots, q\}.\ \exists i \in \{1, \ldots, p\}.\ U'_j \sqsubseteq U_i \wedge T_i \sqsubseteq T'_j$.
4. *If* $U \in \mathsf{ITy}_3$ *and* $U \sqsubseteq U'_1 \sqcap U'_2$ *then* $U = U_1 \sqcap U_2$ *where* $U_1 \sqsubseteq U'_1$ *and* $U_2 \sqsubseteq U'_2$.
5. *If* $\Gamma \in \mathsf{TyEnv}_3$ *and* $\Gamma \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2$ *then* $\Gamma = \Gamma_1 \sqcap \Gamma_2$ *where* $\Gamma_1 \sqsubseteq \Gamma'_1$ *and* $\Gamma_2 \sqsubseteq \Gamma'_2$. □

Proof of Lemma B.1.12.

1. By definition.
2. By induction on U.
    - If $U = a$ ($\mathsf{deg}(U) = \oslash$), nothing to prove.
    - If $U = V \to T$ ($\mathsf{deg}(U) = \oslash$), nothing to prove.
    - If $U = \omega^L$, nothing to prove.
    - If $U = U_1 \sqcap U_2$ ($\mathsf{deg}(U) = \mathsf{deg}(U_1) = \mathsf{deg}(U_2) = L$), by IH we have four cases:

  - If $U_1 = U_2 = \omega^L$ then $U = \omega^L$.
  - If $U_1 = \omega^L$ and $U_2 = \vec{\mathsf{e}}_L \sqcap_{i=1}^k T_i$ where $k \geq 1$ and $\forall i \in \{1, \dots, k\}.\ T_i \in \mathsf{Ty}_3$ then $U = U_2$ (since ω^L is a neutral).
  - If $U_2 = \omega^L$ and $U_1 = \vec{\mathsf{e}}_L \sqcap_{i=1}^k T_i$ where $k \geq 1$ and $\forall i \in \{1, \dots, k\}.\ T_i \in \mathsf{Ty}_3$ then $U = U_1$ (since ω^L is a neutral).
  - If $U_1 = \vec{\mathsf{e}}_L \sqcap_{i=1}^p T_i$ and $U_2 = \vec{\mathsf{e}}_L \sqcap_{i=p+1}^{p+q} T_i$ where $p, q \geq 1$, $\forall i \in \{1, \dots, p+q\}.\ T_i \in \mathsf{Ty}_3$ then $U = \vec{\mathsf{e}}_L \sqcap_{i=1}^{p+q} T_i$.

- If $U = \mathsf{e}_{n_1} V$ ($L = \mathsf{deg}(U) = n_1 :: \mathsf{deg}(V) = n_1 :: K$), by IH we have two cases:
  - If $V = \omega^K$, $U = \mathsf{e}_{n_1}\omega^K = \omega^L$.
  - If $V = \vec{\mathsf{e}}_K \sqcap_{i=1}^p T_i$ where $p \geq 1$ and $\forall i \in \{1, \dots, p\}.\ T_i \in \mathsf{Ty}_3$ then $U = \vec{\mathsf{e}}_L \sqcap_{i=1}^p T_i$ where $p \geq 1$ and $\forall i \in \{1, \dots, p\}.\ T_i \in \mathsf{Ty}_3$.

3. 3a. By induction on $U_1 \sqsubseteq U_2$.

   3b. By induction on K. We do the induction step. Let $U_1 = \mathsf{e}_i U$. By induction on $\mathsf{e}_i U \sqsubseteq U_2$ we obtain $U_2 = \mathsf{e}_i U'$ and $U \sqsubseteq U'$.

   3c. Same proof as in the previous item.

   3d. By induction on the derivation of $U_1 \sqsubseteq U_2$ and then by case on the last rule of the derivation:

   - By rule (ref), $U_1 = U_2$.
   - Case (tr): Let $\dfrac{\sqcap_{i=1}^p \vec{\mathsf{e}}_K(U_i \to T_i) \sqsubseteq U \quad U \sqsubseteq U_2}{\sqcap_{i=1}^p \vec{\mathsf{e}}_K(U_i \to T_i) \sqsubseteq U_2}$.
     By IH, either $U = \omega^K$ and then by 3a., we obtain $U_2 = \omega^K$. Or $U = \sqcap_{j=1}^q \vec{\mathsf{e}}_K(U'_j \to T'_j)$ such that $q \geq 1$ and $\forall j \in \{1, \dots, q\}.\ \exists i \in \{1, \dots, p\}.\ U'_j \sqsubseteq U_i \wedge T_i \sqsubseteq T'_j$. Then by IH again, $U_2 = \omega^K$ or $U_2 = \sqcap_{k=1}^r \vec{\mathsf{e}}_K(U''_k \to T''_k)$ where $r \geq 1$ and $\forall k \in \{1, \dots, r\}.\ \exists j \in \{1, \dots, q\}.\ U''_k \sqsubseteq U'_j \wedge T'_j \sqsubseteq T''_k$. Finally, using rule (tr), we obtain $\forall k \in \{1, \dots, r\}.\ \exists i \in \{1, \dots, p\}.\ U''_k \sqsubseteq U_i \wedge T_i \sqsubseteq T''_k$.
   - By rule ($\sqcap_{\mathsf{E}}$), $U_2 = \omega^K$ or $U_2 = \sqcap_{j=1}^q \vec{\mathsf{e}}_K(U'_j \to T'_j)$ where $q \in \{1, \dots, p\}$ and $\forall j \in \{1, \dots, q\}.\ \exists i \in \{1, \dots, p\}.\ U_i = U'_j \wedge T_i = T'_j$.
   - Case ($\sqcap$) is by IH.
   - Case ($\to$) is trivial.
   - Case ($\sqsubseteq_{\mathsf{exp}}$): Let $\dfrac{\sqcap_{i=1}^p \vec{\mathsf{e}}_L(U_i \to T_i) \sqsubseteq U_2}{\sqcap_{i=1}^p \vec{\mathsf{e}}_K(U_i \to T_i) \sqsubseteq \mathsf{e}_i U_2}$ where $K = i :: L$.
     By IH, $U_2 = \omega^L$ and so $\mathsf{e}_i U_2 = \omega^K$ or $U_2 = \sqcap_{j=1}^q \vec{\mathsf{e}}_L(U'_j \to T'_j)$ so $\mathsf{e}_i U_2 = \sqcap_{j=1}^q \vec{\mathsf{e}}_K(U'_j \to T'_j)$ where $q \geq 1$ and $\forall j \in \{1, \dots, q\}.\ \exists i \in \{1, \dots, p\}.\ U'_j \sqsubseteq U_i \wedge T_i \sqsubseteq T'_j$.

4. By induction on $U \sqsubseteq U'_1 \sqcap U'_2$.

- Case (ref): Let $\overline{U'_1 \sqcap U'_2 \sqsubseteq U'_1 \sqcap U'_2}$. By rule (ref), $U'_1 \sqsubseteq U'_1$ and $U'_2 \sqsubseteq U'_2$.
- Case (tr): Let $\dfrac{U \sqsubseteq U'' \quad U'' \sqsubseteq U'_1 \sqcap U'_2}{U \sqsubseteq U'_1 \sqcap U'_2}$.

  By IH, $U'' = U''_1 \sqcap U''_2$ such that $U''_1 \sqsubseteq U'_1$ and $U''_2 \sqsubseteq U'_2$. Again by IH, $U = U_1 \sqcap U_2$ such that $U_1 \sqsubseteq U''_1$ and $U_2 \sqsubseteq U''_2$.

  So by rule (tr), $U_1 \sqsubseteq U'_1$ and $U_2 \sqsubseteq U'_2$.
- Case $(\sqcap_{\mathsf{E}})$: Let $\overline{(U'_1 \sqcap U'_2) \sqcap U \sqsubseteq U'_1 \sqcap U'_2}$.

  By rule (ref), $U'_1 \sqsubseteq U'_1$ and $U'_2 \sqsubseteq U'_2$. Moreover $\mathsf{deg}(U) = \mathsf{deg}(U'_1 \sqcap U'_2) = \mathsf{deg}(U'_1)$ then by rule $(\sqcap_{\mathsf{E}})$, $U'_1 \sqcap U \sqsubseteq U'_1$.
- Case $(\sqcap)$: Let $\dfrac{U_1 \sqsubseteq U'_1 \quad U_2 \sqsubseteq U'_2}{U_1 \sqcap U_2 \sqsubseteq U'_1 \sqcap U'_2}$.

  Then we are done.
- Case $(\sqcap)$: Let $\dfrac{V_2 \sqsubseteq V_1 \quad T_1 \sqsubseteq T_2}{V_1 \to T_1 \sqsubseteq V_2 \to T_2}$.

  Then $U'_1 = U'_2 = V_2 \to T_2$ and $U = U_1 \sqcap U_2$ such that $U_1 = U_2 = V_1 \to T_1$ and we are done.
- Case $(\sqsubseteq_{\mathsf{exp}})$: Let $\dfrac{U \sqsubseteq U'_1 \sqcap U'_2}{eU \sqsubseteq eU'_1 \sqcap eU'_2}$.

  Then by IH $U = U_1 \sqcap U_2$ such that $U_1 \sqsubseteq U'_1$ and $U_2 \sqsubseteq U'_2$. So, $eU = eU_1 \sqcap eU_2$ and by rule $(\sqsubseteq_{\mathsf{exp}})$, $eU_1 \sqsubseteq eU'_1$ and $eU_2 \sqsubseteq eU'_2$.

5. By induction on $\Gamma \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2$.

- Case (ref): Let $\overline{\Gamma'_1 \sqcap \Gamma'_2 \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2}$.

  By rule (ref), $\Gamma'_1 \sqsubseteq \Gamma'_1$ and $\Gamma'_2 \sqsubseteq \Gamma'_2$.
- Case (tr): Let $\dfrac{\Gamma \sqsubseteq \Gamma'' \quad \Gamma'' \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2}{\Gamma \sqsubseteq \Gamma'_1 \sqcap \Gamma'_2}$.

  By IH, $\Gamma'' = \Gamma''_1 \sqcap \Gamma''_2$ such that $\Gamma''_1 \sqsubseteq \Gamma'_1$ and $\Gamma''_2 \sqsubseteq \Gamma'_2$. Again by IH, $\Gamma = \Gamma_1 \sqcap \Gamma_2$ such that $\Gamma_1 \sqsubseteq \Gamma''_1$ and $\Gamma_2 \sqsubseteq \Gamma''_2$. So by rule (tr), $\Gamma_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$.
- Case $(\sqsubseteq_{\mathsf{c}})$: Let $\dfrac{U_1 \sqsubseteq U_2}{\Gamma, (y^L : U_1) \sqsubseteq \Gamma, (y^L : U_2)}$ where $\Gamma, (y^L : U_2) = \Gamma'_1 \sqcap \Gamma'_2$.
  - If $\Gamma'_1 = \Gamma''_1, (y^L : U'_2)$ and $\Gamma'_2 = \Gamma''_2, (y^L : U''_2)$ such that $U_2 = U'_2 \sqcap U''_2$ then by 4, $U_1 = U'_1 \sqcap U''_1$ such that $U'_1 \sqsubseteq U'_2$ and $U''_1 \sqsubseteq U''_2$. Hence $\Gamma = \Gamma''_1 \sqcap \Gamma''_2$ and $\Gamma, (y^L : U_1) = \Gamma_1 \sqcap \Gamma_2$ where $\Gamma_1 = \Gamma''_1, (y^L : U'_1)$ and $\Gamma_2 = \Gamma''_2, (y^L : U''_1)$ such that $\Gamma_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$ by rule $(\sqsubseteq_{\mathsf{c}})$.
  - If $y^L \notin \mathsf{dom}(\Gamma'_1)$ then $\Gamma = \Gamma'_1 \sqcap \Gamma''_2$ where $\Gamma''_2, (y^L : U_2) = \Gamma'_2$. Hence, $\Gamma, (y^L : U_1) = \Gamma'_1 \sqcap \Gamma_2$ where $\Gamma_2 = \Gamma''_2, (y^L : U_1)$. By rule (ref) and $(\sqsubseteq_{\mathsf{c}})$, $\Gamma'_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$.
  - If $y^L \notin \mathsf{dom}(\Gamma'_2)$ then similar to the above case.

□

Lemma B.1.13. *Let $j \in \{1,2,3\}$, $\Gamma, \Gamma_1, \Gamma_2 \in \mathsf{TyEnv}_j$ and $U, U_1, U_2 \in \mathsf{ITy}_j$.*

1. *Let $\mathsf{ok}(\Gamma)$, $\mathsf{ok}(\Gamma_1)$, and $\mathsf{ok}(\Gamma_2)$*
   - *(a) $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{TyEnv}_j$ and $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$.*
   - *(b) If $j \in \{1,2\}$ and $\Gamma_1, \Gamma_2 \in \mathsf{GTyEnv}$ then $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{GTyEnv}$.*
   - *(c) $e\Gamma \in \mathsf{TyEnv}_j$ and $\mathsf{ok}(e\Gamma)$.*
   - *(d) If $j \in \{1,2\}$ and $\Gamma \in \mathsf{GTyEnv}$ then $e\Gamma \in \mathsf{GTyEnv}$.*
   - *(e) If $j = 2$, $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$ and $\Gamma_1, \Gamma_2 \in \mathsf{GTyEnv}$ then $\Gamma_1 \sqcap \Gamma_2 \sqsubseteq \Gamma_1$.*
2. - *(a) If $((j = 2$ and $\mathsf{deg}(U) \geq I)$ or $(j = 3$ and $\mathsf{deg}(U) \succeq I))$ then $U^{-I} \in \mathsf{ITy}_j$.*
   - *(b) If $((j = 2$ and $\mathsf{deg}(\Gamma) \geq I)$ or $(j = 3$ and $\mathsf{deg}(\Gamma) \succeq I))$ then $\Gamma^{-I} \in \mathsf{TyEnv}_j$.*
3. *Let $j \in \{2,3\}$, $\Gamma_1 \sqsubseteq \Gamma_2$, and $U_1 \sqsubseteq U_2$.*
   - *(a) $\mathsf{ok}(\Gamma_1) \Leftrightarrow \mathsf{ok}(\Gamma_2)$.*
   - *(b) If $((j = 2$ and $U_1 \in \mathsf{GITy}$ and $\mathsf{deg}(U_1) \geq I)$ or $(j = 3$ and $\mathsf{deg}(U_1) \succeq I))$ then $U_1^{-I} \sqsubseteq U'^{-I}$.*
   - *(c) If $((j = 2$ and $\Gamma_1 \in \mathsf{GTyEnv}$ and $\mathsf{deg}(\Gamma_1) \geq I)$ or $(j = 3$ and $\mathsf{deg}(\Gamma_1) \succeq I))$ then $\Gamma_1^{-I} \sqsubseteq \Gamma_2^{-I}$.*
4. *Let $j \in \{2,3\}$ and $\Gamma_1 \diamond \Gamma_2$. If $((j = 2$, $\mathsf{deg}(\Gamma_1) \geq I$, and $\mathsf{deg}(\Gamma_2) \geq I)$ or $(j = 3$, $\mathsf{deg}(\Gamma_1) \succeq I$, and $\mathsf{deg}(\Gamma_2) \succeq I))$ then $\Gamma_1^{-I} \diamond \Gamma_2^{-I}$.*
5. *$\mathsf{ok}(\mathsf{env}^{\varnothing}_{M})$.* □

Proof of Lemma B.1.13.

1. Let $\Gamma_1 = (x_i^{I_i} : U_i) \uplus \Gamma_1'$ and $\Gamma_2 = (x_i^{I_i} : U_i') \uplus \Gamma_2'$ such that $\mathsf{dj}(\mathsf{dom}(\Gamma_1'), \mathsf{dom}(\Gamma_2'))$. Because $\mathsf{ok}(\Gamma_1)$ and $\mathsf{ok}(\Gamma_2)$ then $\mathsf{ok}(\Gamma_1')$, $\mathsf{ok}(\Gamma_2')$, and $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) = I_i = \mathsf{deg}(U_i')$. Therefore, $\Gamma_1 \sqcap \Gamma_2 = \{x^{I_i} \mapsto U_i \sqcap U_i' \mid i \in \{1, \ldots, n\}\} \cup \Gamma_1' \cup \Gamma_2'$.
   - 1a. In the case $j \in \{1,2\}$, we have $\forall i \in \{1, \ldots, n\}.\ U_i \sqcap U_i' \in \mathsf{ITy}_j$ therefore $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{TyEnv}_j$. In the case $j = 3$, we use the fact that $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) = \mathsf{deg}(U_i')$ to obtain $\forall i \in \{1, \ldots, n\}.\ U_i \sqcap U_i' \in \mathsf{ITy}_3$, and finally, $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{TyEnv}_3$.

     Because $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) = I_i = \mathsf{deg}(U_i')$ then we obtain $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i \sqcap U_i') = I_i$. Therefore $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$.
   - 1b. Because $\Gamma_1, \Gamma_2 \in \mathsf{GTyEnv}$ then by definition $\Gamma_1', \Gamma_2' \in \mathsf{GTyEnv}$ and $\forall i \in \{1, \ldots, n\}.\ U_i, U_i' \in \mathsf{GITy}$. Therefore $\forall i \in \{1, \ldots, n\}.\ U_i \sqcap U_i' \in \mathsf{GITy}$. Finally, we obtain $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{GTyEnv}$.

1c. Let $\Gamma = (x_i^{I_i} : U_i)_n$. By hypothesis, $\forall i \in \{1, \ldots, n\}$. $\mathsf{deg}(U_i) = I_i$. Let $j \in \{1, 2\}$. We have $e\Gamma = (x_i^{I_i+1} : eU_i)_n \in \mathsf{TyEnv}_j$. So $\forall i \in \{1, \ldots, n\}$. $\mathsf{deg}(eU_i) = \mathsf{deg}(U_i) + 1 = I_i + 1$. Let $j = 3$ and $e = \mathsf{e}_k$. We have $\mathsf{e}_k\Gamma = (x_i^{k::I_i} : \mathsf{e}_kU_i)_n \in \mathsf{TyEnv}_j$. So, $\forall i \in \{1, \ldots, n\}$. $\mathsf{deg}(\mathsf{e}_kU_i) = k :: \mathsf{deg}(U_i) = k :: I_i$.

1d. Let $\Gamma = (x_i^{I_i} : U_i)_n$. Because $\Gamma \in \mathsf{GTyEnv}$ then $\forall i \in \{1, \ldots, n\}$. $U_i \in \mathsf{GITy}$. Because $e\Gamma = (x_i^{I'_i} : eU_i)_n$. Therefore, $\forall i \in \{1, \ldots, n\}$. $eU_i \in \mathsf{GITy}$ and $e\Gamma \in \mathsf{GTyEnv}$.

1e. Let $\Gamma_1 = (x_i^{n_i} : U_i)_n$ and $\Gamma_2 = (x_i^{n_i} : V_i)_n$. By definition, we have $\forall i \in \{1, \ldots, n\}$. $\mathsf{deg}(U_i) = n_i = \mathsf{deg}(V_i) \wedge U_i, V_i \in \mathsf{GITy}$. Therefore, using rule $(\sqcap_{\mathsf{E}})$ $\forall i \in \{1, \ldots, n\}$. $U_i \sqcap V_i \sqsubseteq U_i$. We have $\Gamma_1 \sqcap \Gamma_2 = (x_i^{n_i} : U_i \sqcap V_i)_n$. Hence, by Lemma 7.3.4.2, $\Gamma_1 \sqcap \Gamma_2 \sqsubseteq \Gamma_1$.

2. 2a. Let $j = 2$ and $m = \mathsf{deg}(U) \geq I = n$. By Lemma 7.2.3.2b, U is of the form $\sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} V_i$ such that $k \geq 1$ and $\exists i \in \{1, \ldots, k\}$. $V_i \in \mathsf{Ty}_2$. Therefore $U^{-}n = \sqcap_{i=1}^{k} \vec{e}_{j(n:m),i} V_i \in \mathsf{ITy}_2$.

Let $j = 3$ and $K = \mathsf{deg}(U) \succeq I = L$. Therefore $K = L :: L'$. By Lemma B.1.12.2:

- ∗ Either $U = \omega^K$. Therefore, $U^{-L} = \omega^{L'} \in \mathsf{ITy}_3$.
- ∗ Or $U = \vec{\mathsf{e}}_K \sqcap_{i=1}^{p} T_i$ where $p \geq 1$ and $\forall i \in \{1, \ldots, p\}$. $T_i \in \mathsf{Ty}_3$. Therefore, $U^{-L} = \vec{\mathsf{e}}_{L'} \sqcap_{i=1}^{p} T_i \in \mathsf{ITy}_3$.

2b. Let $j = 2$, $m = \mathsf{deg}(\Gamma) \geq I = n$, and $\Gamma = (x_i^{n_i} : U_i)_p$. Therefore $\forall i \in \{1, \ldots, p\}$. $n_i \geq m \wedge \mathsf{deg}(U_i) \geq m$ and $\Gamma^{-n} = (x_i^{n_i - n} : U_i{}^{-}n)_p$. Using 2a., we obtain $\Gamma^{-n} \in \mathsf{TyEnv}_2$.

Let $j = 3$, $K = \mathsf{deg}(\Gamma) \succeq I = L$, and $\Gamma = (x_i^{L_i} : U_i)_p$. Therefore $\forall i \in \{1, \ldots, p\}$. $L_i \succeq K \succeq L \wedge L_i = L :: L'_i \wedge \mathsf{deg}(U_i) \succeq K \succeq L$ and $\Gamma^{-L} = (x_i^{L'_i} : U_i^{-L})_p$. Using 2a., we obtain $\Gamma^{-L} \in \mathsf{TyEnv}_3$.

3. 3a. By Lemma 7.3.4.2, $\Gamma_1 = (x_i^{I_i} : U_i)_n$ and $\Gamma_2 = (x_i^{I_i} : U'_i)_n$ and $\forall i \in \{1, \ldots, n\}$. $U_i \sqsubseteq U'_i$. By Lemma 7.3.4.4, $\forall i \in \{1, \ldots, n\}$. $\mathsf{deg}(U_i) = \mathsf{deg}(U'_i)$. Assume $\mathsf{ok}(\Gamma_1)$ then $\forall i \in \{1, \ldots, n\}$. $I_i = \mathsf{deg}(U_i) = \mathsf{deg}(U'_i)$, and so $\mathsf{ok}(\Gamma_2)$. Assume $\mathsf{ok}(\Gamma_2)$ then $\forall i \in \{1, \ldots, n\}$. $I_i = \mathsf{deg}(U'_i) = \mathsf{deg}(U_i)$, and so $\mathsf{ok}(\Gamma_1)$.

3b. Let $j = 2$. Let $\mathsf{deg}(U_1) = n$. By Lemma 7.3.4.4, $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = n$ and $U_1, U_2 \in \mathsf{GITy}$. Using Lemma B.1.11.2a we obtain $U_1 = \sqcap_{i=1}^{m} \vec{e}_{j(1:n),i} T_i$, $U_2 = \sqcap_{i=1}^{m'} \vec{e'}_{j(1:n),i} T'_i$, where $m, m' \geq 1$, $\forall i \in \{1, \ldots, m\}$. $T_i \in \mathsf{Ty}_2$, $\forall i \in \{1, \ldots, m'\}$. $T'_i \in \mathsf{Ty}_2$ and $\forall i \in \{1, \ldots, m'\}$. $\exists k \in \{1, \ldots, m\}$. $\vec{e}_{j(1:n),k} = \vec{e'}_{j(1:n),i} \wedge T_k \sqsubseteq T'_i$. Because $k = I \leq n$ then $U_1^{-k} = \sqcap_{i=1}^{m} \vec{e}_{j(k+1:n),i} T_i$ and $U_2^{-k} = \sqcap_{i=1}^{m'} \vec{e'}_{j(k+1:n),i} T'_i$. Because $U_1 \in \mathsf{GITy}$ then by Lemma 7.2.3.1, one can prove that $\forall i \in \{1, \ldots, m\}.\ T_i \in \mathsf{GITy}$. Therefore using rules ($\sqsubseteq_{\mathsf{exp}}$) and ($\sqcap_{\mathsf{E}}$), one can prove $U_1^{-I} \sqsubseteq U_2^{-I}$.

Let $j = 3$. Let $I = K$. Let $\mathsf{deg}(U_1) = L = K :: K'$. By Lemma B.1.12.2:

- If $U_1 = \omega^L$ then by Lemma B.1.12.3a, $U_2 = \omega^L$ and by rule (ref), $U_1^{-K} = \omega^{K'} \sqsubseteq \omega^{K'} = U_2^{-K}$.
- If $U_1 = \vec{\mathsf{e}}_L \sqcap_{i=1}^p T_i$ where $p \geq 1$ and $\forall i \in \{1, \ldots, p\}.\ T_i \in \mathsf{Ty}_3$, then by Lemma B.1.12.3b, $U_2 = \vec{\mathsf{e}}_L V$ and $\sqcap_{i=1}^p T_i \sqsubseteq V$. Hence, by rule ($\sqsubseteq_{\mathsf{exp}}$), $U_1^{-K} = \vec{\mathsf{e}}_{K'} \sqcap_{i=1}^p T_i \sqsubseteq \vec{\mathsf{e}}_{K'} V = U_2^{-K}$.

3c. By Lemma 7.3.4.2, $\Gamma_1 = (x_i^{I_i} : U_i)_n$, $\Gamma_2 = (x_i^{I_i} : U'_i)_n$, and $\forall i \in \{1, \ldots, n\}.\ U_i \sqsubseteq U'_i$. If $j = 2$ then because $\mathsf{deg}(\Gamma_1) \geq I = k$ and $\Gamma_1 \in \mathsf{GTyEnv}$, by definition we have $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) \geq k \wedge U_i \in \mathsf{GITy}$. If $j = 3$ then because $\mathsf{deg}(\Gamma_1) \succeq I = K$, by definition we have $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) \succeq K$. In both cases, by 3b., $\forall i \in \{1, \ldots, n\}.\ U_i^{-K} \sqsubseteq U_i'^{-I}$ and by Lemma 7.3.4.2, $\Gamma_1^{-I} \sqsubseteq \Gamma_2^{-I}$.

4. Let $x^{I_1} \in \mathsf{dom}(\Gamma_1^{-I})$ and $x^{I_2} \in \mathsf{dom}(\Gamma_2^{-I})$.

   If $j = 2$ then $x^{I+I_1} \in \mathsf{dom}(\Gamma_1)$ and $x^{I+I_2} \in \mathsf{dom}(\Gamma_2)$, hence $I + I_1 = I + I_2$ and so $I_1 = I_2$.

   If $j = 3$ then $x^{I::I_1} \in \mathsf{dom}(\Gamma_1)$ and $x^{I::I_2} \in \mathsf{dom}(\Gamma_2)$, hence $I :: I_1 = I :: I_2$ and so $I_1 = I_2$.

5. By definition, if $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ then $\mathsf{env}_M^{\emptyset} = (x_i^{L_i} : \omega^{L_i})_n$ and by definition, $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(\omega^{L_i}) = L_i$. □

Proof of Theorem 7.3.5. We prove 1. and 2. simultaneously. We prove the results by induction on the derivation $M : \langle \Gamma \vdash_j U \rangle$ and then by case on the last rule of the derivation.

First let us deal with the case where $i \in \{1, 2\}$.

- Let $x^n : \langle (x^n : T) \vdash_1 T \rangle$ such that $T \in \mathsf{GITy}$ and $\mathsf{deg}(T) = n$ be derived using rule (ax) (for system $\vdash_1$). We have $\mathsf{deg}(x^n) = n = \mathsf{deg}(T)$. By definition $x^n \in \mathbb{M}$.

- Let $x^0 : \langle (x^0 : T) \vdash_2 T \rangle$ such that $T \in \mathsf{GITy}$ using rule (ax) (for system $\vdash_2$). We have $\mathsf{deg}(x^0) = 0 = \mathsf{deg}(T)$ using Lemma 7.2.3.2a. By definition $x^0 \in \mathbb{M}$.

- Let $\lambda x^n.M : \langle \Gamma \vdash_i U \to T \rangle$ be derived from $M : \langle \Gamma, (x^n : U) \vdash_i T \rangle$ using rule ($\to_{\mathsf{I}}$) and where $\Gamma = (x_i^{I_i} : U_i)_n$. By IH, $M \in \mathcal{M}_i \cap \mathbb{M}$, $\Gamma, (x^n : U) \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$, $T \in \mathsf{ITy}_i \cap \mathsf{GITy}$, $\mathsf{deg}(U) \geq \mathsf{deg}(M) = \mathsf{deg}(T)$, $\mathsf{ok}(\Gamma)$, $\mathsf{deg}(U) = n$, $\mathsf{deg}(\Gamma) \geq \mathsf{deg}(M)$, and $\mathsf{dom}(\Gamma, (x^n : U)) = \mathsf{fv}(M)$. Therefore $x^n \in \mathsf{fv}(M)$ and we obtain $\lambda x^n.M \in \mathcal{M}_i \cap \mathbb{M}$. If $i = 2$ then $T \in \mathsf{Ty}_2$. Because $U \in \mathsf{GITy}$, we obtain

$U \to T \in \mathsf{ITy}_i \cap \mathsf{GITy}$. If $i = 2$ then $U \to T \in \mathsf{Ty}_2$. Also, $\Gamma \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$. By Lemma 7.2.3.2a, if $i = 2$ then $\mathsf{deg}(U \to T) = \mathsf{deg}(T) = 0$. We have $\mathsf{deg}(U \to T) = \mathsf{deg}(T) = \mathsf{deg}(M) = \mathsf{deg}(\lambda x^n.M)$. Because $\mathsf{dom}(\Gamma, (x^n : U)) = \mathsf{fv}(M)$ then $\mathsf{dom}(\Gamma) = \mathsf{fv}(\lambda x^n.M)$.

- Let $M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i T \rangle$ be derived from $M_1 : \langle \Gamma_1 \vdash_i U \to T \rangle$, $M_2 : \langle \Gamma_2 \vdash_i U \rangle$, and $\Gamma_1 \diamond \Gamma_2$ using rule $(\to_{\mathsf{E}})$. By IH, $M_1, M_2 \in \mathcal{M}_i \cap \mathbb{M}$, $\Gamma_1, \Gamma_2 \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$, $U \to T, U \in \mathsf{ITy}_i \cap \mathsf{GITy}$, $\mathsf{deg}(\Gamma_1) \geq \mathsf{deg}(M_1) = \mathsf{deg}(U \to T)$, $\mathsf{deg}(\Gamma_2) \geq \mathsf{deg}(M_2) = \mathsf{deg}(U)$, $\mathsf{ok}(\Gamma_1)$, $\mathsf{ok}(\Gamma_2)$, $\mathsf{dom}(\Gamma_1) = \mathsf{fv}(M_1)$, and $\mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_2)$. By Lemma 7.2.3.1a, $T \in \mathsf{ITy}_2 \cap \mathsf{GITy}$. If $i = 2$ then $U \to T, T \in \mathsf{Ty}_2$ and therefore by Lemma 7.2.3.2a, $\mathsf{deg}(U \to T) = \mathsf{deg}(T) = 0$. Because $\Gamma_1 \diamond \Gamma_2$, $\mathsf{dom}(\Gamma_1) = \mathsf{fv}(M_1)$, and $\mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_2)$ then $M_1 \diamond M_2$. Also, $\mathsf{deg}(M_1) = \mathsf{deg}(U \to T) \leq \mathsf{deg}(U) = \mathsf{deg}(M_2)$. Therefore $M_1 M_2 \in \mathcal{M}_i \cap \mathbb{M}$. Because $\mathsf{deg}(T) \leq \mathsf{deg}(U)$, we obtain $\mathsf{deg}(M_1 M_2) = \mathsf{deg}(M_1) = \mathsf{deg}(U \to T) = \mathsf{deg}(T)$. By Lemma B.1.13, $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$ and $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$. Because $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$, then $\mathsf{deg}(\Gamma_1 \sqcap \Gamma_2) = \mathsf{min}(\mathsf{deg}(\Gamma_1), \mathsf{deg}(\Gamma_2)) \geq \mathsf{min}(\mathsf{deg}(M_1), \mathsf{deg}(M_2)) = \mathsf{deg}(M_1 M_2)$. Finally, $\mathsf{dom}(\Gamma_1 \sqcap \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_1) \cup \mathsf{fv}(M_2) = \mathsf{fv}(M_1 M_2)$.

- Let $M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i U_1 \sqcap U_2 \rangle$ be derived from $M : \langle \Gamma_1 \vdash_i U_1 \rangle$ and $M : \langle \Gamma_2 \vdash_i U_2 \rangle$ using rule $(\sqcap_{\mathsf{I}})$. By IH, $M \in \mathcal{M}_i \cap \mathbb{M}$, $\Gamma_1, \Gamma_2 \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$ $U_1, U_2 \in \mathsf{ITy}_i \cap \mathsf{GITy}$, $\mathsf{deg}(\Gamma_1) \geq \mathsf{deg}(M) = \mathsf{deg}(U_1)$, $\mathsf{deg}(\Gamma_2) \geq \mathsf{deg}(M) = \mathsf{deg}(U_2)$, $\mathsf{ok}(\Gamma_1)$, $\mathsf{ok}(\Gamma_2)$, $\mathsf{dom}(\Gamma_1) = \mathsf{fv}(M) = \mathsf{dom}(\Gamma_2)$, and if $i = 2$ and $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) \geq k$ then $M^{-k} : \langle \Gamma_1^{-k} \vdash_2 U_1^{-k} \rangle$ and $M^{-k} : \langle \Gamma_2^{-k} \vdash_2 U_2^{-k} \rangle$. By Lemma B.1.13, $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$ and $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$. Because $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$, then $\mathsf{deg}(\Gamma_1 \sqcap \Gamma_2) = \mathsf{min}(\mathsf{deg}(\Gamma_1), \mathsf{deg}(\Gamma_2)) \geq \mathsf{deg}(M)$. Because $\mathsf{deg}(U_1) = \mathsf{deg}(U_2)$ then $U_1 \sqcap U_2 \in \mathsf{ITy}_i \cap \mathsf{GITy}$. We have $\mathsf{deg}(M) = \mathsf{deg}(U_1) = \mathsf{deg}(U_2) = \mathsf{deg}(U_1 \sqcap U_2)$. Also, $\mathsf{dom}(\Gamma_1 \sqcap \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) = \mathsf{fv}(M)$. Finally, let $i = 2$ and $k \in \{0, \ldots, \mathsf{deg}(M)\}$ ($\mathsf{deg}(M) = \mathsf{deg}(U_1 \sqcap U_2)$). We want to prove that $M^{-k} : \langle \Gamma_1 \sqcap \Gamma_2^{-k} \vdash_2 U_1 \sqcap U_2^{-k} \rangle$. By IH, $M^{-k} : \langle \Gamma_1^{-k} \vdash_2 U_1^{-k} \rangle$ and $M^{-k} : \langle \Gamma_2^{-k} \vdash_2 U_2^{-k} \rangle$. Therefore using rule $(\sqcap_{\mathsf{I}})$, $M^{-k} : \langle \Gamma_1^{-k} \sqcap \Gamma_2^{-k} \vdash_2 U_1^{-k} \sqcap U_2^{-k} \rangle$, and we have $\Gamma_1^{-k} \sqcap \Gamma_2^{-k} = \Gamma_1 \sqcap \Gamma_2^{-k}$ and $U_1^{-k} \sqcap U_2^{-k} = U_1 \sqcap U_2^{-k}$.

- Let $M^+ : \langle e\Gamma \vdash_i eU \rangle$ be derived from $M : \langle \Gamma \vdash_i U \rangle$ using rule (exp). By IH, $M \in \mathcal{M}_i \cap \mathbb{M}$, $\Gamma \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$, $U \in \mathsf{ITy}_i \cap \mathsf{GITy}$, $\mathsf{deg}(\Gamma) \geq \mathsf{deg}(M) = \mathsf{deg}(U)$, $\mathsf{ok}(\Gamma)$, $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$, and if $i = 2$ and $\mathsf{deg}(U) \geq k$ then $M^{-k} : \langle \Gamma^{-k} \vdash_2 U^{-k} \rangle$. By Lemma B.1.3.1d, $M \in \mathcal{M}_i \cap \mathbb{M}$. By Lemma B.1.13, $e\Gamma \in \mathsf{TyEnv}_i \cap \mathsf{GTyEnv}$ and $\mathsf{ok}(e\Gamma)$. By Lemma 7.2.3.1c, $eU \in \mathsf{ITy}_i \cap \mathsf{GITy}$. Also, using Lemma B.1.3.1a, $\mathsf{deg}(M^+) = \mathsf{deg}(M) + 1 = \mathsf{deg}(U) + 1 = \mathsf{deg}(eU)$ and $\mathsf{deg}(e\Gamma) = \mathsf{deg}(\Gamma) + 1 \geq \mathsf{deg}(M) + 1 = \mathsf{deg}(M^+)$. Let $\Gamma = (x_j^{n_j} : U_j)_n$ then $e\Gamma = (x_j^{n_j+1} : eU_j)_n$. Therefore $\mathsf{fv}(M) = \{x_j^{n_j} \mid j \in \{1, \ldots, n\}\}$ $\mathsf{dom}(e\Gamma) = \{x_j^{n_j+1} \mid 1 \in \{1, \ldots, n\}\} = \mathsf{fv}(M^+)$ using Lemma B.1.3.1a. Finally, let $i = 2$

and $k \in \{0, \dots, \mathsf{deg}(eU)\}$. Therefore $k \in \{0, \dots, \mathsf{deg}(U)+1\}$. If $k = 0$ then we are done. If $k = k' + 1$ such that $k' \in \{0, \dots, \mathsf{deg}(U)\}$ then $(M^+)^{-k} = (M^+)^{-k'+1} = M^{-k'}$ using Lemma B.1.3.1a, $(e\Gamma)^{-k} = (e\Gamma)^{-k'+1} = \Gamma^{-k'}$, and $(eU)^{-k} = (eU)^{-k'+1} = U^{-k'}$. Because $k' \in \{0, \dots, \mathsf{deg}(U)\}$ and by IH, we obtain $(M^+)^{-k} : \langle (e\Gamma)^{-k} \vdash_2 (eU)^{-k} \rangle$.

- Let $M : \langle \Gamma' \vdash_2 U' \rangle$ be derived from $M : \langle \Gamma \vdash_2 U \rangle$ and $\Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'$ using rule $(\sqsubseteq)$. By Lemma 7.3.4.3, $\Gamma' \sqsubseteq \Gamma$ and $U \sqsubseteq U'$. By IH, $M \in \mathcal{M}_2 \cap \mathbb{M}$, $\Gamma \in \mathsf{TyEnv}_2 \cap \mathsf{GTyEnv}$, $U \in \mathsf{ITy}_2 \cap \mathsf{GITy}$, $\mathsf{deg}(\Gamma) \geq \mathsf{deg}(M) = \mathsf{deg}(U)$, $\mathsf{ok}(\Gamma)$, $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$ and if $\mathsf{deg}(U) \geq k$ then $M^{-k} : \langle \Gamma^{-k} \vdash_2 U^{-k} \rangle$. By Lemma 7.3.4, $\Gamma' \in \mathsf{TyEnv}_2 \cap \mathsf{GTyEnv}$, $U' \in \mathsf{ITy}_2 \cap \mathsf{GITy}$, $\mathsf{deg}(\Gamma') = \mathsf{deg}(\Gamma) \geq \mathsf{deg}(M) = \mathsf{deg}(U) = \mathsf{deg}(U')$, and $\mathsf{dom}(\Gamma') = \mathsf{dom}(\Gamma) = \mathsf{fv}(M)$. By Lemma B.1.13.3a, $\mathsf{ok}(\Gamma')$. Let $k \in \{0, \dots, \mathsf{deg}(U')\}$ then because $\mathsf{deg}(U') = \mathsf{deg}(U)$ by IH, $M^{-k} : \langle \Gamma^{-k} \vdash_2 U^{-k} \rangle$. By Lemmas B.1.13.3b and B.1.13.3c, $\Gamma'^{-k} \sqsubseteq \Gamma^{-k}$ and $U^{-k} \sqsubseteq U'^{-k}$. By Lemma 7.3.4.3, $\Gamma^{-k} \vdash_2 U^{-k} \sqsubseteq \Gamma'^{-k} \vdash_2 U'^{-k}$. By Rule $(\sqsubseteq)$, $M^{-k} : \langle \Gamma'^{-k} \vdash_2 U'^{-k} \rangle$.

