TYPE-THEORETIC METHODOLOGY FOR PRACTICAL PROGRAMMING LANGUAGES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Karl Fredrick Crary

August 1998

Table of Contents

Chapter 3

Type-Theoretic Semantics

Type theory has become a popular framework for formal reasoning in computer science [22, 67, 37] and has formed the basis for a number of automated deduction systems, including Automath, Nuprl, HOL and Coq [32, 19, 39, 8], among others. In addition to formalizing mathematics, these systems are widely used for the analysis and verification of computer programs. To do this, one must draw a connection between the programming language used and the language of type theory; however, these connections have typically been informal translations, diminishing the significance of the formal verification results.

Formal connections have been drawn in the work of Reynolds [86] and Harper and Mitchell [45], each of whom sought to use type-theoretic analysis to explain an entire programming language. Reynolds gave a type-theoretic interpretation of Idealized Algol, and Harper and Mitchell did the same for a simplified fragment of Standard ML. Recently, Harper and Stone [48] have given such an interpretation of full Standard ML (Revised) [72]. However, in each of these cases, the type theories used were not sufficiently rich to form a foundation for mathematical reasoning; for example, they were unable to express equality or induction principles. On the other hand, Kreitz [59] gave an embedding of a fragment of Objective CAML [64] into the foundational type theory of Nuprl. However, this fragment omitted some important constructs, such as recursion and modules.

The difficulty has been that the same features of foundational type theories that make them so expressive also highly constrain their semantics, thereby restricting the constructs that may be introduced into them. For example, as I will discuss below, the existence of induction principles precludes the typing of *fix* that is typical in programming languages. In this chapter I show how to give a semantics to practical programming languages in foundational type theory. In particular, I give an embedding of λ^K into the Nuprl type theory. This embedding is simple and syntax-directed, which has been vital for its use in practical reasoning.

The applications of type-theoretic semantics are not limited to formal reasoning about programs. Using such a semantics it can be considerably easier to prove desirable properties about a programming language, such as type preservation, than with other means. We will see two such examples in Section 3.4. The usefulness of such semantics is also not limited to one particular programming language at a time. If two languages are given type-theoretic semantics, then one may use type theory to show relationships between the two, and when the semantics are simple, those relationships need be no more complicated than the inherent differences between the two. This is particularly useful in the area of type-directed compilation [93, 75, 61, 43, 77]. The process of type-directed compilation consists of (in part) translations between various typed intermediate languages. Embedding each into a common foundational type theory provides an ideal framework for showing the invariance of program meaning throughout the compilation process.

This semantics is also useful even if one ultimately desires a semantics in some framework other than type theory. Martin-Löf type theory is closely tied to a structured operational semantics and has denotational models in many frameworks including partial equivalence relations [5, 40], set theory [54] and domain theory [87, 81, 80]. Thus, foundational type theory may be used as a "semantic intermediate language."

3.1 A Computational Denotational Semantics

My main motivation for a type-theoretic semantics has been to draw formal connections between programming languages and type theory, thereby making type theory a powerful tool for reasoning about languages and programs without sacrificing any formality. However, a type-theoretic semantics is also valuable in its own right as mathematical model of a programming language.

Most programming language semantics are either operational or denotational. A typical operational semantics is specified by giving an evaluation relation on program terms (or an evaluation relation on some abstract machine along with a translation into that machine). Operational semantics have the advantage that they draw direct connections to computation, and explaining how programs compute is one of the prime functions of a semantics. However, operational semantics are typically rather brittle; a slight addition or change to an operational semantics often requires reworking all proofs of properties of that semantics.

In contrast, a denotational semantics specifies, for every program term, a mathematical object that the program term denotes. Typically a term's denotational is determined by composing in some way the denotations of its subterms. The compositionality of denotational semantics usually makes them more robust to change than a typical operational semantics. Furthermore, the equational theory of a denotational semantics is easier to work with since it derives directly from the mathematical objects, without need for an intermediating evaluation relation, and without needing to consider any surrounding context. However, the connection to computation in a typical operational semantics (although present) is much more remote than with an operational semantics.

A denotational semantics in type theory provides the advantages of both a denotational semantics. The type-theoretic semantics I present *is* denotational, and accrues all the attendant advantages of a denotational semantics, but type theory is in essence a programming language itself (with its own operational semantics), so this semantics also draws a strong connection to computation.

3.2 The Language of Type Theory

I begin with an informal overview of the programming features of the Nuprl type theory. It is primarily those programming features that I will use in the embedding. The logic of types is obtained through the propositions-as-types isomorphism [51], but this will not be critical to our purposes in this chapter. I present and discuss the type theory in detail in Chapter 4.

As base types, the theory contains integers (denoted by $\mathbb{Z}$), booleans[1] (denoted by $\mathbb{B}$), strings (denoted by *Atom*), the empty type *Void*, the trivial type *Unit*, and the type *Top* (which

[1]Booleans are actually defined in terms of the disjoint union type.

contains every well-formed term, and in which all well-formed terms are equal). Complex types are built from the base types using various type constructors such as disjoint unions (denoted by $T_1 + T_2$), dependent products[2] (denoted by $\Sigma x{:}T_1.T_2$) and dependent function spaces (denoted by $\Pi x{:}T_1.T_2$). When x does not appear free in T_2, we write $T_1 \times T_2$ for $\Sigma x{:}T_1.T_2$ and $T_1 \to T_2$ for $\Pi x{:}T_1.T_2$.

This gives an account of most of the familiar programming constructs other than polymorphism. To handle polymorphism we want to have functions that can take types as arguments. These can be typed with the dependent types discussed above if one adds a type of all types. Unfortunately, a single type of all types is known to make the theory inconsistent [36, 27, 70, 52], so instead the type theory includes a predicative hierarchy of universes, $\mathbb{U}_1, \mathbb{U}_2, \mathbb{U}_3$, etc. The universe $\mathbb{U}_1$ contains all types built up from the base types only, and the universe $\mathbb{U}_{i+1}$ contains all types built up from the base types and the universes $\mathbb{U}_1, \ldots, \mathbb{U}_i$. In particular, no universe is a member of itself.

Unlike λ^K, which has distinct syntactic classes for kinds, type constructors and terms, Nuprl has only one syntactic class for all expressions. As a result, types are first class citizens and may be computed just as any other term. For example, the expression *if* b *then* $\mathbb{Z}$ *else Top* (where b is a boolean expression) is a valid type. Evaluation is call-by-name, but these constructions may also be used in a call-by-value type theory with little modification.

To state the soundness of the embedding of λ^K, we will require two assertions from the logic of types. These are equality, denoted by $t_1 = t_2$ *in* T, which states that the terms t_1 and t_2 are equal as members of type T, and subtyping, denoted by $T_1 \preceq T_2$, which states that every member of type T_1 is in type T_2 (and that terms equal in T_1 are equal in T_2). A membership assertion, denoted by t *in* T, is defined as $t = t$ *in* T. The basic judgement in Nuprl is $H \vdash_\nu P$, which states that in context H (which contains hypotheses and declarations of variables) the proposition P is true. Often the proposition P will be an assertion of equality or membership in a type.

The basic operators discussed above are summarized in Figure 3.1. Note that the lambda abstractions of Nuprl are untyped, unlike those of λ^K. In addition to the operators discussed here, the type theory contains some other less familiar type constructors: the partial type, set type and very dependent function type. In order to better motivate these type constructors, we defer discussion of them until their point of relevance. The dynamic semantics of all the type-theoretic operators is given in Figure 4.3.

3.2.1 Domain and Category Theory

Some of the mechanisms I use in the following section to give the type-theoretic semantics of λ^K will look similar to mechanisms used to give programming language semantics in domain or category theory. This is not coincidence; the three theories are closely related and some of the mechanisms I use (such as the partial types of Section 3.3.2) are adapted from the folklore of domain theory.

However, domain theory and category theory are not interchangeable with type theory for the purposes in this dissertation. Type theory provides the highest degree of structure of the three theories, domain theory hides some structure in the interest of greater abstraction and generality, and category theory provides the least structure and the most abstraction and generality. Thus, one can easily construct a domain or a category based on the Nuprl type

[2]These are sometimes referred to in the literature as dependent sums, but I prefer the terminology to suggest the connection to the non-dependent type $T_1 \times T_2$.

	Type Formation	Introduction	Elimination
universe i	$\mathbb{U}_i$ (for $i \geq 1$)	type formation operators	
disjoint union	$T_1 + T_2$	$inj_1(e)$ $inj_2(e)$	$case(e, x_1.e_1, x_2.e_2)$
function space	$\Pi x{:}T_1.T_2$	$\lambda x.e$	$e_1 e_2$
product space	$\Sigma x{:}T_1.T_2$	$\langle e_1, e_2 \rangle$	$\pi_1(e)$ $\pi_2(e)$
integers	$\mathbb{Z}$	$\ldots, -1, 0, 1, 2, \ldots$	assorted operations
booleans	$\mathbb{B}$	*true*, *false*	if-then-else
atoms	*Atom*	string literals	equality test ($=_A$)
void	*Void*		
unit	*Unit*	$\star$	
top	*Top*		

Figure 3.1: Type Theory Syntax

theory, but one cannot easily work backwards (although such models of type theory do exist [87, 81, 80]).