We now deal with the case where $i = 3$.

- Let $x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T \rangle$ be derived using rule (ax) (for system $\vdash_3$). By Lemma B.1.12.1 we have $\mathsf{deg}(x^{\oslash}) = \oslash = \mathsf{deg}(T)$.

- Let $M : \langle \mathsf{env}^{\emptyset}_M \vdash_3 \omega^{\mathsf{deg}(M)} \rangle$ be derived using rule (ω). By definition $M \in \mathcal{M}_3$, $\omega^{\mathsf{deg}(M)} \in \mathsf{ITy}_3$, and $\mathsf{dom}(\mathsf{env}^{\emptyset}_M) = \mathsf{fv}(M)$. It is easy to check that $\mathsf{env}^{\emptyset}_M \in \mathsf{TyEnv}_3$. We have $\mathsf{deg}(M) = \mathsf{deg}(\omega^{\mathsf{deg}(M)})$. By Lemma B.1.13.5, $\mathsf{ok}(\mathsf{env}^{\emptyset}_M)$. Let $\mathsf{env}^{\emptyset}_M = (x_i^{L_i} : \omega^{L_i})_n$ By Lemma B.1.1.4, $\forall i \in \{1, \dots, n\}.\ \mathsf{deg}(M) \preceq L_i$. Therefore, by definition of $\mathsf{deg}(\mathsf{env}^{\emptyset}_M) \succeq \mathsf{deg}(M)$. Finally, let $\mathsf{deg}(M) \succeq K$. We want to prove $M^{-K} : \langle (\mathsf{env}^{\emptyset}_M)^{-K} \vdash_3 (\omega^{\mathsf{deg}(M)})^{-K} \rangle$. We have $\mathsf{deg}(M) = K :: K'$ for some K'. By Lemma B.1.5, $M^{-K} \in \mathcal{M}_3$, $\mathsf{deg}(M^{-K}) = K'$, $\forall i \in \{1, \dots, n\}.\ L_i = K :: L'_i$, and $\mathsf{fv}(M^{-K}) = \{x^{L'_1}, \dots, x^{L'_n}\}$. We have $(\mathsf{env}^{\emptyset}_M)^{-K} = (x_i^{L'_i} : \omega^{L'_i})_n = \mathsf{env}^{\emptyset}_{M^{-K}}$. We also have $(\omega^{\mathsf{deg}(M)})^{-K} = (\omega^{K::K'})^{-K} = \omega^{K'} = \omega^{\mathsf{deg}(M^{-K})}$. Therefore, using rule (ω), $M^{-K} : \langle \mathsf{env}^{\emptyset}_{M^{-K}} \vdash_3 \omega^{\mathsf{deg}(M^{-K})} \rangle$.

- Let $\lambda x^L.M : \langle \Gamma \vdash_3 U \to T \rangle$ be derived from $M : \langle \Gamma, (x^L : U) \vdash_3 T \rangle$ using rule $(\to_{\mathsf{I}})$ and where $\Gamma = (x_i^{L_i} : U_i)_n$. By IH, $M \in \mathcal{M}_3$, $\Gamma, (x^L : U) \in \mathsf{TyEnv}_3$, $T \in \mathsf{ITy}_3$, $\mathsf{deg}(U) \succeq \mathsf{deg}(M) = \mathsf{deg}(T)$, $\mathsf{ok}(\Gamma)$, $\mathsf{deg}(U) = L$, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(T)$, and $\mathsf{dom}(\Gamma, (x^L : U)) = \mathsf{fv}(M)$. Therefore $x^L \in \mathsf{fv}(M)$. By hypothesis $T \in \mathsf{Ty}_3$. By Lemma B.1.12.1, we have $\mathsf{deg}(M) = \mathsf{deg}(T) = \oslash$. Therefore $\lambda x^L.M \in \mathcal{M}_3$. Because $\Gamma, (x^L : U) \in \mathsf{TyEnv}_3$, we have $\Gamma \in \mathsf{TyEnv}_3$ and $U \in \mathsf{ITy}_3$. We obtain $U \to T \in \mathsf{ITy}_3$. We have $\mathsf{deg}(U \to T) = \oslash = \mathsf{deg}(M) = \mathsf{deg}(\lambda x^L.M)$. Because $\mathsf{dom}(\Gamma, (x^L : U)) = \mathsf{fv}(M)$ then $\mathsf{dom}(\Gamma) = \mathsf{fv}(\lambda x^L.M)$. Finally, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(T) = \mathsf{deg}(U \to T)$.

- Let $\lambda x^L.M : \langle \Gamma \vdash_3 \omega^L \to T\rangle$ such that $x^L \notin \mathsf{dom}(\Gamma)$ be derived from $M : \langle \Gamma \vdash_3 T\rangle$ using rule $(\to'_{\mathsf{I}})$ and where $\Gamma = (x_i^{L_i} : U_i)_n$. By IH, $M \in \mathcal{M}_3$, $\Gamma \in \mathsf{TyEnv}_3$, $T \in \mathsf{ITy}_3$, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(T) = \mathsf{deg}(M)$, $\mathsf{ok}(\Gamma)$, and $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$. Therefore $x^L \notin \mathsf{fv}(M)$. By hypothesis $T \in \mathsf{Ty}_3$. By Lemma B.1.12.1, we have $\mathsf{deg}(M) = \mathsf{deg}(T) = \oslash$. Therefore $\lambda x^L.M \in \mathcal{M}_3$. We have $\omega^L \to T \in \mathsf{ITy}_3$. We have $\mathsf{deg}(\omega^L \to T) = \oslash = \mathsf{deg}(M) = \mathsf{deg}(\lambda x^L.M)$. Because $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$ and $x^L \notin \mathsf{fv}(M)$, we obtain $\mathsf{dom}(\Gamma) = \mathsf{fv}(\lambda x^L.M)$. Finally, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(T) = \mathsf{deg}(\omega^L \to T)$.

- Let $M_1M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T\rangle$ be derived from $M_1 : \langle \Gamma_1 \vdash_3 U \to T\rangle$, $M_2 : \langle \Gamma_2 \vdash_3 U\rangle$, and $\Gamma_1 \diamond \Gamma_2$ using rule $(\to_{\mathsf{E}})$. By IH, $M_1, M_2 \in \mathcal{M}_3$, $\Gamma_1, \Gamma_2 \in \mathsf{TyEnv}_3$, $U \to T, U \in \mathsf{ITy}_3$, $\mathsf{deg}(\Gamma_1) \succeq \mathsf{deg}(M_1) = \mathsf{deg}(U \to T)$, $\mathsf{deg}(\Gamma_2) \succeq \mathsf{deg}(M_2) = \mathsf{deg}(U)$, $\mathsf{ok}(\Gamma_1)$, $\mathsf{ok}(\Gamma_2)$, $\mathsf{dom}(\Gamma_1) = \mathsf{fv}(M_1)$, and $\mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_2)$. By hypothesis $U \to T \in \mathsf{Ty}_3$ and therefore $T \in \mathsf{Ty}_3$. By Lemma B.1.12.1, we have $\mathsf{deg}(M_1) = \mathsf{deg}(M_1 \to M_2) = \mathsf{deg}(T) = \oslash$. Because $\Gamma_1 \diamond \Gamma_2$, $\mathsf{dom}(\Gamma_1) = \mathsf{fv}(M_1)$, and $\mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_2)$ then $M_1 \diamond M_2$. Therefore $M_1M_2 \in \mathcal{M}_3$. We have $\mathsf{deg}(M_1M_2) = \mathsf{deg}(M_1) = \oslash = \mathsf{deg}(T)$. By Lemma B.1.13, $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{TyEnv}_3$ and $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$. We trivially have $\mathsf{deg}(\Gamma_1 \sqcap \Gamma_2) \succeq \mathsf{deg}(T) = \oslash$. Finally, $\mathsf{dom}(\Gamma_1 \sqcap \Gamma_2) = \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_1) \cup \mathsf{fv}(M_2) = \mathsf{fv}(M_1M_2)$.

- Let $M : \langle \Gamma \vdash_3 U_1 \sqcap U_2\rangle$ be derived from $M : \langle \Gamma \vdash_3 U_1\rangle$ and $M : \langle \Gamma \vdash_3 U_2\rangle$ using rule $(\sqcap_{\mathsf{I}})$. By IH, $M \in \mathcal{M}_3$, $\Gamma \in \mathsf{TyEnv}_3$ $U_1, U_2 \in \mathsf{ITy}_3$, $\mathsf{deg}(M) = \mathsf{deg}(U_1)$, $\mathsf{deg}(M) = \mathsf{deg}(U_2)$, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(M)$, $\mathsf{ok}(\Gamma)$,, $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$, and if $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) \succeq K$ then $M^{-K} : \langle \Gamma^{-K} \vdash_3 U_1^{-K}\rangle$ and $M^{-K} : \langle \Gamma^{-K} \vdash_3 U_2^{-K}\rangle$. Because $\mathsf{deg}(U_1) = \mathsf{deg}(U_2)$ then $U_1 \sqcap U_2 \in \mathsf{ITy}_3$. We have $\mathsf{deg}(M) = \mathsf{deg}(U_1) = \mathsf{deg}(U_2) = \mathsf{deg}(U_1 \sqcap U_2)$. Finally, let $\mathsf{deg}(U_1 \sqcap U_2) \succeq K$. Therefore $\mathsf{deg}(M) = \mathsf{deg}(U_1 \sqcap U_2) \succeq K$. We want to prove that $M^{-K} : \langle \Gamma^{-K} \vdash_2 U_1 \sqcap U_2^{-K}\rangle$. By IH, $M^{-K} : \langle \Gamma^{-K} \vdash_3 U_1^{-K}\rangle$ and $M^{-k} : \langle \Gamma^{-k} \vdash_3 U_2^{-K}\rangle$. Therefore using rule $(\sqcap_{\mathsf{I}})$, $M^{-K} : \langle \Gamma^{-K} \vdash_3 U_1^{-K} \sqcap U_2^{-K}\rangle$, and we have $U_1^{-K} \sqcap U_2^{-K} = U_1 \sqcap U_2^{-K}$.

- Let $M^{+j} : \langle \mathsf{e}_j\Gamma \vdash_3 \mathsf{e}_jU\rangle$ be derived from $M : \langle \Gamma \vdash_3 U\rangle$ using rule (exp). By IH, $M \in \mathcal{M}_3$, $\Gamma \in \mathsf{TyEnv}_3$, $U \in \mathsf{ITy}_3$, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(M) = \mathsf{deg}(U)$, $\mathsf{ok}(\Gamma)$, $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$, and if $\mathsf{deg}(U) \succeq K$ then $M^{-K} : \langle \Gamma^{-K} \vdash_3 U^{-K}\rangle$. By Lemma B.1.5.1, $M^{+j} \in \mathcal{M}_3$. By Lemma B.1.13, $\mathsf{e}_j\Gamma \in \mathsf{TyEnv}_3$ and $\mathsf{ok}(\mathsf{e}_j\Gamma)$. By definition $\mathsf{e}_jU \in \mathsf{ITy}_3$. Also, By Lemma B.1.5.1, $\mathsf{deg}(M^{+j}) = j :: \mathsf{deg}(M) = j :: \mathsf{deg}(U) = \mathsf{deg}(\mathsf{e}_jU)$. Let $\Gamma = (x_i^{L_i} : U_i)_n$. Because $\mathsf{ok}(\Gamma)$, $\forall i \in \{1, \dots, n\}$. $L_i = \mathsf{deg}(U_i)$. Therefore $\mathsf{e}_j\Gamma = (x_i^{j::L_i} : \mathsf{e}_jU_i)_n$. Because $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(U)$ then $\mathsf{deg}(\Gamma) = L$ and $\forall i \in \{1, \dots, n\}$. $L_i \succeq L$. Therefore $\forall i \in \{1, \dots, n\}$. $j :: L_i \succeq j :: L$. We then have $\mathsf{deg}(\mathsf{e}_j\Gamma) \succeq j :: L \succeq j :: \mathsf{deg}(U) = \mathsf{deg}(\mathsf{e}_jU)$. Also, $\mathsf{fv}(M) = \{x_1^{L_1}, \dots, x_n^{L_n}\}$ and so

$\mathsf{dom}(\mathsf{e}_j\Gamma) = \{x_1^{j::L_1}, \dots, x_n^{j::L_n}\} = \mathsf{fv}(M^{+j})$ using Lemma B.1.5.1. Finally, let $\mathsf{deg}(\mathsf{e}_jU) = j :: \mathsf{deg}(U) \succeq K$. If $K = \oslash$ then we are done. Otherwise $K = j :: K'$ for some K' such that $\mathsf{deg}(U) \succeq K'$. We have $(M^{+j})^{-K} = (M^{+j})^{-j::K'} = M^{-K'}$ using Lemma B.1.5.4, $(\mathsf{e}_j\Gamma)^{-K} = (\mathsf{e}_j\Gamma)^{-j::K'} = \Gamma^{-K'}$, and $(\mathsf{e}_jU)^{-K} = (\mathsf{e}_jU)^{-j::K'} = U^{-K'}$. Because $\mathsf{deg}(U) \succeq K'$ and by IH, we obtain $(M^{+j})^{-K} : \langle(\mathsf{e}_j\Gamma)^{-K} \vdash_3 (\mathsf{e}_jU)^{-K}\rangle$.

- Let $M : \langle\Gamma' \vdash_3 U'\rangle$ be derived from $M : \langle\Gamma \vdash_3 U\rangle$ and $\Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'$ using rule $(\sqsubseteq)$. By Lemma 7.3.4.3, $\Gamma' \sqsubseteq \Gamma$ and $U \sqsubseteq U'$. By IH, $M \in \mathcal{M}_3$, $\Gamma \in \mathsf{TyEnv}_3$, $U \in \mathsf{ITy}_3$, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(M) = \mathsf{deg}(U)$, $\mathsf{ok}(\Gamma)$, $\mathsf{dom}(\Gamma) = \mathsf{fv}(M)$ and if $\mathsf{deg}(U) \succeq K$ then $M^{-K} : \langle\Gamma^{-K} \vdash_3 U^{-K}\rangle$. By Lemma 7.3.4, $\Gamma' \in \mathsf{TyEnv}_3$, $U' \in \mathsf{ITy}_3$, $\mathsf{deg}(\Gamma') = \mathsf{deg}(\Gamma) \succeq \mathsf{deg}(M) = \mathsf{deg}(U) = \mathsf{deg}(U')$, and $\mathsf{dom}(\Gamma') = \mathsf{dom}(\Gamma) = \mathsf{fv}(M)$. By Lemma B.1.13.3a, $\mathsf{ok}(\Gamma')$. Let $\mathsf{deg}(U') \succeq K$ then because $\mathsf{deg}(U') = \mathsf{deg}(U)$ by IH, $M^{-K} : \langle\Gamma^{-K} \vdash_3 U^{-K}\rangle$. By Lemmas B.1.13.3b and B.1.13.3c, $\Gamma'^{-K} \sqsubseteq \Gamma^{-K}$ and $U^{-K} \sqsubseteq U'^{-K}$. By Lemma 7.3.4.3, $\Gamma^{-K} \vdash_3 U^{-K} \sqsubseteq \Gamma'^{-K} \vdash_3 U'^{-K}$. By Rule $(\sqsubseteq)$, $M^{-K} : \langle\Gamma'^{-K} \vdash_3 U'^{-K}\rangle$. □

Proof of Remark 7.3.6.

1. Let $M : \langle\Gamma_1 \vdash_3 U_1\rangle$ and $M : \langle\Gamma_2 \vdash_3 U_2\rangle$. By Theorem 7.3.5.2a, $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$. Let $\Gamma_1 = (x_i^{I_i} : V_i)_n$ and $\Gamma_2 = (x_i^{I_i} : V_i')_n$. By Theorem 7.3.5.2, $\forall i \in \{1, \dots, n\}$. $\mathsf{deg}(V_i) = \mathsf{deg}(V_i') = I_i$. By rule $(\sqcap_{\mathsf{E}})$, $V_i \sqcap V_i' \sqsubseteq V_i$ and $V_i \sqcap V_i' \sqsubseteq V_i'$. Hence, by Lemma 7.3.4.2, $\Gamma_1 \sqcap \Gamma_2 \sqsubseteq \Gamma_1$ and $\Gamma_1 \sqcap \Gamma_2 \sqsubseteq \Gamma_2$ and by rules $(\sqsubseteq)$ and $(\sqsubseteq_{\langle\rangle})$, $M : \langle\Gamma_1 \sqcap \Gamma_2 \vdash_3 U_1\rangle$ and $M : \langle\Gamma_1 \sqcap \Gamma_2 \vdash_3 U_2\rangle$. Finally, by rule $(\sqcap_{\mathsf{I}})$, $M : \langle\Gamma_1 \sqcap \Gamma_2 \vdash_3 U_1 \sqcap U_2\rangle$.

2. By Lemma 7.2.3.2, $U = \sqcap_{i=1}^m \vec{e}_{j(1:n),i}T_i$ where $m \geq 1$, and $\forall i \in \{1, \dots, m\}$. $T_i \in \mathsf{Ty}_2 \cap \mathsf{GITy}$. Let $i \in \{1, \dots, m\}$. By Lemma 7.2.3.2, $\mathsf{deg}(T_i) = 0$ and by rule (ax), $x^0 : \langle(x^0 : T_i) \vdash_2 T_i\rangle$. Hence, $x^n : \langle(x^n : \vec{e}_{j(1:n),i}T_i) \vdash_2 \vec{e}_{j(1:n),i}T_i\rangle$ by n applications of rule (exp). Now, by $m-1$ applications of $(\sqcap_{\mathsf{I}})$, $x^n : \langle(x^n : U) \vdash_2 U\rangle$.

3. By Lemma B.1.12, either $U = \omega^L$ so by rule (ω), $x^L : \langle(x^L : \omega^L) \vdash_3 \omega^L\rangle$. Or $U = \sqcap_{i=1}^p \vec{\mathsf{e}}_L T_i$ where $p \geq 1$, and $\forall i \in \{1, \dots, p\}$. $T_i \in \mathsf{Ty}_3$. Let $i \in \{1, \dots, p\}$. By rule (ax), $x^\oslash : \langle(x^\oslash : T_i) \vdash_3 T_i\rangle$, hence by rule (exp), $x^L : \langle(x^L : \vec{\mathsf{e}}_L T_i) \vdash_3 \vec{\mathsf{e}}_L T_i\rangle$. Now, by rule $(\sqcap_{\mathsf{I}}')$, $x^L : \langle(x^L : U) \vdash_3 U\rangle$.

4. By rule $(\sqcap_{\mathsf{E}})$ and since $\omega^{\mathsf{deg}(U)}$ is a neutral. □

B.1.5 Subject reduction and expansion properties of our type systems (Sec. 7.4)

Subject reduction and expansion properties for $\vdash_1$ and $\vdash_2$ (Sec.7.4.1)

Proof of Lemma 7.4.1. 1. By induction on the derivation of $x^n : \langle \Gamma \vdash_1 T \rangle$ and then by case on the last rule of the derivation.

- Case (ax): trivial.
- Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{x^n : \langle \Gamma_1 \vdash_1 U_1 \rangle \quad x^n : \langle \Gamma_2 \vdash_1 U_2 \rangle}{x^n : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_1 U_1 \sqcap U_2 \rangle}$.

  By IH, $\Gamma_1 = (x^n) : U_1$ and $\Gamma_2 = (x^n : U_2)$. Therefore $\Gamma_1 \sqcap \Gamma_2 = (x^n : U_1 \sqcap U_2)$
- Case (exp): Let $\dfrac{x^n : \langle \Gamma \vdash_1 U \rangle}{x^{n+1} : \langle e\Gamma \vdash_1 eU \rangle}$.

  By IH, $\Gamma = (x^n : U)$. Therefore $e\Gamma = (x^{n+1} : eU)$.

2. We prove this result by induction on the derivation of $\lambda x^n.M : \langle \Gamma \vdash_1 T_1 \to T_2 \rangle$ and then by case on the last rule of the derivation:

   - Case $(\to_{\mathsf{I}})$: Trivial.
   - Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{\lambda x^n.M : \langle \Delta \vdash_1 T_1 \to T_2 \rangle \quad \lambda x^n.M : \langle \Delta' \vdash_1 T_1 \to T_2 \rangle}{\lambda x^n.M : \langle \Delta \sqcap \Delta' \vdash_1 T_1 \to T_2 \rangle}$.

     By IH, $M : \langle \Delta, (x^n : T_1) \vdash_1 T_2 \rangle$ and $M : \langle \Delta', (x^n : T_2) \vdash_1 T_2 \rangle$. Using rule $(\sqcap_{\mathsf{I}})$, $M : \langle \Delta \sqcap \Delta', (x^n : T_2) \vdash_1 T_2 \rangle$.

3. By induction on the derivation of $MN : \langle \Gamma \vdash_1 T \rangle$ and then by case on the last rule of the derivation.

   - Case $(\to_{\mathsf{E}})$: Let $\dfrac{M : \langle \Gamma_1 \vdash_1 U \to T \rangle \quad N : \langle \Gamma_2 \vdash_1 U \rangle \quad \Gamma_1 \diamond \Gamma_2}{MN : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_1 T \rangle}$.

     Then we are done with $n = 1$, $m = 0$ and $T'_1 \to T_1 = U \to T$.
   - Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{MN : \langle \Gamma_1 \vdash_1 U_1 \rangle \quad MN : \langle \Gamma_2 \vdash_1 U_2 \rangle}{MN : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_1 U_1 \sqcap U_2 \rangle}$.

     By Theorem 7.3.5, $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = m$. By IH, $\Gamma_1 = \Gamma'_1 \sqcap \Gamma''_1$, $U_1 = \sqcap_{i=1}^{n_1} \vec{e}_{j(1:m),i} T_i$, $n_1 \geq 1$, $M : \langle \Gamma'_1 \vdash_1 \sqcap_{i=1}^{n_1} \vec{e}_{j(1:m),i}(T'_i \to T_i) \rangle$ and $N : \langle \Gamma''_1 \vdash_1 \sqcap_{i=1}^{n_1} \vec{e}_{j(1:m),i} T'_i \rangle$. Again by IH, $\Gamma_2 = \Gamma'_2 \sqcap \Gamma''_2$, $U_2 = \sqcap_{i=n_1+1}^{n_2} \vec{e}_{j(1:m),i} T_i$, $n_2 \geq 1$, $M : \langle \Gamma'_2 \vdash_1 \sqcap_{i=n_1+1}^{n_2} \vec{e}_{j(1:m),i}(T'_i \to T_i) \rangle$ and $N : \langle \Gamma''_2 \vdash_1 \sqcap_{i=n_1+1}^{n_2} \vec{e}_{j(1:m),i} T'_i \rangle$. Therefore $\Gamma_1 \sqcap \Gamma_2 = \Gamma'_1 \sqcap \Gamma'_2 \sqcap \Gamma''_1 \sqcap \Gamma''_2$, and $U_1 \sqcap U_2 = \sqcap_{i=1}^{n_2} \vec{e}_{j(1:m),i} T_i$. Finally, using rule $(\sqcap_{\mathsf{I}})$, $M : \langle \Gamma'_1 \sqcap \Gamma'_2 \vdash_1 \sqcap_{i=1}^{n_2} \vec{e}_{j(1:m),i}(T'_i \to T_i) \rangle$ and $N : \langle \Gamma''_1 \sqcap \Gamma''_2 \vdash_1 \sqcap_{i=1}^{n_2} \vec{e}_{j(1:m),i} T'_i \rangle$.
   - Case (exp): Let $\dfrac{MN : \langle \Gamma \vdash_1 U \rangle}{M^+N^+ : \langle e\Gamma \vdash_1 eU \rangle}$.

We have $m = \mathsf{deg}(eU) = \mathsf{deg}(U) + 1 = m' + 1$. By IH, $\Gamma = \Gamma_1 \sqcap \Gamma_2$, $U = \sqcap_{i=1}^{n} \vec{e}_{j(1:m'),i} T_i$, $n \geq 1$, $M : \langle \Gamma_1 \vdash_1 \sqcap_{i=1}^{n} \vec{e}_{j(1:m'),i}(T'_i \to T_i)\rangle$ and $N : \langle \Gamma_2 \vdash_1 \sqcap_{i=1}^{n} \vec{e}_{j(1:m'),i} T'_i\rangle$. Therefore, $e\Gamma = e\Gamma_1 \sqcap e\Gamma_2$, $eU = \sqcap_{i=1}^{n} e\vec{e}_{j(1:m'),i} T_i$, and using rule (exp), $M^+ : \langle e\Gamma_1 \vdash_1 \sqcap_{i=1}^{n} e\vec{e}_{j(1:m'),i}(T'_i \to T_i)\rangle$ and $N^+ : \langle e\Gamma_2 \vdash_1 \sqcap_{i=1}^{n} e\vec{e}_{j(1:m'),i} T'_i\rangle$.

□

Proof of Lemma 7.4.2. 1. By induction on the derivation of $x^n : \langle \Gamma \vdash_2 U\rangle$ and then by case on the last rule of the derivation.

- Case (ax): trivial.
- Case $(\sqcap_\mathsf{I})$: Let $\dfrac{x^n : \langle \Gamma_1 \vdash_2 U_1\rangle \quad x^n : \langle \Gamma_2 \vdash_2 U_2\rangle}{x^n : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 U_1 \sqcap U_2\rangle}$.

  By IH, $\Gamma_1 = (x^n) : U_1$ and $\Gamma_2 = (x^n : U_2)$. Therefore $\Gamma_1 \sqcap \Gamma_2 = (x^n : U_1 \sqcap U_2)$
- Case (exp): Let $\dfrac{x^n : \langle \Gamma \vdash_2 U\rangle}{x^{n+1} : \langle e\Gamma \vdash_2 eU\rangle}$.

  By IH, $\Gamma = (x^n : U)$. Therefore $e\Gamma = (x^{n+1} : eU)$.
- Case $(\sqsubseteq)$: Let $\dfrac{x^n : \langle \Gamma \vdash_2 U\rangle \quad \Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'}{x^n : \langle \Gamma' \vdash_2 U'\rangle}$.

  By IH, $\Gamma = (x^n : U)$. By Lemma 7.3.4, $\Gamma' = (x^n : U'')$ such that $U'' \sqsubseteq U$ and also $U \sqsubseteq U'$. Therefore using rule (tr), $U'' \sqsubseteq U'$.

2. By induction on the derivation of $\lambda x^n.M : \langle \Gamma \vdash_2 U\rangle$ and then by case on the last rule of the derivation. We have four cases:

- Case $(\to_\mathsf{I})$: If $\dfrac{M : \langle \Gamma, x^n : U \vdash_2 T\rangle}{\lambda x^n.M : \langle \Gamma \vdash_2 U \to T\rangle}$.

  We are done.
- Case $(\sqcap_\mathsf{I})$: Let $\dfrac{\lambda x^n.M : \langle \Gamma_1 \vdash_2 U_1\rangle \quad \lambda x^n.M : \langle \Gamma_2 \vdash_2 U_2\rangle}{\lambda x^n.M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 U_1 \sqcap U_2\rangle}$.

  By Theorem 7.3.5, $U_1 \sqcap U_2 \in \mathsf{GITy}$. $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = m$, $\Gamma_1, \Gamma_2 \in \mathsf{GTyEnv}$, and $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2)$. By Lemma B.1.13.1e, $\Gamma_1 \sqcap \Gamma_2 \sqsubseteq \Gamma_1$ and $\Gamma_1 \sqcap \Gamma_2 \sqsubseteq \Gamma_2$. By IH we have: $U_1 = \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i}(V_i \to T_i)$, $U_2 = \sqcap_{i=k+1}^{k+l} \vec{e}_{j(1:m),i}(V_i \to T_i)$, $\forall i \in \{1, \ldots, k\}$. $M : \langle \Gamma_1, x^n : \vec{e}_{j(1:m),i} V_i \vdash_2 \vec{e}_{j(1:m),i} T_i\rangle$, and $\forall i \in \{k{+}1, \ldots, k{+}l\}$. $M : \langle \Gamma_2, x^n : \vec{e}_{j(1:m),i} V_i \vdash_2 \vec{e}_{j(1:m),i} T_i\rangle$. Hence $U_1 \sqcap U_2 = \sqcap_{i=1}^{k+l} \vec{e}_{j(1:m),i}(V_i \to T_i)$, where $k, l \geq 1$ and by Lemma 7.3.4 and rule $(\sqsubseteq)$, $\forall i \in \{1, \ldots, k{+}l\}$. $M : \langle \Gamma_1 \sqcap \Gamma_2, x^n : \vec{e}_{j(1:m),i} V_i \vdash_2 \vec{e}_{j(1:m),i} T_i\rangle$.
- Case (exp): Let $\dfrac{\lambda x^n.M : \langle \Gamma \vdash_2 U\rangle}{\lambda x^{n+1}.M^+ : \langle e\Gamma \vdash_2 eU\rangle}$.

By IH, because $\mathsf{deg}(U) = m - 1$, $U = \sqcap_{i=1}^{k}\vec{e}_{j(1:m-1),i}(V_i \to T_i)$ where $k \geq 1$ and $\forall i \in \{1, \ldots, k\}$. $M : \langle \Gamma, x^n : \vec{e}_{j(1:m-1),i}V_i \vdash_2 \vec{e}_{j(1:m-1),i}T_i \rangle$. Therefore $eU = \sqcap_{i=1}^{k} e\vec{e}_{j(1:m-1),i}(V_i \to T_i)$ and by rule (exp), $\forall i \in \{1, \ldots, k\}$. $M^+ : \langle \Gamma, x^{n+1} : e\vec{e}_{j(1:m-1),i}V_i \vdash_3 e\vec{e}_{j(1:m-1),i}T_i \rangle$.

- Case ($\sqsubseteq$): Let $\dfrac{\lambda x^n.M : \langle \Gamma \vdash_2 U \rangle \quad \Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'}{\lambda x^n.M : \langle \Gamma' \vdash_2 U' \rangle}$.

  By Lemma 7.3.4.3, $\Gamma' \sqsubseteq \Gamma$ and $U \sqsubseteq U'$. By Theorem 7.3.5, $U, U' \in \mathsf{GITy}$ and $\mathsf{deg}(U) = \mathsf{deg}(U') = m$. By IH, $U = \sqcap_{i=1}^{k}\vec{e}_{j(1:m),i}(V_i \to T_i)$, where $k \geq 1$ and $\forall i \in \{1, \ldots, k\}$. $M : \langle \Gamma, x^n : \vec{e}_{j(1:m),i}V_i \vdash_2 \vec{e}_{j(1:m),i}T_i \rangle$. By Lemma B.1.11.5, $U' = \sqcap_{i=1}^{p}\vec{e'}_{j(1:m),i}(V'_i \to T'_i)$, where $p \geq 1$, and $\forall i \in \{1, \ldots, p\}$. $\exists l \in \{1, \ldots, k\}$. $\vec{e}_{j(1:m),l} = \vec{e'}_{j(1:m),i} \wedge V'_i \sqsubseteq V_l \wedge T_l \sqsubseteq T'_i$. Let $i \in \{1, \ldots, p\}$. Because by Lemma 7.3.4 $\Gamma, x^n : \vec{e}_{j(1:m),l}V_l \vdash_2 \vec{e}_{j(1:m),l}T_l \sqsubseteq \Gamma', x^n : \vec{e'}_{j(1:m),i}V'_i \vdash_2 \vec{e'}_{j(1:m),i}T'_i$, then using rule ($\sqsubseteq$) we obtain $M : \langle \Gamma', x^n : \vec{e'}_{j(1:m),i}V'_i \vdash_2 \vec{e'}_{j(1:m),i}T'_i \rangle$.

3. By induction on the derivation of $MN : \langle \Gamma \vdash_2 U \rangle$ and then by case on the last rule of the derivation.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{M : \langle \Gamma_1 \vdash_i U \to T \rangle \quad N : \langle \Gamma_2 \vdash_i U \rangle \quad \Gamma_1 \diamond \Gamma_2}{MN : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i T \rangle}$.

  Then we are done by taking $k = 1$ and because by Lemma 7.2.3.2a, $\mathsf{deg}(T) = m = 0$.

- Case (exp): Let $\dfrac{MN : \langle \Gamma \vdash_i U \rangle}{(MN)^+ : \langle e\Gamma \vdash_i eU \rangle}$.

  We have $MN^+ = M^+N^+$ and $\mathsf{deg}(eU) = m = \mathsf{deg}(U) + 1 = m' + 1$. By IH, $U = \sqcap_{i=1}^{k}\vec{e}_{j(1:m'),i}T_i$ where $k \geq 1$, $\Gamma = \Gamma_1 \sqcap \Gamma_2$, $M : \langle \Gamma_1 \vdash_2 \sqcap_{i=1}^{k}\vec{e}_{j(1:m'),i}(U_i \to T_i) \rangle$, and $N : \langle \Gamma_2 \vdash_2 \sqcap_{i=1}^{k}\vec{e}_{j(1:m'),i}U_i \rangle$. Therefore, $eU = \sqcap_{i=1}^{k} e\vec{e}_{j(1:m'),i}T_i$ and $e\Gamma = e\Gamma_1 \sqcap e\Gamma_2$. By rule (exp), $M^+ : \langle e\Gamma_1 \vdash_2 \sqcap_{i=1}^{k} e\vec{e}_{j(1:m'),i}(U_i \to T_i) \rangle$, and $N^+ : \langle e\Gamma_2 \vdash_2 \sqcap_{i=1}^{k} e\vec{e}_{j(1:m'),i}U_i \rangle$.

- Case $(\sqcap_{\mathsf{I}})$: $\dfrac{MN : \langle \Gamma_1 \vdash_i V_1 \rangle \quad MN : \langle \Gamma_2 \vdash_i V_2 \rangle}{MN : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i V_1 \sqcap V_2 \rangle}$.

  By Theorem 7.3.5.2a, $\mathsf{deg}(MN) = \mathsf{deg}(V_1) = \mathsf{deg}(V_2) = \mathsf{deg}(V_1 \sqcap V_2) = m$. By IH, $V_1 = \sqcap_{i=1}^{k_1}\vec{e}_{j(1:m),i}T_i$ $V_2 = \sqcap_{i=k_1+1}^{k}\vec{e}_{j(1:m),i}T_i$ where $k > k_1 \geq 1$, $\Gamma_1 = \Gamma'_1 \sqcap \Gamma''_1$, $\Gamma_2 = \Gamma'_2 \sqcap \Gamma''_2$, $M : \langle \Gamma'_1 \vdash_2 \sqcap_{i=1}^{k_1}\vec{e}_{j(1:m),i}(U_i \to T_i) \rangle$, $M : \langle \Gamma'_2 \vdash_2 \sqcap_{i=k_1+1}^{k}\vec{e}_{j(1:m),i}(U_i \to T_i) \rangle$, $N : \langle \Gamma''_1 \vdash_2 \sqcap_{i=1}^{k_1}\vec{e}_{j(1:m),i}U_i \rangle$, and $N : \langle \Gamma''_2 \vdash_2 \sqcap_{i=k_1+1}^{k}\vec{e}_{j(1:m),i}U_i \rangle$. Therefore, $V_1 \sqcap V_2 = \sqcap_{i=1}^{k}\vec{e}_{j(1:m),i}T_i$, $\Gamma_1 \sqcap \Gamma_2 = (\Gamma'_1 \sqcap \Gamma'_2) \sqcap (\Gamma''_1 \sqcap \Gamma''_2)$, and by rule $(\sqcap_{\mathsf{I}})$, $M : \langle \Gamma'_1 \sqcap \Gamma'_2 \vdash_2 \sqcap_{i=1}^{k}\vec{e}_{j(1:m),i}(U_i \to T_i) \rangle$ and $N : \langle \Gamma''_1 \sqcap \Gamma''_2 \vdash_2 \sqcap_{i=1}^{k}\vec{e}_{j(1:m),i}U_i \rangle$.

- Case ($\sqsubseteq$): Let $\dfrac{MN : \langle \Gamma \vdash_2 U \rangle \quad \Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'}{MN : \langle \Gamma' \vdash_2 U' \rangle}$.

By Theorem 7.3.5.2, $\mathsf{deg}(MN) = \mathsf{deg}(U) = \mathsf{deg}(U') = m$ and $U, U' \in \mathsf{GITy}$. By Lemma 7.3.4.3, $\Gamma' \sqsubseteq \Gamma$ and $U \sqsubseteq U'$. By IH, $U = \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} T_i$ where $k \geq 1$, $\Gamma = \Gamma_1 \sqcap \Gamma_2$, $M : \langle \Gamma_1 \vdash_2 \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i}(U_i \to T_i) \rangle$, and $N : \langle \Gamma_2 \vdash_2 \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i} U_i \rangle$. By Lemma B.1.11.8, $\Gamma' = \Gamma'_1 \sqcap \Gamma'_2$ such that $\Gamma'_1 \sqsubseteq \Gamma_1$ and $\Gamma'_2 \sqsubseteq \Gamma_2$. By Lemma B.1.11.2a and using the commutativity of $\sqcap$, $U = \sqcap_{i=1}^{k'} \vec{e}_{j(1:m),i} T'_i$ such that $k' \leq k$ and $\forall i \in \{1, \ldots, k'\}.\ T_i \sqsubseteq T'_i$. Finally, by rule $(\sqsubseteq)$, $M : \langle \Gamma'_1 \vdash_2 \sqcap_{i=1}^{k'} \vec{e}_{j(1:m),i}(U_i \to T'_i) \rangle$, and $N : \langle \Gamma'_2 \vdash_2 \sqcap_{i=1}^{k'} \vec{e}_{j(1:m),i} U_i \rangle$. □

Lemma B.1.14 (Extra Generation for $\vdash_2$)**.**

1. *If* $Mx^n : \langle \Gamma, x^n : U \vdash_2 V \rangle$*,* $\mathsf{deg}(V) = 0$ *and* $x^n \notin \mathsf{fv}(M)$ *then* $V = \sqcap_{i=1}^{k} T_i$ *where* $k \geq 1$ *and* $\forall i \in \{1, \ldots, k\}.\ M : \langle \Gamma \vdash_2 U \to T_i \rangle$*.*

2. *If* $\lambda x^n.Mx^n : \langle \Gamma \vdash_2 U \rangle$ *and* $x^n \notin \mathsf{fv}(M)$ *then* $M : \langle \Gamma \vdash_2 U \rangle$*.* □

Proof of Lemma B.1.14.

1. By induction on the derivation of $Mx^n : \langle \Gamma, x^n : U \vdash_2 V \rangle$ and then by case on the last rule of the derivation. We have three cases:

   - Case $(\to_{\mathsf{E}})$: Let $\dfrac{M : \langle \Gamma \vdash_2 U \to T \rangle \quad x^n : \langle x^n : V \vdash_2 U \rangle \quad \Gamma \diamond (x^n : V)}{Mx^n : \langle \Gamma, x^n : V \vdash_2 T \rangle}$ where $V \sqsubseteq U$ using Lemma 7.4.2.1 and Theorem 7.3.5.2a.