The structure of the Nuprl type theory stems from the fact that Nuprl provides direct access to the primitives of computation and the types of Nuprl speak directly of the computational behavior of terms. This structure is essential for achieving my goal of establishing a direct computational significance for this semantics (recall Section 3.1). This structure also allows type theory to address directly issues that pose significant challenges for domain theory (and that are not directly meaningful in category theory). For example, in type theory we may easily include or exclude divergent terms from types, allowing us to easily distinguish between partial or total functions, and, more importantly, allowing us to use induction properties that are invalid if one includes divergent terms.

3.3 A Type-Theoretic Semantics

I present the embedding of λ^K into type theory in three parts. In the first part I begin by giving embeddings for most of the basic type and term operators. These embeddings are uniformly straightforward. Second, I examine what happens when the embedding is expanded to include *fix*. There we will find it necessary to modify some of the original embeddings of the basic operators. In the third part I complete the semantics by giving embeddings for the kind-level constructs of λ^K. The complete embedding is summarized in Figures 3.3 through 3.6.

The embedding itself could be formulated in type theory, leaving to metatheory only the trivial task of encoding the abstract syntax of the programming language. Were this done, the theorems of Section 3.4 could be proven within the framework of type theory. For simplicity, however, I will state the embedding and theorems in metatheory.

3.3.1 Basic Embedding

The embedding is defined as a syntax-directed mapping (denoted by $[\![\cdot]\!]$) of λ^K expressions to terms of type theory. Recall that in Nuprl all expressions are terms; in particular, types are terms and may be computed just as any other term. Many λ^K expressions are translated directly into type theory:

$$\begin{aligned}
[\![x]\!] &\overset{\text{def}}{=} x \\
[\![\alpha]\!] &\overset{\text{def}}{=} \alpha \\
[\![\lambda x{:}c.e]\!] &\overset{\text{def}}{=} \lambda x.[\![e]\!] \\
[\![e_1 e_2]\!] &\overset{\text{def}}{=} [\![e_1]\!][\![e_2]\!] \\
[\![c_1 \rightarrow c_2]\!] &\overset{\text{def}}{=} [\![c_1]\!] \rightarrow [\![c_2]\!]
\end{aligned}$$

Nothing happens here except that the types are stripped out of lambda abstractions to match the syntax of Nuprl. Functions at the type constructor level are equally easy to embed, but I defer discussion of them until Section 3.3.3.

Since the type theory does not distinguish between functions taking term arguments and functions taking type arguments, polymorphic functions may be embedded just as easily, although a dependent type is required to express the dependency of c on α in the polymorphic type $\forall\alpha{:}\kappa.c$:

$$\begin{aligned}
[\![\Lambda\alpha{:}\kappa.e]\!] &\overset{\text{def}}{=} \lambda\alpha.[\![e]\!] \\
[\![e[c]]\!] &\overset{\text{def}}{=} [\![e]\!][\![c]\!] \\
[\![\forall\alpha{:}\kappa.c]\!] &\overset{\text{def}}{=} \Pi\alpha{:}[\![\kappa]\!].[\![c]\!]
\end{aligned}$$

Just as the type was stripped out of the lambda abstraction above, the kind is stripped out of the polymorphic abstraction. The translation of the polymorphic function type above makes use of the embedding of kinds, but, except for the elementary kind *Type*, I defer discussion of the embedding of kinds until Section 3.3.3. The kind $Type_i$, which contains level-i types, is embedded as the universe containing level-i types:

$$[\![Type_i]\!] \overset{\text{def}}{=} \mathbb{U}_i$$

Records and Disjoint Unions A bit more delicate than the above, but still fairly simple, is the embedding of records. Field labels are taken to be members of type *Atom*, and then records are viewed as functions that map field labels to the contents of the corresponding fields. For example, the record $\{\mathtt{x} = 1, \mathtt{f} = \lambda x{:}int.x\}$, which has type $\{\mathtt{x} : int, \mathtt{f} : int \rightarrow int\}$, is embedded as

$$\lambda a.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ 1\ \mathit{else}\ \mathit{if}\ a =_A \mathtt{f}\ \mathit{then}\ \lambda x.x\ \mathit{else}\ \star$$

where $a =_A a'$ is the equality test on atoms, which returns a boolean when a and a' are atoms.

Since the type of this function's result depends upon its argument, this function must be typed using a dependent type:

$$\Pi a{:}Atom.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ \mathbb{Z}\ \mathit{else}\ \mathit{if}\ a =_A \mathtt{f}\ \mathit{then}\ \mathbb{Z} \rightarrow \mathbb{Z}\ \mathit{else}\ Top$$

In general, records and record types are embedded as follows:

$$\begin{array}{rcl}
[\![\{\ell_1 = e_1, \ldots, \ell_n = e_n\}]\!] & \overset{\text{def}}{=} & \lambda a.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![e_1]\!] \\
& & \quad \vdots \\
& & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![e_n]\!] \\
& & \quad \mathit{else}\ \star \\
[\![\pi_\ell(e)]\!] & \overset{\text{def}}{=} & [\![e]\!]\,\ell \\
[\![\{\ell_1 : c_1, \ldots, \ell_n : c_n\}]\!] & \overset{\text{def}}{=} & \Pi a{:}\mathit{Atom}.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
& & \qquad \mathit{else}\ \mathit{Top}
\end{array}$$

Note that this embedding validates the desired subtyping relationship on records. Since $\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \rightarrow \mathit{int}\} \preceq \{\mathtt{x} : \mathit{int}\}$, we would like the embedding to respect the subtyping relationship: $[\![\{\mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \rightarrow \mathit{int}\}]\!] \preceq [\![\{\mathtt{x} : \mathit{int}\}]\!]$. Fortunately this is the case, since every type is a subtype of *Top*, and in particular the part of the type relating to the omitted field, $\mathit{if}\ a = \mathtt{f}\ \mathit{then}\ \mathbb{Z} \rightarrow \mathbb{Z}\ \mathit{else}\ \mathit{Top}$, is a subtype of *Top*. The use of a type *Top* to catch extra labels is essential for subtyping to work properly makes for a particularly elegant embedding of records, but it is not essential. In the absence of *Top* one could produce a slightly less elegant embedding by restricting the domain to exclude undesired labels using a set type (Section 3.3.3).

Disjoint unions are handled in a similar manner. The injection term $\mathit{inj}_{\mathtt{x}}(1)$ is embedded as the pair $\langle \mathtt{x}, 1\rangle$ of its label and its argument. The types of this term include the sum type $\langle \mathtt{x} : \mathit{int}, \mathtt{f} : \mathit{int} \rightarrow \mathit{int}\rangle$, which is embedded using a dependent type as:

$$\Sigma a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \mathtt{x}\ \mathit{then}\ \mathbb{Z}\ \mathit{else\ if}\ a =_A \mathtt{f}\ \mathit{then}\ \mathbb{Z} \rightarrow \mathbb{Z}\ \mathit{else}\ \mathit{Void}$$

In general, disjoint unions are embedded as follows:

$$\begin{array}{rcl}
[\![\langle \ell_1 : c_1, \ldots, \ell_n : c_n\rangle]\!] & \overset{\text{def}}{=} & \Sigma a{:}\mathit{Atom}.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
& & \qquad \mathit{else}\ \mathit{Void} \\
[\![\mathit{inj}_\ell(e)]\!] & \overset{\text{def}}{=} & \langle \ell, e\rangle \\
[\![\,\mathit{case}(e, \ell_1 \triangleright x_1.e_1, \ldots, \ell_n \triangleright x_n.e_n)]\!] & \overset{\text{def}}{=} & \mathit{if}\ \pi_1(x) =_A \ell_1\ \mathit{then}\ [\![e_1]\!][\pi_2(x)/x_1] \\
& & \vdots \\
& & \mathit{else\ if}\ \pi_1(x) =_A \ell_n\ \mathit{then}\ [\![e_n]\!][\pi_2(x)/x_n] \\
& & \mathit{else}\ \star
\end{array}$$

Again, this relation validates the desired subtyping relationship.

3.3.2 Embedding Recursion

A usual approach to typing general recursive definitions of functions, and the one used in λ^K, is to add a *fix* construct with the typing rule:

$$\frac{H \vdash_\nu e\ \mathit{in}\ T \rightarrow T}{H \vdash_\nu \mathit{fix}(e)\ \mathit{in}\ T} \qquad \text{(wrong)}$$

In effect, this adds recursively defined (and possibly divergent) terms to existing types. Unfortunately, such a broad fixpoint rule makes Martin-Löf type theories inconsistent because of the presence of induction principles. An induction principle on a type specifies the membership of that type; for example, the standard induction principle on the natural numbers specifies that every natural number is either zero or some finite iteration of successor on zero. The ability to add divergent elements to a type would violate the specification implied by that type's induction rule.

One simple way to derive an inconsistency from the above typing rule uses the simplest induction principle, induction on the empty type *Void*. The induction principle for *Void* indirectly specifies that it has no members:

$$\frac{H \vdash_\nu e \text{ } \mathit{in} \text{ } \mathit{Void}}{H \vdash_\nu e \text{ } \mathit{in} \text{ } T}$$

However, it would be easy, using *fix*, to derive a member of *Void*: the identity function can be given type $\mathit{Void} \rightarrow \mathit{Void}$, so $\mathit{fix}(\lambda x.x)$ would have type *Void*. Invoking the induction principle, $\mathit{fix}(\lambda x.x)$ would be a member of every type and, by the propositions-as-types isomorphism, would be a proof of every proposition. It is also worth noting that this inconsistency does not stem from the fact that *Void* is an empty type; similar inconsistencies may be derived (with a bit more work) for almost every type, including function types (to which the *fix* rule of λ^K is restricted).

It is clear, then, that *fix* cannot be used to define new members of the basic types. How then can recursive functions be typed? The solution is to add a new type constructor for ***partial types*** [24, 25, 92]. For any type T, the partial type $\overline{T}$ is a supertype of T that contains all the elements of T and also all divergent terms. (A *total* type is one that contains only convergent terms.) The induction principles on $\overline{T}$ are different than those on T, so we can safely type *fix* with the rule:[3]

$$\frac{H \vdash_\nu e \text{ } \mathit{in} \text{ } \overline{T} \rightarrow \overline{T} \quad H \vdash_\nu T \text{ admiss}}{H \vdash_\nu \mathit{fix}(e) \text{ } \mathit{in} \text{ } \overline{T}}$$

We use partial types to interpret the possibly non-terminating computations of λ^K. When (in λ^K) a term e has type τ, the embedded term $[\![e]\!]$ will have type $\overline{[\![\tau]\!]}$. Moreover, if e is valuable, then $[\![e]\!]$ can still be given the stronger type $[\![\tau]\!]$. Before we can embed *fix* we must re-examine the embedding of function types. In Nuprl, partial functions are viewed as functions with partial result types:[4]

$$\begin{aligned} [\![c_1 \rightarrow c_2]\!] &\overset{\text{def}}{=} [\![c_1]\!] \rightarrow \overline{[\![c_2]\!]} \\ [\![c_1 \Rightarrow c_2]\!] &\overset{\text{def}}{=} [\![c_1]\!] \rightarrow [\![c_2]\!] \\ [\![\forall\alpha{:}\kappa.c]\!] &\overset{\text{def}}{=} \Pi\alpha{:}[\![\kappa]\!].[\![c]\!] \end{aligned}$$

Note that, as desired, $[\![\tau_1 \Rightarrow \tau_2]\!] \preceq [\![\tau_1 \rightarrow \tau_2]\!]$, since $[\![\tau_2]\!] \preceq \overline{[\![\tau_2]\!]}$. If partial polymorphic functions were included in λ^K, they would be embedded as $\Pi\alpha{:}[\![\kappa]\!].\overline{[\![c]\!]}$.

Now suppose we wish to *fix* the function f which (in λ^K) has type $(\tau_1 \rightarrow \tau_2) \rightarrow (\tau_1 \rightarrow \tau_2)$, and suppose, for simplicity only, that f is valuable. Then $[\![f]\!]$ has type $([\![\tau_1]\!] \rightarrow \overline{[\![\tau_2]\!]}) \rightarrow \overline{[\![\tau_1]\!] \rightarrow \overline{[\![\tau_2]\!]}}$.

[3]The second subgoal, that the type T be ***admissible***, is a technical condition related to the notion of admissibility in LCF [38]. All the types used in the embedding are admissible, so I ignore the admissibility condition in the discussion of this chapter. Admissibility is discussed further in Chapter 4 and is examined in detail in Chapter 5.

[4]This terminology can be somewhat confusing. A total type is one that contains only convergent expressions. The partial *function* type $T_1 \rightarrow \overline{T}_2$ contains functions that *return* possibly divergent elements, but those functions themselves converge, so a partial function type is a total type.

This type does not quite fit the *fix* typing rule, which requires the domain type to be partial, so we must coerce $\llbracket f \rrbracket$ to a fixable type. We do this by eta-expanding $\llbracket f \rrbracket$ to gain access to its argument (call it g) and then eta-expanding that argument g:

$$\lambda g.\llbracket f \rrbracket(\lambda x.g\,x) \ \mathit{in}\ \overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}} \to \overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}}$$

Eta-expanding g ensures that it terminates (since $\lambda x.g\,x$ is a canonical form), changing its type from $\overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}}$ to $\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}$. The former type is required by the *fix* rule, but the latter type is expected by $\llbracket f \rrbracket$. Since the coerced $\llbracket f \rrbracket$ fits the *fix* typing rule, we get that $\mathit{fix}(\lambda g.\llbracket f \rrbracket(\lambda x.g\,x))$ has type $\overline{\llbracket \tau_1 \rrbracket \to \overline{\llbracket \tau_2 \rrbracket}}$, as desired. Thus we may embed the *fix* construct as:

$$\llbracket \mathit{fix}_c(e) \rrbracket \ \stackrel{\text{def}}{=}\ \mathit{fix}(\lambda g.\llbracket e \rrbracket(\lambda x.g\,x))$$

Strictness In λ^K, a function may be applied to a possibly divergent argument, but in my semantics functions expect their arguments to be convergent. Therefore we must change the embedding of application to compute function arguments to canonical form before applying the function.[5] This is done using the sequencing construct $\mathit{let}\ x = e_1\ \mathit{in}\ e_2$ which evaluates e_1 to canonical form e_1' and then reduces to $e_2[e_1'/x]$. The sequence term diverges if e_1 or e_2 does and allows x be given a total type:

$$\frac{H \vdash_\nu e_1\ \mathit{in}\ \overline{T_2} \quad H; x{:}T_2 \vdash_\nu e_2\ \mathit{in}\ \overline{T_1}}{H \vdash_\nu \mathit{let}\ x = e_1\ \mathit{in}\ e_2\ \mathit{in}\ \overline{T_1}}$$

Application is then embedded in the expected way:

$$\llbracket e_1 e_2 \rrbracket \ \stackrel{\text{def}}{=}\ \mathit{let}\ x = \llbracket e_2 \rrbracket\ \mathit{in}\ \llbracket e_1 \rrbracket\, x$$

A final issue arises in regard to records and disjoint unions. In the embedding of Section 3.3.1, the record $\{\ell = e\}$ would terminate even if e diverges. This would be unusual in a call-by-value programming language, so we need to ensure that each member of a record is evaluated:

$$\begin{aligned}
\llbracket \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \rrbracket \ \stackrel{\text{def}}{=}\ & \mathit{let}\ x_1 = \llbracket e_1 \rrbracket\ \mathit{in} \\
& \vdots \\
& \mathit{let}\ x_n = \llbracket e_n \rrbracket\ \mathit{in} \\
& \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
& \qquad \vdots \\
& \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
& \qquad \mathit{else} \star
\end{aligned}$$

A similar change must be also be made for disjoint unions:

$$\llbracket \mathit{inj}_\ell(e) \rrbracket \ \stackrel{\text{def}}{=}\ \mathit{let}\ x = \llbracket e \rrbracket\ \mathit{in}\ \langle \ell, x \rangle$$

[5]Polymorphic functions are unaffected because all type expressions converge (Corollary 3.4).

3.3.3 Embedding Kinds

The kind structure of λ^K contains three first-order kind constructors. We have already seen the embedding of the kind *Type*; remaining are the power and singleton kinds. Each of these kinds represents a collection of types, so each will be embedded as something similar to a universe, but unlike the kind $Type_i$, which includes all types of the indicated level, the power and singleton kinds wish to exclude certain undesirable types. The power kind $\mathcal{P}_i(\tau)$ contains only subtypes of τ and the singleton kind $\mathcal{S}_i(\tau)$ contains only types that are equal to τ; other types must be left out.

The mechanism for achieving this exclusion is the *set type* [21]. If S is a type and $P[\cdot]$ is a predicate over S, then the set type $\{z : S \mid P[z]\}$ contains all elements z of S such that $P[z]$ is true. With this type, we can embed the power and singleton kinds as:[6]

$$\begin{aligned}
[\![\mathcal{P}_i(c)]\!] &\stackrel{\text{def}}{=} \{T : \mathbb{U}_i \mid T \preceq [\![c]\!] \wedge [\![c]\!] \ \mathit{in}\ \mathbb{U}_i\} \\
[\![\mathcal{S}_i(c)]\!] &\stackrel{\text{def}}{=} \{T : \mathbb{U}_i \mid T = [\![c]\!] \ \mathit{in}\ \mathbb{U}_i\}
\end{aligned}$$

Among the higher-order type constructors, functions at the type constructor level and their kinds are handled just as at the term level, except that function kinds are permitted to have dependencies but need not deal with partiality or strictness:

$$\begin{aligned}
[\![\lambda\alpha{:}\kappa.c]\!] &\stackrel{\text{def}}{=} \lambda\alpha.[\![c]\!] \\
[\![c_1[c_2]]\!] &\stackrel{\text{def}}{=} [\![c_1]\!][\![c_2]\!] \\
[\![\Pi\alpha{:}\kappa_1.\kappa_2]\!] &\stackrel{\text{def}}{=} \Pi\alpha{:}[\![\kappa_1]\!].[\![\kappa_2]\!]
\end{aligned}$$

Dependent Record Kinds For records at the type constructor level, the embedding of the records themselves is analogous to those at the term level (except that there is no issue of strictness):

$$\begin{aligned}
[\![\{\ell_1 = c_1, \ldots, \ell_n = c_n\}]\!] &\stackrel{\text{def}}{=} \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![c_1]\!] \\
&\qquad\quad \vdots \\
&\qquad\quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![c_n]\!] \\
&\qquad\quad \mathit{else}\,\star \\
[\![\pi_\ell(c)]\!] &\stackrel{\text{def}}{=} [\![c]\!]\,\ell
\end{aligned}$$

However, the embedding of this expression's kind is more complicated. This is because of the need to express dependencies among the fields of the dependent record kind. Recall that the embedding of a non-dependent record type already required a dependent type; to embed a dependent record type will require expressing even more dependency. Consider the dependent record kind $\{\ell \triangleright \alpha : Type_1, \ell' \triangleright \alpha' : \mathcal{P}_1(\alpha)\}$. We might naively attempt to encode this like the non-dependent record type as

$$\Pi a{:}Atom.\ \mathit{if}\ a =_A \ell\ \mathit{then}\ \mathbb{U}_1\ \mathit{else\ if}\ a =_A \ell'\ \mathit{then}\ \{T : \mathbb{U}_1 \mid T \preceq \alpha \wedge \alpha\ \mathit{in}\ \mathbb{U}_1\}\ \mathit{else}\ Top \quad \text{(wrong)}$$

but this encoding is not correct; the variable α is now unbound. We want α to refer to the contents of field ℓ. In the encoding, this means we want α to refer to the value returned by the function when applied to label ℓ. So we want a type of functions whose return type can depend not only upon their arguments but upon their own return values!