     Then because $U \to T \sqsubseteq V \to T$, we have $M : \langle \Gamma \vdash_2 V \to T \rangle$.

   - Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{Mx^n : \langle \Gamma_1, x^n : U'_1 \vdash_2 U_1 \rangle \quad Mx^n : \langle \Gamma_2, x^n : U'_2 \vdash_2 U_2 \rangle}{Mx^n : \langle \Gamma_1 \sqcap \Gamma_2, x^n : U'_1 \sqcap U'_2 \vdash_2 U_1 \sqcap U_2 \rangle}$ where $\mathsf{fv}(M) = \{x_1^{n_1}, \ldots, x_m^{n_m}\}$, $\Gamma_1 = (x_i^{n_i} : V_i)_m$, and $\Gamma_2 = (x_i^{n_i} : V'_i)_m$ using Theorem 7.3.5.2a.

     By Theorem 7.3.5, $U_1 \sqcap U_2, U'_1 \sqcap U'_2 \in \mathsf{GITy}$. and $\forall i \in \{1, \ldots, m\}.\ V_i, V'_i \in \mathsf{GITy}$. By Lemma 7.2.3.1b, $\mathsf{deg}(U'_1) = \mathsf{deg}(U'_2)$, $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = 0$, and $\forall i \in \{1, \ldots, m\}.\ \mathsf{deg}(V_i)V'_i$. By Lemma 7.2.3.1b, $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = 0$. By IH, $U_1 = \sqcap_{i=1}^{k} T_i$, $U_2 = \sqcap_{i=k+1}^{k+l} T_i$, where $k, l \geq 1$, $\forall i \in \{1, \ldots, k\}.\ M : \langle \Gamma_1 \vdash_2 U'_1 \to T_i \rangle$, and $\forall i \in \{k+1, \ldots, k+l\}.\ M : \langle \Gamma_2 \vdash_2 U'_2 \to T_i \rangle$. Using rule $(\sqcap_{\mathsf{E}})$, rule $(\to)$, Lemma 7.3.4.2, rule $(\sqsubseteq_{\langle\rangle})$, rule $(\sqsubseteq)$, we obtain $\forall i \in \{1, \ldots, k+l\}.\ M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 U'_1 \sqcap U'_2 \to T_i \rangle$.

   - Case $(\sqsubseteq)$: Let $\dfrac{Mx^n : \langle \Gamma, x^n : U \vdash_2 V \rangle \quad \Gamma, x^n : U \vdash_2 V \sqsubseteq \Gamma', x^n : U' \vdash_2 V'}{Mx^n : \langle \Gamma', x^n : U' \vdash_2 V' \rangle}$ using Lemma 7.3.4.2.

     By Lemma 7.3.4, $\Gamma' \sqsubseteq \Gamma$, $U' \sqsubseteq U$ and $V \sqsubseteq V'$. By Lemma 7.3.4.4, $\mathsf{deg}(V) = \mathsf{deg}(V') = 0$. By IH, $V = \sqcap_{i=1}^{k} T_i$ where $k \geq 1$ and $\forall i \in \{1, \ldots, k\}.\ M : \langle \Gamma \vdash_2 U \to T_i \rangle$. By Theorem 7.3.5, $V \in \mathsf{GITy}$. By Lemma B.1.11.2, $V' = \sqcap_{i=1}^{p} T'_i$ where $1 \leq p$ and $\forall i \in \{1, \ldots, p\}.\ \exists j \in \{1, \ldots, k\}.\ T_j \sqsubseteq T'_i$. By rule $(\to)$ and Lemma 7.3.4.3, one obtains $\forall i \in \{1, \ldots, p\} \exists j \in$ Therefore, by rule $(\sqsubseteq)$, $\forall i \in \{1, \ldots, p\}.\ M : \langle \Gamma' \vdash_2 U' \to T'_i \rangle$.

2. By Theorem 7.3.5, $m = \mathsf{deg}(U) = \mathsf{deg}(\lambda x^n.Mx^n) = \mathsf{deg}(Mx^n) \leq n$ and $\lambda x^n.Mx^n \in \mathbb{M}$. Therefore, we have $Mx^n \in \mathbb{M}$ and $n = \mathsf{deg}(x^n) \geq \mathsf{deg}(M) = \mathsf{deg}(Mx^n) = m$. By Lemma 7.4.2.2, $U = \sqcap_{i=1}^{k} \vec{e}_{j(1:m),i}(V_i \to T_i)$ where $k \geq 1$ and $\forall i \in \{1,\dots,k\}$. $Mx^n : \langle \Gamma, x^n : \vec{e}_{j(1:m),i}V_i \vdash_2 \vec{e}_{j(1:m),i}T_i \rangle$. If $m > 0$ then, by Theorem 7.3.5.2d and by 1., $\forall i \in \{1,\dots,k\}$. $M^{-m}x^{n-m} : \langle \Gamma^{-m}, x^{n-m} : V_i \vdash_2 T_i \rangle \wedge M^{-m} : \langle \Gamma^{-m} \vdash_2 V_i \to T_i \rangle$. Now, by m applications of rule (exp), $M : \langle \Gamma \vdash_2 \vec{e}_{j(1:m),i}(V_i \to T_i) \rangle$. Finally, by $k-1$ applications of rule $(\sqcap_{\mathsf{I}})$, $M : \langle \Gamma \vdash_2 U \rangle$. □

Lemma B.1.15. *Let $i \in \{1,2,3\}$ and $M : \langle \Gamma \vdash_i U \rangle$. We have:*

1. *If $M : \langle \Delta \vdash_i V \rangle$ then $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Delta)$.*

2. *Assume $N : \langle \Delta \vdash_i V \rangle$. We have $\Gamma \diamond \Delta$ iff $M \diamond N$.*

3. *If N is a subterm of M then there are Δ, V such that $N : \langle \Delta \vdash_i V \rangle$.*

4. *If $\Gamma = \Gamma_1 \sqcap \Gamma_2 \sqcap \Gamma_3$ then $\Gamma_1 \diamond \Gamma_2$.*

5. *If $\Gamma = \Gamma_1 \sqcap \Gamma_2$ and $\Gamma_3 \sqsubseteq \Gamma_1$ then $\Gamma_3 \sqcap \Gamma_2 \sqsubseteq \Gamma$* □

Proof of Lemma B.1.15.

1. Corollary of Theorem 7.3.5.2a because $\mathsf{dom}(\Gamma) = \mathsf{fv}(M) = \mathsf{dom}(\Delta)$.

2. Use Theorem 7.3.5.2a.

3. By induction on the derivation of $M : \langle \Gamma \vdash_i U \rangle$ and then by case on the last rule of the derivation.

4. By Theorem 7.3.5.2a, $\mathsf{dom}(\Gamma) = \mathsf{deg}(M)$. Let $x^{n_1} \in \mathsf{dom}(\Gamma_1)$ and $x^{n_2} \in \mathsf{dom}(\Gamma_2)$. Then, $x^{n_1}, x^{n_2} \in \mathsf{dom}(\Gamma) = \mathsf{deg}(M)$. Finally, by Lemma B.1.1.1, $M \diamond M$, and so $n_1 = n_2$ and $\Gamma_1 \diamond \Gamma_2$.

5. By definition $\Gamma_1 = \Gamma_1' \uplus \Gamma_1''$ and $\Gamma_2 = \Gamma_2' \uplus \Gamma_2''$ be such that $\mathsf{dj}(\mathsf{dom}(\Gamma_1''), \mathsf{dom}(\Gamma_2''))$, $\Gamma_1' = (x_i^{I_i} : U_i)_n$, $\Gamma_2' = (x_i^{I_i} : V_i)_n$, and $\forall i \in \{1,\dots,n\}$. $\mathsf{deg}(U_i) = \mathsf{deg}(V_i)$. Therefore $\Gamma = (x_i^{I_i} : U_i \sqcap V_i)_n \uplus \Gamma_1'' \uplus \Gamma_2''$. By Lemma 7.3.4.2, $\Gamma_3 = (x_i^{I_i} : U_i')_n \uplus \Gamma_3'$ such that $\Gamma_3' \sqsubseteq \Gamma_1''$, $\mathsf{dom}(\Gamma_3') = \mathsf{dom}(\Gamma_1'')$, and $\forall i \in \{1,\dots,n\}$. $U_i' \sqsubseteq U_i$. Therefore we have $\Gamma_3 \sqcap \Gamma_2 = (x_i^{I_i} : U_i' \sqcap V_i)_n \uplus \Gamma_3' \uplus \Gamma_2''$ Using rules $(\sqcap)$ and (ref) we obtain $\forall i \in \{1,\dots,n\}$. $U_i' \sqcap V_i \sqsubseteq U_i \sqcap V_i$. Finally, again by Lemma 7.3.4.2, $\Gamma_3 \sqcap \Gamma_2 \sqsubseteq \Gamma$. □

Proof of Remark 7.4.3. By Lemma B.1.15.3, $(\lambda x^n.M_1)M_2$ is typable.

- Case $\vdash_1$. By induction on the typing of $(\lambda x^n.M_1)M_2$. The only interesting case is rule $(\to_{\mathsf{E}})$ where $M = (\lambda x^n.M_1)M_2$ is the subterm in question:

$$\frac{\lambda x^n.M_1 : \langle \Gamma_1 \vdash_1 T_1 \to T_2 \rangle \quad M_2 : \langle \Gamma_2 \vdash_1 T_1 \rangle \quad \Gamma_1 \diamond \Gamma_2}{(\lambda x^n.M_1)M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_1 T_2 \rangle}$$

By Lemma 7.4.1.2, $M_1 : \langle \Gamma_1, x^n : T_1 \vdash_1 T_2 \rangle$. By Theorem 7.3.5, $n = \mathsf{deg}(T_1) = \mathsf{deg}(M_2)$. Hence, $(\lambda x^n.M_1)M_2 \rightarrowtriangle_\beta M_1[x^n := M_2]$.

- Case $\vdash_2$. By induction on the typing of $(\lambda x^n.M_1)M_2$. We consider only the rule $(\to_{\mathsf{E}})$

  $$\frac{\lambda x^n.M_1 : \langle \Gamma_1 \vdash_2 V \to T \rangle \quad M_2 : \langle \Gamma_2 \vdash_2 V \rangle \quad \Gamma_1 \diamond \Gamma_2}{(\lambda x^n.M_1)M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T \rangle}$$

  By Lemma 7.2.3.2, $\mathsf{deg}(V \to T) = 0$. By Lemma 7.4.2.2, $V \to T = \sqcap_{i=1}^{k}(V_i \to T_i)$ where $k \geq 1$ and $\forall i \in \{1, \ldots, k\}$. $M_1 : \langle \Gamma_1, x^n : V_i \vdash_2 T_i \rangle$. Hence $k = 1$, $V_1 = V$, $T_1 = T$ and $M_1 : \langle \Gamma_1, x^n : V \vdash_2 T \rangle$. By Theorem 7.3.5, $\mathsf{deg}(M_2) = \mathsf{deg}(V) = n$. So, $(\lambda x^n.M_1)M_2 \rightarrowtriangle_\beta M_1[x^n := M_2]$. □

Proof of Lemma 7.4.4. By Lemma 7.3.7.3, $\Gamma \diamond \Delta$.

By induction on the derivation of $M : \langle \Gamma, x^n : U \vdash_2 V \rangle$ (note that using Theorem 7.3.5, $x^n \in \mathsf{fv}(M)$), making use of Theorem 7.3.5.

- Case (ax): Let $\dfrac{T \in \mathsf{GITy}}{x^0 : \langle (x^0 : T) \vdash_2 T \rangle}$.

  Because $N : \langle \Delta \vdash_2 T \rangle$, then $N = x^0[x^0 := N] : \langle \Delta \vdash_2 T \rangle$.

- Case $(\to_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma, x^n : U, y^m : U' \vdash_2 T \rangle}{\lambda y^m.M : \langle \Gamma, x^n : U \vdash_2 U' \to T \rangle}$.

  Let y^m be such that $\forall m'.\ y^{m'} \notin \mathsf{dom}(\Delta)$. Since $\Gamma \diamond \Delta$, $(\Gamma, y^m : U') \diamond \Delta$ and we also have $y^m \notin \mathsf{dom}(\Delta)$. By IH, $M[x^n := N] : \langle (\Gamma \sqcap \Delta), y^m : U' \vdash_2 T \rangle$. By rule $(\to_{\mathsf{I}})$, $(\lambda y^m.M)[x^n := N] = \lambda y^m.M[x^n := N] : \langle \Gamma \sqcap \Delta \vdash_2 U' \to T \rangle$.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{M_1 : \langle \Gamma_1, x^n : U_1 \vdash_2 V \to T \rangle \quad M_2 : \langle \Gamma_2, x^n : U_2 \vdash_2 V \rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2, x^n : U_1 \sqcap U_2 \vdash_2 T \rangle}$
  where $x^n \in \mathsf{fv}(M_1) \cap \mathsf{fv}(M_2)$. (The cases $x^n \in \mathsf{fv}(M_1) \setminus \mathsf{fv}(M_2)$ are $x^n \in \mathsf{fv}(M_2) \setminus \mathsf{fv}(M_1)$ are similar.)

  We have $N : \langle \Delta \vdash_2 U_1 \sqcap U_2 \rangle$ and $(\Gamma_1 \sqcap \Gamma_2) \diamond \Delta$. By rules $(\sqcap_{\mathsf{E}})$ and $(\sqsubseteq)$, $N : \langle \Delta \vdash_2 U_1 \rangle$ and $N : \langle \Delta \vdash_2 U_2 \rangle$. Now use IH and rule $(\to_{\mathsf{E}})$.

- Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma_1, x^n : U_1' \vdash_2 U_1 \rangle \quad M : \langle \Gamma_2, x^n : U_2' \vdash_2 U_2 \rangle}{M : \langle \Gamma_1 \sqcap \Gamma_2, x^n : U \vdash_2 U_1 \sqcap U_2 \rangle}$ (because $x^n \in \mathsf{fv}(M)$ and using Theorem 7.3.5) where $U = U_1' \sqcap U_2'$.

  By Theorem 7.3.5, $\mathsf{deg}(U_1') = n = \mathsf{deg}(U_2')$ and $U_1', U_2' \in \mathsf{GITy}$. Using rule $(\sqcap_{\mathsf{E}})$, $U \sqsubseteq U_1'$ and $U \sqsubseteq U_2'$. Using rules $(\sqsubseteq_{\mathsf{c}})$, (ref), $(\sqsubseteq_{\langle\rangle})$, and $(\sqsubseteq)$, $M : \langle \Gamma_1, x^n : U \vdash_2 U_1 \rangle$ and $M : \langle \Gamma_2, x^n : U \vdash_2 U_2 \rangle$ By IH, $M[x^n := N] : \langle \Gamma_1 \sqcap \Delta \vdash_2 U_1 \rangle$ and $M[x^n := N] : \langle \Gamma_2 \sqcap \Delta \vdash_2 U_2 \rangle$ Therefore by rule $(\sqcap_{\mathsf{I}})$. $M[x^n := N] : \langle \Gamma_1 \sqcap \Gamma_2 \sqcap \Delta \vdash_2 U_1 \sqcap U_2 \rangle$.

- Case (exp): Let $\dfrac{M : \langle \Gamma, x^n : U \vdash_2 V\rangle}{M^{+:}\langle e\Gamma, x^{n+1} : eU \vdash_2 eV\rangle}$.

  We have $N : \langle \Delta \vdash_2 eU\rangle$ and $e\Gamma \diamond \Delta$. By Theorem 7.3.5, $\mathsf{deg}(N) = \mathsf{deg}(eU) = \mathsf{deg}(U) + 1 > 0$. Hence, by Lemmas B.1.3.1 and 7.3.5.2, $N = P^+$ and $P : \langle \Delta^- \vdash_2 U\rangle$. Because $e\Gamma \diamond \Delta$ then by Lemma B.1.13.4, $\Gamma \diamond \Delta^-$. By IH, $M[x^n := P] : \langle \Gamma \sqcap \Delta^- \vdash_2 V\rangle$. By rule (exp) and Lemma B.1.3.2, $M^+[x^{n+1} := N] : \langle e\Gamma \sqcap \Delta \vdash_2 eV\rangle$.

- Case $(\sqsubseteq)$: Let $\dfrac{M : \langle \Gamma', x^n : U' \vdash_2 V'\rangle \quad \Gamma', x^n : U' \vdash_2 V' \sqsubseteq \Gamma, x^n : U \vdash_2 V}{M : \langle \Gamma, x^n : U \vdash_2 V\rangle}$ (note the use of Lemma 7.3.4).

  By Lemma 7.3.4, $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$, $\Gamma \sqsubseteq \Gamma'$, $U \sqsubseteq U'$ and $V' \sqsubseteq V$. Hence $\Gamma' \diamond \Delta$, by rule $(\sqsubseteq)$ $N : \langle \Delta \vdash_2 U'\rangle$ and, by IH, $M[x^n := N] : \langle \Gamma' \sqcap \Delta \vdash_2 V'\rangle$. By Lemma B.1.15.5, $\Gamma \sqcap \Delta \sqsubseteq \Gamma' \sqcap \Delta$. Hence, $\Gamma' \sqcap \Delta \vdash_2 V' \sqsubseteq \Gamma \sqcap \Delta \vdash_2 V$ and $M[x^n := N] : \langle \Gamma \sqcap \Delta \vdash_2 V\rangle$. □

Lemma B.1.16. *If $M : \langle \Gamma \vdash_2 U\rangle$ and $M \to\!\!\!\rhd_\beta N$ then $N : \langle \Gamma \vdash_2 U\rangle$.* □

Proof of Lemma B.1.16. By induction on the derivation of $M : \langle \Gamma \vdash_2 U\rangle$. Cases $(\to_{\mathsf{I}})$, $(\sqcap_{\mathsf{I}})$ and $(\sqsubseteq)$ are by IH. We give the remaining two cases.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{M_1 : \langle \Gamma_1 \vdash_2 U \to T\rangle \quad M_2 : \langle \Gamma_2 \vdash_2 U\rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle}$.

  For the cases $N = M_1 N_2$ where $M_2 \to\!\!\!\rhd_\beta N_2$ or $N = N_1 M_2$ where $M_1 \to\!\!\!\rhd_\beta N_1$ use IH. Assume $M_1 = \lambda x^n.P$ and $M_1 M_2 = (\lambda x^n.P)M_2 \to\!\!\!\rhd_\beta P[x^n := M_2] = N$ where $\mathsf{deg}(M_2) = n$. By Lemma 7.2.3.2a, $\mathsf{deg}(U \to T) = 0$. Because $\lambda x^n.P : \langle \Gamma_1 \vdash_2 U \to T\rangle$ then, by Lemma 7.4.2.2, $P : \langle \Gamma_1, x^n : U \vdash_2 T\rangle$. By Lemma 7.4.4, $P[x^n := M_2] : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle$.

- Case (exp): Let $\dfrac{M : \langle \Gamma \vdash_2 U\rangle}{M^+ : \langle e\Gamma \vdash_2 eU\rangle}$.

  Because $M^+ \to\!\!\!\rhd_\beta N$ then by Lemma 7.1.11.2, $\mathsf{deg}(M^+) = \mathsf{deg}(N)$. By Lemmas B.1.3.1a and B.1.4.2, $\mathsf{deg}(N) = \mathsf{deg}(M) + 1 > 0$ and $M \to\!\!\!\rhd_\beta N^-$. By IH, $N^- : \langle \Gamma \vdash_2 U\rangle$ and, by Lemma B.1.3.1b and rule (exp), $N : \langle e\Gamma \vdash_2 eU\rangle$. □

The next lemma will be used in the proof of subject expansion for β.

Lemma B.1.17. *Let $(\lambda x^n.M_1)M_2 : \langle \Gamma \vdash_2 U\rangle$ then $\Gamma = \Gamma_1 \sqcap \Gamma_2$ and there exists $V \in \mathsf{ITy}_2$ such that $M_1 : \langle \Gamma_1, (x^n : V) \vdash_2 U\rangle$ and $M_2 : \langle \Gamma_2 \vdash_2 V\rangle$.* □

Proof of Lemma B.1.17. By induction on the derivation of $(\lambda x^n.M_1)M_2 : \langle \Gamma \vdash_2 U\rangle$. and then by case on the last rule of the derivation.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{\lambda x^n.M_1 : \langle \Gamma_1 \vdash_2 V \to T\rangle \quad M_2 : \langle \Gamma_2 \vdash_2 V\rangle \quad \Gamma_1 \diamond \Gamma_2}{(\lambda x^n.M_1)M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle}$.

  Since $\mathsf{deg}(V \to T) = 0$, then by Lemma 7.4.2.2 $M_1 : \langle \Gamma_1, (x^n : V) \vdash_2 T\rangle$.

- Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{(\lambda x^n.M_1)M_2 : \langle \Gamma_1 \vdash_2 U_1\rangle \quad (\lambda x^n.M_1)M_2 : \langle \Gamma_2 \vdash_2 U_2\rangle}{(\lambda x^n.M_1)M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 U_1 \sqcap U_2\rangle}$.

  By IH, $\Gamma_1 = \Gamma'_1 \sqcap \Gamma'_2$, $\Gamma_2 = \Gamma''_1 \sqcap \Gamma''_2$, $\exists\ V, V' \in \mathsf{ITy}_2$, such that $M_1 : \langle \Gamma'_1, (x^n : V) \vdash_2 U_1\rangle$, $M_2 : \langle \Gamma'_2 \vdash_2 V\rangle$, $M_1 : \langle \Gamma''_1, (x^n : V') \vdash_2 U_2\rangle$, and $M_2 : \langle \Gamma''_2 \vdash_2 V'\rangle$. By rule $(\sqcap_{\mathsf{I}})$, $M_1 : \langle \Gamma'_1 \sqcap \Gamma''_1, (x^n : V \sqcap V') \vdash_2 U_1 \sqcap U_2\rangle$, and $M_2 : \langle \Gamma'_2 \sqcap \Gamma''_2 \vdash_2 V \sqcap V'\rangle$. Finally, we have $\Gamma_1 \sqcap \Gamma_2 = \Gamma'_1 \sqcap \Gamma''_1 \sqcap \Gamma'_2 \sqcap \Gamma''_2$ and $V \sqcap V' \in \mathsf{ITy}_2$.

- Case (exp): Let $\dfrac{(\lambda x^n.M_1)M_2 : \langle \Gamma \vdash_2 U\rangle}{(\lambda x^{n+1}.M_1{}^+)M_2{}^+ : \langle e\Gamma \vdash_2 eU\rangle}$.

  By IH, $\Gamma = \Gamma_1 \sqcap \Gamma_2$ and $\exists V \in \mathsf{ITy}_2$, such that $M_1 : \langle \Gamma_1, (x^n : V) \vdash_2 U\rangle$ and $M_2 : \langle \Gamma_2 \vdash_2 V\rangle$. So by rule (exp), $M_1{}^+ : \langle e\Gamma_1, (x^{n+1} : eV) \vdash_2 eU\rangle$ and $M_2{}^+ : \langle e\Gamma_2 \vdash_2 eV\rangle$.

- Case $(\sqsubseteq)$: Let $\dfrac{(\lambda x^n.M_1)M_2 : \langle \Gamma' \vdash_2 U'\rangle \quad \Gamma' \vdash_2 U' \sqsubseteq \Gamma \vdash_2 U}{(\lambda x^n.M_1)M_2 : \langle \Gamma \vdash_2 U\rangle}$.

  By Lemma 7.3.4.3, $\Gamma \sqsubseteq \Gamma'$ and $U' \sqsubseteq U$. By IH, $\Gamma' = \Gamma'_1 \sqcap \Gamma'_2$ and $\exists V \in \mathsf{ITy}_2$, such that $M_1 : \langle \Gamma'_1, (x^n : V) \vdash_2 U'\rangle$ and $M_2 : \langle \Gamma'_2 \vdash_2 V\rangle$. By Lemma B.1.11.8, $\Gamma = \Gamma_1 \sqcap \Gamma_2$ such that $\Gamma_1 \sqsubseteq \Gamma'_1$ and $\Gamma_2 \sqsubseteq \Gamma'_2$. So by rule $(\sqsubseteq)$, $M_1 : \langle \Gamma_1, (x^n : V) \vdash_2 U\rangle$ and $M_2 : \langle \Gamma_2 \vdash_2 V\rangle$. □

Now, we give the basic block in the proof of subject expansion for β.

Lemma B.1.18. *If $N : \langle \Gamma \vdash_2 U\rangle$ and $M \rightarrow\!\!\!\triangleright_\beta N$ then $M : \langle \Gamma \vdash_2 U\rangle$* □

Proof of Lemma B.1.18. By induction on the derivation of $N : \langle \Gamma \vdash_2 U\rangle$ and then by case on the last rule of the derivation.

- Case (ax): Let $\dfrac{T \in \mathsf{GITy}}{x^0 : \langle (x^0 : T) \vdash_2 T\rangle}$ and $M \rightarrow\!\!\!\triangleright_\beta x^0$.

  By cases on M, we can show that $M = (\lambda y^0.y^0)x^0$. Because $T \in \mathsf{GITy}$, by rule (ax), $y^0 : \langle (y^0 : T) \vdash_2 T\rangle$ then by rule $(\rightarrow_{\mathsf{I}})$, $\lambda y^0.y^0 : \langle () \vdash_2 T \rightarrow T\rangle$, and so by rule $(\rightarrow_{\mathsf{E}})$, $(\lambda y^0.y^0)x^0 : \langle (x^0 : T) \vdash_2 T\rangle$.

- Case $(\rightarrow_{\mathsf{I}})$: Let $\dfrac{N : \langle \Gamma, (x^n : U) \vdash_2 T\rangle}{\lambda x^n.N : \langle \Gamma \vdash_2 U \rightarrow T\rangle}$ and $M \rightarrow\!\!\!\triangleright_\beta \lambda x^n.N$.

  By cases on M.

  - If M is a variable this is not possible.
  - If $M = \lambda x^n.M'$ such that $M' \rightarrow\!\!\!\triangleright_\beta N$ and $x^n \in \mathsf{fv}(M') \cap \mathsf{fv}(N)$ then by IH, $M : \langle \Gamma, (x^n : U) \vdash_2 T\rangle$ and by rule $(\rightarrow_{\mathsf{I}})$, $M : \langle \Gamma \vdash_2 U \rightarrow T\rangle$.
  - If M is an application term then the reduction must be at the root. Hence, $M = (\lambda y^m.M_1)M_2 \rightarrow\!\!\!\triangleright_\beta M_1[y^m := M_2] = \lambda x^n.N$ where $y^m \in \mathsf{fv}(M_1)$ and $\mathsf{deg}(M_2) = m$. There are two cases (M_1 cannot be an application term):

- If $M_1 = y^m$ then $M_2 = \lambda x^n.N$ and $\mathsf{deg}(N) = \mathsf{deg}(M_2) = m$. By Theorem 7.3.5.2, $m = \mathsf{deg}(N) = \mathsf{deg}(T) = 0$. So $M = (\lambda y^0.y^0)(\lambda x^n.N)$. Because by Theorem 7.3.5.2, $U\to T \in \mathsf{GITy} \cap \mathsf{ITy}_2$, by rule (ax), $y^0 : \langle (y^0 : U\to T) \vdash_2 U\to T\rangle$, by rule $(\to_{\mathsf{I}})$, $\lambda y^0.y^0 : \langle () \vdash_2 (U\to T)\to(U\to T)\rangle$, and by rule $(\to_{\mathsf{E}})$, $(\lambda y^0.y^0)(\lambda x^n.N) : \langle \Gamma \vdash_2 U\to T\rangle$.

- If $M_1 = \lambda x^n.M_1'$ such that $\forall n'.\ x^{n'} \notin \mathsf{fv}(M_2) \cup \{y^m\}$ then $M_1[y^m := M_2] = \lambda x^n.M_1'[y^m := M_2] = \lambda x^n.N$ and $\mathsf{deg}(M_2) = m$. Since $(\lambda y^m.M_1')M_2 \rightarrowtriangle_\beta M_1'[y^m := M_2] = N$, by IH, $(\lambda y^m.M_1')M_2 : \langle \Gamma, (x^n : U) \vdash_2 T\rangle$. By Lemma B.1.17, $\Gamma, (x^n : U) = \Gamma_1 \sqcap \Gamma_2$ and $\exists V \in \mathsf{ITy}$ such that $M_1' : \langle \Gamma_1, (y^m : V) \vdash_2 T\rangle$ and $M_2 : \langle \Gamma_2 \vdash_2 V\rangle$. By Theorem 7.3.5.2a, $\mathsf{dom}(\Gamma_2) = \mathsf{fv}(M_2)$. Because $x^n \notin \mathsf{fv}(M_2)$ then $\Gamma = \Gamma_1' \sqcap \Gamma_2$ and $\Gamma_1 = \Gamma_1', (x^n : U)$. Hence by rule $(\to_{\mathsf{I}})$, $\lambda x^n.M_1' : \langle \Gamma_1', (y^m : V) \vdash_2 U\to T\rangle$, again by rule $(\to_{\mathsf{I}})$, $\lambda y^m.\lambda x^n.M_1' : \langle \Gamma_1' \vdash_2 V\to U\to T\rangle$, and since by Lemma B.1.15.4, $\Gamma_1' \diamond \Gamma_2$, by rule $(\to_{\mathsf{E}})$, $M = (\lambda y^m.\lambda x^n.M_1')M_2 : \langle \Gamma \vdash_2 U\to T\rangle$.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{N_1 : \langle \Gamma_1 \vdash_2 U\to T\rangle \quad N_2 : \langle \Gamma_2 \vdash_2 U\rangle \quad \Gamma_1 \diamond \Gamma_2}{N_1N_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle}$ and $M \rightarrowtriangle_\beta N_1N_2$.

  - If $M = M_1N_2 \rightarrowtriangle_\beta N_1N_2$ where $M_1 \diamond N_2$, $N_1 \diamond N_2$ (by Lemma B.1.1) and $M_1 \rightarrowtriangle_\beta N_1$ then by IH, $M_1 : \langle \Gamma_1 \vdash_2 U\to T\rangle$, and by rule $(\to_{\mathsf{E}})$, $M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle$.

  - If $M = N_1M_2 \rightarrowtriangle_\beta N_1N_2$ where $N_1 \diamond M_2$, $N_1 \diamond N_2$ (by Lemma B.1.1) and $M_2 \rightarrowtriangle_\beta N_2$ then by IH, $M_2 : \langle \Gamma_2 \vdash_2 U\rangle$, and by rule $(\to_{\mathsf{E}})$, $M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle$.

  - If $M = (\lambda x^n.M_1)M_2 \rightarrowtriangle_\beta M_1[x^n := M_2] = N_1N_2$ where $\mathsf{deg}(M_2) = n$ and $x^n \in \mathsf{fv}(M_1)$. By cases on M_1 (M_1 cannot be an abstraction):

    - If $M_1 = x^n$ then $M_2 = N_1N_2$, $\mathsf{deg}(N_1N_2) = \mathsf{deg}(M_2) = n$, and $M = (\lambda x^0.x^0)(N_1N_2)$ because by Theorem 7.3.5, $n = \mathsf{deg}(N_1N_2) = \mathsf{deg}(T) = 0$ and $T \in \mathsf{GITy}$. By rule (ax), $x^0 : \langle (x^0 : T) \vdash_2 T\rangle$, hence by rule $(\to_{\mathsf{I}})$, $\lambda x^0.x^0 : \langle () \vdash_2 T\to T\rangle$, and by rule $(\to_{\mathsf{E}})$, $(\lambda x^0.x^0)(N_1N_2) : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle$.

    - If $M_1 = M_1'M_1''$ then $M_1[x^n := M_2] = M_1'[x^n := M_2]M_1''[x^n := M_2] = N_1N_2$. So, $M_1'[x^n := M_2] = N_1$ and $M_1''[x^n := M_2] = N_2$.

      - If $x^n \in \mathsf{fv}(M_1')$ and $x^n \in \mathsf{fv}(M_1'')$ then $(\lambda x^n.M_1')M_2 \rightarrowtriangle_\beta N_1$ and $(\lambda x^n.M_1'')M_2 \rightarrowtriangle_\beta N_2$. By IH, $(\lambda x^n.M_1')M_2 : \langle \Gamma_1 \vdash_2 U\to T\rangle$ and $(\lambda x^n.M_1'')M_2 : \langle \Gamma_2 \vdash_2 U\rangle$. By Lemma B.1.17 twice, $\Gamma_1 = \Gamma_1' \sqcap \Gamma_1''$, $\Gamma_2 = \Gamma_2' \sqcap \Gamma_2''$, and $\exists\ V, V' \in \mathsf{ITy}$ such that $M_1' : \langle \Gamma_1', (x^n : V) \vdash_2 U\to T\rangle$, $M_2 : \langle \Gamma_1'' \vdash_2 V\rangle$, $M_1'' : \langle \Gamma_2', (x^n : V') \vdash_2 U\rangle$ and $M_2 : \langle \Gamma_2'' \vdash_2 V'\rangle$. Therefore, $\Gamma_1 \sqcap \Gamma_2 = \Gamma_1' \sqcap \Gamma_1'' \sqcap \Gamma_2' \sqcap \Gamma_2''$. By rule $(\sqcap_{\mathsf{I}})$,

$M_2 : \langle \Gamma''_1 \sqcap \Gamma''_2 \vdash_2 V \sqcap V' \rangle$. Because by Lemma B.1.15.4, $\Gamma'_1 \diamond \Gamma'_2$, then by rule $(\to_{\mathsf{E}})$, $M'_1 M''_1 : \langle \Gamma'_1 \sqcap \Gamma'_2, (x^n : V \sqcap V') \vdash_2 T \rangle$. Using rule $(\to_{\mathsf{I}})$, $\lambda x^n.M'_1 M''_1 : \langle \Gamma'_1 \sqcap \Gamma'_2 \vdash_2 (V \sqcap V') \to T \rangle$. Finally, by rule $(\to_{\mathsf{E}})$ and because by Lemma B.1.15.4, $\Gamma'_1 \sqcap \Gamma'_2 \diamond \Gamma''_1 \sqcap \Gamma''_2$, we obtain $(\lambda x^n.M'_1 M''_1) M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T \rangle$.

- If $x^n \in \mathsf{fv}(M'_1)$ and $x^n \notin \mathsf{fv}(M''_1)$ then $M'_1[x^n := M_2] = N_1$ and $M''_1 = N_2$. We have $(\lambda x^n.M'_1) M_2 \rightarrowtriangle_\beta N_1$, so by IH, $(\lambda x^n.M'_1) M_2 : \langle \Gamma_1 \vdash_2 U \to T \rangle$. By Lemma B.1.17, $\Gamma_1 = \Gamma'_1 \sqcap \Gamma''_1$ and $\exists V \in \mathsf{ITy}$ such that $M'_1 : \langle \Gamma'_1, (x^n : V) \vdash_2 U \to T \rangle$ and $M_2 : \langle \Gamma''_1 \vdash_2 V \rangle$. Therefore $\Gamma_1 \sqcap \Gamma_2 = \Gamma'_1 \sqcap \Gamma''_2 \sqcap \Gamma_2$. Because by Lemma B.1.15.4, $\Gamma'_1 \diamond \Gamma_2$, by rule $(\to_{\mathsf{E}})$, $M'_1 M''_1 : \langle \Gamma'_1 \sqcap \Gamma_2, (x^n : V) \vdash_2 T \rangle$, and by rule $(\to_{\mathsf{I}})$, $\lambda x^n.M'_1 M''_1 : \langle \Gamma'_1 \sqcap \Gamma_2 \vdash_2 V \to T \rangle$. Finally, by rule $(\to_{\mathsf{E}})$ and because by Lemma B.1.15.4, $\Gamma'_1 \sqcap \Gamma_2 \diamond \Gamma''_1$, $(\lambda x^n.M'_1 M''_1) M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T \rangle$.
- If $x^n \notin \mathsf{fv}(M'_1)$ and $x^n \in \mathsf{fv}(M''_1)$ then the proof is similar to the previous case.

- Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{N : \langle \Gamma_1 \vdash_2 U_1 \rangle \quad N : \langle \Gamma_2 \vdash_2 U_2 \rangle}{N : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 U_1 \sqcap U_2 \rangle}$ and $M \rightarrowtriangle_\beta N$.

  By IH, $M : \langle \Gamma_1 \vdash_2 U_1 \rangle$ and $M : \langle \Gamma_2 \vdash_2 U_2 \rangle$, hence by rule $(\sqcap_{\mathsf{I}})$, $M : \langle \Gamma \vdash_2 U_1 \sqcap U_2 \rangle$.

- Case (exp): Let $\dfrac{N : \langle \Gamma \vdash_2 U \rangle}{N^+ : \langle e\Gamma \vdash_2 eU \rangle}$ and $M \rightarrowtriangle_\beta N^+$.

  By Lemmas B.1.5.8 and B.1.5.4, $M^- \rightarrowtriangle_\beta N$, and by IH, $M^- : \langle \Gamma \vdash_2 U \rangle$. By Lemma B.1.3.1b, $(M^-)^+ = M$ and by rule (exp), $M : \langle e\Gamma \vdash_2 eU \rangle >$.

- Case $(\sqsubseteq)$: Let $\dfrac{N : \langle \Gamma \vdash_2 U \rangle \quad \Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'}{N : \langle \Gamma' \vdash_2 U' \rangle}$ and $M \rightarrowtriangle_\beta N$.

  By IH, $M : \langle \Gamma \vdash_2 U \rangle$ and by rule $(\sqsubseteq)$, $M : \langle \Gamma' \vdash_2 U' \rangle$. □

Proof of Lemma 7.4.6.

1. 1 By induction on the length of the derivation of $M \rightarrowtriangle_\beta^* N$ using Lemma B.1.16.

2. 2 By induction on the length of the derivation of $M \rightarrowtriangle_\beta^* N$ using Lemma B.1.18. □

Subject reduction and expansion properties for $\vdash_3$ (Sec. 7.4.2)

Proof of Lemma 7.4.7. 1. By induction on the derivation $x^L : \langle \Gamma \vdash_3 U \rangle$. We have five cases:

- Case (ax): Let $\overline{x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T \rangle}$.

  Then it is done using rule (ref).

- Case (ω): Let $\overline{x^L : \langle (x^L : \omega^L) \vdash_3 \omega^L \rangle}$.

  Then it is done using rule (ref).

- Case ($\sqcap_{\mathsf{I}}$): Let $\dfrac{x^L : \langle \Gamma \vdash_3 U_1 \rangle \quad x^L : \langle \Gamma \vdash_3 U_2 \rangle}{x^L : \langle \Gamma \vdash_3 U_1 \sqcap U_2 \rangle}$.

  By IH, $\Gamma = (x^L : V)$, $V \sqsubseteq U_1$ and $V \sqsubseteq U_2$ then by rule ($\sqcap$), $V \sqsubseteq U_1 \sqcap U_2$.

- Case (exp): Let $\dfrac{x^L : \langle \Gamma \vdash_3 U \rangle}{x^{i::L} : \langle \mathsf{e}_i\Gamma \vdash_3 \mathsf{e}_i U \rangle}$.

  Then by IH, $\Gamma = (x^L : V)$ and $V \sqsubseteq U$, so $\mathsf{e}_i\Gamma = (x^{i::L} : \mathsf{e}_i V)$ and by rule ($\sqsubseteq_{\mathsf{exp}}$), $\mathsf{e}_i V \sqsubseteq \mathsf{e}_i U$,

- Case ($\sqsubseteq$): Let $\dfrac{x^L : \langle \Gamma' \vdash_3 U' \rangle \quad \Gamma' \vdash_3 U' \sqsubseteq \Gamma \vdash_3 U}{x^L : \langle \Gamma \vdash_3 U \rangle}$.

  By Lemma 7.3.4.3, $\Gamma \sqsubseteq \Gamma'$ and $U' \sqsubseteq U$ and, by IH, $\Gamma' = (x^L : V')$ and $V' \sqsubseteq U'$. Then, by Lemma 7.3.4.2, $\Gamma = (x^L : V)$, $V \sqsubseteq V'$ and, by rule (tr), $V \sqsubseteq U$.

2. By induction on the derivation $\lambda x^L.M : \langle \Gamma \vdash_3 U \rangle$. We have five cases:

- Case (ω): Let $\overline{\lambda x^L.M : \langle \mathsf{env}^{\emptyset}_{\lambda x^L.M} \vdash_3 \omega^{\mathsf{deg}(\lambda x^L.M)} \rangle}$.

  We are done.

- Case ($\to_{\mathsf{I}}$): Let $\dfrac{M : \langle \Gamma, x^L : U \vdash_3 T \rangle}{\lambda x^L.M : \langle \Gamma \vdash_3 U \to T \rangle}$.

  Then $\mathsf{deg}(U \to T) = \oslash$ and we are done.

- Case ($\sqcap_{\mathsf{I}}$): Let $\dfrac{\lambda x^L.M : \langle \Gamma \vdash_3 U_1 \rangle \quad \lambda x^L.M : \langle \Gamma \vdash_3 U_2 \rangle}{\lambda x^L.M : \langle \Gamma \vdash_3 U_1 \sqcap U_2 \rangle}$.

  Then $\mathsf{deg}(U_1 \sqcap U_2) = \mathsf{deg}(U_1) = \mathsf{deg}(U_2) = K$. By IH, we have four cases:

  - If $U_1 = U_2 = \omega^K$ then $U_1 \sqcap U_2 = \omega^K$.
  - If $U_1 = \omega^K$, $U_2 = \sqcap_{i=1}^{p} \vec{\mathsf{e}}_K (V_i \to T_i)$ where $p \geq 1$ and $\forall i \in \{1, \ldots, p\}$. $M : \langle \Gamma, x^L : \vec{\mathsf{e}}_K V_i \vdash_3 \vec{\mathsf{e}}_K T_i \rangle$ then $U_1 \sqcap U_2 = U_2$ (ω^K is a neutral element).
  - If $U_2 = \omega^K$, $U_1 = \sqcap_{i=1}^{p} \vec{\mathsf{e}}_K (V_i \to T_i)$ where $p \geq 1$ and $\forall i \in \{1, \ldots, p\}$. $M : \langle \Gamma, x^L : \vec{\mathsf{e}}_K V_i \vdash_3 \vec{\mathsf{e}}_K T_i \rangle$ then $U_1 \sqcap U_2 = U_1$ (ω^K is a neutral element).
  - If $U_1 = \sqcap_{i=1}^{p} \vec{\mathsf{e}}_K (V_i \to T_i)$, $U_2 = \sqcap_{i=p+1}^{p+q} \vec{\mathsf{e}}_K (V_i \to T_i)$ (hence $U_1 \sqcap U_2 = \sqcap_{i=1}^{p+q} \vec{\mathsf{e}}_K (V_i \to T_i)$) where $p, q \geq 1$ and $\forall i \in \{1, \ldots, p+q\}$. $M : \langle \Gamma, x^L : \vec{\mathsf{e}}_K V_i \vdash_3 \vec{\mathsf{e}}_K T_i \rangle$.

- Case (exp): Let $\dfrac{\lambda x^L.M : \langle \Gamma \vdash_3 U \rangle}{\lambda x^{i::L}.M^{+i} : \langle \mathsf{e}_i\Gamma \vdash_3 \mathsf{e}_i U \rangle}$.

  We have $\mathsf{deg}(\mathsf{e}_i U) = i :: \mathsf{deg}(U) = i :: K' = K$. By IH, we have two cases:

  - If $U = \omega^{K'}$ then $\mathsf{e}_i U = \omega^K$.

    - If $U = \sqcap_{j=1}^{p}\vec{\mathsf{e}}_{K'}(V_j \to T_j)$, where $p \geq 1$ and $\forall j \in \{1,\ldots,p\}$. $M : \langle \Gamma, x^L : \vec{\mathsf{e}}_{K'}V_j \vdash_3 \vec{\mathsf{e}}_{K'}T_j\rangle$. So $\mathsf{e}_iU = \sqcap_{j=1}^{p}\vec{\mathsf{e}}_K(V_j \to T_j)$ and by rule (exp), $\forall j \in \{1,\ldots,p\}$. $M^{+i} : \langle \mathsf{e}_i\Gamma, x^{i::L} : \vec{\mathsf{e}}_K V_j \vdash_3 \vec{\mathsf{e}}_K T_j\rangle$.

  - Case $(\sqsubseteq)$: Let $\dfrac{\lambda x^L.M : \langle \Gamma \vdash_3 U\rangle \quad \Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'}{\lambda x^L.M : \langle \Gamma' \vdash_3 U'\rangle}$.

    By Lemma 7.3.4.3, $\Gamma' \sqsubseteq \Gamma$ and $U \sqsubseteq U'$ and by Lemma 7.3.4.4, $\mathsf{deg}(U) = \mathsf{deg}(U') = K$. By IH, we have two cases:

    - If $U = \omega^K$ then, by Lemma B.1.12.3a, $U' = \omega^K$.
    - If $U = \sqcap_{i=1}^{p}\vec{\mathsf{e}}_K(V_i \to T_i)$, where $p \geq 1$ and $\forall i \in \{1,\ldots,p\}$. $M : \langle \Gamma, x^L : \vec{\mathsf{e}}_K V_i \vdash_3 \vec{\mathsf{e}}_K T_i\rangle$. By Lemma B.1.12.3d:
      * Either $U' = \omega^K$.
      * Or $U' = \sqcap_{i=1}^{q}\vec{\mathsf{e}}_K(V'_i \to T'_i)$, where $q \geq 1$ and $\forall i \in \{1,\ldots,q\}$. $\exists j \in \{1,\ldots,p\}$. $V'_i \sqsubseteq V_j \wedge T_j \sqsubseteq T'_i$. Let $i \in \{1,\ldots,q\}$. Because, by Lemma 7.3.4.3, $(\Gamma, x^L : \vec{\mathsf{e}}_K V_j \vdash_3 \vec{\mathsf{e}}_K T_j) \sqsubseteq (\Gamma', x^L : \vec{\mathsf{e}}_K V'_i \vdash_3 \vec{\mathsf{e}}_K T'_i)$ then $M : \langle \Gamma', x^L : \vec{\mathsf{e}}_K V'_i \vdash_3 \vec{\mathsf{e}}_K T'_i\rangle$.

3. Similar as the proof of 2.

4. By induction on the derivation $Mx^L : \langle \Gamma, x^L : U \vdash_3 T\rangle$. We have only two cases:

   - Case $(\to_{\mathsf{E}})$: Let $\dfrac{M : \langle \Gamma \vdash_3 V \to T\rangle \quad x^L : \langle (x^L : U) \vdash_2 V\rangle \quad \Gamma \diamond (x^L : U)}{Mx^L : \langle \Gamma, (x^L : U) \vdash_3 T\rangle}$ using Theorem 7.3.5.

     By 1., $U \sqsubseteq V$. Because $V \to T \sqsubseteq U \to T$, then we have $M : \langle \Gamma \vdash_3 U \to T\rangle$.

   - Case $(\sqsubseteq)$: Let $\dfrac{Mx^L : \langle \Gamma', (x^L : U') \vdash_3 V'\rangle}{Mx^L : \langle \Gamma, (x^L : U) \vdash_3 V\rangle}$ where $\Gamma', (x^L : U') \vdash_3 V' \sqsubseteq \Gamma, (x^L : U) \vdash_3 V$, using Lemma 7.3.4.

     By Lemma 7.3.4, $\Gamma \sqsubseteq \Gamma'$, $U \sqsubseteq U'$, and $V' \sqsubseteq V$. By IH, $M : \langle \Gamma' \vdash_3 U' \to V'\rangle$ and by rule $(\sqsubseteq)$, $M : \langle \Gamma \vdash_3 U \to V\rangle$. □

Proof of Lemma 7.4.8. By Lemma 7.3.7.3, $\Gamma \diamond \Delta$. By Theorem 7.3.5, $M, N \in \mathcal{M}_3$, $\mathsf{deg}(N) = \mathsf{deg}(U) = L$, $\mathsf{ok}(\Delta)$ and $\mathsf{ok}(\Gamma, x^L : U)$. By Lemma B.1.13.1a, $\mathsf{ok}(\Gamma \sqcap \Delta)$. By Lemma B.1.1.5, $M[x^L := N] \in \mathcal{M}_3$. By Lemma 7.3.5.2a, $x^L \in \mathsf{fv}(M)$. By Lemma B.1.1.5, $\mathsf{deg}(M[x^L := N]) = \mathsf{deg}(M)$.

We prove the lemma by induction on the derivation $M : \langle \Gamma, x^L : U \vdash_3 V\rangle$.

- Case (ax): Let $\overline{x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T\rangle}$ and $N : \langle \Delta \vdash_3 T\rangle$.

  Then $x^{\oslash}[x^{\oslash} := N] = N : \langle \Delta \vdash_3 T\rangle$.

- Case (ω): Let $\overline{M : \langle \mathsf{env}^{\varnothing}_{\mathsf{fv}(M)\setminus\{x^L\}}, (x^L : \omega^L) \vdash_3 \omega^{\mathsf{deg}(M)}\rangle}$ and $N : \langle \Delta \vdash_3 \omega^L\rangle$.

By rule (ω), $M[x^L := N] : \langle \mathsf{env}^{\varnothing}_{M[x^L:=N]} \vdash_3 \omega^{\mathsf{deg}(M[x^L:=N])}\rangle$. Because $x^L \in \mathsf{fv}(M)$, we have $\mathsf{fv}(M[x^L := N]) = (\mathsf{fv}(M) \setminus \{x^L\}) \cup \mathsf{fv}(N)$. We can prove that $\mathsf{env}^{\varnothing}_{\mathsf{fv}(M)\setminus\{x^L\}} \sqcap \Delta \sqsubseteq \mathsf{env}^{\varnothing}_{(\mathsf{fv}(M)\setminus\{x^L\})\cup\mathsf{fv}(N)} = \mathsf{env}^{\varnothing}_{M[x^L:=N]}$. Therefore, by rule $(\sqsubseteq)$, $M[x^L := N] : \langle \mathsf{env}^{\varnothing}_{\mathsf{fv}(M)\setminus\{x^L\}} \sqcap \Delta \vdash_3 \omega^{\mathsf{deg}(M)}\rangle$.

- Case $(\to_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma, x^L : U, y^K : U' \vdash_3 T\rangle}{\lambda y^K.M : \langle \Gamma, x^L : U \vdash_3 U' \to T\rangle}$ such that $\forall K'.\ y^{K'} \notin \mathsf{fv}(N) \cup \{x^L\}$.

  So $(\lambda y^K.M)[x^L := N] = \lambda y^K.M[x^L := N]$. By Lemma B.1.1.2b, $M \diamond N$. By Theorem 7.3.5, $y^K \notin \mathsf{dom}(\Delta)$. By IH, $M[x^L := N] : \langle \Gamma \sqcap \Delta, y^K : U' \vdash_3 T\rangle$. By rule $(\to_{\mathsf{I}})$, $(\lambda y^K.M)[x^L := N] : \langle \Gamma \sqcap \Delta \vdash_3 U' \to T\rangle$.

- Case $(\to_{\mathsf{I}}')$: Let $\dfrac{M : \langle \Gamma, x^L : U \vdash_3 T\rangle \quad y^K \notin \mathsf{dom}(\Gamma, x^L : U)}{\lambda y^K.M : \langle \Gamma, x^L : U \vdash_3 \omega^K \to T\rangle}$ such that $\forall K'.\ y^{K'} \notin \mathsf{fv}(N) \cup \{x^L\}$.

  So $(\lambda y^K.M)[x^L := N] = \lambda y^K.M[x^L := N]$. By Lemma B.1.1.2b, $M \diamond N$. By IH, $M[x^L := N] : \langle \Gamma \sqcap \Delta \vdash_3 T\rangle$. By Theorem 7.3.5, $y^K \notin \mathsf{dom}(\Delta)$. By rule $(\to_{\mathsf{I}}')$, $(\lambda y^K.M)[x^L := N] : \langle \Gamma \sqcap \Delta \vdash_3 \omega^K \to T\rangle$.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{M_1 : \langle \Gamma_1, x^L : U_1 \vdash_3 V \to T\rangle \quad M_2 : \langle \Gamma_2, x^L : U_2 \vdash_3 V\rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2, x^L : U_1 \sqcap U_2 \vdash_3 T\rangle}$ where we consider $x^L \in \mathsf{fv}(M_1) \cap \mathsf{fv}(M_2)$, using Theorem 7.3.5.2a, and where $N : \langle \Delta \vdash_3 U_1 \sqcap U_2\rangle$.

  By Lemma B.1.1.2a, $M_1 \diamond N$ and $M_2 \diamond N$. By rules $(\sqcap_{\mathsf{E}})$ and $(\sqsubseteq)$, $N : \langle \Delta \vdash_3 U_1\rangle$ and $N : \langle \Delta \vdash_3 U_2\rangle$. By IH $M_1[x^L := N] : \langle \Gamma_1 \sqcap \Delta \vdash_3 V \to T\rangle$ and $M_1[x^L := N] : \langle \Gamma_2 \sqcap \Delta \vdash_3 V\rangle$. By Theorem 7.3.5.2a and Lemma B.1.1.3, $\Gamma_1 \sqcap \Delta \diamond \Gamma_2 \sqcap \Delta$. Finally by rule $(\to_{\mathsf{E}})$, $M[x^L := N] : \langle \Gamma_1 \sqcap \Gamma_2 \sqcap \Delta \vdash_3 T\rangle$.

  The cases $x^L \in \mathsf{fv}(M_1) \setminus \mathsf{fv}(M_2)$ or $x^L \in \mathsf{fv}(M_2) \setminus \mathsf{fv}(M_1)$ are similar.

- Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma, x^L : U \vdash_3 U_1\rangle \quad M : \langle \Gamma, x^L : U \vdash_3 U_2\rangle}{M : \langle \Gamma, x^L : U \vdash_3 U_1 \sqcap U_2\rangle}$.

  Use IH and rule $(\sqcap_{\mathsf{I}})$.

- Case (exp): Let $\dfrac{M : \langle \Gamma, x^L : U \vdash_3 V\rangle}{M^{+i} : \langle \mathsf{e}_i\Gamma, x^{i::L} : \mathsf{e}_i U \vdash_3 \mathsf{e}_i V\rangle}$ and $N : \langle \Delta \vdash_3 \mathsf{e}_i U\rangle$.

  By Theorem 7.3.5, $\mathsf{deg}(N) = \mathsf{deg}(\mathsf{e}_i U) = i :: \mathsf{deg}(U)$. and $N^{-i} : \langle \Delta^{-i} \vdash_3 U\rangle$. By Lemma B.1.5, $(N^{-i})^{+i} = N$ and $M \diamond N^{-i}$. By IH, $M[x^L := N^{-i}] : \langle \Gamma \sqcap \Delta^{-i} \vdash_3 V\rangle$. By rule (exp) and Lemma B.1.5.5, $M^{+i}[x^{i::L} := N] : \langle \mathsf{e}_i\Gamma \sqcap \Delta \vdash_3 \mathsf{e}_i V\rangle$.

- Case $(\sqsubseteq)$: Let $\dfrac{M : \langle \Gamma', x^L : U' \vdash_3 V'\rangle \quad \Gamma', x^L : U' \vdash_3 V' \sqsubseteq \Gamma, x^L : U \vdash_3 V}{M : \langle \Gamma, x^L : U \vdash_3 V\rangle}$ (using Lemma 7.3.4).

By Lemma 7.3.4, $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$, $\Gamma \sqsubseteq \Gamma'$, $U \sqsubseteq U'$ and $V' \sqsubseteq V$. Hence $N : \langle \Delta \vdash_3 U' \rangle$ and, by IH, $M[x^L := N] : \langle \Gamma' \sqcap \Delta \vdash_3 V' \rangle$. It is easy to show that $\Gamma \sqcap \Delta \sqsubseteq \Gamma' \sqcap \Delta$. Hence, $\Gamma' \sqcap \Delta \vdash_3 V' \sqsubseteq \Gamma \sqcap \Delta \vdash_3 V$ and $M[x^L := N] : \langle \Gamma \sqcap \Delta \vdash_3 V \rangle$. □

The next lemma is needed in the proofs.

Lemma B.1.19.

1. *If* $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$ *then* $\mathsf{env}^{\varnothing}_M\upharpoonright_N = \mathsf{env}^{\varnothing}_N$.
2. *If* $\mathsf{ok}(\Gamma_1)$, $\mathsf{ok}(\Gamma_2)$, $\mathsf{fv}(M) \subseteq \mathsf{dom}(\Gamma_1)$ *and* $\mathsf{fv}(N) \subseteq \mathsf{dom}(\Gamma_2)$ *then* $(\Gamma_1 \sqcap \Gamma_2)\upharpoonright_{MN} \sqsubseteq (\Gamma_1\upharpoonright_M) \sqcap (\Gamma_2\upharpoonright_N)$.
3. $\mathsf{e}_i(\Gamma\upharpoonright_M) = (\mathsf{e}_i\Gamma)\upharpoonright_{M^{+i}}$ □

Proof of Lemma B.1.19.

1. Let $\mathsf{fv}(M) = \mathsf{fv}(N) \cup \{x_1^{L_1}, \ldots, x_n^{L_n}\}$. Then $\mathsf{env}^{\varnothing}_M = \mathsf{env}^{\varnothing}_N, (x_i^{L_i} : \omega^{L_i})_n$. and $\mathsf{env}^{\varnothing}_M\upharpoonright_N = \mathsf{env}^{\varnothing}_N$.
2. By Lemma B.1.13.1a, $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$. Also, $\mathsf{ok}(\Gamma_1\upharpoonright_M)$, $\mathsf{ok}(\Gamma_2\upharpoonright_N)$ and $\mathsf{dom}((\Gamma_1 \sqcap \Gamma_2)\upharpoonright_{MN}) = \mathsf{fv}(MN) = \mathsf{fv}(M) \cup \mathsf{fv}(N) = \mathsf{dom}(\Gamma_1\upharpoonright_M) \cup \mathsf{dom}(\Gamma_2\upharpoonright_N) = \mathsf{dom}((\Gamma_1\upharpoonright_M) \sqcap (\Gamma_2\upharpoonright_N))$. Now, we show by cases that if $((\Gamma_1 \sqcap \Gamma_2)\upharpoonright_{MN})(x^L) = U_1$ and $((\Gamma_1\upharpoonright_M) \sqcap (\Gamma_2\upharpoonright_N))(x^L) = U_2$ then $U_1 \sqsubseteq U_2$:
   - If $x^L \in \mathsf{fv}(M) \cap \mathsf{fv}(N)$ then $\Gamma_1(x^L) = U_1'$, $\Gamma_2(x^L) = U_1''$, and $U_1 = U_1' \sqcap U_1'' = U_2$.
   - If $x^L \in \mathsf{fv}(M) \setminus \mathsf{fv}(N)$ then:
     - If $x^L \in \mathsf{dom}(\Gamma_2)$ then $\Gamma_1(x^L) = U_2$, $\Gamma_2(x^L) = U_1'$ and $U_1 = U_1' \sqcap U_2 \sqsubseteq U_2$.
     - If $x^L \notin \mathsf{dom}(\Gamma_2)$ then $\Gamma_1(x^L) = U_2$ and $U_1 = U_2$.
   - If $x^L \in \mathsf{fv}(N) \setminus \mathsf{fv}(M)$ then:
     - If $x^L \in \mathsf{dom}(\Gamma_1)$ then $\Gamma_1(x^L) = U_1'$, $\Gamma_2(x^L) = U_2$ and $U_1 = U_1' \sqcap U_2 \sqsubseteq U_2$.
     - If $x^L \notin \mathsf{dom}(\Gamma_1)$ then $\Gamma_2(x^L) = U_2$ and $U_1 = U_2$.
3. Let $\Gamma = (x_j^{L_j} : U_j)_n, (y_j^{L_j'} : U_j')_p$ and let $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\} \uplus \{z_1^{L_1''}, \ldots, z_m^{L_m''}\}$. such that $\mathsf{dj}(\{y_1^{L_1'}, \ldots, y_p^{L_p'}\}, \{z_1^{L_1''}, \ldots, z_m^{L_m''}\})$. Therefore $\Gamma\upharpoonright_M = (x_j^{L_j} : U_j)_n$ and $\mathsf{e}_i(\Gamma\upharpoonright_M) = (x_j^{i::L_j} : \mathsf{e}_iU_j)_n$. Because $\mathsf{e}_i\Gamma = (x_j^{i::L_j} : \mathsf{e}_iU_j)_n, (y_j^{i::L_j'} : \mathsf{e}_iU_j')_p$, and by Lemma B.1.5.1, $\mathsf{fv}(M^{+i}) = \{x_1^{i::L_1}, \ldots, x_n^{i::L_n}\} \uplus \{z_1^{i::L_1''}, \ldots, z_m^{i::L_m''}\}$ such that $\mathsf{dj}(\{y_1^{i::L_1'}, \ldots, y_p^{i::L_p'}\}, \{z_1^{i::L_1''}, \ldots, z_m^{i::L_m''}\})$, then $(\mathsf{e}_i\Gamma)\upharpoonright_{M^{+i}} = (x_j^{i::L_j} : \mathsf{e}_iU_j)_n$. □

The next two theorems are needed in the proof of subject reduction.

Theorem B.1.20. *If $M : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrowtriangle_\beta N$ then $N : \langle \Gamma\upharpoonright_N \vdash_3 U \rangle$.* □

Proof of Lemma B.1.20. By induction on the derivation $M : \langle \Gamma \vdash_3 U \rangle$.

- Case (ω) follows by Theorem 7.1.11.2 and Lemma B.1.19.1.
- Case $(\to_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma, (x^L : U) \vdash_3 T \rangle}{\lambda x^L.M : \langle \Gamma \vdash_3 U \to T \rangle}$.

  Then $N = \lambda x^L.N'$ and $M \rightarrowtriangle_\beta N'$. By IH, $N' : \langle (\Gamma, (x^L : U))\upharpoonright_{N'} \vdash_3 T \rangle$. If $x^L \in \mathsf{fv}(N')$ then $N' : \langle \Gamma\upharpoonright_{\mathsf{fv}(\lambda x^L.N')}, (x^L : U) \vdash_3 T \rangle$ and by rule $(\to_{\mathsf{I}})$, $\lambda x^L.N' : \langle \Gamma\upharpoonright_{\lambda x^L.N'} \vdash_3 U \to T \rangle$. Else $N' : \langle \Gamma\upharpoonright_{\mathsf{fv}(\lambda x^L.N')} \vdash_3 T \rangle$ so by rule $(\to'_{\mathsf{I}})$, $\lambda x^L.N' : \langle \Gamma\upharpoonright_{\lambda x^L.N'} \vdash_3 \omega^L \to T \rangle$ and since by Theorem 7.3.5.2 and Lemma 7.3.6.4, $U \sqsubseteq \omega^L$, by rule $(\sqsubseteq)$, $\lambda x^L.N' : \langle \Gamma\upharpoonright_{\lambda x^L.N'} \vdash_3 U \to T \rangle$.
- Case $(\to'_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma \vdash_3 T \rangle \quad x^L \notin \mathsf{dom}(\Gamma)}{\lambda x^L.M : \langle \Gamma \vdash_3 \omega^L \to T \rangle}$.

  Then $N = \lambda x^L.N'$ and $M \rightarrowtriangle_\beta N'$. Because $x^L \notin \mathsf{fv}(M)$ (using Theorem 7.3.5), by Theorem 7.1.11.2, $x^L \notin \mathsf{fv}(N')$. By IH, $N' : \langle \Gamma\upharpoonright_{\mathsf{fv}(N')\setminus\{x^L\}} \vdash_3 T \rangle$ so by rule $(\to'_{\mathsf{I}})$, $\lambda x^L.N' : \langle \Gamma\upharpoonright_{\lambda x^L.N'} \vdash_3 \omega^L \to T \rangle$.
- Case $(\to_{\mathsf{E}})$: Let $\dfrac{M_1 : \langle \Gamma_1 \vdash_3 U \to T \rangle \quad M_2 : \langle \Gamma_2 \vdash_3 U \rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle}$.

  Using Lemma B.1.19.2, case $M_1 \rightarrowtriangle_\beta N_1$ and $N = N_1 M_2$ and case $M_2 \rightarrowtriangle_\beta N_2$ and $N = M_1 N_2$ are easy. Let $M_1 = \lambda x^L.M'_1$ and $N = M'_1[x^L := M_2]$. By Lemma 7.3.7.3 and Lemma B.1.1.2, $M'_1 \diamond M_2$. If $x^L \in \mathsf{fv}(M'_1)$ then by Lemma 7.4.7.2, $M'_1 : \langle \Gamma_1, x^L : U \vdash_3 T \rangle$. By Lemma 7.4.8, $M'_1[x^L := M_2] : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle$. If $x^L \notin \mathsf{fv}(M'_1)$ then by Lemma 7.4.7.3, $M'_1[x^L := M_2] = M'_1 : \langle \Gamma_1 \vdash_3 T \rangle$ and by rule $(\sqsubseteq)$, $N : \langle (\Gamma_1 \sqcap \Gamma_2)\upharpoonright_N \vdash_3 T \rangle$.
- Case $(\sqcap_{\mathsf{I}})$ is by IH.
- case (exp): Let $\dfrac{M : \langle \Gamma \vdash_3 U \rangle}{M^{+i} : \langle \mathsf{e}_i\Gamma \vdash_3 \mathsf{e}_i U \rangle}$.

  If $M^{+i} \rightarrowtriangle_\beta N$ then by Lemma B.1.5.9, there is $P \in \mathcal{M}_3$ such that $P^{+i} = N$ and $M \rightarrowtriangle_\beta P$. By IH, $P : \langle \Gamma\upharpoonright_P \vdash_3 U \rangle$ and by rule (exp) and Lemma B.1.19.3, $N : \langle (\mathsf{e}_i\Gamma)\upharpoonright_N \vdash_3 \mathsf{e}_i U \rangle$.
- Case $(\sqsubseteq)$: Let $\dfrac{M : \langle \Gamma \vdash_3 U \rangle \quad \Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'}{M : \langle \Gamma' \vdash_3 U' \rangle}$.

  Then by IH, Lemma 7.3.4.3 and rule $(\sqsubseteq)$, $N : \langle \Gamma'\upharpoonright_N \vdash_3 U' \rangle$. □

Theorem B.1.21. *If $M : \langle \Gamma \vdash_3 U \rangle$ and $M \rightarrowtriangle_\eta N$ then $N : \langle \Gamma \vdash_3 U \rangle$.* □

Proof of Lemma B.1.21. By induction on the derivation $M : \langle \Gamma \vdash_3 U \rangle$.

- Case (ω): Let $\overline{M : \langle \mathsf{env}^{\varnothing}_{M} \vdash_3 \omega^{\mathsf{deg}(M)} \rangle}$.

  Then by Lemma 7.1.11.1, $\mathsf{deg}(M) = \mathsf{deg}(N)$ and $\mathsf{fv}(M) = \mathsf{fv}(N)$, and by rule (ω), $N : \langle \mathsf{env}^{\varnothing}_{N} \vdash_3 \omega^{\mathsf{deg}(N)} \rangle$.

- Case $(\rightarrow_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma, (x^L : U) \vdash_3 T \rangle}{\lambda x^L.M : \langle \Gamma \vdash_3 U \rightarrow T \rangle}$.

  then we have two cases:

  - $M = N x^L$ such that $x^L \notin \mathsf{fv}(N)$ and so by Lemma 7.4.7.4, $N : \langle \Gamma \vdash_3 U \rightarrow T \rangle$.
  - $N = \lambda x^L.N'$ and $M \rightarrowtriangle_{\eta} N'$. By IH, $N' : \langle \Gamma, (x^L : U) \vdash_3 T \rangle$ and by rule $(\rightarrow_{\mathsf{I}})$, $N : \langle \Gamma \vdash_3 U \rightarrow T \rangle$.

- Case $(\rightarrow'_{\mathsf{I}})$: Let $\dfrac{M : \langle \Gamma \vdash_3 T \rangle \quad x^L \notin \mathsf{dom}(\Gamma)}{\lambda x^L.M : \langle \Gamma \vdash_3 \omega^L \rightarrow T \rangle}$.

  Therefore by Theorem 7.3.5, $x^L \notin \mathsf{fv}(M)$. Then $N = \lambda x^L.N'$ and $M \rightarrowtriangle_{\eta} N'$. By Lemma 7.1.11.1, $\mathsf{fv}(M) = \mathsf{fv}(N')$. By IH, $N' : \langle \Gamma \vdash_3 T \rangle$, and by rule $(\rightarrow'_{\mathsf{I}})$, $N : \langle \Gamma \vdash_3 \omega^L \rightarrow T \rangle$.

- Case $(\rightarrow_{\mathsf{E}})$: Let $\dfrac{M_1 : \langle \Gamma_1 \vdash_3 U \rightarrow T \rangle \quad M_2 : \langle \Gamma_2 \vdash_3 U \rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle}$.

  Then we have two cases:

  - $M_1 \rightarrowtriangle_{\eta} N_1$ and $N = N_1 M_2$. By IH $N_1 : \langle \Gamma_1 \vdash_3 U \rightarrow T \rangle$ and by rule $(\rightarrow_{\mathsf{E}})$, $N : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle$.
  - $M_2 \rightarrowtriangle_{\eta} N_2$ and $N = M_1 N_2$. By IH $N_2 : \langle \Gamma_2 \vdash_3 U \rangle$ and by rule $(\rightarrow_{\mathsf{E}})$, $N : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle$.

- Case $(\sqcap_{\mathsf{I}})$ is by IH and rule $(\sqcap_{\mathsf{I}})$.

- Case (exp): Let $\dfrac{M : \langle \Gamma \vdash_3 U \rangle}{M^{+i} : \langle \mathsf{e}_i \Gamma \vdash_3 \mathsf{e}_i U \rangle}$.

  Then by Lemma B.1.5.9, there exists $P \in \mathcal{M}_3$ such that $P^{+i} = N$ and $M \rightarrowtriangle_{\eta} P$. By IH, $P : \langle \Gamma \vdash_3 U \rangle$ and by rule (exp), $N : \langle \mathsf{e}_i \Gamma \vdash_3 \mathsf{e}_i U \rangle$.

- Case $(\sqsubseteq)$: Let $\dfrac{M : \langle \Gamma \vdash_3 U \rangle \quad \Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'}{M : \langle \Gamma' \vdash_3 U' \rangle}$.

  Then by IH and rule $(\sqsubseteq)$, $N : \langle \Gamma' \vdash_3 U' \rangle$. □

Proof of Theorem 7.4.10. Proof is by induction on the reduction $M \rightarrowtriangle^{*}_{\beta\eta} N$ using Theorem B.1.20 and Theorem B.1.21. □

Proof of Lemma 7.4.12. By Theorem 7.3.5.2, we have $M[x^L := N] \in \mathcal{M}_3$. By Lemma B.1.1.5a, $M \diamond N$ and $\mathsf{deg}(N) = L$. Let us prove the result by induction on the derivation $M[x^L := N] : \langle \Gamma \vdash_3 U \rangle$ and then by case on the last rule of the derivation.

- Case (ax): Let $\overline{y^{\oslash} : \langle (y^{\oslash} : T) \vdash_3 T \rangle}$

  Then $M = x^{\oslash}$ and $N = y^{\oslash}$. By rule (ax), $x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T \rangle$.

- Case (ω): Let $\overline{M[x^L := N] : \langle \mathsf{env}^{\varnothing}_{M[x^L:=N]} \vdash_3 \omega^{\mathsf{deg}(M[x^L:=N])} \rangle}$.

  By Lemma B.1.1.5b, $\mathsf{deg}(M) = \mathsf{deg}(M[x^L := N])$. Therefore, by rule (ω), $M : \langle \mathsf{env}^{\varnothing}_{\mathsf{fv}(M)\setminus\{x^L\}}, (x^L : \omega^L) \vdash_3 \omega^{\mathsf{deg}(M)} \rangle$ and $N : \langle \mathsf{env}^{\varnothing}_N \vdash_3 \omega^L \rangle$ and because $\mathsf{fv}(M[x^L := N]) = (\mathsf{fv}(M) \setminus \{x^L\}) \cup \mathsf{fv}(N)$, $\mathsf{env}^{\varnothing}_{\mathsf{fv}(M)\setminus\{x^L\}} \sqcap \mathsf{env}^{\varnothing}_N = \mathsf{env}^{\varnothing}_{M[x^L:=N]}$.

- Case $(\to_{\mathsf{I}})$: Let $\dfrac{M[x^L := N] : \langle \Gamma, (y^K : W) \vdash_3 T \rangle}{\lambda y^K.M[x^L := N] : \langle \Gamma \vdash_3 W \to T \rangle}$ where $\forall K'.\ y^{K'} \notin \mathsf{fv}(N) \cup \{x^L\}$.

  By IH, $\exists\ V, \Gamma_1, \Gamma_2$ such that $M : \langle \Gamma_1, x^L : V \vdash_3 T \rangle$, $N : \langle \Gamma_2 \vdash_3 V \rangle$ and $(\Gamma, y^K : W) = \Gamma_1 \sqcap \Gamma_2$. By Theorem 7.3.5.2a, $\mathsf{fv}(N) = \mathsf{dom}(\Gamma_2)$ and $\mathsf{fv}(M) = \mathsf{dom}(\Gamma_1) \cup \{x^L\}$. Because $y^K \notin \mathsf{fv}(N)$, $x^K \notin \mathsf{dom}(\Gamma_2)$ and $\Gamma_1 = \Delta_1, y^K : W$. Hence $M : \langle \Delta_1, y^K : W, x^L : V \vdash_3 T \rangle$. By rule $(\to_{\mathsf{I}})$, $\lambda y^K.M : \langle \Delta_1, x^L : V \vdash_3 W \to T \rangle$. Finally, $\Gamma = \Delta_1 \sqcap \Gamma_2$.

- Case $(\to'_{\mathsf{I}})$: Let $\dfrac{M[x^L := N] : \langle \Gamma \vdash_3 T \rangle \quad y^K \notin \mathsf{dom}(\Gamma)}{\lambda y^K.M[x^L := N] : \langle \Gamma \vdash_3 \omega^K \to T \rangle}$ where $\forall K'.\ y^{K'} \notin \mathsf{fv}(N) \cup \{x^L\}$.

  By IH, $\exists\ V, \Gamma_1, \Gamma_2$ such that $M : \langle \Gamma_1, x^L : V \vdash_3 T \rangle$, $N : \langle \Gamma_2 \vdash_3 V \rangle$ and $\Gamma = \Gamma_1 \sqcap \Gamma_2$. Since $y^K \notin \mathsf{dom}(\Gamma_1) \cup \{x^L\}$, $\lambda y^K.M : \langle \Gamma_1, x^L : V \vdash_3 \omega^K \to T \rangle$.

- Case $(\to_{\mathsf{E}})$: Let $\dfrac{M_1[x^L := N] : \langle \Gamma_1 \vdash_3 W \to T \rangle \quad M_2[x^L := N] : \langle \Gamma_2 \vdash_3 W \rangle}{M_1[x^L := N]M_2[x^L := N] : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle}$ where $\Gamma_1 \diamond \Gamma_2$ and $M = M_1M_2$.

  By Lemmas B.1.1.1 and B.1.1.2a, $M_1 \diamond M_2$, We have three cases:

  - If $x^L \in \mathsf{fv}(M_1) \cap \mathsf{fv}(M_2)$ then by IH, $\exists\ V_1, V_2, \Delta_1, \Delta_2, \Delta'_1, \Delta'_2$ such that $M_1 : \langle \Delta_1, (x^L : V_1) \vdash_3 W \to T \rangle$, $M_2 : \langle \Delta'_1, (x^L : V_2) \vdash_3 W \rangle$, $N : \langle \Delta_2 \vdash_3 V_1 \rangle$, $N : \langle \Delta'_2 \vdash_3 V_2 \rangle$, $\Gamma_1 = \Delta_1 \sqcap \Delta_2$ and $\Gamma_2 = \Delta'_1 \sqcap \Delta'_2$. Because $M_1 \diamond M_2$, then by Lemma B.1.15.2, $(\Delta_1, (x^L : V_1)) \diamond (\Delta'_1, (x^L : V_2))$. Then, by rule $(\to_{\mathsf{E}})$, $M_1M_2 : \langle \Delta_1 \sqcap \Delta'_1, (x^L : V_1 \sqcap V_2) \vdash_3 T \rangle$ and by rule $(\sqcap'_{\mathsf{I}})$, $N : \langle \Delta_2 \sqcap \Delta'_2 \vdash_3 V_1 \sqcap V_2 \rangle$. Finally, $\Gamma_1 \sqcap \Gamma_2 = \Delta_1 \sqcap \Delta_2 \sqcap \Delta'_1 \sqcap \Delta'_2$.
  - If $x^L \in \mathsf{fv}(M_1) \setminus \mathsf{fv}(M_2)$ then $M_2[x^L := N] = M_2$ and by IH, $\exists\ V, \Delta_1, \Delta_2$ such that $M_1 : \langle \Delta_1, (x^L : V) \vdash_3 W \to T \rangle$, $N : \langle \Delta_2 \vdash_3 V \rangle$, and $\Gamma_1 = \Delta_1 \sqcap \Delta_2$.

  Because $M_1 \diamond M_2$, then by Lemma B.1.15.2, $(\Delta_1, (x^L : V)) \diamond \Gamma_2$. By rule $(\to_{\mathsf{E}})$, $M_1 M_2 : \langle \Delta_1 \sqcap \Gamma_2, (x^L : V) \vdash_3 T\rangle$ and $\Gamma_1 \sqcap \Gamma_2 = \Delta_1 \sqcap \Delta_2 \sqcap \Gamma_2$.

  - If $x^L \in \mathsf{fv}(M_2) \setminus \mathsf{fv}(M_1)$ then $M_1[x^L := N] = M_2$ and by IH, $\exists\ V, \Delta_1, \Delta_2$ such that $M_2 : \langle \Delta_1, (x^L : V) \vdash_3 W\rangle$, $N : \langle \Delta_2 \vdash_3 V\rangle$, and $\Gamma_2 = \Delta_1 \sqcap \Delta_2$. Because $M_1 \diamond M_2$, then by Lemma B.1.15.2, $(\Delta_1, (x^L : V)) \diamond \Gamma_1$. By rule $(\to_{\mathsf{E}})$, $M_1 M_2 : \langle \Gamma_1 \sqcap \Delta_1, (x^L : V) \vdash_3 T\rangle$ and $\Gamma_1 \sqcap \Gamma_2 = \Gamma_1 \sqcap \Delta_1 \sqcap \Delta_2$.

- Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{M[x^L := N] : \langle \Gamma \vdash_3 U_1\rangle \quad M[x^L := N] : \langle \Gamma \vdash_3 U_2\rangle}{M[x^L := N] : \langle \Gamma \vdash_3 U_1 \sqcap U_2\rangle}$.