[6]The second clause in the embedding of the power kind ($[\![c]\!]$ *in* $\mathbb{U}_i$) is used for technical reasons that require that well-formedness of $\mathcal{P}_i(\tau)$ imply that $\tau : Type_i$.

The type I will use for this embedding is Hickey's *very dependent function type* [49]. This type is a generalization of the dependent function type (itself a generalization of the ordinary function type) and like it, the very dependent function type's members are just lambda abstractions. The difference is in the specification of a function's return type. The type is denoted by $\{f \mid x{:}T_1 \rightarrow T_2\}$ where f and x are binding occurrences that may appear free in T_2 (but not in T_1).

As with the dependent function type, x stands for the function's argument, but the additional variable f refers to the function itself. A function g belongs to the type $\{f \mid x{:}T_1 \rightarrow T_2\}$ if g takes an argument from T_1 (call it t) and returns a member of $T_2[t, g/x, f]$.[7]

For example, the kind $\{\ell \triangleright \alpha : \mathit{Type}_1, \ell' \triangleright \alpha' : \mathcal{P}_1(\alpha)\}$ discussed above is encoded as a very dependent function type as:

$$\{f \mid a{:}\mathit{Atom} \rightarrow \mathit{if}\ a =_A \ell\ \mathit{then}\ \mathbb{U}_1\ \mathit{else\ if}\ a =_A \ell'\ \mathit{then}\ \{T : \mathbb{U}_1 \mid T \preceq f\,\ell \wedge f\,\ell\ \mathit{in}\ \mathbb{U}_1\}\ \mathit{else}\ \mathit{Top}\}$$

To understand where this type constructor fits in with the more familiar type constructors, consider the "type triangle" shown in Figure 3.2. On the right are the non-dependent type constructors and in the middle are the dependent type constructors. Arrows are drawn from type constructors to weaker ones that may be implemented with them. Horizontal arrows indicate when a weaker constructor may be obtained by dropping a possible dependency from a stronger one; for example, the function type $T_1 \rightarrow T_2$ is a degenerate form of the dependent function type $\Pi x{:}T_1.T_2$ where the dependent variable x is not used in T_2. Diagonal arrows indicate when a weaker constructor may be implemented with a stronger one by performing case analysis on a boolean; for example, the disjoint union type $T_1 + T_2$ is equivalent to the type $\Sigma b{:}\mathbb{B}.\ \mathit{if}\ b\ \mathit{then}\ T_1\ \mathit{else}\ T_2$.[8]

If we ignore the very dependent function type, the type triangle illustrates how the basic type constructors may be implemented by the dependent function and dependent product types. The very dependent function type completes this picture: the dependent function is a degenerate form where the f dependency is not used, and the dependent product may be implemented by switching on a boolean. Thus, the very dependent function type is a single unified type constructor from which all the basic type constructors may be constructed.

In general, dependent record kinds are encoded using a very dependent function type as follows:

$$\begin{aligned}
[\![\{\ell_1 \triangleright \alpha_1 : \kappa_1, \ldots, \ell_n \triangleright \alpha_n : \kappa_n\}]\!] \stackrel{\text{def}}{=} \{f \mid a{:}\mathit{Atom}\rightarrow\ & \\
& \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\kappa_1]\!] \\
& \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\kappa_2]\!][f\,\ell_1/\alpha_1] \\
& \vdots \\
& \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
& \quad [\![\kappa_n]\!][f\,\ell_1 \cdots f\,\ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
& \mathit{else}\ \mathit{Top}\}
\end{aligned}$$

[7]To avoid the apparent circularity, in order for $\{f \mid x{:}T_1 \rightarrow T_2\}$ to be well-formed we require that T_2 may only use the result of f when applied to elements of T_1 that are less than x with regard to some well-founded order. This restriction will not be a problem for this embedding because the order in which field labels appear in a dependent record kind is a perfectly good well-founded order.

[8]By switching on a label, instead of a boolean, record types and tagged variant types could be implemented and placed along the diagonals as well.

$$\begin{tikzcd}
\{f \mid x{:}T_1 \to T_2\} \arrow[r] \arrow[dr] & \Pi x{:}T_1.T_2 \arrow[r] \arrow[dr] & T_1 \to T_2 \\
 & \Sigma x{:}T_1.T_2 \arrow[r] \arrow[dr] & T_1 \times T_2 \\
 & & T_1 + T_2
\end{tikzcd}$$

Figure 3.2: The Type Triangle

3.3.4 Embedding Modules

As discussed in Section 2.4.1, λ^K uses a phase-splitting interpretation of modules, where modules (including higher-order modules) are considered to consist of two components: a compile-time component and a run-time component. This is reflected in the type-theoretic semantics by an embedding that explicitly phase-splits modules. The technique used is derived from Harper, *et al.* [46].

The embeddings for modules and signatures are given by two syntax-directed mappings, $[\![\cdot]\!]_c$ and $[\![\cdot]\!]_r$, one for the compile-time component of the given expression and one for the run time component. Given these, the embedding of a module is a pair of the compile-time and run-time components:

$$[\![m]\!] \stackrel{\text{def}}{=} \langle [\![m]\!]_c, [\![m]\!]_r \rangle$$

In signatures, the types of run-time fields may depend upon a compile-time member, as in the signature $\{\texttt{foo} \triangleright s_{\texttt{foo}} : \langle \mathit{Type} \rangle, \texttt{bar} : \langle\!\langle \mathit{ext}(s_{\texttt{foo}}) \rangle\!\rangle\}$ (corresponding to the KML module `sig tycon foo : type val bar : foo end`). Consequently, the embedding of a signature is the dependent product of the compile-time component and the run-time component. The run-time component's embedding is a function that takes as an argument the compile-time member on which it depends. In the embedding of the full signature, that function is applied to the variable standing for the compile-time member:

$$[\![\sigma]\!] \stackrel{\text{def}}{=} \Sigma v{:}[\![\sigma]\!]_c.[\![\sigma]\!]_r v$$

The embeddings of basic modules and signatures are simple. The run-time component is trivial for $\langle \kappa \rangle$ signatures and $\langle c \rangle$ modules, and the compile-time component is trivial for $\langle\!\langle c \rangle\!\rangle$ signatures and $\langle\!\langle e \rangle\!\rangle$. Module variables s are split into separate variables s_c and s_r for each component.

$$\begin{array}{rcll}
[\![\langle c\rangle]\!]_c & \stackrel{\text{def}}{=} & [\![c]\!] & \text{(module compile-time component)} \\
[\![\langle c\rangle]\!]_r & \stackrel{\text{def}}{=} & \star & \text{(module run-time component—trivial)} \\
[\![\langle \kappa\rangle]\!]_c & \stackrel{\text{def}}{=} & [\![\kappa]\!] & \text{(signature compile-time component)} \\
[\![\langle \kappa\rangle]\!]_r & \stackrel{\text{def}}{=} & \lambda v.\mathit{Top} & \text{(signature run-time component—trivial)} \\
[\![\langle\!\langle e\rangle\!\rangle]\!]_c & \stackrel{\text{def}}{=} & \star & \text{(module compile-time component—trivial)} \\
[\![\langle\!\langle e\rangle\!\rangle]\!]_r & \stackrel{\text{def}}{=} & [\![e]\!] & \text{(module run-time component)} \\
[\![\langle\!\langle c\rangle\!\rangle]\!]_c & \stackrel{\text{def}}{=} & \mathit{Top} & \text{(signature compile-time component—trivial)} \\
[\![\langle\!\langle c\rangle\!\rangle]\!]_r & \stackrel{\text{def}}{=} & \lambda v.[\![c]\!] & \text{(signature run-time component)} \\
[\![s]\!]_c & \stackrel{\text{def}}{=} & s_c & \text{(module compile-time component)} \\
[\![s]\!]_r & \stackrel{\text{def}}{=} & s_r & \text{(module run-time component)}
\end{array}$$

For each of the signatures above it is impossible for there to be any dependency of the run-time component on the compile-time component, since for $\langle\kappa\rangle$ there is no nontrivial run-time to depend on anything, and for $\langle\!\langle c\rangle\!\rangle$ there is no nontrivial compile-time for anything to depend on. As a result, the variable v is ignored in each case.