  By IH, $\exists\ V_1, V_2, \Delta_1, \Delta_2, \Delta'_1, \Delta'_2$ such that $M : \langle \Delta_1, x^L : V_1 \vdash_3 U_1\rangle$, $M : \langle \Delta'_1, x^L : V_2 \vdash_3 U_2\rangle$, $N : \langle \Delta_2 \vdash_3 V_1\rangle$, $N : \langle \Delta'_2 \vdash_3 V_2\rangle$, and $\Gamma = \Delta_1 \sqcap \Delta_2 = \Delta'_1 \sqcap \Delta'_2$. By rule $(\sqcap'_{\mathsf{I}})$, $M : \langle \Delta_1 \sqcap \Delta'_1, x^L : V_1 \sqcap V_2 \vdash_3 U_1 \sqcap U_2\rangle$ and $N : \langle \Delta_2 \sqcap \Delta'_2 \vdash_3 V_1 \sqcap V_2\rangle$. Finally, $\Gamma = \Delta_1 \sqcap \Delta_2 \sqcap \Delta'_1 \sqcap \Delta'_2$.

- Case (exp): Let $\dfrac{M[x^L := N] : \langle \Gamma \vdash_3 U\rangle}{M^{+j}[x^{j::L} := N^{+j}] : \langle \mathsf{e}_j\Gamma \vdash_3 \mathsf{e}_j U\rangle}$ using Lemma B.1.5.5.

  By IH, $\exists\ V, \Gamma_1, \Gamma_2$ such that $M : \langle \Gamma_1, x^L : V \vdash_3 U\rangle$, $N : \langle \Gamma_2 \vdash_3 V\rangle$ and $\Gamma = \Gamma_1 \sqcap \Gamma_2$. So by rule (exp), $M^{+j} : \langle \mathsf{e}_j\Gamma_1, x^{j::L} : \mathsf{e}_j V \vdash_3 \mathsf{e}_j U\rangle$, $N : \langle \mathsf{e}_j\Gamma_2 \vdash_3 \mathsf{e}_j V\rangle$ and $\mathsf{e}_j\Gamma = \mathsf{e}_j\Gamma_1 \sqcap \mathsf{e}_j\Gamma_2$.

- Case $(\sqsubseteq)$: Let $\dfrac{M[x^L := N] : \langle \Gamma' \vdash_3 U'\rangle \quad \Gamma' \vdash_3 U' \sqsubseteq \Gamma \vdash_3 U}{M[x^L := N] : \langle \Gamma \vdash_3 U\rangle}$.

  By Lemma 7.3.4.2, $\Gamma \sqsubseteq \Gamma'$ and $U' \sqsubseteq U$. By IH, $\exists\ V, \Gamma'_1, \Gamma'_2$ such that $M : \langle \Gamma'_1, x^L : V \vdash_3 U'\rangle$, $N : \langle \Gamma'_2 \vdash_3 V\rangle$ and $\Gamma' = \Gamma'_1 \sqcap \Gamma'_2$. By Lemma B.1.12.5, $\Gamma = \Gamma_1 \sqcap \Gamma_2$, $\Gamma_1 \sqsubseteq \Gamma'_1$, and $\Gamma_2 \sqsubseteq \Gamma'_2$. Finally, by rule $(\sqsubseteq)$, $M : \langle \Gamma_1, x^L : V \vdash_3 U\rangle$ and $N : \langle \Gamma_2 \vdash_3 V\rangle$. □

The next lemma is useful to prove that subject expansion w.r.t. β holds in $\vdash_3$.

Lemma B.1.22. *If $M[x^L := N] : \langle \Gamma \vdash_3 U\rangle$, $L \succeq \mathsf{deg}(M)$, and $\overline{ix} = \mathsf{fv}((\lambda x^L.M)N)$ then $(\lambda x^L.M)N : \langle \Gamma\uparrow^{\overline{ix}} \vdash_3 U\rangle$.* □

Proof of Lemma B.1.22. Let $\mathsf{deg}(U) = K$. By Theorem 7.3.5.2, $M[x^L := N] \in \mathcal{M}_3$. By Lemma B.1.1.5a, $M \diamond N$ and $\mathsf{deg}(N) = L$. By definition $\lambda x^L.M \in \mathcal{M}_3$. By Lemma B.1.1.2a, $\lambda x^L.M \diamond N$. By definition, $(\lambda x^L.M)N \in \mathcal{M}_3$. By Lemma B.1.1.5b and Theorem 7.3.5.2, $\mathsf{deg}(\Gamma) \succeq \mathsf{deg}(U) = K = \mathsf{deg}(M[x^L := N]) = \mathsf{deg}(M) = \mathsf{deg}((\lambda x^L.M)N)$. So $L \succeq K$ and there exists K' such that $L = K :: K'$. We have two cases:

- If $x^L \in \mathsf{fv}(M)$ then, by Lemma 7.4.12, $\exists\ V, \Gamma_1, \Gamma_2$ such that $M : \langle \Gamma_1, x^L : V \vdash_3 U\rangle$, $N : \langle \Gamma_2 \vdash_3 V\rangle$, and $\Gamma = \Gamma_1 \sqcap \Gamma_2$. By Theorem 7.3.5.2, $\mathsf{ok}(\Gamma_1)$ and $\mathsf{ok}(\Gamma_2)$. By Lemma B.1.13.1a, $\mathsf{ok}(\Gamma_1 \sqcap \Gamma_2)$. So, it is easy to prove, using Lemma B.1.13.5, that $\mathsf{ok}(\Gamma\uparrow^{\overline{ix}})$. By Lemma 7.3.7.3, $(\Gamma_1, x^L : V) \diamond \Gamma_2$, so $\Gamma_1 \diamond \Gamma_2$.

By Theorem 7.3.5.2, $\mathsf{deg}(\Gamma_1) \succeq \mathsf{deg}(M) = \mathsf{deg}(U) = K$ and $L = \mathsf{deg}(N) = \mathsf{deg}(V) \preceq \mathsf{deg}(\Gamma_2)$. By Lemma B.1.12.2, we have two cases :

- If $U = \omega^K$ then by Lemma 7.3.7.2, $(\lambda x^L.M)N : \langle \Gamma\uparrow^{\overline{ix}} \vdash_3 U\rangle$.
- If $U = \vec{\mathsf{e}}_K \sqcap_{i=1}^p T_i$ where $p \geq 1$ and $\forall i \in \{1,\dots,p\}$. $T_i \in \mathsf{Ty}_3$ then by Theorem 7.3.5.2, $M^{-K} : \langle \Gamma_1^{-K}, x^{K'} : V^{-K} \vdash_3 \sqcap_{i=1}^p T_i\rangle$. By rule $(\sqsubseteq)$, $\forall i \in \{1,\dots,p\}$. $M^{-K} : \langle \Gamma_1^{-K}, x^{K'} : V^{-K} \vdash_3 T_i\rangle$, so by rule $(\to_\mathsf{I})$, $\lambda x^{K'}.M^{-K} : \langle \Gamma_1^{-K} \vdash_3 V^{-K} \to T_i\rangle$. Again by Theorem 7.3.5.2, $N^{-K} : \langle \Gamma_2^{-K} \vdash_3 V^{-K}\rangle$ and because $\Gamma_1 \diamond \Gamma_2$, then by Lemma B.1.13.4, $\Gamma_1^{-K} \diamond \Gamma_2^{-K}$, so by rule $(\to_\mathsf{E})$, $\forall i \in \{1,\dots,p\}$. $(\lambda x^{K'}.M^{-K})N^{-K} : \langle \Gamma_1^{-K} \sqcap \Gamma_2^{-K} \vdash_3 T_i\rangle$. Finally by rules $(\sqcap_\mathsf{I})$ and (exp), $(\lambda x^L.M)N : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 U\rangle$, so $(\lambda x^L.M)N : \langle \Gamma\uparrow^{\overline{ix}} \vdash_3 U\rangle$.

- If $x^L \notin \mathsf{fv}(M)$ then $M : \langle \Gamma \vdash_3 U\rangle$. By Theorem 7.3.5.2, $\mathsf{ok}(\Gamma)$. So, it is easy to prove, using Lemma B.1.13.5, that $\mathsf{ok}(\Gamma\uparrow^{\overline{ix}})$. By Lemma B.1.12.2, we have two cases :
  - If $U = \omega^K$ then by Lemma 7.3.7.2, $(\lambda x^L.M)N : \langle \Gamma\uparrow^{\overline{ix}} \vdash_3 U\rangle$.
  - If $U = \vec{\mathsf{e}}_K \sqcap_{i=1}^p T_i$ where $p \geq 1$ and $\forall i \in \{1,\dots,p\}$. $T_i \in \mathsf{Ty}_3$, and by Theorem 7.3.5.2, $M^{-K} : \langle \Gamma^{-K} \vdash_3 \sqcap_{i=1}^p T_i\rangle$. By rule $(\sqsubseteq)$, $\forall i \in \{1,\dots,p\}$. $M^{-K} : \langle \Gamma^{-K} \vdash_3 T_i\rangle$. Using Lemma B.1.5.1 and by induction on K, we can prove that $x^{K'} \notin \mathsf{fv}(M^{-K})$. So by Theorem 7.3.5.2a, $x^{K'} \notin \mathsf{dom}(\Gamma^{-K})$. So by rule $(\to_\mathsf{I}')$, $\lambda x^{K'}.M^{-K} : \langle \Gamma^{-K} \vdash_3 \omega^{K'} \to T_i\rangle$. By rule (ω), $N^{-K} : \langle \mathsf{env}^{\varnothing}_{N^{-K}} \vdash_3 \omega^{K'}\rangle$ and $N : \langle \mathsf{env}^{\varnothing}_N \vdash_3 \omega^L\rangle$. By Theorem 7.3.5.2, $\mathsf{deg}(\mathsf{env}^{\varnothing}_N) \succeq \mathsf{deg}(N) = L$. By Lemma 7.3.7.3, $\Gamma \diamond \mathsf{env}^{\varnothing}_N$. By Lemma B.1.13.4, $\Gamma^{-K} \diamond \mathsf{env}^{\varnothing}_{N^{-K}}$. By rule $(\to_\mathsf{E})$, $\forall i \in \{1,\dots,p\}$. $(\lambda x^{K'}.M^{-K})N^{-K} : \langle \Gamma^{-K} \sqcap \mathsf{env}^{\varnothing}_{N^{-K}} \vdash_3 T_i\rangle$. Finally by rules $(\sqcap_\mathsf{I})$ and (exp), $(\lambda x^L.M)N : \langle \Gamma \sqcap \mathsf{env}^{\varnothing}_N \vdash_3 U\rangle$, so $(\lambda x^L.M)N : \langle \Gamma\uparrow^{\overline{ix}} \vdash_3 U\rangle$.

□

Next, we give the main block for the proof of subject β-expansion.

Theorem B.1.23. *If $N : \langle \Gamma \vdash_3 U\rangle$ and $M \to\!\!\!\!\!\rhd_\beta N$ then $M : \langle \Gamma\uparrow^M \vdash_3 U\rangle$.* □

Proof of Lemma B.1.23. By induction on the derivation $N : \langle \Gamma \vdash_3 U\rangle$ and then by case on the last rule of the derivation.

- Case (ax): Let $\overline{x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T\rangle}$ and $M \to\!\!\!\!\!\rhd_\beta x^{\oslash}$.

  Then $M = (\lambda y^K.M_1)M_2$, and $x^{\oslash} = M_1[y^K := M_2]$. Because $M \in \mathcal{M}_3$ then $K \succeq \mathsf{deg}(M_1)$. By Lemma B.1.22, $M : \langle (x^{\oslash} : T)\uparrow^M \vdash_3 T\rangle$.

- Case (ω): Let $\overline{N : \langle \mathsf{env}^{\varnothing}_N \vdash_3 \omega^{\mathsf{deg}(N)}\rangle}$ and $M \to\!\!\!\!\!\rhd_\beta N$.

  By Theorem 7.1.11.2, $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$ and $\mathsf{deg}(M) = \mathsf{deg}(N)$. We have $(\mathsf{env}^{\varnothing}_N)\uparrow^M = \mathsf{env}^{\varnothing}_M$. By rule (ω), $M : \langle \mathsf{env}^{\varnothing}_M \vdash_3 \omega^{\mathsf{deg}(M)}\rangle$. Hence, $M : \langle (\mathsf{env}^{\varnothing}_N)\uparrow^M \vdash_3 \omega^{\mathsf{deg}(N)}\rangle$.

- Case ($\to_{\mathsf{I}}$): Let $\dfrac{N : \langle \Gamma, x^L : U \vdash_3 T\rangle}{\lambda x^L.N : \langle \Gamma \vdash_3 U \to T\rangle}$ and $M \twoheadrightarrow_\beta \lambda x^L.N$.

  We have two cases:

  - If $M = \lambda x.M'$ where $M' \twoheadrightarrow_\beta N$ then by IH, $M' : \langle (\Gamma, (x^L : U))\uparrow^{M'} \vdash_3 T\rangle$. Since by Theorem 7.1.11.2 and Theorem 7.3.5.2a, $x^L \in \mathsf{fv}(N) \subseteq \mathsf{fv}(M')$ then we have $(\Gamma, (x^L : U))\uparrow^{\mathsf{fv}(M')} = \Gamma\uparrow^{\mathsf{fv}(M')\setminus\{x^L\}}, (x^L : U)$ and $\Gamma\uparrow^{\mathsf{fv}(M')\setminus\{x^L\}} = \Gamma\uparrow^{\lambda x^L.M'}$. Hence, $M' : \langle \Gamma\uparrow^{\lambda x^L.M'}, (x^L : U) \vdash_3 T\rangle$ and finally, by rule ($\to_{\mathsf{I}}$), $\lambda x^L.M' : \langle \Gamma\uparrow^{\lambda x^L.M'} \vdash_3 U \to T\rangle$.
  - If $M = (\lambda y^K.M_1)M_2$ and $\lambda x^L.N = M_1[y^K := M_2]$ then, because $M \in \mathcal{M}_3$ then $K \succeq \mathsf{deg}(M_1)$, and by Lemma B.1.22, because $M_1[y^K := M_2] : \langle \Gamma \vdash_3 U \to T\rangle$, we have $(\lambda y^K.M_1)M_2 : \langle \Gamma\uparrow^{(\lambda y^K.M_1)M_2} \vdash_3 U \to T\rangle$.

- Case ($\to'_{\mathsf{I}}$): Let $\dfrac{N : \langle \Gamma \vdash_3 T\rangle \quad x^L \notin \mathsf{dom}(\Gamma)}{\lambda x^L.N : \langle \Gamma \vdash_3 \omega^L \to T\rangle}$ and $M \twoheadrightarrow_\beta N$.

  Then this case is similar to the above case.

- Case ($\to_{\mathsf{E}}$): Let $\dfrac{N_1 : \langle \Gamma_1 \vdash_3 U \to T\rangle \quad N_2 : \langle \Gamma_2 \vdash_3 U\rangle \quad \Gamma_1 \diamond \Gamma_2}{N_1N_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T\rangle}$ and $M \twoheadrightarrow_\beta N_1N_2$.

  We have three cases:

  - $M = M_1N_2$ where $M_1 \twoheadrightarrow_\beta N_1$ and $M_1 \diamond N_2$ using Lemma B.1.1. By IH, $M_1 : \langle \Gamma_1\uparrow^{M_1} \vdash_3 U \to T\rangle$. It is easy to show that $(\Gamma_1 \sqcap \Gamma_2)\uparrow^{M_1N_2} = \Gamma_1\uparrow^{M_1} \sqcap \Gamma_2$. Since $M_1 \diamond N_2$, by Lemma 7.3.7.3, $\Gamma_1\uparrow^{M_1} \diamond \Gamma_2$. Finally, use rule ($\to_{\mathsf{E}}$).
  - $M = N_1M_2$ where $M_2 \twoheadrightarrow_\beta N_2$. Similar to the above case.
  - If $M = (\lambda x^L.M_1)M_2$ and $N_1N_2 = M_1[x^L := M_2]$ then, because $M \in \mathcal{M}_3$ then $L \succeq \mathsf{deg}(M_1)$, and by Lemma B.1.22, $(\lambda x^L.M_1)M_2 : \langle (\Gamma_1 \sqcap \Gamma_2)\uparrow^{(\lambda x^L.M_1)M_2} \vdash_3 T\rangle$.

- Case ($\sqcap_{\mathsf{I}}$): Let $\dfrac{N : \langle \Gamma \vdash_3 U_1\rangle \quad N : \langle \Gamma \vdash_3 U_2\rangle}{N : \langle \Gamma \vdash_3 U_1 \sqcap U_2\rangle}$ and $M \twoheadrightarrow_\beta N$.

  Then use IH.

- Case (exp): Let $\dfrac{N : \langle \Gamma \vdash_3 U\rangle}{N^{+j} : \langle \mathsf{e}_j\Gamma \vdash_3 \mathsf{e}_jU\rangle}$.

  By Lemma B.1.5.8 then there is $P \in \mathcal{M}_3$ such that $M = P^{+j}$ and $P \twoheadrightarrow_\beta N$. By IH, $P : \langle \Gamma\uparrow^P \vdash_3 U\rangle$ and by rule (exp), $M : \langle (\mathsf{e}_j\Gamma)\uparrow^M \vdash_3 \mathsf{e}_jU\rangle$ (it is easy to prove that $\mathsf{e}_j(\Gamma\uparrow^P) = (\mathsf{e}_j\Gamma)\uparrow^M$).

- Case ($\sqsubseteq$): Let $\dfrac{N : \langle \Gamma \vdash_3 U\rangle \quad \Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'}{N : \langle \Gamma' \vdash_3 U'\rangle}$ and $M \twoheadrightarrow_\beta N$.

By Lemma 7.3.4.3, $\Gamma' \sqsubseteq \Gamma$ and $U \sqsubseteq U'$. It is easy to show that $\Gamma'\!\uparrow^M \sqsubseteq \Gamma\!\uparrow^M$ and hence by Lemma 7.3.4.3, $\Gamma\!\uparrow^M \vdash_3 U \sqsubseteq \Gamma'\!\uparrow^M \vdash_3 U'$. By IH, $M : \langle \Gamma\!\uparrow^M \vdash_3 U\rangle$. Hence, by rule $(\sqsubseteq)$, we have $M : \langle \Gamma'\!\uparrow^M \vdash_3 U'\rangle$. □

Proof of Theorem 7.4.14. By induction on the length of the derivation $M \rightarrow\!\!\!\!\triangleright^*_\beta N$ using Theorem B.1.23 and the fact that if $\mathsf{fv}(P) \subseteq \mathsf{fv}(Q)$ then $(\Gamma\!\uparrow^P)\!\uparrow^Q = \Gamma\!\uparrow^Q$. □

B.2 Realisability semantics and their completeness (Ch. 8)

B.2.1 Realisability (Sec. 8.1)

Proof of Lemma 8.1.2. 1. easy.

2. If $M \rightarrow\!\!\!\!\triangleright^*_r N^+$ where $N \in \overline{M}$, then, by Lemma 7.1.11.1, Lemma B.1.3.1 and Lemma B.1.4.3, $M = P^+$ and $P \rightarrow\!\!\!\!\triangleright_\beta N$. Because $\overline{M} \in \mathsf{SAT}^r$, $P \in \overline{M}$ and so $P^+ = M \in \overline{M}^+$.

3. If $M \rightarrow\!\!\!\!\triangleright^*_r N^{+i}$ where $N \in \overline{M}$, then by Lemma B.1.5.8, $M = P^{+i}$ such that $P \in \mathcal{M}_3$ and $P \rightarrow\!\!\!\!\triangleright_r N$. Because $\overline{M} \in \mathsf{SAT}^r$, $P \in \overline{M}$ and so $P^{+i} = M \in \overline{M}^{+i}$.

4. Let $i \in \{1,2,3\}$, $M \in \overline{M}_1 \rightsquigarrow \overline{M}_2$ and $N \rightarrow\!\!\!\!\triangleright^*_r M$. If $P \in \overline{M}_1$ such that $P \diamond N$ then by Lemma B.1.2.1, $P \diamond M$. So, by definition, $MP \in \overline{M}_2$. Because $\overline{M}_2 \subseteq \mathcal{M}_i$ then $MP \in \mathcal{M}_i$. In case $i = 3$, because $MP \in \mathcal{M}_3$, $\mathsf{deg}(M) \preceq \mathsf{deg}(P)$ and by Lemma 7.1.11, $\mathsf{deg}(M) = \mathsf{deg}(N)$. So $NP \in \mathcal{M}_i$ and $NP \rightarrow\!\!\!\!\triangleright^*_r MP$. Because $MP \in \overline{M}_2$ and $\overline{M}_2 \in \mathsf{SAT}^r$ then $NP \in \overline{M}_2$. Hence, $N \in \overline{M}_1 \rightsquigarrow \overline{M}_2$.

5. Let $M \in (\overline{M}_1 \rightsquigarrow \overline{M}_2)^+$ then $M = N^+$ and $N \in \overline{M}_1 \rightsquigarrow \overline{M}_2$. If $P \in \overline{M}_1{}^+$ such that $M \diamond P$ then $P = Q^+$, $Q \in \overline{M}_1$ and $MP = N^+Q^+ = (NQ)^+$. By Lemma B.1.3.1(c)i, $N \diamond Q$ and hence $NQ \in \overline{M}_2$ and $MP \in \overline{M}_2{}^+$. Thus $M \in \overline{M}_1{}^+ \rightsquigarrow \overline{M}_2{}^+$.

6. Let $M \in (\overline{M}_1 \rightsquigarrow \overline{M}_2)^{+i}$ then $M = N^{+i}$ and $N \in \overline{M}_1 \rightsquigarrow \overline{M}_2$. Let $P \in \overline{M}_1^{+i}$ such that $M \diamond P$. Then $P = Q^{+i}$ such that $Q \in \overline{M}_1$. Because $M \diamond P$ then by Lemma B.1.5.2, $N \diamond Q$. So $NQ \in \overline{M}_2$. Because $\overline{M}_2 \subseteq \mathcal{M}_3$ then $NQ \in \mathcal{M}_3$. Because $(NQ)^{+i} = N^{+i}Q^{+i} = MP$ then $MP \in \overline{M}_2^{+i}$. Hence, $M \in \overline{M}_1^{+i} \rightsquigarrow \overline{M}_2^{+i}$.

7. let $M \in \overline{M}^+ \rightsquigarrow \overline{M}_2{}^+$. Because $\overline{M}_1{}^+ \wr \overline{M}_2{}^+$ then there is $N \in \overline{M}_1{}^+$ such that $M \diamond N$. We have $MN \in \overline{M}_2{}^+$ then $MN = P^+$ where $P \in \overline{M}_2$. Hence, $M = M_1{}^+$. Let $N_1 \in \overline{M}_1$ such that $M_1 \diamond N_1$. We have $N_1{}^+ \in \overline{M}_1{}^+$. By Lemma B.1.3.1(c)i, $M \diamond N_1{}^+$ and we have $(M_1N_1)^+ = M_1{}^+N_1{}^+ \in \overline{M}_2{}^+$. Hence $M_1N_1 \in \overline{M}_2$. Thus $M_1 \in \overline{M}_1 \rightsquigarrow \overline{M}_2$ and $M = M_1{}^+ \in (\overline{M}_1 \rightsquigarrow \overline{M}_2)^+$.

8. Let $M \in \overline{M}_1^{+i} \leadsto \overline{M}_2^{+i}$ such that $\overline{M}_1^{+i} \wr \overline{M}_2^{+i}$. By hypothesis, there exists $P \in \overline{M}_1^{+i}$ such that $M \diamond P$. Then $MP \in \overline{M}_2^{+i}$. Hence $MP = Q^{+i}$ such that $Q \in \overline{M}_2$. Because $\overline{M}_2 \subseteq \mathcal{M}_3$ then $Q \in \mathcal{M}_3$ and by Lemma B.1.5.1, $MP \in \mathcal{M}_3$. Hence by definition $M \in \mathcal{M}_3$ and by Lemma B.1.5.1, $\mathsf{deg}(M) = \mathsf{deg}(Q^{+i}) = i :: \mathsf{deg}(Q)$. So by Lemma B.1.5.7, there exists $M_1 \in \mathcal{M}_3$ such that $M = M_1^{+i}$. Let $N_1 \in \overline{M}_1$ such that $M_1 \diamond N_1$. By definition $N_1^{+i} \in \overline{M}_1^{+i}$ and by Lemma B.1.5.2, $M \diamond N_1^{+i}$, i.e., $M_1^{+i} \diamond N_1^{+i}$. So, $MN_1^{+i} \in \overline{M}_2^{+i}$. Hence, $M_1N_1 \in \overline{M}_2$. Thus, $M_1 \in \overline{M}_1 \leadsto \overline{M}_2$ and $M = M_1^{+i} \in (\overline{M}_1 \leadsto \overline{M}_2)^{+i}$.

9. If $M \mathrel{-\!\!\triangleright}^*_\beta N$ and $N \in \mathbb{M} \cap \mathcal{M}_2^n$ then by Lemma 7.1.11.2, $M \in \mathbb{M} \cap \mathcal{M}_2^n$. □

Proof of Lemma 8.1.4.

1. 1a. By induction on U using Lemma 8.1.2.

   1b. We prove $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}_x^L \subseteq \mathcal{I}(U) \subseteq \mathcal{M}_3^L$ by induction on U. Case $U = a$: by definition. Case $U = \omega^L$: We have $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}_x^L \subseteq \mathcal{M}_3^L \subseteq \mathcal{M}_3^L$. Case $U = U_1 \sqcap U_2$ (resp. $U = \mathsf{e}_i V$): use IH since $\mathsf{deg}(U_1) = \mathsf{deg}(U_2)$ (resp. $\mathsf{deg}(U) = i :: \mathsf{deg}(V)$, $\forall x \in \mathsf{Var}_1.\ (\mathsf{VAR}_x^K)^{+i} = \mathsf{VAR}_x^{i::K}$ and $(\mathcal{M}_3^K)^{+i} = \mathcal{M}_3^{i::K}$). Case $U = V \to T$: by definition, $K = \mathsf{deg}(V) \succeq \mathsf{deg}(T) = \oslash$.

   - Let $x \in \mathsf{Var}_1$, $N_1, \ldots, N_k \in \mathcal{M}_3$ such that $(\forall i \in \{1, \ldots, k\}.\ \mathsf{deg}(N_i) \succeq \oslash)$, and $\diamond\{x^\oslash, N_1, \ldots, N_k\}$. Let $N \in \mathcal{I}(V)$ such that $(x^\oslash N_1 \ldots N_k) \diamond N$. By IH, $N \in \mathcal{M}_3^K$ and $\mathsf{deg}(N) = K \succeq \oslash$. Again, by IH, $x^\oslash N_1 \ldots N_k N \in \mathcal{I}(T)$. Thus $x^\oslash N_1 \ldots N_k \in \mathcal{I}(V \to T)$.
   - Let $M \in \mathcal{I}(V \to T)$. Let $x \in \mathsf{Var}_1$ such that $\forall L.\ x^L \notin \mathsf{fv}(M)$. By IH, $x^K \in \mathcal{I}(V)$ then $Mx^K \in \mathcal{I}(T)$ and, by IH, $\mathsf{deg}(Mx^K) = \oslash$ (using Lemma B.1.12.1). Thus $\mathsf{deg}(M) = \oslash$.

   1c By definition, $x^n \in \mathsf{VAR}_x^n$. We prove $\mathsf{VAR}_x^n \subseteq \mathcal{I}(U) \subseteq \mathbb{M}^n$ by induction on $U \in \mathsf{GITy}$. Case $U = a$: by definition. Case $U = U \sqcap V$ (resp. $U = eU'$): use IH since by Lemma 7.2.3, $U, V \in \mathsf{GITy}$ and $\mathsf{deg}(U) = \mathsf{deg}(V)$ (resp. $U' \in \mathsf{GITy}$, $\mathsf{deg}(U) = \mathsf{deg}(U') + 1$, $(\mathsf{VAR}_x^n)^+ = \mathsf{VAR}_x^{n+1}$ and $(\mathcal{M}_2^n)^+ = \mathcal{M}_2^{n+1}$). Case $U = U \to T$: Lemma 7.2.3, $U, T \in \mathsf{GITy}$ and $m = \mathsf{deg}(U) \geq \mathsf{deg}(T) = n$.

   - Let $x^n N_1 \ldots N_k \in \mathcal{M}_2$ and $N \in \mathcal{I}(U)$ such that $(x^n N_1 \ldots N_k) \diamond N$. By IH, $\mathsf{deg}(N) = m \geq n$ and $N \in \mathbb{M}^m$. Therefore $N \in \mathcal{M}_2$. We have $x^n N_1 \ldots N_k N \in \mathcal{M}_2$. Hence, $x^n N_1 \ldots N_k N \in \mathsf{VAR}_x^n$. By IH, $x^n N_1 \ldots N_k N \in \mathcal{I}(T)$. Thus $x^n N_1 \ldots N_k \in \mathcal{I}(U \to T)$.
   - Let $M \in \mathcal{I}(U \to T)$. Let $x \in \mathsf{Var}_1$ such that $\forall p.\ x^p \notin \mathsf{fv}(M)$. Hence, $M \diamond x^m$. By IH, $x^m \in \mathcal{I}(U)$. Then $Mx^m \in \mathcal{I}(T)$, and so by IH $Mx^m \in \mathbb{M}^n$. By Lemma 7.1.6, $M \in \mathbb{M}$ and $\mathsf{deg}(M) \leq m$. Since $\mathsf{deg}(Mx^m) = \mathsf{min}(\mathsf{deg}(M), m) = n$, $\mathsf{deg}(M) = n$ and so $M \in \mathbb{M}^n$.

2. By induction of the derivation $U \sqsubseteq V$. □

Proof of Lemma 8.1.6.

- Case $\vdash_1$ / $\vdash_2$: Let $i \in \{1,2\}$. We prove the result by induction on the derivation of $M : \langle (x_i^{n_i} : U_i)_n \vdash_i U\rangle$ and then by case on the last rule of the derivation. First note, by Theorem 7.3.5 and Lemma 8.1.4.1c, $M \in \mathcal{M}_2$, $\forall i \in \{1,\dots,n\}.\ U_i \in \mathsf{GITy} \wedge \mathsf{deg}(U_i) = n_i \wedge N_i \in \mathbb{M}^{n_i}$, and $\forall V \in \mathsf{GITy} \cap \mathsf{ITy}_1.\ \mathcal{I}(V) \neq \varnothing$. By Lemma B.1.1.5a, $M[(x_i^{n_i} := N_i)_n] \in \mathcal{M}_2$.
  - Case (ax) of $\vdash_1$: Let $\dfrac{T \in \mathsf{GITy} \quad \mathsf{deg}(T) = n}{x^n : \langle (x^n : T) \vdash_1 T\rangle}$ and $N_1 \in \mathcal{I}(T)$.

    Then $x^n[x^n := N_1] = N_1 \in \mathcal{I}(T)$.
  - Case (ax) of $\vdash_2$: Let $\dfrac{T \in \mathsf{GITy}}{x^0 : \langle (x^0 : T) \vdash_2 T\rangle}$ and $N_1 \in \mathcal{I}(T)$.

    Then $x^0[x^0 := N_1] = N_1 \in \mathcal{I}(T)$.
  - Case $(\to_{\mathsf{I}})$: Let $\dfrac{M : \langle (x_i^{n_i} : U_i)_n, (x^m : U) \vdash_i T\rangle}{\lambda x^m.M : \langle (x_i^{n_i} : U_i)_n \vdash_i U \to T\rangle}$.

    We take $\forall i \in \{1,\dots,n\}.\ N_i \in \mathcal{I}(U_i) \wedge \forall m'.\ x^{m'} \notin \mathsf{fv}(N_i)$. By Theorem 7.3.5, $U, T \in \mathsf{GITy}$ and $\mathsf{deg}(U) = m$. Let $N \in \mathcal{I}(U)$ such that $(\lambda x^m.M)[(x_i^{n_i} := N_i)_n] \diamond N$. By Lemma 8.1.4, $N \in \mathbb{M}^m$. Since $(\lambda x^m.M[(x_i^{n_i} := N_i)_n]) \diamond N$, by Lemma B.1.1, $M[(x_i^{n_i} := N_i)_n] \diamond N$ and $M[(x_i^{n_i} := N_i)_n][x^m := N] = M[(x_i^{n_i} := N_i)_n, x^m := N] \in \mathcal{M}_2$. Hence, by IH, $M[(x_i^{n_i} := N_i)_n, x^m := N] \in \mathcal{I}(T)$ and $(\lambda x^m.M[(x_1^{n_1} := N_1)_n])N \rhd_\beta M[(x_i^{n_i} := N_i)_n, x^m := N] \in \mathcal{I}(T)$. Since, by Lemma 8.1.4, $\mathcal{I}(T)$ is β-saturated then $(\lambda x^m.M[(x_1^{n_1} := N_1)_n])N \in \mathcal{I}(T)$ and hence $\lambda x^m.M[(x_i^{n_i} := N_i)_n] \in \mathcal{I}(U) \rightsquigarrow \mathcal{I}(T) = \mathcal{I}(U \to T)$.
  - Case $(\to_{\mathsf{E}})$: Let $\dfrac{M_1 : \langle \Gamma_1 \vdash_i U \to T\rangle \quad M_2 : \langle \Gamma_2 \vdash_i U\rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1 M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_i T\rangle}$.

    Let $\Gamma_1 = (x_i^{n_i} : U_i)_n, (y_j^{m_j} : V_j)_m$, $\Gamma_2 = (x_i^{n_i} : U_i')_n, (z_k^{p_k} : W_k)_p$ and $\Gamma_1 \sqcap \Gamma_2 = (x_i^{n_i} : U_i \sqcap U_i')_n, (y_j^{m_j} : V_j)_m, (z_k^{p_k} : W_k)_p$. Let $\forall i \in \{1,\dots,n\}.\ P_i \in \mathcal{I}(U_i \sqcap U_i')$, $\forall j \in \{1,\dots,m\}.\ Q_j \in \mathcal{I}(V_j)$ and $\forall k \in \{1,\dots,r\}.\ R_k \in \mathcal{I}(W_k)$ where $(M_1 M_2)[(x_i^{n_i} := P_i)_n, (y_j^{m_j} := Q_j)_m, (z_k^{p_k} := R_k)_p] \in \mathcal{M}_2$. Let $N_1 = M_1[(x_i^{n_i} := P_i)_n, (y_j^{m_j} := Q_j)_m]$ and $N_2 = M_2[(x_i^{n_i} := P_i)_n, (z_k^{p_k} := R_k)_p]$. By Theorem 7.3.5.2a, $\mathsf{fv}(M_1) = \mathsf{dom}(\Gamma_1)$ and $\mathsf{fv}(M_2) = \mathsf{dom}(\Gamma_2)$. Hence, $(M_1 M_2)[(x_i^{n_i} := P_i)_n, (y_j^{m_j} := Q_j)_m, (z_k^{p_k} := R_k)_p] = N_1 N_2$. By Lemma B.1.1, $N_1 \in \mathcal{M}_2$, $N_2 \in \mathcal{M}_2$, and $N_1 \diamond N_2$. By IH, $N_1 \in \mathcal{I}(U) \rightsquigarrow \mathcal{I}(T)$ and $N_2 \in \mathcal{I}(U)$. Hence, $N_1 N_2 = (M_1 M_2)[(x_i^{n_i} := P_i)_n, (y_j^{m_j} := Q_j)_m, (z_k^{p_k} := R_k)_p] \in \mathcal{I}(T)$.
  - Case $(\sqcap_{\mathsf{I}})$: Let $\dfrac{M : \langle (x_i^{n_i} : U_i)_n \vdash_i U\rangle \quad M : \langle (x_i^{n_i} : V_i)_n \vdash_i V\rangle}{M : \langle (x_i^{n_i} : U_i \sqcap V_i)_n \vdash_i U \sqcap V\rangle}$ (note the use Theorem 7.3.5.2a).

We have, $\forall i \in \{1, \ldots, n\}.\ N_i \in \mathcal{I}(U_i \sqcap V_i) = \mathcal{I}(U_i) \cap \mathcal{I}(V_i)$ By IH, $M[(x_i^{n_i} := N_i)_n] \in \mathcal{I}(U)$ and $M[(x_i^{n_i} := N_i)_n] \in \mathcal{I}(V)$. Hence, $M[(x_i^{n_i} := N_i)_n] \in \mathcal{I}(U \sqcap V)$.

- Case (exp): Let $\dfrac{M : \langle (x_i^{n_i} : T_i)_n \vdash_i U\rangle}{M^+ : \langle (x_i^{n_i+1} : eT_i)_n \vdash_i eU\rangle}$.

  Let $\forall i \in \{1, \ldots, n\}.\ N_i \in \mathcal{I}(eT_i) = \mathcal{I}(T_i)^+$ where $M^+[(x_i^{n_i+1} := N_i)_n] \in \mathcal{M}_2$. Then $\forall i \in \{1, \ldots, n\}.\ N_i = P_i{}^+ \wedge P_i \in \mathcal{I}(T_i)$. By Lemma B.1.3.1(c)i, $\diamond\{M, P_1, \ldots, P_n\}$. By IH, $M[(x_i^{n_i} := P_i)_n] \in \mathcal{I}(U)$. Hence, by lemma B.1.3.2, $M^+[(x_i^{n_i+1} := P_i{}^+)_n] = (M[(x_i^{n_i} := P_i)_n])^+ \in \mathcal{I}(U)^+ = \mathcal{I}(eU)$.

- Case $(\sqsubseteq)$: Let $\dfrac{M : \Gamma \vdash_2 U \quad \Gamma \vdash_2 U \sqsubseteq \Gamma' \vdash_2 U'}{M : \Gamma' \vdash_2 U'}$.

  By Lemma 7.3.4, we have $\Gamma = (x_i^{n_i} : U_i)_n$ and $\Gamma' = (x_i^{n_i} : U'_i)_n$, where $\forall i \in \{1, \ldots, n\}.\ U'_i \sqsubseteq U_i$, and $U \sqsubseteq U'$. By Lemma 8.1.4.2, $\forall i \in \{1, \ldots, n\}.\ N_i \in \mathcal{I}(U_i)$. By IH, $M[(x_i^{n_i} := N_i)_n] \in \mathcal{I}(U')$. By Lemma 8.1.4.2, $M[(x_i^{n_i} := N_i)_n] \in \mathcal{I}(U)$.

- Case $\vdash_3$: We prove the result by induction on the derivation $M : \langle (x_j^{L_j} : U_j)_n \vdash_3 U\rangle$ and then by case on the last rule of the derivation. First note, by Theorem 7.3.5 and Lemma 8.1.4.1b, $M \in \mathcal{M}_3$, $\forall j \in \{1, \ldots, n\}.\ \mathsf{deg}(U_j) = L_j \wedge N_j \in \mathcal{M}_3^{L_j}$, and $\forall V \in \mathsf{ITy}_3.\ \mathcal{I}(V) \neq \varnothing$. By Lemma B.1.1.5a, $M[(x_j^{L_j} := N_j)_n] \in \mathcal{M}_3$.

  - Case (ax): Let $\overline{x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T\rangle}$.

    Let $N \in \mathcal{I}(T)$ then $x^{\oslash}[x^{\oslash} := N] = N \in \mathcal{I}(T)$.

  - Case (ω): Let $\overline{M : \langle \mathsf{env}^{\varnothing}_M \vdash_3 \omega^{\mathsf{deg}(M)}\rangle}$.

    Let $\mathsf{env}^{\varnothing}_M = (x_j^{L_j} : \omega^{L_j})_n$ so $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$. By Lemma B.1.1.5, $\mathsf{deg}(M[(x_j^{L_j} := N_j)_n]) = \mathsf{deg}(M)$ and $M[(x_j^{L_j} := N_j)_n] \in \mathcal{M}_3^{\mathsf{deg}(M)} = \mathcal{I}(\omega^{\mathsf{deg}(M)})$.