Functors For function modules and signatures, everything looks familiar in the compile-time component, where the run-time material is ignored:

$$\begin{array}{rcl}
[\![\lambda s{:}\sigma.m]\!]_c & \stackrel{\text{def}}{=} & \lambda s_c.[\![m]\!]_c \\
[\![m_1 m_2]\!]_c & \stackrel{\text{def}}{=} & [\![m_1]\!]_c[\![m_2]\!]_c \\
[\![\Pi s{:}\sigma_1.\sigma_2]\!]_c & \stackrel{\text{def}}{=} & \Pi s_c{:}[\![\sigma_1]\!]_c.[\![\sigma_2]\!]_c
\end{array}$$

However, the run-time component may not similarly ignore the compile-time component, because of possible dependencies. Therefore, the run-time component abstracts over both the compile-time and run-time components of its argument:

$$\begin{array}{rcl}
[\![\lambda s{:}\sigma.m]\!]_r & \stackrel{\text{def}}{=} & \lambda s_c.\,\lambda s_r.\,[\![m]\!]_r \\
[\![m_1 m_2]\!]_r & \stackrel{\text{def}}{=} & \mathit{let}\ x = [\![m_2]\!]_r\ \mathit{in}\ [\![m_1]\!]_r[\![m_2]\!]_c x
\end{array}$$

The signature's run-time component, then, takes an argument v representing the compile-time component and returns a curried function type. The result type is the run-time component of the result signature and that depends on the compile-time component. Fortunately, the result's compile-time component is available (as $v\, s_c$), since v maps the compile-time component of the argument to the compile-time component of the result.

$$[\![\Pi s{:}\sigma_1.\sigma_2]\!]_r \stackrel{\text{def}}{=} \lambda v.\,\Pi s_c{:}[\![\sigma_1]\!]_c.\,[\![\sigma_1]\!]_r s_c \rightarrow [\![\sigma_2]\!]_r(v\, s_c)$$

Structures For dependent record modules and signatures, the compile-time component looks like dependent record kinds and the type constructor records that belong to them:

$$
\begin{array}{rcl}
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_{\mathrm{c}} & \overset{\mathrm{def}}{=} & \lambda a.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![m_1]\!]_{\mathrm{c}} \\
& & \quad \vdots \\
& & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![m_n]\!]_{\mathrm{c}} \\
& & \quad \mathit{else}\ \star \\
[\![\pi_\ell(m)]\!]_{\mathrm{c}} & \overset{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{c}} \ell \\
[\![\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}]\!]_{\mathrm{c}} & \overset{\mathrm{def}}{=} & \{f \mid a{:}\mathit{Atom} \rightarrow \\
& & \qquad \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\sigma_1]\!]_{\mathrm{c}} \\
& & \qquad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{c}}[f\,\ell_1/s_{1\mathrm{c}}] \\
& & \qquad \vdots \\
& & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
& & \qquad\quad [\![\sigma_n]\!]_{\mathrm{c}}[f\,\ell_1 \cdots f\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{(n-1)\mathrm{c}}] \\
& & \qquad \mathit{else}\ \mathit{Top}\}
\end{array}
$$

The run-time component of modules looks much like the embedding of record terms (as with those, a series of opening lets is necessary to ensure strictness):

$$
\begin{array}{rcl}
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_{\mathrm{r}} & \overset{\mathrm{def}}{=} & \mathit{let}\ x_1 = [\![m_1]\!]_{\mathrm{r}}\ \mathit{in} \\
& & \vdots \\
& & \mathit{let}\ x_n = [\![m_n]\!]_{\mathrm{r}}\ \mathit{in} \\
& & \lambda a.\, \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
& & \quad \vdots \\
& & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
& & \quad \mathit{else}\ \star \\
& & \text{(where } x_i \text{ does not appear free in } m_i) \\
[\![\pi_\ell(m)]\!]_{\mathrm{r}} & \overset{\mathrm{def}}{=} & [\![m]\!]_{\mathrm{r}} \ell
\end{array}
$$

The run-time component of signatures is also familiar, except that it must deal with dependencies on the compile-time component. Again, the run-time component takes an argument v representing the compile-time component. The run-time component $[\![\sigma_i]\!]_{\mathrm{r}}$ of each field is applied to the compile-time component of that field, which is $v\,\ell_i$. Also, each field may have dependencies on the compile-time components of *earlier* fields. These dependencies will have been expressed by free occurrences of the variables s_{ic}, into which we substitute the corresponding compile-time components $v\,\ell_i$. It is worthwhile to note that these substitutions for s_{ic} are the only places where dependencies on the argument v are introduced.

$$
\begin{array}{l}
[\![\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}]\!]_{\mathrm{r}} \\
\quad \overset{\mathrm{def}}{=} \quad \lambda v.\, \Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![\sigma_1]\!]_{\mathrm{r}}(v\,\ell_1) \\
\qquad\qquad\qquad\qquad\quad \mathit{if}\ a =_A \ell_2\ \mathit{then}\ [\![\sigma_2]\!]_{\mathrm{r}}(v\,\ell_2)[v\,\ell_1/s_{1\mathrm{c}}] \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
\qquad\qquad\qquad\qquad\qquad [\![\sigma_n]\!]_{\mathrm{r}}(v\,\ell_n)[v\,\ell_1 \cdots v\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{(n-1)\mathrm{c}}] \\
\qquad\qquad\qquad\qquad\quad \mathit{else}\ \mathit{Top}
\end{array}
$$

$$
\begin{aligned}
\llbracket \mathit{Type}_i \rrbracket &\overset{\text{def}}{=} \{T : \mathbb{U}_i \mid T \text{ admiss} \wedge T \text{ total}\} \\
\llbracket \Pi\alpha{:}\kappa_1.\kappa_2 \rrbracket &\overset{\text{def}}{=} \Pi\alpha{:}\llbracket\kappa_1\rrbracket.\llbracket\kappa_2\rrbracket \\
\llbracket \{\ell_1 \rhd \alpha_1 : \kappa_1, \ldots, \ell_n \rhd \alpha_n : \kappa_n\} \rrbracket &\overset{\text{def}}{=} \{f \mid a{:}\mathit{Atom} \rightarrow \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket\kappa_1\rrbracket \\
&\qquad\qquad\qquad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ \llbracket\kappa_2\rrbracket[f\,\ell_1/\alpha_1] \\
&\qquad\qquad\qquad \vdots \\
&\qquad\qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
&\qquad\qquad\qquad\qquad \llbracket\kappa_n\rrbracket[f\,\ell_1 \cdots f\,\ell_{n-1}/\alpha_1 \cdots \alpha_{n-1}] \\
&\qquad\qquad\qquad \mathit{else\ Top}\} \\
&\qquad \text{(where } f, a \text{ do not appear free in } \kappa_i\text{)} \\
\llbracket \mathcal{P}_i(c) \rrbracket &\overset{\text{def}}{=} \{T : \mathbb{U}_i \mid T \preceq \llbracket c\rrbracket \wedge \llbracket c\rrbracket\ \mathit{in}\ \mathbb{U}_i \wedge T \text{ admiss}\} \\
&\qquad \text{(where } T \text{ does not appear free in } c\text{)} \\
\llbracket \mathcal{S}_i(c) \rrbracket &\overset{\text{def}}{=} \{T : \mathbb{U}_i \mid T = \llbracket c\rrbracket\ \mathit{in}\ \mathbb{U}_i\} \\
&\qquad \text{(where } T \text{ does not appear free in } c\text{)} \\
\\
\llbracket \alpha \rrbracket &\overset{\text{def}}{=} \alpha \\
\llbracket \lambda\alpha{:}\kappa.c \rrbracket &\overset{\text{def}}{=} \lambda\alpha.\llbracket c\rrbracket \\
\llbracket c_1[c_2] \rrbracket &\overset{\text{def}}{=} \llbracket c_1\rrbracket\llbracket c_2\rrbracket \\
\llbracket \{\ell_1 = c_1, \ldots, \ell_n = c_n\} \rrbracket &\overset{\text{def}}{=} \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket c_1\rrbracket \\
&\qquad \vdots \\
&\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ \llbracket c_n\rrbracket \\
&\qquad \mathit{else}\ \star \\
&\qquad \text{(where } a \text{ does not appear free in } c_i\text{)} \\
\llbracket \pi_\ell(c) \rrbracket &\overset{\text{def}}{=} \llbracket c\rrbracket\,\ell \\
\llbracket c_1 \rightarrow c_2 \rrbracket &\overset{\text{def}}{=} \llbracket c_1\rrbracket \rightarrow \overline{\llbracket c_2\rrbracket} \\
\llbracket c_1 \Rightarrow c_2 \rrbracket &\overset{\text{def}}{=} \llbracket c_1\rrbracket \rightarrow \llbracket c_2\rrbracket \\
\llbracket \forall\alpha{:}\kappa.c \rrbracket &\overset{\text{def}}{=} \Pi\alpha{:}\llbracket\kappa\rrbracket.\llbracket c\rrbracket \\
\llbracket \{\ell_1 : c_1, \ldots, \ell_n : c_n\} \rrbracket &\overset{\text{def}}{=} \Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket c_1\rrbracket \\
&\qquad\qquad \vdots \\
&\qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ \llbracket c_n\rrbracket \\
&\qquad\qquad\quad \mathit{else\ Top} \\
&\qquad \text{(where } a \text{ does not appear free in } c_i\text{)} \\
\llbracket \langle\ell_1 : c_1, \ldots, \ell_n : c_n\rangle \rrbracket &\overset{\text{def}}{=} \Sigma a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket c_1\rrbracket \\
&\qquad\qquad \vdots \\
&\qquad\qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ \llbracket c_n\rrbracket \\
&\qquad\qquad\quad \mathit{else\ Void} \\
&\qquad \text{(where } a \text{ does not appear free in } c_i\text{)} \\
\llbracket \mathit{ext}(m) \rrbracket &\overset{\text{def}}{=} \llbracket m\rrbracket_{\mathsf{c}}
\end{aligned}
$$