  - Case $(\to_{\mathsf{I}})$: Let $\dfrac{M : \langle (x_j^{L_j} : U_j)_n, (x^K : V) \vdash_3 T\rangle}{\lambda x^K.M : \langle (x_j^{L_j} : U_j)_n \vdash_3 V \to T\rangle}$ such that $\forall K'.\ \forall j \in \{1, \ldots, n\}.\ x^{K'} \notin \mathsf{fv}(N_j)$.

    We have, $(\lambda x^K.M)[(x_j^{L_j} := N_j)_n] = \lambda x^K.M[(x_j^{L_j} := N_j)_n]$. Let $N \in \mathcal{I}(V)$ such that $(\lambda x^K.M)[(x_j^{L_j} := N_j)_n] \diamond N$. By Theorem 7.3.5.2, $\mathsf{deg}(V) = K$. Because $N \in \mathcal{I}(V)$ and by Lemma 8.1.4.1, $\mathcal{I}(V) \subseteq \mathcal{M}_3^K$, we have $\mathsf{deg}(N) = K$. By Lemma B.1.1.2 and Lemma B.1.1.5, $M[(x_j^{L_j} := N_j)_n] \diamond N$ and $M[(x_j^{L_j} := N_j)_n][x^K := N] = M[(x_j^{L_j} := N_j)_n, x^K := N] \in \mathcal{M}_3$. Hence, $(\lambda x^K.M[(x_j^{L_j} := N_j)_n])N \in \mathcal{M}_3$ and $(\lambda x^K.M[(x_j^{L_j} := N_j)_n])N \rightarrowtriangle_r M[(x_j^{L_j} := N_j)_n, (x^K := N)]$. By IH, $M[(x_j^{L_j} := N_j)_n, (x^K := N)] \in \mathcal{I}(T)$. Because, by Lemma 8.1.4.1, $\mathcal{I}(T)$ is r-saturated then $(\lambda x^K.M[(x_j^{L_j} := N_j)_n])N \in \mathcal{I}(T)$ and finally $\lambda x^K.M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(V) \rightsquigarrow \mathcal{I}(T) = \mathcal{I}(V \to T)$.

- Case ($\to_{\mathsf{I}}'$): Let $\dfrac{M : \langle (x_j^{L_j} : U_j)_n \vdash_3 T\rangle \quad x^K \notin \mathsf{dom}((x_j^{L_j} : U_j)_n)}{\lambda x^K.M : \langle (x_j^{L_j} : U_j)_n \vdash_3 \omega^K \to T\rangle}$ such that $\forall K'.\ \forall j \in \{1,\ldots,n\}.\ x^{K'} \notin \mathsf{fv}(N_j)$.

  Let $N \in \mathcal{I}(\omega^K) = \mathcal{M}_3^L$ such that $(\lambda x^K.M)[(x_j^{L_j} := N_j)_n] \diamond N$. By Theorem 7.3.5.2a, $x^K \notin \mathsf{fv}(M)$. We have, $(\lambda x^K.M)[(x_j^{L_j} := N_j)_n] = \lambda x^K.M[(x_j^{L_j} := N_j)_n]$. Because $N \in \mathcal{I}(\omega^K) = \mathcal{M}_3^K$, by Lemma 8.1.4.1, $\mathsf{deg}(N) = K$. By Lemma B.1.1.2 and Lemma B.1.1.5, $M[(x_j^{L_j} := N_j)_n] \diamond N$ and $M[(x_j^{L_j} := N_j)_n][x^K := N] = M[(x_j^{L_j} := N_j)_n, x^K := N] = M[(x_j^{L_j} := N_j)_n] \in \mathcal{M}_3$. Hence, $(\lambda x^K.M[(x_j^{L_j} := N_j)_n])N \in \mathcal{M}_3$ and $(\lambda x^K.M[(x_j^{L_j} := N_j)_n])N \rightarrow\!\!\!\triangleright_r M[(x_j^{L_j} := N_j)_n, (x^K := N)]$. By IH, $M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(T)$. Because, by Lemma 8.1.4.1, $\mathcal{I}(T)$ is r-saturated then $(\lambda x^K.M[(x_j^{L_j} := N_j)_n])N \in \mathcal{I}(T)$ and so $\lambda x^K.M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(\omega^K) \rightsquigarrow \mathcal{I}(T) = \mathcal{I}(\omega^K \to T)$.

- Case ($\to_{\mathsf{E}}$): Let $\dfrac{M_1 : \langle \Gamma_1 \vdash_3 V \to T\rangle \quad M_2 : \langle \Gamma_2 \vdash_3 V\rangle \quad \Gamma_1 \diamond \Gamma_2}{M_1M_2 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T\rangle}$.

  Let $\Gamma_1 = (x_j^{L_j} : U_j)_n, (y_j^{K_j} : V_j)_m$, $\Gamma_2 = (x_j^{L_j} : U_j')_n, (z_j^{K_j'} : W_j)_p$ such that $\mathsf{dj}(\{y_1^{K_1},\ldots,y_m^{K_m}\}, \{z_1^{K_1'},\ldots,z_p^{K_p'}\})$ and $\Gamma_1 \sqcap \Gamma_2 = (x_j^{L_j} : U_j \sqcap U_j')_n, (y_j^{K_j} : V_j)_m, (z_j^{K_j'} : W_j)_p$. Let $\forall j \in \{1,\ldots,n\}.\ P_j \in \mathcal{I}(U_j \sqcap U_j')$, $\forall j \in \{1,\ldots,m\}.\ Q_j \in \mathcal{I}(V_j)$, and $\forall j \in \{1,\ldots,p\}.\ R_j \in \mathcal{I}(W_j)$. Therefore, $\forall j \in \{1,\ldots,n\}.\ P_j \in \mathcal{I}(U_j) \cap \mathcal{I}(U_j')$. By hypothesis, $(M_1M_2)[(x_j^{L_j} := P_j)_n, (y_j^{K_j} := Q_j)_m, (z_j^{K_j'} := R_j)_p] = N_1N_2 \in \mathcal{M}_3$ where using Theorem 7.3.5, $N_1 = M_1[(x_j^{L_j} := P_j)_n, (y_j^{K_j} := Q_j)_m] \in \mathcal{M}_3$ and $N_2 = M_2[(x_j^{L_j} := P_j)_n, (z_j^{K_j'} := R_j)_p] \in \mathcal{M}_3$ and $N_1 \diamond N_2$. By IH, $N_1 \in \mathcal{I}(V) \rightsquigarrow \mathcal{I}(T)$ and $N_2 \in \mathcal{I}(V)$. Hence, $N_1N_2 \in \mathcal{I}(T)$.

- Case ($\sqcap_{\mathsf{I}}$): Let $\dfrac{M : \langle (x_j^{L_j} : U_j)_n \vdash_3 V_1\rangle \quad M : \langle (x_j^{L_j} : U_j)_n \vdash_3 V_2\rangle}{M : \langle (x_j^{L_j} : U_j)_n \vdash_3 V_1 \sqcap V_2\rangle}$.

  By IH, $M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(V_1)$ and $M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(V_2)$. Hence, $M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(V_1 \sqcap V_2)$.

- Case (exp): Let $\dfrac{M : \langle (x_k^{L_k} : U_k)_n \vdash_3 U\rangle}{M^{+j} : \langle (x_k^{j::L_k} : \mathsf{e}_jU_k)_n \vdash_3 \mathsf{e}_jU\rangle}$.

  We take, $\forall k \in \{1,\ldots,n\}.\ N_k \in \mathcal{I}(\mathsf{e}_jU_k) = \mathcal{I}(U_k)^{+j}$. Then $\forall k \in \{1,\ldots,n\}.\ N_k = P_k^{+j} \wedge P_k \in \mathcal{I}(U_k)$. By Lemma 8.1.4.1b, $\forall k \in \{1,\ldots,n\}.\ P_k \in \mathcal{M}_3^{L_k}$. By Lemma B.1.5.3, $\diamond\{M\} \cup \{P_k \mid k \in \{1,\ldots,n\}\}$. By Lemma B.1.1.5, $M[(x_k^{L_k} := P_k)_n] \in \mathcal{M}_3$. By IH, $M[(x_k^{L_k} := P_k)_n] \in \mathcal{I}(T)$. Hence, by Lemma B.1.5.5, $M^{+j}[(x_k^{j::L_k} := N_k)_n] = (M[(x_k^{L_k} := P_k)_n])^{+j} \in \mathcal{I}(U)^{+j} = \mathcal{I}(\mathsf{e}_jU)$.

- Case ($\sqsubseteq$): Let $\dfrac{M : \Gamma \vdash_3 U \quad \Gamma \vdash_3 U \sqsubseteq \Gamma' \vdash_3 U'}{M : \Gamma' \vdash_3 U'}$.

  By Lemma 7.3.4, we have $\Gamma' = (x_j^{L_j} : U_j')_n$ and $\Gamma = (x_j^{L_j} : U_j)_n$, such that

$\forall j \in \{1,\dots,n\}$. $U'_j \sqsubseteq U_j$ and $U \sqsubseteq U'$. By Lemma 8.1.4.2, $N_j \in \mathcal{I}(U_j)$ then, by IH, $M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(U)$ and, by Lemma 8.1.4.2, $M[(x_j^{L_j} := N_j)_n] \in \mathcal{I}(U')$.

□

Next we give a lemma concerning reductions in $\lambda I^{\mathbb{N}}$ that will be used in the rest of the article.

Lemma B.2.1.

1. *If $M[y^{I_1} := x^{I_2}] \mathrel{-\!\!\triangleright}_\beta N$ then $M \mathrel{-\!\!\triangleright}_\beta N'$ where $N = N'[y^{I_1} := x^{I_2}]$.*
2. *If $M[y^{I_1} := x^{I_2}]$ has a β-normal form then M has a β-normal form.*
3. *Let $k \geq 1$. If $Mx_1^{I_1} \dots x_k^{I_k}$ is normalisable then M is normalisable.*
4. *Let $k \geq 1$, $i \in \{1,\dots,k\}$, $l \geq 0$, $x_i^{I_i} N_1 \dots N_l$ be in normal form and M be closed. If $Mx_1^{I_1} \dots x_k^{I_k} \mathrel{-\!\!\triangleright}_\beta^* x_i^{I_i} N_1 \dots N_l$ then for some $m \geq i$ and $n \leq l$, $M \mathrel{-\!\!\triangleright}_\beta^* \lambda x_1^{I_1}.\dots.\lambda x_m^{I_m}.x_i^{I_i} M_1 \dots M_n$ where $n + k = m + l$, $M_j \simeq_\beta N_j$ for every $j \in \{1,\dots,n\}$ and $N_{n+j} \simeq_\beta x_{m+j}^{I_{m+j}}$ for every $j \in \{1,\dots,k-m\}$.* □

Proof of Lemma B.2.1.

1. By induction on $M[y^{I_1} := x^{I_2}] \mathrel{-\!\!\triangleright}_\beta N$.
2. $M[y^{I_1} := x^{I_2}] \mathrel{-\!\!\triangleright}_\beta^* P$ where P is in β-normal form. The proof is by induction on $M[y^{I_1} := x^{I_2}] \mathrel{-\!\!\triangleright}_\beta^* P$ using 1.
3. By induction on $k \geq 1$. We only prove the basic case. The proof is by cases.
   - If $Mx_1^{I_1} \mathrel{-\!\!\triangleright}_\beta^* M'x_1^{I_1}$ where $M'x_1^{I_1}$ is in β-normal form and $M \mathrel{-\!\!\triangleright}_\beta^* M'$ then M' is in β-normal form and M is β-normalising.
   - If $Mx_1^{I_1} \mathrel{-\!\!\triangleright}_\beta^* (\lambda y^{I_1}.N)x_1^{I_1} \mathrel{-\!\!\triangleright}_\beta N[y^{I_1} := x_1^{I_1}] \mathrel{-\!\!\triangleright}_\beta^* P$ where P is in β-normal form and $M \mathrel{-\!\!\triangleright}_\beta^* \lambda y^{I_1}.N$ then by 2., N has a β-normal form and so, $\lambda y^{I_1}.N$ has a β-normal form. Hence, M has a β-normal form.
4. By 3., M is normalisable, and, since M is closed, its normal form is as follows: $\lambda x_1^{I_1}.\dots.\lambda x_m^{I_m}.z^I M_1 \dots M_n$ for $n, m \geq 0$ and where each M_i is a normal form. Using Theorem 7.1.13, $x_i^{I_i} N_1 \dots N_l \simeq_\beta (\lambda x_1^{I_1}.\dots.\lambda x_m^{I_m}.z^I M_1 \dots M_n)x_1^{I_1} \dots x_k^{I_k}$. Hence $m \leq k$ and $x_i^{I_i} N_1 \dots N_l \simeq_\beta z^I M_1 \dots M_n x_{m+1}^{I_{m+1}} \dots x_k^{I_k}$. Finally, $z^I = x_i^{I_i}$, $n \leq l$, $i \leq m$, $l = n + k - m$, $\forall j \in \{1,\dots,n\}$. $M_j \simeq_\beta N_j$, and $\forall j \in \{1,\dots,k-m\}$. $N_{n+j} \simeq_\beta x_{m+j}^{I_{m+j}}$. □

Proof of Example 8.1.9.

1. Let $y \in \mathsf{Var}_2$ and take $\overline{M} = \{M \in \mathbb{M}^0 \mid M \rightarrowtriangle^*_\beta y^0 \vee (k \geq 0 \wedge x \in \mathsf{Var}_1 \wedge M \rightarrowtriangle^*_\beta x^0 N_1 \ldots N_k)\}$. The set $\overline{M}$ is β-saturated and $\forall x \in \mathsf{Var}_1$. $\mathsf{VAR}^0_x \subseteq \overline{M} \subseteq \mathbb{M}^0$. Let $\mathcal{I}$ be a β_1-interpretation such that $\mathcal{I}(\mathsf{a}) = \mathcal{I}(\mathsf{b}) = \overline{M}$. If $M \in [(\mathsf{a} \sqcap \mathsf{b}) \to \mathsf{a}]_{\beta_1}$ then M is closed and $M \in \overline{M} \rightsquigarrow \overline{M}$. Since $My^0 \in \overline{M}$ (because $y^0 \in \overline{M}$ and $M \diamond y^0$), M is closed, and $x^0 \neq y^0$, by Lemma 7.1.11.3, $My^0 \rightarrowtriangle^*_\beta y^0$. Hence, by Lemma B.2.1.4, $M \rightarrowtriangle^*_\beta \lambda y^0.y^0$. By Lemma 7.1.11.3, $\mathsf{deg}(M) = \mathsf{deg}(\lambda y^0.y^0) = 0$ and $M \in \mathbb{M}^0$.

   Conversely, let $M \in \mathbb{M}^0$ and $M \rightarrowtriangle^*_\beta \lambda y^0.y^0$. By Lemma 7.1.11.3, M is closed. Let $\mathcal{I}$ be a β_1-interpretation and $N \in \mathcal{I}(\mathsf{a} \sqcap \mathsf{b})$. Because M is closed, we have $M \diamond N$. Since $\mathcal{I}(\mathsf{a})$ is saturated, $N \in \mathcal{I}(\mathsf{a})$ and $MN \rightarrowtriangle^*_\beta N$, then $MN \in \mathcal{I}(\mathsf{a})$ and hence $M \in \mathcal{I}(\mathsf{a} \sqcap \mathsf{b}) \rightsquigarrow \mathcal{I}(\mathsf{a})$. Finally, $M \in [(\mathsf{a} \sqcap \mathsf{b}) \to \mathsf{a}]_{\beta_1}$.

2 If $\lambda y^0.y^0 : \langle () \vdash_1 (\mathsf{a} \sqcap \mathsf{b}) \to \mathsf{a} \rangle$, then by Lemma 7.4.1.2, $y^0 : \langle (y^0 : \mathsf{a} \sqcap \mathsf{b}) \vdash_1 \mathsf{a} \rangle$ and by Lemma 7.4.1.1, $\mathsf{a} = \mathsf{a} \sqcap \mathsf{b}$. Absurd because $\mathsf{a} \neq \mathsf{b}$.

3. Easy using rule $(\sqsubseteq)$.

4. Let $y \in \mathsf{Var}_2$ and $\overline{M} = \{M \in \mathcal{M}^\oslash_3 \mid (k \geq 0 \wedge x \in \mathsf{Var}_1 \wedge M \rightarrowtriangle^*_\beta x^\oslash N_1 \ldots N_k) \vee M \rightarrowtriangle^*_\beta y^\oslash\}$. The set $\overline{M}$ is β-saturated and $\forall x \in \mathsf{Var}_1$. $\mathsf{VAR}^\oslash_x \subseteq \overline{M} \subseteq \mathcal{M}^\oslash_3$. Take a β_3-interpretation $\mathcal{I}$ such that $\mathcal{I}(\mathsf{a}) = \overline{M}$. If $M \in [\mathsf{id}_0]_{\beta_3}$ then M is closed and $M \in \overline{M} \rightsquigarrow \overline{M}$. Because $y^\oslash \in \overline{M}$ and $M \diamond y^\oslash$ then $My^\oslash \in \overline{M}$ and $((My^\oslash \rightarrowtriangle^*_\beta x^\oslash N_1 \ldots N_k$ where $k \geq 0$ and $x \in \mathsf{Var}_1)$ or $My^\oslash \rightarrowtriangle^*_\beta y^\oslash)$. Because M is closed and $x^\oslash \neq y^\oslash$, by Lemma 7.1.11.2, $My^\oslash \rightarrowtriangle^*_\beta y^\oslash$. Hence, by Lemma B.2.1.4, $M \rightarrowtriangle^*_\beta \lambda y^\oslash.y^\oslash$ and, by Lemma 7.1.11.2, $M \in \mathcal{M}^\oslash_3$.

   Conversely, let $M \in \mathcal{M}^\oslash_3$ such that M is closed and $M \rightarrowtriangle^*_\beta \lambda y^\oslash.y^\oslash$. Let $\mathcal{I}$ be a β_3-interpretation and $N \in \mathcal{I}(\mathsf{a})$ such that $M \diamond N$. By Lemma 8.1.4.1b, $N \in \mathcal{M}^\oslash_3$, so $MN \in \mathcal{M}^\oslash_3$. Since $\mathcal{I}(\mathsf{a})$ is β-saturated and $MN \rightarrowtriangle^*_\beta N$, $MN \in \mathcal{I}(\mathsf{a})$. Therefore $M \in \mathcal{I}(\mathsf{a}) \rightsquigarrow \mathcal{I}(\mathsf{a})$ and $M \in [\mathsf{id}_0]_{\beta_3}$.

5. By Lemma 8.1.8 and 4., $[\mathsf{id}_1]_{\beta_3} = [\mathsf{e}_1(\mathsf{a} \to \mathsf{a})]_{\beta_3} = [\mathsf{a} \to \mathsf{a}]^{+1}_{\beta_3} = [\mathsf{id}_0]^{+1}_{\beta_3} = \{M \in \mathcal{M}^{(1)}_3 \mid M \rightarrowtriangle^*_\beta \lambda y^{(1)}.y^{(1)}\}$.

6. Let $y \in \mathsf{Var}_2$, $\overline{M}_1 = \{M \in \mathcal{M}^\oslash_3 \mid M \rightarrowtriangle^*_\beta y^\oslash \vee (k \geq 0 \wedge x \in \mathsf{Var}_1 \wedge M \rightarrowtriangle^*_\beta x^\oslash N_1 \ldots N_k)\}$ and $\overline{M}_2 = \{M \in \mathcal{M}^\oslash_3 \mid M \rightarrowtriangle^*_\beta y^\oslash y^\oslash \vee (k \geq 0 \wedge x \in \mathsf{Var}_1 \wedge (M \rightarrowtriangle^*_\beta x^\oslash N_1 \ldots N_k \vee M \rightarrowtriangle^*_\beta y^\oslash(x^\oslash N_1 \ldots N_k)))\}$. The sets $\overline{M}_1$, $\overline{M}_2$ are β-saturated and $\forall x \in \mathsf{Var}_1$. $\forall i \in \{1,2\}$. $\mathsf{VAR}^\oslash_x \subseteq \overline{M}_i \subseteq \mathcal{M}^\oslash_3$. Let $\mathcal{I}$ be a β_3-interpretation such that $\mathcal{I}(\mathsf{a}) = \overline{M}_1$ and $\mathcal{I}(\mathsf{b}) = \overline{M}_2$. If $M \in [\mathsf{d}]_{\beta_3}$ then M is closed (hence $M \diamond y^\oslash$) and $M \in (\overline{M}_1 \cap (\overline{M}_1 \rightsquigarrow \overline{M}_2)) \rightsquigarrow \overline{M}_2$. Because $y^\oslash \in \overline{M}_1$ and $y^\oslash \in \overline{M}_1 \rightsquigarrow \overline{M}_2$, $y^\oslash \in \overline{M}_1 \cap (\overline{M}_1 \rightsquigarrow \overline{M}_2)$ and $My^\oslash \in \overline{M}_2$. Since $x^\oslash \neq y^\oslash$, by Lemma 7.1.11.2, $My^\oslash \rightarrowtriangle^*_\beta y^\oslash y^\oslash$. Hence, by Lemma B.2.1.4, $M \rightarrowtriangle^*_\beta \lambda y^\oslash.y^\oslash y^\oslash$ and, by Lemma 7.1.11.2, $\mathsf{deg}(M) = \oslash$ and $M \in \mathcal{M}^\oslash_3$.

Conversely, let $M \in \mathcal{M}_3^{\oslash}$ such that M is closed and $M \rightarrowtriangle^*_\beta \lambda y^{\oslash}.y^{\oslash}y^{\oslash}$. Let $\mathcal{I}$ be a β_3-interpretation and $N \in \mathcal{I}(\mathsf{a} \sqcap (\mathsf{a}{\to}\mathsf{b})) = \mathcal{I}(\mathsf{a}) \cap (\mathcal{I}(\mathsf{a}) \leadsto \mathcal{I}(\mathsf{b}))$ such that $M \diamond N$. By Lemma 8.1.4.1b and Lemma B.1.1.1, $N \in \mathcal{M}_3^{\oslash}$ and $N \diamond N$. So $NN, MN \in \mathcal{M}_3^{\oslash}$. Since $\mathcal{I}(\mathsf{b})$ is β-saturated, $NN \in \mathcal{I}(\mathsf{b})$ and $MN \rightarrowtriangle^*_\beta NN$, we have $MN \in \mathcal{I}(\mathsf{b})$ and hence $M \in \mathcal{I}(\mathsf{a} \sqcap (\mathsf{a}{\to}\mathsf{b})) \leadsto \mathcal{I}(\mathsf{b})$. Therefore, $M \in [\mathsf{d}]_{\beta_3}$.

7. Let $f, y \in \mathsf{Var}_2$ such that $f \neq y$ and take $\overline{M} = \{M \in \mathcal{M}_3^{\oslash} \mid k, n \geq 0 \wedge x \in \mathsf{Var}_1 \wedge (M \rightarrowtriangle^*_\beta (f^{\oslash})^n(x^{\oslash}N_1 \ldots N_k) \vee M \rightarrowtriangle^*_\beta (f^{\oslash})^n y^{\oslash})\}$. The set $\overline{M}$ is β-saturated and $\forall x \in \mathsf{Var}_1$. $\mathsf{VAR}_x^{\oslash} \subseteq \overline{M} \subseteq \mathcal{M}_3^{\oslash}$. Let $\mathcal{I}$ be a β_3-interpretation such that $\mathcal{I}(\mathsf{a}) = \overline{M}$. If $M \in [\mathsf{nat}_0]_{\beta_3}$ then M is closed and $M \in (\overline{M} \leadsto \overline{M}) \leadsto (\overline{M} \leadsto \overline{M})$. We have $f^{\oslash} \in \overline{M} \leadsto \overline{M}$, $y^{\oslash} \in \overline{M}$ and $\diamond\{M, f^{\oslash}, y^{\oslash}\}$ then $Mf^{\oslash}y^{\oslash} \in \overline{M}$ and $(Mf^{\oslash}y^{\oslash} \rightarrowtriangle^*_\beta (f^{\oslash})^n(x^{\oslash}N_1 \ldots N_k)$ or $Mf^{\oslash}y^{\oslash} \rightarrowtriangle^*_\beta (f^{\oslash})^n y^{\oslash})$ where $n, k \geq 0$ and $x \in \mathsf{Var}_1$. Since M is closed and $\mathsf{dj}(\{x^{\oslash}\}, \{y^{\oslash}, f^{\oslash}\})$, by Lemma 7.1.11.2, $Mf^{\oslash}y^{\oslash} \rightarrowtriangle^*_\beta (f^{\oslash})^n y^{\oslash}$ where $n \geq 1$. Hence, by Lemma B.2.1.4, $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.f^{\oslash}$ or $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.\lambda y^{\oslash}.(f^{\oslash})^n y^{\oslash}$ where $n \geq 1$. Moreover, by Lemma 7.1.11.2, $\mathsf{deg}(M) = \oslash$ and $M \in \mathcal{M}_3^{\oslash}$.

   Conversely, let $M \in \mathcal{M}_3^{\oslash}$ such that M is closed and $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.f^{\oslash}$ or $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.\lambda y^{\oslash}.(f^{\oslash})^n y^{\oslash}$ where $n \geq 1$. Let $\mathcal{I}$ be a β_3-interpretation, $N \in \mathcal{I}(\mathsf{a}{\to}\mathsf{a}) = \mathcal{I}(\mathsf{a}) \leadsto \mathcal{I}(\mathsf{a})$ and $N' \in \mathcal{I}(\mathsf{a})$ such that $\diamond\{M, N, N'\}$. By Lemma 8.1.4.1b, $N, N' \in \mathcal{M}_3^{\oslash}$, so $MNN', (N)^m N' \in \mathcal{M}_3^{\oslash}$, where $m \geq 0$. It is easy to show, by induction on $m \geq 0$, that $(N)^m N' \in \mathcal{I}(\mathsf{a})$. Since $MNN' \rightarrowtriangle^*_\beta (N)^m N'$ where $m \geq 0$ and $(N)^m N' \in \mathcal{I}(\mathsf{a})$ which is β-saturated, then $MNN' \in \mathcal{I}(\mathsf{a})$. Hence, $M \in (\mathcal{I}(\mathsf{a}) \leadsto \mathcal{I}(\mathsf{a})) \leadsto (\mathcal{I}(\mathsf{a}) \leadsto \mathcal{I}(\mathsf{a}))$ and $M \in [\mathsf{nat}_0]_{\beta_3}$.

8. By Lemma 8.1.8, $[\mathsf{nat}_1]_{\beta_3} = [\mathsf{e}_1\mathsf{nat}_0]_{\beta_3} = [\mathsf{nat}_0]_{\beta_3}^{+1}$. By 7., $[\mathsf{nat}_1]_{\beta_3} = [\mathsf{nat}_0]_{\beta_3}^{+1} = \{M \in \mathcal{M}_3^{(1)} \mid M \rightarrowtriangle^*_\beta \lambda f^{(1)}.f^{(1)} \vee M \rightarrowtriangle^*_\beta \lambda f^{(1)}.\lambda y^{(1)}.(f^{(1)})^n y^{(1)}$ where $n \geq 1\}$.

9. Let $f, y \in \mathsf{Var}_2$ and take $\overline{M} = \{M \in \mathcal{M}_3^{\oslash} \mid k, n \geq 0 \wedge \mathsf{deg}(Q_i) \succeq (1) \wedge (M \rightarrowtriangle^*_\beta x^{\oslash}P_1 \ldots P_k \vee M \rightarrowtriangle^*_\beta f^{\oslash}(x^{(1)}Q_1 \ldots Q_n) \vee M \rightarrowtriangle^*_\beta y^{\oslash} \vee M \rightarrowtriangle^*_\beta f^{\oslash}y^{(1)})\}$. The set $\overline{M}$ is β-saturated and $\forall x \in \mathsf{Var}_1$. $\mathsf{VAR}_x^{\oslash} \subseteq \overline{M} \subseteq \mathcal{M}_3^{\oslash}$. Let $\mathcal{I}$ be a β_3-interpretation such that $\mathcal{I}(\mathsf{a}) = \overline{M}$. If $M \in [\mathsf{nat}_0']_{\beta_3}$ then M is closed and $M \in (\overline{M}^{+1} \leadsto \overline{M}) \leadsto (\overline{M}^{+1} \leadsto \overline{M})$. Let $N \in \overline{M}^{+1}$ such that $N \diamond f^{\oslash}$. We have $N \rightarrowtriangle^*_\beta x^{(1)}P_1^{+1} \ldots P_k^{+1}$ or $N \rightarrowtriangle^*_\beta y^{(1)}$, for some $k \geq 0$ and $P_1, \ldots, P_k$. Therefore $f^{\oslash}N \rightarrowtriangle^*_\beta f^{\oslash}(x^{(1)}P_1^{+1} \ldots P_k^{+1}) \in \overline{M}$ or $f^{\oslash}N \rightarrowtriangle^*_\beta f^{\oslash}y^{(1)} \in \overline{M}$, thus $f^{\oslash} \in \overline{M}^{+1} \leadsto \overline{M}$. We have $f^{\oslash} \in \overline{M}^{+1} \leadsto \overline{M}$, $y^{(1)} \in \overline{M}^{+1}$ and $\diamond\{M, f^{\oslash}, y^{(1)}\}$, then $Mf^{\oslash}y^{(1)} \in \overline{M}$. Because M is closed and $\mathsf{dj}(\{x^{\oslash}, x^{(1)}, y^{\oslash}\}, \{y^{(1)}, f^{\oslash}\})$, by Lemma 7.1.11.2, $Mf^{\oslash}y^{(1)} \rightarrowtriangle^*_\beta f^{\oslash}y^{(1)}$. Hence, by Lemma B.2.1.4, $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.f^{\oslash}$ or $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.\lambda y^{(1)}.f^{\oslash}y^{(1)}$. Moreover, by Lemma 7.1.11.2, $\mathsf{deg}(M) = \oslash$ and $M \in \mathcal{M}_3^{\oslash}$.

   Conversely, let $M \in \mathcal{M}_3^{\oslash}$ such M is closed and $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.f^{\oslash}$ or $M \rightarrowtriangle^*_\beta \lambda f^{\oslash}.\lambda y^{(1)}.f^{\oslash}y^{(1)}$. Let $\mathcal{I}$ be an β_3-interpretation, $N \in \mathcal{I}(\mathsf{e}_1\mathsf{a}{\to}\mathsf{a}) = \mathcal{I}(\mathsf{a})^{+1} \leadsto$

$\mathcal{I}(\mathsf{a})$ and $N' \in \mathcal{I}(\mathsf{a})^{+1}$ where $\diamond\{M, N, N'\}$. By Lemma 8.1.4.1b, $N \in \mathcal{M}_3^{\oslash}$ and $N' \in \mathcal{M}_3^{(1)}$, so $MNN', NN' \in \mathcal{M}_3^{\oslash}$. Since $MNN' \rightarrow\!\!\!\triangleright_\beta^* NN'$, $NN' \in \mathcal{I}(\mathsf{a})$ and $\mathcal{I}(\mathsf{a})$ is β-saturated then $MNN' \in \mathcal{I}(\mathsf{a})$. Hence, $M \in (\mathcal{I}(\mathsf{a})^{+1} \rightsquigarrow \mathcal{I}(\mathsf{a})) \rightsquigarrow (\mathcal{I}(\mathsf{a})^{+1} \rightsquigarrow \mathcal{I}(\mathsf{a}))$ and $M \in [\mathsf{nat}'_0]_{\beta_3}$. □

B.2.2 Completeness challenges in $\lambda I^{\mathbb{N}}$ (Sec. 8.2)

Completeness for $\vdash_2$ fails with more than one E-variable (Sec. 8.2.2)

Proof of Remark 8.2.2. 1. For every interpretation $\mathcal{I}$, $\mathcal{I}(\mathsf{e}_1\mathsf{a}\to\mathsf{a}) = \mathcal{I}(\mathsf{e}_2\mathsf{a}\to\mathsf{a}) = \mathcal{I}(\mathsf{a})^+ \rightsquigarrow \mathcal{I}(\mathsf{a})$. Let $M \in \mathcal{I}(\mathsf{a})^+ \rightsquigarrow \mathcal{I}(\mathsf{a})$. By Lemma 8.1.4.1c, $\mathsf{deg}(M) = 0$. We have $M \diamond \lambda f^0.f^0$. $(\lambda f^0.f^0)M \rightarrow\!\!\!\triangleright_\beta M \in \mathcal{I}(\mathsf{a})^+ \rightsquigarrow \mathcal{I}(\mathsf{a})$. By Lemma 8.1.4.1a, $(\lambda f^0.f^0)M \in \mathcal{I}(\mathsf{a})^+ \rightsquigarrow \mathcal{I}(\mathsf{a})$. Therefore, $\lambda y^0.y^0 \in [\mathsf{nat}''_0]_{\beta_2}$.

2. If $\lambda f^0.f^0 : \langle() \vdash_2 \mathsf{nat}''_0\rangle$, by Lemmas 7.4.2.2 and 7.4.2.1, $f^0 : \langle f^0 : \mathsf{e}_1\mathsf{a}\to\mathsf{a} \vdash_2 \mathsf{e}_2\mathsf{a}\to\mathsf{a}\rangle$ and $\mathsf{e}_1\mathsf{a}\to\mathsf{a} \sqsubseteq \mathsf{e}_2\mathsf{a}\to\mathsf{a}$. Thus, by Lemma B.1.11.4, $\mathsf{e}_2\mathsf{a} \sqsubseteq \mathsf{e}_1\mathsf{a}$. Again, by Lemma B.1.11.3, $\mathsf{e}_1\mathsf{a} = \mathsf{e}_2 U$ where $\mathsf{a} \sqsubseteq U$. This is impossible because $\mathsf{e}_1 \neq \mathsf{e}_2$. □

Completeness for $\vdash_2$ with only one E-variable (Sec. 8.2.3)

Proof of Lemma 8.2.3. 1. We prove the result by induction on U and then by case on the last rule.

- Let $U = U_1 \sqcap U_2$. By definition $\mathsf{deg}(U_1), \mathsf{deg}(U_2) > 0$. Therefore by IH, $\mathsf{e}_1{U_1}^- = U_1$ and $\mathsf{e}_2{U_2}^- = U_2$. Finally, $\mathsf{e}_1U^- = \mathsf{e}_1U_1 \sqcap \mathsf{e}_1{U_2}^- = \mathsf{e}_1{U_1}^- \sqcap \mathsf{e}_1{U_2}^- = U_1 \sqcap U_2 = U$.
- Let $U = \mathsf{e}_1U_1$. Therefore $\mathsf{e}_1U^- = \mathsf{e}_1\mathsf{e}_1{U_1}^- = \mathsf{e}_1U_1$.
- Cases $U = U_1\to T$ and $U = a$ are trivial because by Lemma 7.2.3.2a, $\mathsf{deg}(U) = 0$.

2. If $U^- = V^-$ then $\mathsf{e}_1U^- = \mathsf{e}_1V^-$ and by 1., $U = V$. □

Lemma B.2.2.

1. *If* $\mathsf{deg}(U) = n$ *then* DVar_U *is an infinite set* $\{y^n \mid y \in \mathsf{Var}_2\}$.
2. *If* $U \neq V$ *and* $\mathsf{deg}(U) = \mathsf{deg}(V) = n$ *then* $\mathsf{dj}(\mathsf{DVar}_U, \mathsf{DVar}_V)$.
3. *If* $y^n \in \mathsf{DVar}_U$ *then* $y^{n+1} \in \mathsf{DVar}_{\mathsf{e}_1U}$.
4. *If* $y^{n+1} \in \mathsf{DVar}_U$ *then* $y^n \in \mathsf{DVar}_{U^-}$. □

Proof of Lemma B.2.2.

1. We prove this result by induction on n. Let $n = 0$ then we conclude by definition. Let $n = m + 1$. Then $\mathsf{DVar}_U = \{y^{n+1} \mid y^n \in \mathsf{DVar}_{U^-}\}$. By IH, DVar_{U^-} is an infinite set $\{y^m \mid y \in \mathsf{Var}_2\}$. Therefore DVar_U is an infinite set $\{y^n \mid y \in \mathsf{Var}_2\}$.

2. We prove the result by induction on n. Let $n = 0$ then we conclude by definition. Let $n = m + 1$. Then $\mathsf{DVar}_U = \{y^{n+1} \mid y^n \in \mathsf{DVar}_{U^-}\}$ and $\mathsf{DVar}_V = \{y^{n+1} \mid y^n \in \mathsf{DVar}_{V^-}\}$. By Lemma 8.2.3.2, $U^- \neq V^-$, and by definition, $\mathsf{deg}(U^-) = \mathsf{deg}(V^-) = m$ By IH, $\mathsf{dj}(\mathsf{DVar}_{U^-}, \mathsf{DVar}_{V^-})$. Therefore, $\mathsf{dj}(\mathsf{DVar}_U, \mathsf{DVar}_V)$.

3. Because $(\mathsf{e}_1 U)^- = U$.

4. By definition. □

Lemma B.2.3.

1. *If* $\Gamma \subseteq \mathsf{BPreEnv}^n$ *then* $\mathsf{e}_1\Gamma \subseteq \mathsf{BPreEnv}^{n+1}$.

2. *If* $\Gamma \subseteq \mathsf{BPreEnv}^{n+1}$ *then* $\Gamma^- \subseteq \mathsf{BPreEnv}^n$.

3. *If* $\Gamma_1 \subseteq \mathsf{BPreEnv}^n$, $\Gamma_2 \subseteq \mathsf{BPreEnv}^m$ *and* $m \geq n$ *then* $\Gamma_1 \sqcap \Gamma_2 \subseteq \mathsf{BPreEnv}^n$. □

Proof of Lemma B.2.3.

1. Because $\Gamma \subseteq \mathsf{BPreEnv}^n$, $\Gamma = (y_i^{n_i} : U_i)_m$ such that $\forall i \in \{1, \ldots, m\}.\ \mathsf{deg}(U_i) = n_i \wedge n_i \geq n \wedge y_i^{n_i} \in \mathsf{DVar}_{U_i}$. Therefore, $\mathsf{e}_1\Gamma = (y_i^{n_i+1} : \mathsf{e}_1 U_i)_m$ and by Lemma B.2.2.3, $\forall i \in \{1, \ldots, m\}.\ \mathsf{deg}(\mathsf{e}_1 U_i) = n_i + 1 \wedge n_i + 1 \geq n + 1 \wedge y_i^{n_i+1} \in \mathsf{DVar}_{\mathsf{e}_1 U_i}$. Finally, $\mathsf{e}_1\Gamma \subseteq \mathsf{BPreEnv}^{n+1}$.

2. Because $\Gamma \subseteq \mathsf{BPreEnv}^{n+1}$, $\Gamma = (y_i^{n_i} : U_i)_m$ such that $\forall i \in \{1, \ldots, m\}.\ \mathsf{deg}(U_i) = n_i \wedge n_i \geq n + 1 \wedge y_i^{n_i} \in \mathsf{DVar}_{U_i}$. Therefore, $\Gamma^- = (y_i^{n_i-1} : U_i{}^-)_m$ and $\forall i \in \{1, \ldots, m\}.\ \mathsf{deg}(U_i{}^-) = n'_i \wedge n_i = n'_i + 1 \wedge n'_i \geq n \wedge y_i^{n'_i+1} \in \mathsf{DVar}_{U_i}$. By Lemma B.2.2.4, $\forall i \in \{1, \ldots, m\}.\ y_i^{n'_i} \in \mathsf{DVar}_{U_i{}^-}$. Finally, $\Gamma^- \subseteq \mathsf{BPreEnv}^n$.

3. Note that $\mathsf{BPreEnv}^m \subseteq \mathsf{BPreEnv}^n$. Therefore $\Gamma_1, \Gamma_2 \subseteq \mathsf{BPreEnv}^n$. Let $(\Gamma_1 \sqcap \Gamma_2)(x^p) = U_1 \sqcap U_2$ such that $\Gamma_1(x^p) = U_1$ and $\Gamma_2(x^p) = U_2$. Then $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = p \geq n$ and $x^p \in \mathsf{DVar}_{U_1} \cap \mathsf{DVar}_{U_2}$. Hence, by Lemma B.2.2.2, $U_1 = U_2$. Finally, we can prove that $\Gamma_1 \sqcap \Gamma_2 = \Gamma_1 \cup \Gamma_2 \subset \mathsf{BPreEnv}^n$. □

Lemma B.2.4.

1. $(\mathsf{OPEN}^n)^+ = \mathsf{OPEN}^{n+1}$.

2. *If* $y \in \mathsf{Var}_2$ *and* $(My^m) \in \mathsf{OPEN}^n$ *then* $M \in \mathsf{OPEN}^n$.

3. *If $M \in \mathsf{OPEN}^n$, $M \diamond N$, $N \in \mathbb{M}$ and $\mathsf{deg}(N) = m \geq n$ then $MN \in \mathsf{OPEN}^n$.*

4. *If $\mathsf{deg}(M) = n$, $m \geq n$, $M \diamond N$, $M \in \mathbb{M}$ and $N \in \mathsf{OPEN}^m$ then $MN \in \mathsf{OPEN}^n$.* □

Proof of Lemma B.2.4. 1. By Lemma B.1.3.1a. 2. By definition $x^i \in \mathsf{fv}(My^m)$ and $i \geq n$. Because $x \neq y$ then $x^i \in \mathsf{fv}(M)$. Therefore $M \in \mathsf{OPEN}^n$. 3. By hypothesis, $M \in \mathbb{M}^n$ and $x^i \in \mathsf{fv}(M)$ such that $x \in \mathsf{Var}_1$ and $i \geq n$. By definition $MN \in \mathbb{M}^n$ and therefore $MN \in \mathsf{OPEN}^n$. 4. Similar to 3. □

Proof of Lemma 8.2.8.

1. First we show that $\mathbb{I}(a)$ is β-saturated. Let $M \rightarrow\!\!\triangleright^*_\beta N$ and $N \in \mathbb{I}(a)$.

   - If $N \in \mathsf{OPEN}^0$ then $N \in \mathbb{M}^0$ and x^i for some $x \in \mathsf{Var}_1$, $i \geq 0$ and $x^i \in \mathsf{fv}(N)$. By Lemma 8.1.2.9, $\mathbb{M}^0$ is β-saturated and so, $M \in \mathbb{M}^0$. By Lemma 7.1.11.3, $\mathsf{fv}(M) = \mathsf{fv}(N)$ and so, $x^i \in \mathsf{fv}(M)$. Hence, $M \in \mathsf{OPEN}^0$
   - If $N \in \{M \in \mathcal{M}^0_2 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 a\rangle\}$ then $\exists\ \Gamma \subseteq \mathsf{BPreEnv}^0$, such that $N : \langle \Gamma \vdash_2 a\rangle$. By subject expansion corollary 7.4.6, $M : \langle \Gamma \vdash_2 a\rangle$ and by Lemma 7.1.11.3, $\mathsf{deg}(M) = \mathsf{deg}(N)$. Hence, $M \in \{M \in \mathcal{M}^0_2 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 a\rangle\}$.

   Now we show that $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}^0_x \subseteq \mathbb{I}(a) \subseteq \mathbb{M}^0$.

   - Let $x \in \mathsf{Var}_1$ and $M \in \mathsf{VAR}^0_x$. Hence, $M = x^0 N_1 \ldots N_k \in \mathbb{M}^0$, and $x^0 \in \mathsf{fv}(M)$. Thus, $M \in \mathsf{OPEN}^0$.
   - Let $M \in \mathbb{I}(a)$. If $M \in \mathsf{OPEN}^0$ then $M \in \mathbb{M}^0$. Else, $\exists\ \Gamma \subseteq \mathsf{BPreEnv}^0$ such that $M : \langle \Gamma \vdash_2 a\rangle$. Since by Theorem 7.3.5, $M \in \mathbb{M}$ and $\mathsf{deg}(M) = \mathsf{deg}(a) = 0$, $M \in \mathbb{M}^0$.

2. By induction on $U \in \mathsf{GITy}$.

   - Let $U = a$: By definition of $\mathbb{I}$ and by 1.
   - Let$U = \mathsf{e}_1 V$: $\mathsf{deg}(V) = n - 1$ and, by Lemma 7.2.3, $V \in \mathsf{GITy}$. By IH and Lemma B.2.4.1, $\mathbb{I}(\mathsf{e}_1 V) = (\mathbb{I}(V))^+ = (\mathsf{OPEN}^{n-1} \cup \{M \in \mathbb{M}^{n-1} \mid M : \langle \mathsf{BPreEnv}^{n-1} \vdash_2 V\rangle\})^+ = \mathsf{OPEN}^n \cup (\{M \in \mathbb{M}^{n-1} \mid M : \langle \mathsf{BPreEnv}^{n-1} \vdash_2 V\rangle\})^+$.
     - If $M \in \mathbb{M}^{n-1}$ and $M : \langle \mathsf{BPreEnv}^{n-1} \vdash_2 V\rangle$ then $M : \langle \Gamma \vdash_2 V\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^{n-1}$. By rule (exp) and Lemma B.2.3.1, $M^+ : \langle \mathsf{e}_1\Gamma \vdash_2 \mathsf{e}_1 V\rangle$ and $\mathsf{e}_1\Gamma \subseteq \mathsf{BPreEnv}^n$. Thus by Theorem 7.3.5.2, $M^+ \in \mathbb{M}^n$ and $M^+ : \langle \mathsf{BPreEnv}^n \vdash_2 \mathsf{e}_1 V\rangle$.

  - If $M \in \mathbb{M}^n$ and $M : \langle \mathsf{BPreEnv}^n \vdash_2 \mathsf{e}_1 V\rangle$ then $M : \langle \Gamma \vdash_2 \mathsf{e}_1 V\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^n$. By Theorem 7.3.5.2, and Lemma B.2.3.2, $M^- : \langle \Gamma^- \vdash_2 V\rangle$ and $\Gamma^- \subseteq \mathsf{BPreEnv}^{n-1}$. Thus, by Lemma B.1.3.(1b. and 1d.), $M = (M^-)^+$ and $M^- \in \mathbb{M}^{n-1}$. Hence, $M^- \in \{M \in \mathbb{M}^{n-1} \mid M : \langle \mathsf{BPreEnv}^{n-1} \vdash_2 V\rangle\}$.

  Hence $(\{M \in \mathbb{M}^{n-1} \mid M : \langle \mathsf{BPreEnv}^{n-1} \vdash_2 V\rangle\})^+ = \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U\rangle\}$ and finally, $\mathbb{I}(U) = \mathsf{OPEN}^n \cup \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U\rangle\}$.

- Let $U = U_1 \sqcap U_2$: By Lemma 7.2.3.1b, $U_1, U_2 \in \mathsf{GITy}$ and $\mathsf{deg}(U_1) = \mathsf{deg}(U_2) = n$. By IH, $\mathbb{I}(U_1 \sqcap U_2) = \mathbb{I}(U_1) \cap \mathbb{I}(U_2) = (\mathsf{OPEN}^n \cup \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1\rangle\}) \cap (\mathsf{OPEN}^n \cup \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U_2\rangle\}) = \mathsf{OPEN}^n \cup (\{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1\rangle\} \cap \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U_2\rangle\})$.

  - If $M \in \mathbb{M}^n$, $M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1\rangle$ and $M : \langle \mathsf{BPreEnv}^n \vdash_2 U_2\rangle$ then $M : \langle \Gamma_1 \vdash_2 U_1\rangle$ and $M : \langle \Gamma_2 \vdash_2 U_2\rangle$ where $\Gamma_1, \Gamma_2 \subseteq \mathsf{BPreEnv}^n$. By Remark 7.3.6, $M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 U_1 \sqcap U_2\rangle$. Because by Lemma B.2.3.3, $\Gamma_1 \sqcap \Gamma_2 \subseteq \mathsf{BPreEnv}^n$, we obtain $M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1 \sqcap U_2\rangle$.
  - If $M \in \mathbb{M}^n$ and $M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1 \sqcap U_2\rangle$ then $M : \langle \Gamma \vdash_2 U_1 \sqcap U_2\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^n$. By rule ($\sqsubseteq$), $M : \langle \Gamma \vdash_2 U_1\rangle$ and $M : \langle \Gamma \vdash_2 U_2\rangle$. Hence, $M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1\rangle$ and $M : \langle \mathsf{BPreEnv}^n \vdash_2 U_2\rangle$.

  We deduce that $\mathbb{I}(U_1 \sqcap U_2) = \mathsf{OPEN}^n \cup \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U_1 \sqcap U_2\rangle\}$.

- Let $U = V \to T$: By Lemma 7.2.3, $V, T \in \mathsf{GITy}$ and let $m = \mathsf{deg}(V) \geq \mathsf{deg}(T) = 0$. By IH, $\mathbb{I}(V) = \mathsf{OPEN}^m \cup \{M \in \mathbb{M}^m \mid M : \langle \mathsf{BPreEnv}^m \vdash_2 V\rangle\}$ and $\mathbb{I}(T) = \mathsf{OPEN}^0 \cup \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 T\rangle\}$. By definition, $\mathbb{I}(V \to T) = \mathbb{I}(V) \rightsquigarrow \mathbb{I}(T)$.

  - Let $M \in \mathbb{I}(V) \rightsquigarrow \mathbb{I}(T)$. By Lemma B.2.2.1, let $y^m \in \mathsf{DVar}_V$ such that $y \in \mathsf{Var}_2$, and $\forall n, y^n \notin \mathsf{fv}(M)$. Then $y^m \diamond M$. By remark 7.3.6, $y^m : \langle (y^m : V) \vdash_2 V\rangle$. Hence, $y^m : \langle \mathsf{BPreEnv}^m \vdash_2 V\rangle$ and so $y^m \in \mathbb{I}(V)$ and $My^m \in \mathbb{I}(T)$.
    - ∗ If $My^m \in \mathsf{OPEN}^0$ then since $y \in \mathsf{Var}_2$, by Lemma B.2.4.2, $M \in \mathsf{OPEN}^0$.
    - ∗ If $My^m \in \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 T\rangle\}$ then $My^m \in \mathbb{M}^0$ and $My^m : \langle \mathsf{BPreEnv}^0 \vdash_2 T\rangle$. So $My^m : \langle \Gamma \vdash_2 T\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^0$. Since $y^m \in \mathsf{fv}(My^m)$ and since by Theorem 7.3.5, $\mathsf{dom}(\Gamma) = \mathsf{fv}(My^m)$, $\Gamma = \Gamma', (y^m : V')$, and $\mathsf{deg}(V') = m$. Since $(\!| y^m, V' |\!) \in \mathsf{BPreEnv}^0$, $\mathsf{deg}(V') = m$ and $y^m \in \mathsf{DVar}_{V'}$, by Lemma B.2.2.2, $V = V'$. So $My^m : \langle \Gamma', (y^m : V) \vdash_2 T\rangle$ and by

Lemma B.1.14.1, $M : \langle \Gamma' \vdash_2 V \to T\rangle$ and by Theorem 7.3.5.2, $M \in \mathbb{M}$ and $\mathsf{deg}(M) = 0$. Since $\Gamma' \subseteq \mathsf{BPreEnv}^0$, $M : \langle \mathsf{BPreEnv}^0 \vdash_2 V \to T\rangle$. And so, $M \in \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 V \to T\rangle\}$.

- Let $M \in \mathsf{OPEN}^0 \cup \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 V \to T\rangle\}$ and $N \in \mathbb{I}(V) = \mathsf{OPEN}^m \cup \{M \in \mathbb{M}^m \mid M : \langle \mathsf{BPreEnv}^m \vdash_2 V\rangle\}$ such that $M \diamond N$. Then, $\mathsf{deg}(N) = m$.
  - ∗ Case $M \in \mathsf{OPEN}^0$. Since $N \in \mathbb{M}$, by Lemma B.2.4.3, $MN \in \mathsf{OPEN}^0 \subseteq \mathbb{I}(T)$.
  - ∗ Case $M \in \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 V \to T\rangle\}$, so $M \in \mathbb{M}^0$.
    - · If $N \in \mathsf{OPEN}^m$ then, by Lemma B.2.4.4, $MN \in \mathsf{OPEN}^0 \subseteq \mathbb{I}(T)$.
    - · If $N \in \{M \in \mathbb{M}^m \mid M : \langle \mathsf{BPreEnv}^m \vdash_2 V\rangle\}$, then $M : \langle \Gamma_1 \vdash_2 V \to T\rangle$ and $N : \langle \Gamma_2 \vdash_2 V\rangle$ where $\Gamma_1 \subseteq \mathsf{BPreEnv}^0$ and $\Gamma_2 \subseteq \mathsf{BPreEnv}^m$. Because $M \diamond N$, then by Lemma B.1.15.2, $\Gamma_1 \diamond \Gamma_2$. So by rule $(\to_{\mathsf{E}})$, $MN : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_2 T\rangle$. By Lemma B.2.3.3, $\Gamma_1 \sqcap \Gamma_2 \subseteq \mathsf{BPreEnv}^0$. Therefore $MN : \langle \mathsf{BPreEnv}^0 \vdash_2 T\rangle$. By Theorem 7.3.5, $MN \in \mathbb{M}^0$. Hence, $MN \in \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 T\rangle\} \subseteq \mathbb{I}(T)$.

  In all cases, $M \in \mathbb{I}(V \to T)$.

We deduce that $\mathbb{I}(V \to T) = \mathsf{OPEN}^0 \cup \{M \in \mathbb{M}^0 \mid M : \langle \mathsf{BPreEnv}^0 \vdash_2 V \to T\rangle\}$.

□

Proof of Theorem 8.2.9. By definition we have: $[U]_{\beta_2} = \{M \in \mathcal{M}_2 \mid \mathsf{closed}(M) \wedge M \in \bigcap_{\mathcal{I} \in \mathsf{Interp}^{\beta_2}} \mathcal{I}(U)\}$.

1. Let $M \in [U]_{\beta_2}$. Then M is a closed term and $M \in \mathbb{I}(U)$. Hence, by Lemma 8.2.8, $M \in \mathsf{OPEN}^n \cup \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U\rangle\}$. Because M is closed, $M \notin \mathsf{OPEN}^n$. Hence, $M \in \{M \in \mathbb{M}^n \mid M : \langle \mathsf{BPreEnv}^n \vdash_2 U\rangle\}$ and so, $M : \langle \Gamma \vdash_2 U\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^n$. Since M is closed, by Theorem 7.3.5.2a, $\Gamma = ()$ and therefore $M : \langle () \vdash_2 U\rangle$.

   Conversely, let $M \in \mathbb{M}^n$ where $M : \langle () \vdash_2 U\rangle$. By Theorem 7.3.5.2a, M is closed. Let $\mathcal{I}$ be a β_2-interpretation. By soundness Lemma 8.1.6, $M \in \mathcal{I}(U)$. Thus, $M \in [U]_{\beta_2}$.

2. Let $M \in [U]_{\beta_2}$ and $M \rightarrow\!\!\!\!\triangleright_{\beta}^{*} N$. By 1., $M \in \mathbb{M}^n$ and $M : \langle () \vdash_2 U\rangle$. By subject reduction Corollary 7.4.6, $N : \langle () \vdash_2 U\rangle$. By Lemma 7.1.11.3, $\mathsf{deg}(N) = \mathsf{deg}(M) = n$. By Theorem 7.3.5.2, $N \in \mathbb{M}$. Hence, by 1., $N \in [U]_{\beta_2}$.

3. Let $N \in [U]_{\beta_2}$ and $M \rightarrow\!\!\!\triangleright_\beta^* N$. By 1., $N \in \mathbb{M}^n$ and $N : \langle () \vdash_2 U \rangle$. By subject expansion Corollary 7.4.6, $M : \langle () \vdash_2 U \rangle$. By Lemma 7.1.11.3, $\mathsf{deg}(N) = \mathsf{deg}(M) = n$. By Theorem 7.3.5.2, $M \in \mathbb{M}$. Hence, by 1., $M \in [U]_{\beta_2}$. □

B.2.3 Completeness for $\lambda^{\mathcal{L}_\mathbb{N}}$ (Sec. 8.3)

Proof of Lemma 8.3.2. 1. Let $\mathsf{deg}(U) = L_1$ and $\mathsf{deg}(V) = L_2$ such that $L_1 = L :: L'_1$ and $L_2 = L :: L'_2$. By Lemma B.1.12.2:

- Either $U = \omega^{L::L'_1} = \mathsf{e}_L \omega^{L'_1}$.
- Or $U = \vec{\mathsf{e}}_{L::L'_1} \sqcap_{i=1}^p T_i = \vec{\mathsf{e}}_L \vec{\mathsf{e}}_{L'_1} \sqcap_{i=1}^p T_i$ such that $p \geq 1$ and $\forall i \in \{1, \ldots, p\}.\ T_i \in \mathsf{Ty}_3$.

In both cases there exists U' such that $U = \mathsf{e}_L U'$. Similarly, there exists V' such that $V = \mathsf{e}_L V'$. If $U^- L = V^{-L}$ then $U' = V'$ and therefore $U = V$.

2. Easy induction on L

3. We have $\mathsf{DVar}_U = \{y^L \mid y^\oslash \in \mathsf{DVar}_{U^{-L}}\}$ and $\mathsf{DVar}_V = \{y^L \mid y^\oslash \in \mathsf{DVar}_{V^{-L}}\}$. By 1., $U^{-L} \neq V^{-L}$. By Lemma B.1.12, $\mathsf{deg}(U^{-L}) = \mathsf{deg}(V^{-L}) = \oslash$. Therefore by definition, $\mathsf{dj}(\mathsf{DVar}_{U^{-L}}, \mathsf{DVar}_{V^{-L}})$, and finally, $\mathsf{dj}(\mathsf{DVar}_U, \mathsf{DVar}_V)$.

4. We prove the result by induction on L. The case $L = \oslash$ is by definition. Let $L = i :: L'$. By IH, $\bigcup_{U \in \mathsf{ITy}_3^{L'}} \mathsf{DVar}_U = \mathsf{Var}^{L'}$. Let $y^L \in \bigcup_{U \in \mathsf{ITy}_3^L} \mathsf{DVar}_U$ then $y^{L'} \in \mathsf{DVar}_{U^{-i}}$ for some $U \in \mathsf{ITy}_3^L$. We have, $U^{-i} \in \mathsf{ITy}_3^{L'}$. Therefore, $y^{L'} \in \mathsf{Var}^{L'}$. Finally, $y^L \in \mathsf{Var}^L$. Let $y^L \in \mathsf{Var}^L$ then $y^{L'} \in \mathsf{Var}^{L'}$. Therefore, $y^{L'} \in \mathsf{DVar}_U$ for some $U \in \mathsf{ITy}_3^{L'}$. We have, $\mathsf{e}_i U \in \mathsf{ITy}_3^L$. and $\mathsf{e}_i U^{-i} = U$. Therefore, $y^L \in \mathsf{DVar}_{\mathsf{e}_i U}$. Finally, $y^L \bigcup_{U \in \mathsf{ITy}_3^L} \mathsf{DVar}_U$.

5. Let $y^L \in \mathsf{DVar}_U$ then because $\mathsf{e}_i U^{-i} = U$, we obtain by definition $y^{i::L} \in \mathsf{DVar}_{\mathsf{e}_i U}$.

6. By definition. □

Proof of Lemma 8.3.4.

1. Let $\Gamma \subseteq \mathsf{BPreEnv}^L$. By definition, we have $\Gamma = (x_i^{L_i} : U_i)_n$ such that $\forall i \in \{1, \ldots, n\}.\ x^{L_i} \in \mathsf{DVar}_{U_i} \wedge U_i \in \mathsf{ITy}_3^{L_i} \wedge L_i \succeq L$. Therefore $\forall i \in \{1, \ldots, n\}.\ \mathsf{deg}(U_i) = L_i$, i.e., $\mathsf{ok}(\Gamma)$.

2. Let $\Gamma \subseteq \mathsf{BPreEnv}^L$ then by definition $\Gamma = (x_j^{L_j} : U_j)_n$ such that $\forall j \in \{1, \ldots, n\}.\ x^{L_j} \in \mathsf{DVar}_{U_j} \wedge U_j \in \mathsf{ITy}_3^{L_j} \wedge L_j \succeq L$. Therefore, $\mathsf{e}_i \Gamma = (x_j^{i::L_j} : \mathsf{e}_i U_j)_n$ and by Lemma 8.3.2.5, $\forall j \in \{1, \ldots, n\}.\ x^{i::L_j} \in \mathsf{DVar}_{\mathsf{e}_i U_j} \wedge \mathsf{e}_i U_j \in \mathsf{ITy}_3^{i::L_j} \wedge i :: L_j \succeq i :: L$. By definition, we obtain $\mathsf{e}_i \Gamma \subseteq \mathsf{BPreEnv}^{i::L}$.

3. Let $\Gamma \subseteq \mathsf{BPreEnv}^{i::L}$. then by definition $\Gamma = (x_j^{L_j} : U_j)_n$ such that $\forall j \in \{1, \ldots, n\}.\ x^{L_j} \in \mathsf{DVar}_{U_j} \wedge U_j \in \mathsf{ITy}_3^{L_j} \wedge L_j \succeq i :: L$. By Lemma 8.3.2.6 and Lemma B.1.12, $\Gamma = (x_j^{i::L'_j} : \mathsf{e}_i U'_j)_n$ such that $\forall j \in \{1, \ldots, n\}.\ x^{L'_j} \in \mathsf{DVar}_{U'_j} \wedge U_j = \mathsf{e}_i U'_j \wedge L_j = i :: L'_j \wedge U_j \in \mathsf{ITy}_3^{i::L'_j} \wedge L'_j \succeq L$. We then have $\Gamma^{-i} = (x_j^{L'_j} : U'_j)_n$ such that $\forall j \in \{1, \ldots, n\}.\ x^{L'_j} \in \mathsf{DVar}_{U'_j} \wedge U'_j \in \mathsf{ITy}_3^{L'_j} \wedge L'_j \succeq L$, i.e., $\Gamma^{-i} \subseteq \mathsf{BPreEnv}^L$.

4. Let $\Gamma_1 \subseteq \mathsf{BPreEnv}^L$, $\Gamma_2 \subseteq \mathsf{BPreEnv}^K$, and $L \preceq K$. By definition, we have $\Gamma_1 = (x_i^{L_i} : U_i)_n$ and $\Gamma_2 = (y_i^{K_i} : V_i)_m$ such that $\forall i \in \{1, \ldots, n\}.\ x^{L_i} \in \mathsf{DVar}_{U_i} \wedge U_i \in \mathsf{ITy}_3^{L_i} \wedge L_i \succeq L$ and $\forall i \in \{1, \ldots, m\}.\ y^{K_i} \in \mathsf{DVar}_{V_i} \wedge V_i \in \mathsf{ITy}_3^{K_i} \wedge K_i \succeq K$. By 1, $\mathsf{ok}(\Gamma_1)$ and $\mathsf{ok}(\Gamma_2)$, therefore $\Gamma_1 \sqcap \Gamma_2$ is well-defined. Let $(\Gamma_1 \sqcap \Gamma_2)(x^{L'}) = U$. Either $x^{L'} \in \mathsf{dom}(\Gamma_1) \setminus \mathsf{dom}(\Gamma_2)$ then by hypothesis, $x^{L'} \in \mathsf{DVar}_U$, $U \in \mathsf{ITy}_3^{L'}$, and $L' \succeq L$. Or $x^{L'} \in \mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1)$ then by hypothesis, $x^{L'} \in \mathsf{DVar}_U$, $U \in \mathsf{ITy}_3^{L'}$, and $L' \succeq K \succeq L$. Or $x^{L'} \in \mathsf{dom}(\Gamma_2) \cap \mathsf{dom}(\Gamma_1)$ then $U = U_1 \sqcap U_2$ such that $\Gamma_1(x^{L'} = U_1)$ and $\Gamma_2(x^{L'} = U_2)$. By hypothesis, $y^{L'} \in \mathsf{DVar}_{U_1} \cap \mathsf{DVar}_{U_2}$, $U_1, U_2 \in \mathsf{ITy}_3^{L'}$, and $L' \succeq K \succeq L$. Because $\mathsf{dom}(U_1) = \mathsf{dom}(U_2) = L'$ then by Lemma 8.3.2.3, we have $U_1 = U_2$. and $U_1 \sqcap U_2 = U_1 = U_2 \in \mathsf{ITy}_3^{L'}$. We then have that $\Gamma_1 \sqcap \Gamma_2 \in \mathsf{BPreEnv}^L$. □

Proof of Lemma 8.3.6.

1. Let $M \in (\mathsf{OPEN}^L)^{+i}$ then $M = N^{+i}$ such that $N \in \mathsf{OPEN}^L$. By definition $N \in \mathcal{M}_3^L$ such that $x^K \in \mathsf{fv}(N)$, $x \in \mathsf{Var}_1$, and $K \succeq L$. By Lemma B.1.5.1, $M \in \mathcal{M}_3^{i::L}$, $x^{i::K} \in \mathsf{fv}(M)$, and $i :: K \succeq i :: L$. Hence, $M \in \mathsf{OPEN}^{i::L}$.

   Let $M \in \mathsf{OPEN}^{i::L}$. Then $M \in \mathcal{M}_3^{i::L}$, $x^K \in \mathsf{fv}(M)$, $x^K \in \mathsf{Var}_1$, and $K \succeq i :: L$. Therefore, $K = i :: K'$, $K_0 \succeq L$, and $\mathsf{deg}(M) = i :: L$. By Lemma B.1.5, $M = N^{+i}$ such that $N \in \mathcal{M}_3^L$ and $x^{K'} \in \mathsf{fv}(N)$. Hence $N \in \mathsf{OPEN}^L$ and $M \in (\mathsf{OPEN}^L)^{+i}$.

2. Let $y \in \mathsf{Var}_2$, $My^K \in \mathsf{OPEN}^L$, then $My^K \in \mathcal{M}_3^L$, $x^{L'} \in \mathsf{fv}(My^K)$, and $K' \succeq L$. Because $x \neq y$ then $x^{L'} \in \mathsf{fv}(M)$. By definition, $M \in \mathcal{M}_3^L$, therefore $M \in \mathsf{OPEN}^L$.

3. By definition of OPEN^L.

4. By definition of OPEN^L. □

Proof of Lemma 8.3.8.

1. We do two cases ($r = \beta\eta$ and $r = \beta$).

   Case $r = \beta\eta$. It is easy to see that $\forall x \in \mathsf{Var}_1.\ \mathsf{VAR}_x^{\oslash} \subseteq \mathsf{OPEN}^{\oslash} \subseteq \mathbb{I}_{\beta\eta}(a)$. Now we show that $\mathbb{I}_{\beta\eta}(a)$ is $\beta\eta$-saturated. Let $M \rightarrow\!\!\!\!\triangleright_{\beta\eta}^* N$ and $N \in \mathbb{I}_{\beta\eta}(a)$.

- If $N \in \mathsf{OPEN}^{\oslash}$ then $N \in \mathcal{M}_3^{\oslash}$, $x \in \mathsf{Var}_1$, and $x^L \in \mathsf{fv}(N)$ for some L. By Theorem 7.1.11.2, $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$ and $\mathsf{deg}(M) = \mathsf{deg}(N)$, hence, $M \in \mathsf{OPEN}^{\oslash}$
- If $N \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* a\rangle\}$ then $N \mathrel{-\!\!\rhd}^*_{\beta\eta} N'$ and $\exists\ \Gamma \subseteq \mathsf{BPreEnv}^{\oslash}$, such that $N' : \langle \Gamma \vdash_3 a\rangle$. Hence $M \mathrel{-\!\!\rhd}^*_{\beta\eta} N'$ and since by Theorem 7.1.11.2, $\mathsf{deg}(M) = \mathsf{deg}(N')$, $M \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* a\rangle\}$.

Case $r = \beta$. It is easy to see that $\forall x \in \mathsf{Var}_1$. $\mathsf{VAR}_x^{\oslash} \subseteq \mathsf{OPEN}^{\oslash} \subseteq \mathbb{I}_\beta(a)$. Now we show that $\mathbb{I}_\beta(a)$ is β-saturated. Let $M \mathrel{-\!\!\rhd}^*_{\beta} N$ and $N \in \mathbb{I}_\beta(a)$.

- If $N \in \mathsf{OPEN}^{\oslash}$ then $N \in \mathcal{M}_3^{\oslash}$, $x \in \mathsf{Var}_1$, and $x^L \in \mathsf{fv}(N)$ for some L. By Theorem 7.1.11.2, $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$ and $\mathsf{deg}(M) = \mathsf{deg}(N)$, hence, $M \in \mathsf{OPEN}^{\oslash}$
- If $N \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 a\rangle\}$ then $\exists\ \Gamma \subseteq \mathsf{BPreEnv}^{\oslash}$, such that $N : \langle \Gamma \vdash_3 a\rangle$. By Theorem 7.4.14, $M : \langle \Gamma\uparrow^M \vdash_3 a\rangle$. Since by Theorem 7.1.11.2, $\mathsf{fv}(N) \subseteq \mathsf{fv}(M)$, let $\mathsf{fv}(N) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ and $\mathsf{fv}(M) = \mathsf{fv}(N) \cup \{x_{n+1}^{L_{n+1}}, \ldots, x_{n+m}^{L_{n+m}}\}$. So $\Gamma\uparrow^M = \Gamma, (x_{n+1}^{L_{n+1}} : \omega^{L_{n+1}}, \ldots, x_{n+m}^{L_{n+m}} : \omega^{L_{n+m}})$. For each $i \in \{n+1, \ldots, n+m\}$, take U_i such that $x_i^{L_i} \in \mathsf{DVar}_{U_i}$. Then $\Gamma, (x_{n+1}^{L_{n+1}} : U_{n+1}, \ldots, x_{n+m}^{L_{n+m}} : U_{n+m}) \subseteq \mathsf{BPreEnv}^{\oslash}$ and by Remark.7.3.6.4 and rule $(\sqsubseteq)$, $M : \langle \Gamma, (x_{n+1}^{L_{n+1}} : U_{n+1}, \ldots, x_{n+m}^{L_{n+m}} : U_{n+m}) \vdash_3 a\rangle$. Thus $M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 a\rangle$ and since by Theorem 7.1.11.2, $\mathsf{deg}(M) = \mathsf{deg}(N)$, $M \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 a\rangle\}$.

2. By induction on U.

   - $U = a$: By definition of $\mathbb{I}_{\beta\eta}$.
   - $U = \omega^L$: By definition, $\mathbb{I}_{\beta\eta}(\omega^L) = \mathcal{M}_3^L$. Hence, $\mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* \omega^L\rangle\} \subseteq \mathbb{I}_{\beta\eta}(\omega^L)$. Let $M \in \mathbb{I}_{\beta\eta}(\omega^L)$ where $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ then $M \in \mathcal{M}_3^L$. For each $i \in \{1, \ldots, n\}$, take U_i such that $x_i^{L_i} \in \mathsf{DVar}_{U_i}$. Then $\Gamma = (x_i^{L_i} : U_i)_n \subseteq \mathsf{BPreEnv}^L$. By Lemma 7.3.7.2 and Lemma 8.3.4, $M : \langle \Gamma \vdash_3 \omega^L\rangle$. Hence $M : \langle \mathsf{BPreEnv}^L \vdash_3 \omega^L\rangle$. Therefore, $\mathbb{I}_{\beta\eta}(\omega^L) \subseteq \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* \omega^L\rangle\}$. We deduce $\mathbb{I}_{\beta\eta}(\omega^L) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* \omega^L\rangle\}$.
   - $U = \mathsf{e}_i V$: $L = i :: K$ and $\mathsf{deg}(V) = K$. By IH and Lemma 8.3.6, $\mathbb{I}_{\beta\eta}(\mathsf{e}_i V) = (\mathbb{I}_{\beta\eta}(V))^{+i} = (\mathsf{OPEN}^K \cup \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\})^{+i} = \mathsf{OPEN}^L \cup (\{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\})^{+i}$.
     - If $M \in \mathcal{M}_3^K$ and $M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle$ then $M \mathrel{-\!\!\rhd}^*_{\beta\eta} N$ and $N : \langle \Gamma \vdash_3 V\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^K$. By rule (exp), Lemmas B.1.5.6 and 8.3.4.2, $N^{+i} : \langle \mathsf{e}_i\Gamma \vdash_3 \mathsf{e}_i V\rangle$, $M^{+i} \mathrel{-\!\!\rhd}^*_{\beta\eta} N^{+}i$ and $\mathsf{e}_i\Gamma \subseteq \mathsf{BPreEnv}^L$. Thus $M^{+i} \in \mathcal{M}_3^L$ and $M^{+i} : \langle \mathsf{BPreEnv}^L \vdash_3^* U\rangle$.

   - If $M \in \mathcal{M}_3^L$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U\rangle$, then $M \rightarrowtriangle^*_{\beta\eta} N$ and $N : \langle \Gamma \vdash_3 U\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^L$. By Lemmas B.1.5, 7.3.5, and 8.3.4.3, $M^{-i} \rightarrowtriangle^*_{\beta\eta} N^{-i}$, $N^{-i} : \langle \Gamma^{-i} \vdash_3 V\rangle$, and $\Gamma^{-i} \subseteq \mathsf{BPreEnv}^K$, and $M = (M^{-i})^{+i}$. Therefore $M^{-i} \in \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\}$.

  Finally, $(\{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\})^{+i} = \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U\rangle\}$ and $\mathbb{I}_{\beta\eta}(U) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U\rangle\}$.

- $U = U_1 \sqcap U_2$: By IH, $\mathbb{I}_{\beta\eta}(U_1 \sqcap U_2) = \mathbb{I}_{\beta\eta}(U_1) \cap \mathbb{I}_{\beta\eta}(U_2) = (\mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1\rangle\}) \cap (\mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_2\rangle\}) = \mathsf{OPEN}^L \cup (\{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1\rangle\} \cap \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_2\rangle\})$.

   - If $M \in \mathcal{M}_3^L$, $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1\rangle$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_2\rangle$ then $M \rightarrowtriangle^*_{\beta\eta} N_1$, $M \rightarrowtriangle^*_{\beta\eta} N_2$, $N_1 : \langle \Gamma_1 \vdash_3 U_1\rangle$ and $N_2 : \langle \Gamma_2 \vdash_3 U_2\rangle$ where $\Gamma_1, \Gamma_2 \subseteq \mathsf{BPreEnv}^L$. By confluence Theorem 7.1.13 and subject reduction Theorem 7.4.10, $\exists$ M' such that $N_1 \rightarrowtriangle^*_{\beta\eta} M'$ and $N_2 \rightarrowtriangle^*_{\beta\eta} M'$, $M' : \langle \Gamma_1 \restriction_{M'} \vdash_3 U_1\rangle$ and $M' : \langle \Gamma_2 \restriction_{M'} \vdash_3 U_2\rangle$. Hence by Remark 7.3.6, Lemma 7.1.11, Theorem 7.3.5.2a, and Lemma B.1.19.2, $M' : \langle (\Gamma_1 \sqcap \Gamma_2)\restriction_{M'} \vdash_3 U_1 \sqcap U_2\rangle$ and, by Lemma 8.3.4.4, $(\Gamma_1 \sqcap \Gamma_2)\restriction_{M'} \subseteq \Gamma_1 \sqcap \Gamma_2 \subseteq \mathsf{BPreEnv}^L$. Thus, $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1 \sqcap U_2\rangle$.
   - If $M \in \mathcal{M}_3^L$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1 \sqcap U_2\rangle$ then $M \rightarrowtriangle^*_{\beta\eta} N$, $N : \langle \Gamma \vdash_3 U_1 \sqcap U_2\rangle$ and $\Gamma \subseteq \mathsf{BPreEnv}^L$. By rule $(\sqsubseteq)$, $N : \langle \Gamma \vdash_3 U_1\rangle$ and $N : \langle \Gamma \vdash_3 U_2\rangle$. Hence, $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1\rangle$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_2\rangle$.

  We deduce that $\mathbb{I}_{\beta\eta}(U_1 \sqcap T_2) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3^* U_1 \sqcap U_2\rangle\}$.

- $U = V \to T$: Let $\mathsf{deg}(T) = \oslash \preceq K = \mathsf{deg}(V)$. By IH, $\mathbb{I}_{\beta\eta}(V) = \mathsf{OPEN}^K \cup \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\}$ and $\mathbb{I}_{\beta\eta}(T) = \mathsf{OPEN}^\oslash \cup \{M \in \mathcal{M}_3^\oslash \mid M : \langle \mathsf{BPreEnv}^\oslash \vdash_3^* T\rangle\}$. By definition, $\mathbb{I}_{\beta\eta}(V \to T) = \mathbb{I}_{\beta\eta}(V) \leadsto \mathbb{I}_{\beta\eta}(T)$.

   - Let $M \in \mathbb{I}_{\beta\eta}(V) \leadsto \mathbb{I}_{\beta\eta}(T)$ and, by Lemma 8.3.2, let $y^K \in \mathsf{DVar}_V$ such that $\forall K.\ y^K \notin \mathsf{fv}(M)$. Then $M \diamond y^K$. By remark 7.3.6.3, $y^K : \langle (y^K : V) \vdash_3^* V\rangle$. Hence $y^K : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle$. Thus, $y^K \in \mathbb{I}_{\beta\eta}(V)$ and $My^K \in \mathbb{I}_{\beta\eta}(T)$.
      - $*$ If $My^K \in \mathsf{OPEN}^\oslash$ then since $y \in \mathsf{Var}_2$, by Lemma 8.3.6, $M \in \mathsf{OPEN}^\oslash$.
      - $*$ If $My^K \in \{M \in \mathcal{M}_3^\oslash \mid M : \langle \mathsf{BPreEnv}^\oslash \vdash_3^* T\rangle\}$ then $My^K \rightarrowtriangle^*_{\beta\eta} N$ and $N : \langle \Gamma \vdash_3 T\rangle$ such that $\Gamma \subseteq \mathsf{BPreEnv}^\oslash$, hence, $\lambda y^K.My^K \rightarrowtriangle_{\beta\eta} \lambda y^K.N$. We have two cases:
         - $\cdot$ If $y^K \in \mathsf{dom}(\Gamma)$ then $\Gamma = \Delta, (y^K : V)$ and by rule $(\to_{\mathsf{I}})$, $\lambda y^K.N : \langle \Delta \vdash_3 V \to T\rangle$.

- If $y^K \notin \mathsf{dom}(\Gamma)$, let $\Delta = \Gamma$. By rule $(\to'_{\mathsf{I}})$, $\lambda y^K.N : \langle \Delta \vdash_3 \omega^K \to T\rangle$. By rule $(\sqsubseteq)$, since $(\Delta \vdash_3 \omega^K \to T) \sqsubseteq (\Delta \vdash_3 V \to T)$ using Remark 7.3.6.4, we have $\lambda y^K.N : \langle \Delta \vdash_3 V \to T\rangle$.

Note that $\Delta \subseteq \mathsf{BPreEnv}^{\oslash}$. Because $\lambda y^K.My^K \twoheadrightarrow_{\beta\eta} M$ and $\lambda y^K.My^K \twoheadrightarrow_{\beta\eta} \lambda y^K.N$, by confluence Theorem 7.1.13 and subject reduction Theorem 7.4.10, there is M' such that $M \twoheadrightarrow_{\beta\eta} M'$, $\lambda y^K.N \twoheadrightarrow_{\beta\eta} M'$, $M' : \langle \Delta\upharpoonright_{M'} \vdash_3 V \to T\rangle$. Since $\Delta\upharpoonright_{M'} \subseteq \Delta \subseteq \mathsf{BPreEnv}^{\oslash}$, $M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* V \to T\rangle$.