Figure 3.3: Embedding Kinds and Types

$$
\begin{array}{rcl}
\llbracket x \rrbracket & \overset{\text{def}}{=} & x \\
\llbracket \lambda x{:}c.e \rrbracket & \overset{\text{def}}{=} & \lambda x.\llbracket e \rrbracket \\
\llbracket e_1 e_2 \rrbracket & \overset{\text{def}}{=} & \mathit{let}\ x = \llbracket e_2 \rrbracket\ \mathit{in}\ \llbracket e_1 \rrbracket\, x \\
 & & \text{(where } x \text{ does not appear free in } e_1\text{)} \\
\llbracket \Lambda\alpha{:}\kappa.e \rrbracket & \overset{\text{def}}{=} & \lambda\alpha.\llbracket e \rrbracket \\
\llbracket e[c] \rrbracket & \overset{\text{def}}{=} & \llbracket e \rrbracket \llbracket c \rrbracket \\
\llbracket \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \rrbracket & \overset{\text{def}}{=} & \mathit{let}\ x_1 = \llbracket e_1 \rrbracket\ \mathit{in} \\
 & & \vdots \\
 & & \mathit{let}\ x_n = \llbracket e_n \rrbracket\ \mathit{in} \\
 & & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
 & & \qquad \vdots \\
 & & \qquad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
 & & \qquad \mathit{else}\ \star \\
 & & \text{(where } x_i \text{ does not appear free in } e_j\text{)} \\
\llbracket \pi_\ell(e) \rrbracket & \overset{\text{def}}{=} & \llbracket e \rrbracket\, \ell \\
\llbracket \mathit{inj}_\ell(e) \rrbracket & \overset{\text{def}}{=} & \mathit{let}\ x = \llbracket e \rrbracket\ \mathit{in}\ \langle \ell, x \rangle \\
\llbracket \mathit{case}(e, \ell_1 \triangleright x_1.e_1, \ldots, \ell_n \triangleright x_n.e_n) \rrbracket & \overset{\text{def}}{=} & \mathit{let}\ x = \llbracket e \rrbracket\ \mathit{in} \\
 & & \mathit{if}\ \pi_1(x) =_A \ell_1\ \mathit{then}\ e_1[\pi_2(x)/x_1] \\
 & & \vdots \\
 & & \mathit{else\ if}\ \pi_1(x) =_A \ell_n\ \mathit{then}\ e_n[\pi_2(x)/x_n] \\
 & & \mathit{else}\ \star \\
 & & \text{(where } x \text{ does not appear free in } e_i\text{)} \\
\llbracket \mathit{fix}_c(e) \rrbracket & \overset{\text{def}}{=} & \mathit{fix}(\lambda g.\llbracket e \rrbracket(\lambda x.g\,x)) \\
 & & \text{(where } g \text{ does not appear free in } e\text{)} \\
\llbracket \mathit{ext}(m) \rrbracket & \overset{\text{def}}{=} & \llbracket m \rrbracket_{\mathrm{r}}
\end{array}
$$

Figure 3.4: Embedding Terms

3.4 Properties of the Embedding

I conclude my presentation of the type-theoretic semantics of λ^K by examining some of the important properties of the semantics. We want the embedding to validate the intuitive meaning of the judgements of λ^K's static semantics. If κ_1 is a subkind of κ_2 then we want the embedded kind $\llbracket\kappa_1\rrbracket$ to be a subtype of $\llbracket\kappa_2\rrbracket$; if c_1 and c_2 are equal in kind κ, we want the embedded constructors $\llbracket c_1\rrbracket$ and $\llbracket c_2\rrbracket$ to be equal (in $\llbracket\kappa\rrbracket$); and if e has type τ we want $\llbracket e\rrbracket$ to have type $\overline{\llbracket\tau\rrbracket}$ (and $\llbracket\tau\rrbracket$ if e is valuable). Similar properties are desired for the module judgements. This is stated in Theorem 3.1:

Theorem 3.1 (Semantic Soundness) *For every λ^K context Γ, let $\llbracket\Gamma\rrbracket$ be defined as follows:*

$$
\begin{array}{rcl}
\llbracket \bullet \rrbracket & \overset{\text{def}}{=} & \epsilon \\
\llbracket \Gamma[\alpha : \kappa] \rrbracket & \overset{\text{def}}{=} & \llbracket\Gamma\rrbracket; \alpha{:}\llbracket\kappa\rrbracket \\
\llbracket \Gamma[x : c] \rrbracket & \overset{\text{def}}{=} & \llbracket\Gamma\rrbracket; x{:}\llbracket c\rrbracket \\
\llbracket \Gamma[s : \sigma] \rrbracket & \overset{\text{def}}{=} & \llbracket\Gamma\rrbracket; s_{\mathrm{c}}{:}\llbracket\sigma\rrbracket_{\mathrm{c}}; s_{\mathrm{r}}{:}\llbracket\sigma\rrbracket_{\mathrm{r}} s_{\mathrm{c}}
\end{array}
$$

Then the following implications hold:

1. *If $\Gamma \vdash_K \kappa = \kappa'$ then $\mathit{level}(\kappa) = \mathit{level}(\kappa')$ and $\llbracket\Gamma\rrbracket \vdash_\nu \llbracket\kappa\rrbracket = \llbracket\kappa'\rrbracket$ in $\mathbb{U}_{\mathit{level}(\kappa)+1}$*

$$
\begin{array}{rcl}
\llbracket\sigma\rrbracket & \overset{\text{def}}{=} & \Sigma v{:}\llbracket\sigma\rrbracket_{\mathrm{c}}.\llbracket\sigma\rrbracket_{\mathrm{r}}v \\
 & & \text{(where } v \text{ does not appear free in } \sigma\text{)} \\
\llbracket\langle\kappa\rangle\rrbracket_{\mathrm{c}} & \overset{\text{def}}{=} & \llbracket\kappa\rrbracket \\
\llbracket\langle\kappa\rangle\rrbracket_{\mathrm{r}} & \overset{\text{def}}{=} & \lambda v.\,\mathit{Top} \\
\llbracket\langle\!\langle c\rangle\!\rangle\rrbracket_{\mathrm{c}} & \overset{\text{def}}{=} & \mathit{Top} \\
\llbracket\langle\!\langle c\rangle\!\rangle\rrbracket_{\mathrm{r}} & \overset{\text{def}}{=} & \lambda v.\llbracket c\rrbracket \\
 & & \text{(where } v \text{ does not appear free in } c\text{)} \\
\llbracket\Pi s{:}\sigma_1.\sigma_2\rrbracket_{\mathrm{c}} & \overset{\text{def}}{=} & \Pi s_{\mathrm{c}}{:}\llbracket\sigma_1\rrbracket_{\mathrm{c}}.\llbracket\sigma_2\rrbracket_{\mathrm{c}} \\
\llbracket\Pi s{:}\sigma_1.\sigma_2\rrbracket_{\mathrm{r}} & \overset{\text{def}}{=} & \lambda v.\,\Pi s_{\mathrm{c}}{:}\llbracket\sigma_1\rrbracket_{\mathrm{c}}.\,\llbracket\sigma_1\rrbracket_{\mathrm{r}}s_{\mathrm{c}} \to \llbracket\sigma_2\rrbracket_{\mathrm{r}}(v\,s_{\mathrm{c}}) \\
 & & \text{(where } v, s \text{ do not appear free in } \sigma_i\text{)} \\
\llbracket\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}\rrbracket_{\mathrm{c}} & \overset{\text{def}}{=} & \{f \mid a{:}\mathit{Atom} \to \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket\sigma_1\rrbracket_{\mathrm{c}} \\
 & & \qquad\qquad\quad \mathit{else\ if}\ a =_A \ell_2\ \mathit{then}\ \llbracket\sigma_2\rrbracket_{\mathrm{c}}[f\,\ell_1/s_{1\mathrm{c}}] \\
 & & \qquad\qquad\quad \vdots \\
 & & \qquad\qquad\quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad\qquad\qquad \llbracket\sigma_n\rrbracket_{\mathrm{c}}[f\,\ell_1 \cdots f\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{n-1\mathrm{c}}] \\
 & & \qquad\qquad\quad \mathit{else}\ \mathit{Top}\} \\
 & & \text{(where } f, a \text{ do not appear free in } \sigma_i\text{)} \\
\llbracket\{\ell_1 \triangleright s_1 : \sigma_1, \ldots, \ell_n \triangleright s_n : \sigma_n\}\rrbracket_{\mathrm{r}} & \overset{\text{def}}{=} & \lambda v.\,\Pi a{:}\mathit{Atom}.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ \llbracket\sigma_1\rrbracket_{\mathrm{r}}(v\,\ell_1) \\
 & & \qquad\qquad\quad \mathit{if}\ a =_A \ell_2\ \mathit{then}\ \llbracket\sigma_2\rrbracket_{\mathrm{r}}(v\,\ell_2)[v\,\ell_1/s_{1\mathrm{c}}] \\
 & & \qquad\qquad\quad \vdots \\
 & & \qquad\qquad\quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then} \\
 & & \qquad\qquad\qquad \llbracket\sigma_n\rrbracket_{\mathrm{r}}(v\,\ell_n) \\
 & & \qquad\qquad\qquad\quad [v\,\ell_1 \cdots v\,\ell_{n-1}/s_{1\mathrm{c}} \cdots s_{n-1\mathrm{c}}] \\
 & & \qquad\qquad\quad \mathit{else}\ \mathit{Top}\} \\
 & & \text{(where } v, a \text{ do not appear free in } \sigma_i\text{)}
\end{array}
$$

Figure 3.5: Embedding Signatures

2. *If* $\Gamma \vdash_K \kappa \preceq \kappa'$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu (\llbracket\kappa\rrbracket\ \mathit{in}\ \mathbb{U}_{\mathit{level}(\kappa)+1} \wedge \llbracket\kappa'\rrbracket\ \mathit{in}\ \mathbb{U}_{\mathit{level}(\kappa')+1} \wedge \llbracket\kappa\rrbracket \preceq \llbracket\kappa'\rrbracket)$.
3. *If* $\Gamma \vdash_K c = c' : \kappa$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu \llbracket c\rrbracket = \llbracket c'\rrbracket\ \mathit{in}\ \llbracket\kappa\rrbracket$.
4. *If* $\Gamma \vdash_K c \preceq c'$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu \llbracket c\rrbracket \preceq \llbracket c'\rrbracket$.
5. *If* $\Gamma \vdash_K e : c$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu \llbracket e\rrbracket\ \mathit{in}\ \overline{\llbracket c\rrbracket}$.
6. *If* $\Gamma \vdash_K e \downarrow c$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu \llbracket e\rrbracket\ \mathit{in}\ \llbracket c\rrbracket$.
7. *If* $\Gamma \vdash_K \sigma \preceq \sigma'$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu (\llbracket\sigma\rrbracket\ \mathit{in}\ \mathbb{U}_{\mathit{level}(\sigma)+1} \wedge \llbracket\sigma'\rrbracket\ \mathit{in}\ \mathbb{U}_{\mathit{level}(\sigma')+1} \wedge \llbracket\sigma\rrbracket \preceq \llbracket\sigma'\rrbracket)$.
8. *If* $\Gamma \vdash_K m : \sigma$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu (\llbracket m\rrbracket\ \mathit{in}\ \overline{\llbracket\sigma\rrbracket} \wedge \llbracket m\rrbracket_{\mathrm{c}}\ \mathit{in}\ \llbracket\sigma\rrbracket_{\mathrm{c}})$.
9. *If* $\Gamma \vdash_K m \downarrow \sigma$ *then* $\llbracket\Gamma\rrbracket \vdash_\nu \llbracket m\rrbracket\ \mathit{in}\ \llbracket\sigma\rrbracket$.

Proof

By induction on the derivations of the λ^K judgements.

We may observe two immediate consequences of the soundness theorem. One is the desirable property of type preservation: evaluation does not change the type of a program. Figure 4.3 gives a small-step evaluation relation for the Nuprl type theory (denoted by $t \mapsto t'$ when t evaluates in one step to t'). Type preservation of λ^K (Corollary 3.3) follows directly from soundness and type preservation of Nuprl (Proposition 3.2).

$$
\begin{array}{rcl}
[\![m]\!] & \stackrel{\text{def}}{=} & \langle [\![m]\!]_c, [\![m]\!]_r \rangle \\
[\![s]\!]_c & \stackrel{\text{def}}{=} & s_c \\
[\![s]\!]_r & \stackrel{\text{def}}{=} & s_r \\
[\![\langle c \rangle]\!]_c & \stackrel{\text{def}}{=} & [\![c]\!] \\
[\![\langle c \rangle]\!]_r & \stackrel{\text{def}}{=} & \star \\
[\![\langle\!\langle e \rangle\!\rangle]\!]_c & \stackrel{\text{def}}{=} & \star \\
[\![\langle\!\langle e \rangle\!\rangle]\!]_r & \stackrel{\text{def}}{=} & [\![e]\!] \\
[\![\lambda s{:}\sigma.m]\!]_c & \stackrel{\text{def}}{=} & \lambda s_c. [\![m]\!]_c \\
[\![\lambda s{:}\sigma.m]\!]_r & \stackrel{\text{def}}{=} & \lambda s_c.\, \lambda s_r.\, [\![m]\!]_r \\
[\![m_1 m_2]\!]_c & \stackrel{\text{def}}{=} & [\![m_1]\!]_c [\![m_2]\!]_c \\
[\![m_1 m_2]\!]_r & \stackrel{\text{def}}{=} & \mathit{let}\ x = [\![m_2]\!]_r\ \mathit{in}\ [\![m_1]\!]_r [\![m_2]\!]_c x \\
 & & \text{(where } x \text{ does not appear free in } m_1,\, m_2\text{)} \\
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_c & \stackrel{\text{def}}{=} & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ [\![m_1]\!]_c \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ [\![m_n]\!]_c \\
 & & \quad \mathit{else}\ \star \\
 & & \text{(where } a \text{ does not appear free in } m_i\text{)} \\
[\![\{\ell_1 = m_1, \ldots, \ell_n = m_n\}]\!]_r & \stackrel{\text{def}}{=} & \mathit{let}\ x_1 = [\![m_1]\!]_r\ \mathit{in} \\
 & & \vdots \\
 & & \mathit{let}\ x_n = [\![m_n]\!]_r\ \mathit{in} \\
 & & \lambda a.\ \mathit{if}\ a =_A \ell_1\ \mathit{then}\ x_1 \\
 & & \quad \vdots \\
 & & \quad \mathit{else\ if}\ a =_A \ell_n\ \mathit{then}\ x_n \\
 & & \quad \mathit{else}\ \star \\
 & & \text{(where } x_i \text{ does not appear free in } m_i\text{)} \\
[\![\pi_\ell(m)]\!]_c & \stackrel{\text{def}}{=} & [\![m]\!]_c \ell \\
[\![\pi_\ell(m)]\!]_r & \stackrel{\text{def}}{=} & [\![m]\!]_r \ell \\
[\![m : \sigma]\!]_c & \stackrel{\text{def}}{=} & [\![m]\!]_c \\
[\![m : \sigma]\!]_r & \stackrel{\text{def}}{=} & [\![m]\!]_r
\end{array}
$$

Figure 3.6: Embedding Modules

Proposition 3.2 *If $\vdash_\nu t$ in T and $t \mapsto^* t'$ then $\vdash_\nu t'$ in T.*

Proof

Direct from Corollary 4.4.

Corollary 3.3 (Type Preservation) *If $\vdash_K e : \tau$ and $[\![e]\!] \mapsto^* t$ then $\vdash_\nu t$ in $[\![\tau]\!]$.*

Another consequence of the soundness theorem is that the phase distinction [16, 46] is respected in λ^K: all type expressions converge and therefore types may be computed in a compile-time phase. This is expressed by Corollary 3.4:

Corollary 3.4 (Phase Distinction) *If $\vdash_K c : \kappa$ then there exists canonical t such that $[\![c]\!] \mapsto^* t$.*

Proof

For any well-formed λ^K kind κ, the embedded kind $[\![\kappa]\!]$ can easily be shown to be a total type. (Intuitively, every type is total unless it is constructed using the partial type constructor, which is not used in the embedding of kinds.) The conclusion follows directly.

3.5 Prospects for Extension

In order to embed λ^K into type theory, I abandoned impredicativity. This was because the standard semantics for Nuprl (discussed in Section 4.4) does not support it. However, Mendler [69] has developed a semantic model of Nuprl enhanced with recursive types and some impredicative polymorphism. In that type theory, it is entirely straightforward to handle second-order impredicativity. I have not used Mendler's type theory in this thesis because its semantics is very complicated, and because it is not clear how easily it can be extended to support partial types. An alternative type theory to be explored is the Calculus of Constructions [29, 28], which also supplies impredicative features and could likely support the semantics discussed in this chapter.

Another important avenue for future work is to extend the semantics in this chapter to explain stateful computation. One promising device for doing this is to encode stateful computations as monads [82, 60], but this raises two difficulties. In order to encode references in monads, all expressions that may side-effect the store must take the store as an argument. The problem is how to assign a type to the store. Since side-effecting functions may be in the store themselves, the store must be typed using a recursive type, and since side-effecting expressions take the store as an argument, that recursive type will include *negative* occurrences of the variable of recursion. Mendler's type theory may express recursive types with only positive occurrences, but to allow negative occurrences is an open problem.[9]

Finally, a particularly compelling direction is to extend the semantics to account for objects, and that turns out to require only the same mechanisms discussed above. In a weak sense, this semantics can already support objects; the existential object encoding of Pierce and Turner [83] uses only constructs available within the type theory used here. Unfortunately, that encoding is not practical in a predicative system, because it involves quantification over the types of an object's hidden instance variables. That quantification results in objects always belonging to a universe one level higher than their underlying code, which prevents such object from being first-class.[10] However, in an impredicative type theory, Pierce and Turner's object encoding can be used quite satisfactorily. In a type theory that additionally supplies recursive types (with negative occurrences), a variety of other object encodings become possible as well [15, 13, 3, 31, 14]. Alternatively, the object encoding of Hickey [50] works entirely within the existing Nuprl type theory and shows promise of being extendable to a practical object system.

3.6 Conclusions

I have shown how to give a type-theoretic semantics to an expressive programming calculus that supports modular and object-oriented features. This semantics makes it possible to use formal type-theoretic reasoning about programs and programming languages without informal embeddings and without sacrificing core expressiveness of the programming language.