- Let $M \in \mathsf{OPEN}^{\oslash} \cup \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* V \to T\rangle\}$ and $N \in \mathbb{I}_{\beta\eta}(V) = \mathsf{OPEN}^K \cup \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\}$ such that $M \diamond N$. Then, $\mathsf{deg}(N) = K \succeq \oslash = \mathsf{deg}(M)$.
  - If $M \in \mathsf{OPEN}^{\oslash}$ then, by Lemma 8.3.6.3, $MN \in \mathsf{OPEN}^{\oslash}$.
  - If $M \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* V \to T\rangle\}$ then:
    - If $N \in \mathsf{OPEN}^K$ then, by Lemma 8.3.6.3, $MN \in \mathsf{OPEN}^{\oslash}$.
    - If $N \in \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3^* V\rangle\}$ then $M \twoheadrightarrow_{\beta\eta}^* M_1$, $N \twoheadrightarrow_{\beta\eta}^* N_1$, $M_1 : \langle \Gamma_1 \vdash_3 V \to T\rangle$ and $N_1 : \langle \Gamma_2 \vdash_3 V\rangle$ where $\Gamma_1 \subseteq \mathsf{BPreEnv}^{\oslash}$ and $\Gamma_2 \subseteq \mathsf{BPreEnv}^K$. By Lemma B.1.2.1 and Theorem 7.1.11.2 $\mathsf{deg}(M) = \mathsf{deg}(M_1)$, $\mathsf{deg}(N) = \mathsf{deg}(N_1)$, and $M_1 \diamond N_2$. Therefore, $MN \twoheadrightarrow_{\beta\eta}^* M_1N_1$. By rule $(\to_{\mathsf{E}})$ and Lemma 7.3.7.3, $M_1N_1 : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T\rangle$. By Lemma 8.3.4.4, $\Gamma_1 \sqcap \Gamma_2 \subset \mathsf{BPreEnv}^{\oslash}$. Therefore $MN : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* T\rangle$.

We deduce that $\mathbb{I}_{\beta\eta}(V \to T) = \mathsf{OPEN}^{\oslash} \cup \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3^* V \to T\rangle\}$.

3. We only do the case $r = \beta$. By induction on U.

- $U = a$: By definition of $\mathbb{I}_\beta$.
- $U = \omega^L$: By definition, $\mathbb{I}_\beta(\omega^L) = \mathcal{M}_3^L$. Hence, $\mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 \omega^L\rangle\} \subseteq \mathbb{I}_\beta(\omega^L)$. Let $M \in \mathbb{I}_\beta(\omega^L)$ where $\mathsf{fv}(M) = \{x_1^{L_1}, \ldots, x_n^{L_n}\}$ then $M \in \mathcal{M}_3^L$. For each $i \in \{1, \ldots, n\}$, we take U_i to be the type such that $x_i^{L_i} \in \mathsf{DVar}_{U_i}$. Then $\Gamma = (x_i^{L_i} : U_i)_n \subseteq \mathsf{BPreEnv}^L$. By Lemma 7.3.7.2 and Lemma 8.3.4.1, $M : \langle \Gamma \vdash_3 \omega^L\rangle$. Hence $M : \langle \mathsf{BPreEnv}^L \vdash_3 \omega^L\rangle$. Therefore, $\mathbb{I}_\beta(\omega^L) \subseteq \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 \omega^L\rangle\}$. Finally, $\mathbb{I}_\beta(\omega^L) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 \omega^L\rangle\}$.
- $U = \mathsf{e}_i V$: $L = i :: K$ and $\mathsf{deg}(V) = K$. By IH and Lemma 8.3.6.1, $\mathbb{I}_\beta(\mathsf{e}_i V) = (\mathbb{I}_\beta(V))^{+i} = (\mathsf{OPEN}^K \cup \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle\})^{+i} = \mathsf{OPEN}^L \cup (\{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle\})^{+i}$.
  - If $M \in \mathcal{M}_3^K$ and $M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle$ then $M : \langle \Gamma \vdash_3 V\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^K$. By rule (exp) and Lemma 8.3.4.2, $M^{+i} : \langle \mathsf{e}_i\Gamma \vdash_3 \mathsf{e}_i V\rangle$ and $\mathsf{e}_i\Gamma \subseteq \mathsf{BPreEnv}^L$. Thus $M^{+i} \in \mathcal{M}_3^L$ and $M^{+i} : \langle \mathsf{BPreEnv}^L \vdash_3 U\rangle$.

  - If $M \in \mathcal{M}_3^L$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3 U\rangle$, then $M : \langle \Gamma \vdash_3 U\rangle$ where $\Gamma \subseteq \mathsf{BPreEnv}^L$. By Lemmas 7.3.5, and 8.3.4.3, $M^{-i} : \langle \Gamma^{-i} \vdash_3 V\rangle$ and $\Gamma^{-i} \subseteq \mathsf{BPreEnv}^K$. Thus by Lemma B.1.5, $M = (M^{-i})^{+i}$ and $M^{-i} \in \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle\}$.

  Finally, $(\{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle\})^{+i} = \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U\rangle\}$ and $\mathbb{I}_\beta(U) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U\rangle\}$.

- $U = U_1 \sqcap U_2$: By IH, $\mathbb{I}_\beta(U_1 \sqcap U_2) = \mathbb{I}_\beta(U_1) \cap \mathbb{I}_\beta(U_2) = (\mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1\rangle\}) \cap (\mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U_2\rangle\}) = \mathsf{OPEN}^L \cup (\{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1\rangle\} \cap \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U_2\rangle\})$.

  - If $M \in \mathcal{M}_3^L$, $M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1\rangle$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3 U_2\rangle$ then $M : \langle \Gamma_1 \vdash_3 U_1\rangle$ and $M : \langle \Gamma_2 \vdash_3 U_2\rangle$ where $\Gamma_1, \Gamma_2 \subseteq \mathsf{BPreEnv}^L$. Hence by Remark 7.3.6.1, $M : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 U_1 \sqcap U_2\rangle$ and, by Lemma 8.3.4.4, $\Gamma_1 \sqcap \Gamma_2 \subset \mathsf{BPreEnv}^L$. Thus $M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1 \sqcap U_2\rangle$.
  - If $M \in \mathcal{M}_3^L$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1 \sqcap U_2\rangle$ then $M : \langle \Gamma \vdash_3 U_1 \sqcap U_2\rangle$ and $\Gamma \subseteq \mathsf{BPreEnv}^L$. By rule ($\sqsubseteq$), $M : \langle \Gamma \vdash_3 U_1\rangle$ and $M : \langle \Gamma \vdash_3 U_2\rangle$. Hence, $M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1\rangle$ and $M : \langle \mathsf{BPreEnv}^L \vdash_3 U_2\rangle$.

  We deduce that $\mathbb{I}_\beta(U_1 \sqcap T_2) = \mathsf{OPEN}^L \cup \{M \in \mathcal{M}_3^L \mid M : \langle \mathsf{BPreEnv}^L \vdash_3 U_1 \sqcap U_2\rangle\}$.

- $U = V \to T$: Let $\mathsf{deg}(T) = \oslash \preceq K = \mathsf{deg}(V)$. By IH, $\mathbb{I}_\beta(V) = \mathsf{OPEN}^K \cup \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle\}$ and $\mathbb{I}_\beta(T) = \mathsf{OPEN}^\oslash \cup \{M \in \mathcal{M}_3^\oslash \mid M : \langle \mathsf{BPreEnv}^\oslash \vdash_3 T\rangle\}$. Note that $\mathbb{I}_\beta(V \to T) = \mathbb{I}_\beta(V) \rightsquigarrow \mathbb{I}_\beta(T)$.

  - Let $M \in \mathbb{I}_\beta(V) \rightsquigarrow \mathbb{I}_\beta(T)$ and, by Lemma 8.3.2, let $y^K \in \mathsf{DVar}_V$ such that $\forall K.\ y^K \notin \mathsf{fv}(M)$. Then $M \diamond y^K$. By remark 7.3.6.3, $y^K : \langle (y^K : V) \vdash_3^* V\rangle$. Hence $y^K : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle$. Thus, $y^K \in \mathbb{I}_\beta(V)$ and $My^K \in \mathbb{I}_\beta(T)$.
    - If $My^K \in \mathsf{OPEN}^\oslash$ then since $y \in \mathsf{Var}_2$, by Lemma 8.3.6.2, $M \in \mathsf{OPEN}^\oslash$.
    - If $My^K \in \{M \in \mathcal{M}_3^\oslash \mid M : \langle \mathsf{BPreEnv}^\oslash \vdash_3 T\rangle\}$ then $My^K : \langle \Gamma \vdash_3 T\rangle$ such that $\Gamma \subseteq \mathsf{BPreEnv}^\oslash$. By Theorem 7.3.5.2a, $\mathsf{dom}(\Gamma) = \mathsf{fv}(My^K)$ and $y^K \in \mathsf{fv}(My^K)$, $\Gamma = \Delta, (y^K : V')$. Since $(y^K : V') \in \mathsf{BPreEnv}^\oslash$, by Lemma 8.3.2.3, $V = V'$. So $My^K : \langle \Delta, (y^K : V) \vdash_3 T\rangle$ and by Lemma B.1.14.1, $M : \langle \Delta \vdash_3 V \to T\rangle$. Note that $\Delta \subseteq \mathsf{BPreEnv}^\oslash$, hence $M : \langle \mathsf{BPreEnv}^\oslash \vdash_3 V \to T\rangle$.
  - Let $M \in \mathsf{OPEN}^\oslash \cup \{M \in \mathcal{M}_3^\oslash \mid M : \langle \mathsf{BPreEnv}^\oslash \vdash_3 V \to T\rangle\}$ and $N \in \mathbb{I}_\beta(V) = \mathsf{OPEN}^K \cup \{M \in \mathcal{M}_3^K \mid M : \langle \mathsf{BPreEnv}^K \vdash_3 V\rangle\}$ such that $M \diamond N$. Then, $\mathsf{deg}(N) = K \succeq \oslash = \mathsf{deg}(M)$.

  - $*$ If $M \in \mathsf{OPEN}^{\oslash}$ then, by Lemma 8.3.6.3, $MN \in \mathsf{OPEN}^{\oslash}$.
  - $*$ If $M \in \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 V \to T \rangle\}$ then
    - $\cdot$ If $N \in \mathsf{OPEN}^{K}$ then, by Lemma 8.3.6.4, $MN \in \mathsf{OPEN}^{\oslash}$.
    - $\cdot$ If $N \in \{M \in \mathcal{M}_3^{K} \mid M : \langle \mathsf{BPreEnv}^{K} \vdash_3 V \rangle\}$ then $M : \langle \Gamma_1 \vdash_3 V \to T \rangle$ and $N : \langle \Gamma_2 \vdash_3 V \rangle$ where $\Gamma_1 \subseteq \mathsf{BPreEnv}^{\oslash}$ and $\Gamma_2 \subseteq \mathsf{BPreEnv}^{K}$. By rule $(\to_{\mathsf{E}})$ and Lemma 7.3.7.3, $MN : \langle \Gamma_1 \sqcap \Gamma_2 \vdash_3 T \rangle$. By Lemma 8.3.4.4, $\Gamma_1 \sqcap \Gamma_2 \subseteq \mathsf{BPreEnv}^{\oslash}$. Therefore $MN : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 T \rangle$.

We deduce that $\mathbb{I}_{\beta}(V \to T) = \mathsf{OPEN}^{\oslash} \cup \{M \in \mathcal{M}_3^{\oslash} \mid M : \langle \mathsf{BPreEnv}^{\oslash} \vdash_3 V \to T \rangle\}$.

□

B.3 Embedding of a system close to **CDV** in our type system $\vdash_3$

Let us now present a sketched proof of the embedding of a restricted version [27, 28], which we call RCDV, of the well known intersection type system CDV, both introduced by Coppo, Dezani, and Venneri [28] and recalled by Van Bakel [4], in our type system $\vdash_3$.

Let us provide an alternative presentation of RCDV's normalised types:

$$
\begin{array}{lll}
\varphi \in \mathsf{RCDVTyVar} & & \text{(a countably infinite set of type variables)} \\
\phi \in \mathsf{RCDVTy} & ::= & \varphi \mid \sigma \to \phi \\
\sigma \in \mathsf{RCDVITy} & ::= & \omega \mid \phi_1 \cap \cdots \cap \phi_n, \text{ where } n \geq 1
\end{array}
$$

Even though we provide an alternative presentation of RCDV we shall prefix entities and rules names of this system with "RCDV" in this section.

Let the form $\cap_n \sigma_i$ be a notation for $\phi_1 \cap \cdots \cap \phi_n$. A basis (set of type assignments) is written B ($\in \mathsf{RCDVBasis}$) and $\cap_n B_i$ is similar to our intersection of type environments (without indexes).

Let us now recall their type system (the original version of RCDV is presented in a natural deduction fashion):

$$\frac{}{x : \phi \vdash x : \phi}\ (\mathsf{RCDV\text{-}Ax}) \qquad \frac{B_1 \vdash M : \phi_1 \quad \cdots \quad B_n \vdash M : \phi_n}{\cap_n B_i \vdash M : \cap_n \phi_i}\ (\mathsf{RCDV\text{-}\cap I})$$

$$\frac{}{\vdash M : \omega}\ (\mathsf{RCDV\text{-}}\omega) \qquad \frac{B_1 \vdash M : \sigma \to \phi \quad B_2 \vdash N : \sigma}{B_1 \cap B_2 \vdash MN : \phi}\ (\mathsf{RCDV\text{-}\to E})$$

$$\frac{B, x : \sigma \vdash M : \phi}{B \vdash \lambda x.M : \sigma \to \phi}\ (\mathsf{RCDV\text{-}\to I}) \qquad \frac{B \vdash M : \phi \quad x \text{ does not occur in } B}{B \vdash \lambda x.M : \omega \to \phi}\ (\mathsf{RCDV\text{-}a})$$

Coppo, Dezani and Venneri [28] allow the ω type to be a normalised type in their RCDV system. They then consider many restrictions on normalised types and in their typing rules to disallow the use of ω at many places, which is why we chose to consider an alternative presentation of their system.

Let us now define an erasure function on our types and type environments. Informally, this erasure remove all the indexes and expansion variables from our different syntactic objects. Let us assume that there exists a bijective function $\mathsf{bijtyvar}$ from TyVar to $\mathsf{RCDVTyVar}$. The erasure on types is as follows: $\mathsf{er}(a) = \mathsf{bijtyvar}(a)$, $\mathsf{er}(U{\rightarrow}T) = \mathsf{er}(U){\rightarrow}\mathsf{er}(T)$, $\mathsf{er}(U_1 \sqcap U_2) = \mathsf{er}(U_1) \cap \mathsf{er}(U_2)$, $\mathsf{er}(\omega^L) = \omega$ and $\mathsf{er}(\mathsf{e}_i U) = \mathsf{er}(U)$. One can check that the erasure of a type in ITy_3 is in RCDVTy and that the erasure of a type in Ty_3 is in $\mathsf{RCDVITy}$. We trivially extend the erasure function to type environments.

Let us define a decoration function to decorate λ-terms. Let $\mathsf{dec}(x) = x^{\oslash}$, $\mathsf{dec}(\lambda x.M) = \lambda x^{\oslash}.\mathsf{dec}(M)$ and $\mathsf{dec}(MN) = \mathsf{dec}(M)\mathsf{dec}(N)$. One can check (by induction on the structure of M) that the decoration of an undecorated λ-term M (such that each variable is decorated with the index $\oslash$) is in $\mathcal{M}_3^{\oslash}$. In our simple embedding the untyped λ-calculus is embedded in $\mathcal{M}_3^{\oslash}$ which is the range of our decoration function.

Let us prove that if $\phi \in \mathsf{RCDVTy}$ is a normalised type then there exists $T \in \mathsf{Ty}_3$ such that $\mathsf{er}(T) = \phi$, if $\sigma \in \mathsf{RCDVITy}$ is a normalised intersection type then there exists $U \in \mathsf{ITy}_3$ such that $\mathsf{er}(U) = \sigma$, if $B \in \mathsf{RCDVBasis}$ then there exists a type environment Γ such that $\mathsf{er}(\Gamma) = B$, and if $B \vdash M : \sigma$ then there exists Γ and U such that $\mathsf{er}(\Gamma) = B$, $\mathsf{er}(U) = \sigma$, and $\mathsf{dec}(M) : \langle \Gamma{\uparrow}^{\mathsf{dec}(M)} \vdash_3 U\rangle$.

Let $\phi \in \mathsf{RCDVTy}$ be a normalised type and $\sigma \in \mathsf{RCDVITy}$ be a normalised intersection type. We now provide a sketch of the proof (by induction on the structures of ϕ and σ) that there exists $T \in \mathsf{Ty}_3$ such that $\mathsf{er}(T) = \phi$ and that there exists $U \in \mathsf{ITy}_3$ such that $\mathsf{er}(U) = \sigma$: let $\phi = \varphi$ then there exists $a \in \mathsf{TyVar}$ such that $\mathsf{bijtyvar}(a) = \varphi$ and $\mathsf{er}(a) = \mathsf{bijtyvar}(a) = \varphi$; let $\phi = \sigma{\rightarrow}\phi'$ then σ is a normalised intersection type and ϕ' is a normalised type, by induction hypothesis there exists $T \in \mathsf{Ty}_3$ such that $\mathsf{er}(T) = \phi'$ and $U \in \mathsf{ITy}_3$ such that $\mathsf{er}(U) = \sigma$, so $\mathsf{er}(U{\rightarrow}T) = \phi$; let $\sigma = \cap_n \phi_i$ then for all i, ϕ_i is a normalised type, by induction hypothesis, for all i, there exists $T_i \in \mathsf{Ty}_3$ such that $\mathsf{er}(T_i) = \phi_i$, so, $\mathsf{er}(T_1 \sqcap \cdots \sqcap T_n) = \sigma$; let $\sigma = \omega$ then take $U = \omega^{\oslash}$ for example.

Let us provide a sketch of the proof that if $B \vdash M : \sigma$ then there exists Γ and U such that $\mathsf{er}(\Gamma) = B$, $\mathsf{er}(U) = \sigma$ and $\mathsf{dec}(M) : \langle \Gamma{\uparrow}^{\mathsf{dec}(M)} \vdash_3 U\rangle$.

- (RCDV-Ax): let $x : \phi \vdash x : \phi$. We proved that there exists $T \in \mathsf{Ty}_3$ such that $\mathsf{er}(T) = \phi$ and $x^{\oslash} : \langle (x^{\oslash} : T) \vdash_3 T\rangle$ by rule (ax).

- (RCDV-ω): let $\vdash M : \omega$ then using rule (ω), $\mathsf{dec}(M) : \langle \mathsf{env}^{\varnothing}_{\mathsf{dec}(M)} \vdash_3 \omega^{\oslash}\rangle$.

- (RCDV-${\rightarrow}\mathsf{I}$): let $B \vdash \lambda x.M : \sigma{\rightarrow}\phi$ such that $B, x : \sigma \vdash M : \phi$. By induction hypothesis, there exists Γ' and T such that $\mathsf{er}(\Gamma') = (B, x : \sigma)$, $\mathsf{er}(T) = \phi$ and $\mathsf{dec}(M) : \langle \Gamma'{\uparrow}^{\mathsf{dec}(M)} \vdash_3 T \rangle$. Because $x \in \mathsf{fv}(M)$ then we can prove that $x^{\oslash} \in \mathsf{fv}(\mathsf{dec}(M))$ and $\Gamma'{\uparrow}^{\mathsf{dec}(M)} = \Gamma{\uparrow}^{\mathsf{dec}(\lambda x.M)}, (x^{\oslash} : U)$ such that $\mathsf{er}(U) = \sigma$. By rule $(\rightarrow_{\mathsf{I}})$, $\lambda x^{\oslash}.\mathsf{dec}(M) : \langle \Gamma{\uparrow}^{\mathsf{dec}(\lambda x.M)} \vdash_3 U \rightarrow T \rangle$.

- (RCDV-a): let $B \vdash \lambda x.M : \omega \rightarrow \phi$ such that $B \vdash M : \phi$ and where x does not occur in B. By induction hypothesis, there exists Γ and T such that $\mathsf{er}(\Gamma) = B$, $\mathsf{er}(T) = \phi$ and $\mathsf{dec}(M) : \langle \Gamma{\uparrow}^{\mathsf{dec}(M)} \vdash_3 T \rangle$. Because x does not occur in B then $x \notin \mathsf{fv}(M)$ and by rule $(\rightarrow_{\mathsf{I}}')$, $\lambda x^{\oslash}.\mathsf{dec}(M) : \langle \Gamma{\uparrow}^{\mathsf{dec}(\lambda x.M)} \vdash_3 \omega^{\oslash} \rightarrow T \rangle$.

- (RCDV-$\rightarrow$E): let $B_1 \cap B_2 \vdash MN : \phi$ such that $B_1 \vdash M : \sigma \rightarrow \phi$ and $B_2 \vdash N : \sigma$. By induction hypothesis we can prove that there exit Γ_1, Γ_2, U and T such that $\mathsf{er}(\Gamma_1) = B_1$, $\mathsf{er}(\Gamma_2) = B_2$, $\mathsf{er}(U) = \sigma$, $\mathsf{er}(T) = \phi$, $\mathsf{dec}(M) : \langle \Gamma_1{\uparrow}^{\mathsf{dec}(M)} \vdash_3 U \rightarrow T \rangle$ and $\mathsf{dec}(N) : \langle \Gamma_2{\uparrow}^{\mathsf{dec}(N)} \vdash_3 U \rangle$. Because $\Gamma_1{\uparrow}^{\mathsf{dec}(M)}$ and $\Gamma_2{\uparrow}^{\mathsf{dec}(N)}$ are compatible then by rule $(\rightarrow_{\mathsf{E}})$, $MN : \langle \Gamma_1{\uparrow}^{\mathsf{dec}(M)} \sqcap \Gamma_2{\uparrow}^{\mathsf{dec}(N)} \vdash_3 T \rangle$ and we can prove that $\Gamma_1{\uparrow}^{\mathsf{dec}(M)} \sqcap \Gamma_2{\uparrow}^{\mathsf{dec}(N)} = (\Gamma_1 \sqcap \Gamma_2){\uparrow}^{\mathsf{dec}(MN)}$ and that $\mathsf{er}(\Gamma_1 \sqcap \Gamma_2) = \sqcap\{B_1, B_2\}$.

- (RCDV-$\cap$I): let $\cap_n B_i \vdash M : \cap_n \phi_i$ such that $B_i \vdash M : \phi_i$, for all i. Then we can conclude using Remark 7.3.6.

The type system introduced at the beginning of this section can then be embedded into our type system without making use of expansion variables and restraining the space of meaning $\mathcal{M}_3$ to the basis $\mathcal{M}_3^{\oslash}$.

Unfortunately, as mentioned in Ch. 9, we do not believe that it would be possible to embed RCDV in our system such that we would make use of the expansion variables "as much as possible".

Bibliography

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, 1999.

[2] Andrew W. Appel. A critique of Standard ML. *J. Funct. Program.*, 3(4):391–429, 1993.

[3] Andrea Asperti and Enrico Tassi. Modified realizability and inductive types. Technical report, Department of Computer Science, University of Bologna, 2006.

[4] Steffen Van Bakel. Strict intersection types for the lambda calculus; a survey. Located at `http://www.doc.ic.ac.uk/~svb/Research/`, 2011.

[5] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[6] Henk P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. Oxford University Press, Inc., New York, NY, USA, 1992.

[7] Henk P. Barendregt, Jan A. Bergstra, Jan W. Klop, and Henri Volken. Degrees, reductions and representability in the lambda calculus. Technical Report Preprint no. 22, University of Utrecht, Department of Mathematics, 1976.

[8] Henk P. Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completness of type assignment. *The Journal of Symbolic Logic*, 48(4), 1983.

[9] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4):17–30, 1993.

[10] Marc Bezem, Jan Willem Klop, Roel de Vrijer, Erik Barendsen, Inge Bethke, Jan Heering, Richard Kennaway, Paul Klint, Vincent van Oostrom, Femke van

Raamsdonk, Fer-Jan de Vries, and Hans Zantema. *Term Rewriting Systems.*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, March 2003.

[11] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, November 1997.

[12] Matthias Blume. Dependency analysis for Standard ML. *ACM Trans. Program. Lang. Syst.*, 21(4):790–812, 1999.

[13] Corrado Böhm, editor. *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975*, volume 37 of *Lecture Notes in Computer Science*. Springer, 1975.

[14] Nabil El Boustani and Jurriaan Hage. Improving type error messages for Generic Java. In Germán Puebla and Germán Vidal, editors, *PEPM*, pages 131–140. ACM, 2009.

[15] Nabil El Boustani and Jurriaan Hage. Corrective hints for type incorrect Generic Java programs. In John P. Gallagher and Janis Voigtländer, editors, *PEPM*, pages 5–14. ACM, 2010.

[16] Bernd Braßel. TypeHope - there is hope for your type errors. In Grelck et al. [54], pages 185–198.

[17] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[18] Felice Cardone and J. Roger Hindley. History of lambda-calculus and combinatory logic. Technical report, Swansea University Mathematics Department, 2006.

[19] Sébastien Carlier, Jeff Polakow, J. B. Wells, and Assaf J. Kfoury. System E: Expansion variables for flexible typing with linear and non-linear types and intersection types. In David A. Schmidt, editor, *ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2004.

[20] Sébastien Carlier and J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. *Electr. Notes Theor. Comput. Sci.*, 136:173–202, 2005.

[21] Alonzo Church. A set of postulates for the foundations of logic. *The Annals of Mathematics*, 33(2):346–366, April 1932.

[22] Alonzo Church. A proof of freedom from contradiction. *Proceedings of the National Academy of Sciences of the United States of America*, 21(5):275–281, May 1935.

[23] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[24] Alonzo Church and John B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[25] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: mini-ML. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 13–27, New York, NY, USA, 1986. ACM.

[26] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4), 1979.

[27] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ-calculus semantic. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560. J.R. Hindley and J.P. Seldin, 1980.

[28] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Functional characters of solvable terms. *Mathematische Logik Und Grundlagen der Mathematik*, 27:45–58, 1981.

[29] Thierry Coquand. Completeness theorems and lambda-calculus. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2005.

[30] Haskell B. Curry. Functionality in combinatory logic. *Proc. Nat. Acad, Sci. USA*, 20:584–590, 1934.

[31] Haskell B. Curry and Robert Feys. *Combinatory Logic I.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1958.

[32] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL82*, pages 207–212, New York, NY, USA, 1982. ACM.

[33] Luis M. M. Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1984.

[34] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 5(34):381–392, January 1972.

[35] Mariangiola Dezani-Ciancaglini, Furio Honsell, and Fabio Alessi. A complete characterization of complete intersection-type preorders. *ACM Trans. Comput. Log.*, 4(1):120–147, 2003.

[36] Dominic Duggan. Correct type explanation. In *In ACM SIGPLAN Workshop on ML*, pages 49–58, 1998.

[37] Dominic Duggan and Frederick Bent. Explaining type inference. *Sci. Comput. Program.*, 27(1):37–83, 1996.

[38] Michael A. E. Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.

[39] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1995.

[40] Samir Farkh and Karim Nour. Résultats de complétude pour des classes de types du système AF2. *Theoretical Informatics and Applications*, 31(6):513–537, 1998.

[41] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 415–431, London, UK, 1994. Springer-Verlag.

[42] John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11-12):609–636, 1998.

[43] Jean H. Gallier. On Girard's "candidats de reductibilité". In P. Odifreddi, editor, *Logic and Computer Science*, pages 123–203. Academic Press, 1990.

[44] Jean H. Gallier. On the correspondence between proofs and λ-terms. *Cahiers du centre de logique*, 8:55–138, 1995.

[45] Jean H. Gallier. Proving properties of typed λ-terms using realisability, covers, and sheaves. *Theoretical Computer Science*, 142(2):299–368, 1995.

[46] Jean H. Gallier. Typing untyped λ-terms, or realisability strikes again!. *Annals of Pure and Applied Logic*, 91:231–270, 1998.

[47] Holger Gast. Explaining ML type errors by data flows. In Grelck et al. [54], pages 72–89.

[48] Silvia Ghilezan and Viktor Kunčak. Confluence of untyped lambda calculus via simple types. *Lecture Notes in Computer Science*, 2202:38–49, 2001.

[49] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92, 1971.

[50] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Universite de Paris VII, 1972.

[51] Kurt Gödel. On undecidable propositions of formal mathematical systems. Lecture notes taken by S. C. Kleene and B. Rosser at the Institute for Advanced Study (reprinted in Davis, 1965, 39–74), 1934.

[52] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130, New York, NY, USA, 1978. ACM.

[53] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[54] Clemens Grelck, Frank Huch, Greg Michaelson, and Philip W. Trinder, editors. *16th Int'l Workshop, IFL 2004*, volume 3474 of *LNCS*. Springer, 2005.

[55] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In Olivier Danvy and Andrzej Filinski, editors, *PADO*, volume 2053 of *LNCS*, pages 63–83. Springer, 2001.

[56] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In Pierpaolo Degano, editor, *ESOP*, volume 2618 of *LNCS*, pages 284–301. Springer, 2003.

[57] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

[58] Jurriaan Hage and Bastiaan Heeren. Ordering type constraints: A structured approach. Technical report, Institute of information and computing sciences, utrecht university, 2005.

[59] Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *18th Int'l Symp., IFL 2006*, volume 4449 of *LNCS*, pages 199–216. Springer, 2007.

[60] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electron. Notes Theor. Comput. Sci.*, 236:163–183, 2009.

[61] Robert Harper. Programming in Standard ML, 2009. Working draft of August 20, 2009.

[62] Bastiaan Heeren and Jurriaan Hage. Type class directives. In Manuel V. Hermenegildo and Daniel Cabeza, editors, *7th Int'l Symp., PADL 2005*, volume 3350 of *LNCS*, pages 253–267. Springer, 2005.

[63] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Constraint based type inferencing in Helium. In M.-C. Silaghi and M. Zanker, editors, *Workshop Proceedings of Immediate Applications of Constraint Programming*, pages 59 – 80, Cork, September 2003.

[64] Bastiaan Heeren, Johan Jeuring, S. Doaitse Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. Technical report, Utrecht University, 2002.

[65] Bastiaan J. Heeren. *Top Quality Type Error Messages.* PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[66] Fritz Henglein. Type inference and semi-unification. In *LISP and functional programming*, pages 184–197, New York, NY, USA, 1988. ACM.

[67] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[68] J. Roger Hindley. Reductions of residuals are finite. *Transaction of the American Mathematical Society.*, 240:345–361, 1978.

[69] J. Roger Hindley. The simple semantics for Coppo-Dezani-Sallé types. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 1982.

[70] J. Roger Hindley. The completeness theorem for typing lambda-terms. *Theor. Comput. Sci.*, 22:1–17, 1983.

[71] J. Roger Hindley. Curry's types are complete with respect to F-semantics too. *Theoretical Computer Science*, 22:127–133, 1983.

[72] J. Roger Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[73] J. Roger Hindley and Giuseppe Longo. Lambda-calculus models and extensionality. *Zeit. Math. Logik*, 26:289–310, 1980.

[74] Pieter J. W. Hofstra. *Completions in realizability.* PhD thesis, University of Utrecht, 2003.

[75] Pieter J. W. Hofstra. All realizability is relative. *Mathematical Proceedings of the Cambridge Philosophical Society*, 141(2):239–264, 2006.

[76] W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. J.R. Hindley and J.P. Seldin, 1969.

[77] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27:1–164, May 1992.

[78] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. *SIGPLAN Lisp Pointers*, V(1):193–204, 1992.

[79] Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A modern Perspective on Type Theory. From its Origins until Today.*, volume 29. Applied Logic Series, 2004.

[80] Fairouz Kamareddine and Karim Nour. A completeness result for a realisability semantics for an intersection type system. *Annals of Pure and Applied Logic*, 146:180–198, 2007.

[81] Fairouz Kamareddine, Karim Nour, Vincent Rahli, and J. B. Wells. Challenges and solutions to realisability semantics for intersection types with expansion variables. Submitted to Fundamenta Informaticae, 2008.

[82] Fairouz Kamareddine, Karim Nour, Vincent Rahli, and J. B. Wells. A complete realisability semantics for intersection types and arbitrary expansion variables. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC*, volume 5160 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2008.

[83] Fairouz Kamareddine, Karim Nour, Vincent Rahli, and J. B. Wells. Developing realisability semantics for intersection types and expansion variables.

Presented to ITRS'08, 4th Workshop on Intersection Types and Related Systems, Turin, Italy, 25 March 2008, 2008.

[84] Fairouz Kamareddine and Vincent Rahli. Simplified reducibility proofs of Church-Rosser for β- and $\beta\eta$-reduction. *Electr. Notes Theor. Comput. Sci.*, 247:85–101, 2009.

[85] Fairouz Kamareddine, Vincent Rahli, and J. B. Wells. Reducibility proofs in the λ-calculus. Presented to ITRS'08, 4th Workshop on Intersection Types and Related Systems, Turin, Italy, 25 March 2008, 2008.

[86] Paris C. Kanellakis, Harry G. Mairson, and John C. Mitchell. Unification and ML type reconstruction. Technical report, Providence, RI, USA, 1990.

[87] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem (preliminary report). In *STOC*, pages 468–476. ACM, 1990.

[88] Assaf J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL*, pages 161–174, 1999.

[89] Stephen C. Kleene. On the interpretation of intuitionistic number theory. *The Journal of Symbolic Logic*, 10(4):109–124, 1945.

[90] Stephen C. Kleene. *Mathematical Logic.* John Wiley & Sons, 1967.

[91] Stephen C. Kleene and John B. Rosser. The inconsistency of certain foraml logics. *The Annals of Mathematics*, 36(3):630–636, 1935.

[92] Jan W. Klop. *Combinatory Reductions Systems.* PhD thesis, Mathematisch Centrum, Amsterdam, 1980.

[93] George Koletsos. Church-Rosser theorem for typed functional systems. *Journal of Symbolic Logic*, 50(3):782–790, 1985.

[94] George Koletsos and Yiorgos Stavrinos. Church-Rosser property and intersection types. *Australasian Journal of Logic*, 2007.

[95] Georg Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland Publishing, 1959.

[96] Jean-Louis Krivine. *Lambda-calcul, types et modèles.* Masson, 1990.

[97] R. Labib-Sami. Typer avec (ou sans) types auxilières.

[98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, jul 1998.

[99] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In Jeanne Ferrante and Kathryn S. McKinley, editors, *ACM SIGPLAN 2007 Conf. PLDI*. ACM, 2007.

[100] Xavier Leroy. An overview of types in compilation. In *In Lecture Notes in Computer Science*, pages 1–8. Springer-Verlag, 1998.

[101] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–302, New York, NY, USA, 1991. ACM.

[102] Jean-Jacques Lévy. An algebraic interpretation of the *lambda beta* K-calculus; and an application of a labelled *lambda*-calculus. *Theoretical Compututer Science*, 2(1):97–114, 1976.

[103] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[104] Bruce J. McAdam. On the unification of substitutions in type inference. In Kevin Hammond, Antony J. T. Davie, and Chris Clack, editors, *10th Int'l Workshop, IFL'98*, volume 1595 of *LNCS*, pages 137–152. Springer, 1999.

[105] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[106] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[107] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[108] Martin Müller. A constraint-based recast of ML-polymorphism (extended abstract). Technical report, 8th International Workshop on unification, 1994.

[109] Martin Müller. Notes on HM(X), 1998.

[110] Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard Robinet, editors, *Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1984.

[111] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In Colin Runciman and Olin Shivers, editors, *8th ACM SIGPLAN Int'l Conf., ICFP 2003*, pages 15–26. ACM, 2003.

[112] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[113] Jaap Van Oosten. Realizability: a historical essay. *Mathematical. Structures in Comp. Sci.*, 12(3):239–263, 2002.

[114] François Pottier. Simplifying subtyping constraints. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 122–133, New York, NY, USA, 1996. ACM.

[115] François Pottier. A modern eye on ML type inference: old techniques and recent developments. Lecture notes for the APPSEM Summer School, September 2005.

[116] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[117] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. J.R. Hindley and J.P. Seldin, 1980.

[118] Vincent Rahli, J. B. Wells, and Fairouz Kamareddine. A constraint system for a SML type error slicer. Technical Report HW-MACS-TR-0079, Heriot-Watt University, MACS, ULTRA group, 2010.

[119] Gene F. Rose. Propositional calculus and realizability. *Transactions of the American Mathematical Society*, 75(1):1–19, 1953.

[120] John B. Rosser. Highlights of the history of the lambda-calculus. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 216–225, New York, NY, USA, 1982. ACM Press.

[121] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.

[122] Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, 1988.

[123] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics.* Elsevier Science, 2006.

[124] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, 2000.

[125] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in Haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 72–83, New York, NY, USA, 2003. ACM.

[126] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 80–91, New York, NY, USA, 2004. ACM.

[127] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Type processing by constraint reasoning. In Naoki Kobayashi, editor, *4th Asian Symp., APLAS 2006*, volume 4279 of *LNCS*, pages 1–25. Springer, 2006.

[128] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical report, 1999.

[129] Martin Franz Sulzmann. *A general framework for Hindley/Milner type systems with constraints.* PhD thesis, Yale University, New Haven, CT, USA, 2000. Director - Paul Hudak.

[130] W. W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[131] Masako Takahashi. Parallel reductions in lambda-calculus. *Journal of Symbolic Computation*, 7(2):113–123, 1989.

[132] TES team. Type error slicing, project webpage, 2010. `http://www.macs.hw.ac.uk/ultra/compositional-analysis/type-error-slicing/slicing.cgi`.

[133] Frank Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, 10(1):5–55, 2001.

[134] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, 1990.

[135] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics.* Elsevier Science, 1988.

[136] Alan M. Turing. Computability and lambda-definability. *J. Symb. Log.*, 2(4):153–163, 1937.

[137] Pawel Urzyczyn. Type reconstruction in F_ω. *Mathematical Structures in Computer Science*, 7(4):329–358, 1997.

[138] Jean van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, Massachusetts, 1967.

[139] Mitchell Wand. Finding the source of type errors. In *13th ACM SIGACT-SIGPLAN Symp., POPL'86*, pages 38–43, New York, NY, USA, 1986. ACM.

[140] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[141] Jeremy Wazny. *Type inference and type error diagnosis for Hindley/Milner with extensions.* PhD thesis, University of Melbourne, Australia, 2006.

[142] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1-3):111–156, 1999.

[143] J. B. Wells. The essence of principal typings. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th Int'l Colloq., ICALP 2002*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

[144] Alfred N. Whitehead and Bertrand Russell. *Principia mathematica.* Cambridge University Press, 1910.

[145] David A. Wolfram. Intractable unifiability problems and backtracking. In Ehud Y. Shapiro, editor, *ICLP*, volume 225 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 1986.

[146] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, 1995.

[147] Jun Yang. Explaining type errors by finding the source of a type conflict. In *SFP'99: Selected papers from the 1st Scottish Functional Programming Workshop*, pages 59–67, Exeter, UK, UK, 2000. Intellect Books.

[148] Jun Yang, Greg Michaelson, and Phil Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

[149] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. Improved type error reporting. In Markus Mohnen and Pieter W. M. Koopman, editors, *12th Int'l Workshop, IFL 2000*, volume 2011 of *LNCS*, pages 71–86. Springer, 2001.

Index