Formal reasoning aside, embedding programming languages into type theory allows a researcher to bring the full power of type theory to bear on a programming problem. For example, in Crary [30] I use a type-theoretic interpretation to expose the relation of power kinds to a nonconstructive set type. Adjusting this interpretation to make the power kind constructive

[9]See Birkedal and Harper [11] for a promising approach that may lead to a solution of this problem.

[10]However, in some contexts it is not necessary for objects to be first class. For example, Jackson [57] independently used an encoding essentially the same as Pierce and Turner's to implement computational abstract algebra. In that context, algebras were rarely intermingled with the elements of an algebra, and when they were, an increase in the universe level was acceptable.

results in the proof-passing technique used to implement higher-order coercive subtyping in KML.

Furthermore, the simplicity of the semantics makes it attractive to use as a mathematical model similar in spirit, if not in detail, to the Scott-Strachey program [90]. This semantics works out so neatly because type theory provides built-in structure well-suited for analysis of programming. Most importantly, type theory provides structured data and an intrinsic notion of computation. Non-type-theoretic models of type theory can expose the "scaffolding" when one desires the details of how that structure may be implemented (Section 4.4).

As a theory of structured data and computation, type theory is itself a very expressive programming language. Practical programming languages are less expressive, but offer properties that foundational type theory does not, such as decidable type checking. I suggest that it is profitable to take type theory as a foundation for programming, and to view practical programming languages as *tractable approximations* of type theory. The semantics in this chapter illustrates how to formalize these approximations. This view not only helps to *explain* programming languages and their features, as I have done here, but also provides a greater insight into how we can bring more of the expressiveness of type theory into programming languages.

$t \in T$	iff $t = t \in T$
T type	iff $T = T$
$T_1 = T_2$	iff $\exists T_1', T_2'. (T_1 \Downarrow T_1') \wedge (T_2 \Downarrow T_2') \wedge (T_1' = T_2')$
$t_1 = t_2 \in T$	iff $\exists t_1', t_2', T'. (t_1 \Downarrow t_1') \wedge (t_2 \Downarrow t_2') \wedge (T \Downarrow T') \wedge (t_1' = t_2' \in T')$
$\neg(t_1 = t_2 \in \mathit{Void})$	
$a = a' \in \mathit{Atom}$ (a, a' atoms)	iff $a \equiv a'$
$n = n' \in \mathbb{Z}$ (n, n' integers)	iff $n \equiv n'$
$t = t' \in \mathbb{E}$	iff $t\downarrow \wedge t'\downarrow$
$\mathit{inj}_1(a) = \mathit{inj}_1(a') \in A + B$	iff $A + B$ type $\wedge\, a = a' \in A$
$\mathit{inj}_2(b) = \mathit{inj}_2(b') \in A + B$	iff $A + B$ type $\wedge\, b = b' \in B$
$\langle a, b\rangle = \langle a', b'\rangle \in \Sigma x{:}A.B$	iff $\Sigma x{:}A.B$ type $\wedge\, (a = a' \in A) \wedge (b = b' \in B[a/x])$
$\lambda x.b = \lambda x.b' \in \Pi x{:}A.B$	iff $\Pi x{:}A.B$ type $\wedge\, \forall a = a' \in A.\, b[a/x] = b'[a'/x] \in B[a/x]$
$\lambda x.b = \lambda x.b' \in \{f \mid x{:}A \to B\}$	iff $\{f \mid x{:}A \to B\}$ type $\wedge\, \forall a = a' \in A.\, b[a/x] = b'[a'/x] \in B[\lambda x.b, a/f, x]$
$t = t' \in \overline{T}$	iff $\overline{T}$ type $\wedge\, (t\downarrow \Leftrightarrow t'\downarrow) \wedge (t\downarrow \Rightarrow t = t' \in T)$
$a = a' \in \{x : A \mid B\}$	iff $\{x : A \mid B\}$ type $\wedge\, a = a' \in A \wedge \exists b.\, b \in B[a/x]$
$a = a' \in xy{:}A//B$	iff $(xy{:}A//B)$ type $\wedge\, a \in A \wedge a' \in A \wedge \exists b.\, b \in B[a, a'/x, y]$
$\star \in (a = a'\ \mathit{in}\ A)$	iff $(a = a'\ \mathit{in}\ A)$ type $\wedge\, (a = a' \in A)$
$\star \in (A \preceq B)$	iff $(A \preceq B)$ type $\wedge\, \forall a = a' \in A.\, a = a' \in B$
$\star \in (t \leq t')$	iff $(t \leq t')$ type $\wedge\, \exists n, n'.\, t \Downarrow n \wedge t' \Downarrow n' \wedge n \leq n'$
$\star \in (a\ \mathit{in!}\ A)$	iff $(a\ \mathit{in!}\ A)$ type $\wedge\, a\downarrow$
$\star \in (A$ total$)$	iff $(A$ total$)$ type $\wedge\, \forall t \in A.\, t\downarrow$
$\star \in (A$ admiss$)$	iff $(A$ admiss$)$ type $\wedge$ Adm(A)

For $T \simeq T'$ iff $T = T'$, and for $T \simeq T'$ iff $T = T' \in \mathbb{U}_i$:

$\mathit{Void} \simeq \mathit{Void}$ $\mathit{Atom} \simeq \mathit{Atom}$ $\mathbb{Z} \simeq \mathbb{Z}$ $\mathbb{E} \simeq \mathbb{E}$	
$A + B \simeq A' + B'$	iff $A \simeq A' \wedge B \simeq B'$
$\Sigma x{:}A.B \simeq \Sigma x{:}A'.B'$	iff $A \simeq A' \wedge \forall a = a' \in A.\, B[a/x] \simeq B'[a'/x]$
$\Pi x{:}A.B \simeq \Pi x{:}A'.B'$	iff $A \simeq A' \wedge \forall a = a' \in A.\, B[a/x] \simeq B'[a'/x]$
$\{f \mid x{:}A \to B\} \simeq \{f \mid x{:}A' \to B'\}$	iff $A \simeq A' \wedge$
	$\exists P, <.\, \forall a_1 = a_1' \in A.\, \forall a_2 = a_2' \in A.\, P[a_1, a_2/x, y] \simeq P[a_1', a_2'/x, y] \wedge$
	$\forall a, a'.\, a < a' \Leftrightarrow (\exists e.\, e \in P[a, a'/x, y]) \wedge <$ is well-founded $\wedge$
	$\forall a = a' \in A.\, \forall t = t' \in \{f \mid x{:}\{e : A \mid P[e, a/x, y]\} \to B\}.$
	$B[t, a/f, x] \simeq B'[t', a'/f, x]$
$\overline{T} \simeq \overline{T'}$	iff $T \simeq T' \wedge \forall t \in T.\, t\downarrow$
$\{x : A \mid B\} \simeq \{x : A' \mid B'\}$	iff $A \simeq A' \wedge \forall a = a' \in A.\, B[a/x] \simeq B[a'/x] \wedge$
	$\forall a = a' \in A.\, B'[a/x] \simeq B'[a'/x] \wedge$
	$\exists t.\, t \in \Pi x{:}A.\, B \to B' \wedge \exists t.\, t \in \Pi x{:}A.\, B' \to B$
$(xy{:}A//B) \simeq (xy{:}A'//B')$	iff $A \simeq A' \wedge$
	$\forall a_1 = a_1' \in A.\, \forall a_2 = a_2' \in A.\, B[a_1, a_2/x, y] \simeq B[a_1', a_2'/x, y] \wedge$
	$\forall a_1 = a_1' \in A.\, \forall a_2 = a_2' \in A.\, B'[a_1, a_2/x, y] \simeq B'[a_1', a_2'/x, y] \wedge$
	$\exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, B \to B' \wedge \exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, B' \to B \wedge$
	$\exists t.\, t \in \Pi x{:}A.\, B[x/y] \wedge \exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, B \to B[y, x/x, y] \wedge$
	$\exists t.\, t \in \Pi x{:}A.\, \Pi y{:}A.\, \Pi z{:}A.\, B \to B[y, z/x, y] \to B[z/y]$
$(a_1 = a_2\ \mathit{in}\ A) \simeq$	
$(a_1' = a_2'\ \mathit{in}\ A')$	iff $A \simeq A' \wedge a_1 = a_1' \in A \wedge a_2 = a_2' \in A$
$(A \preceq B) \simeq (A' \preceq B')$	iff $A \simeq A' \wedge B \simeq B'$
$(t_1 \preceq t_2) \simeq (t_1' \preceq t_2')$	iff $t_1 = t_1' \in \mathbb{Z} \wedge t_2 = t_2' \in \mathbb{Z}$
$(a\ \mathit{in!}\ A) \simeq (a'\ \mathit{in!}\ A')$	iff $A \simeq A' \wedge a = a' \in \overline{A}$
$(A$ total$) \simeq (A'$ total$)$	iff $A \simeq A'$
$(A$ admiss$) \simeq (A'$ admiss$)$	iff $A \simeq A'$
$\mathbb{U}_i$ type	
$\mathbb{U}_i \in \mathbb{U}_j$	iff $i < j$

Figure 4.5: Type Specifications

Chapter 5

Admissibility

One of the earliest logical theorem provers was the LCF system [38], based on the logic of partial computable functions [88, 89]. Although LCF enjoyed many groundbreaking successes, one problem it faced was that, although it supported a natural notion of *partial* function, it had difficulty expressing the notion of a *total* function. Later theorem provers based on constructive type theory, such as Nuprl [19], based on Martin-Löf type theory [68], and Coq [8], based on the Calculus of Constructions [29], faced the opposite problem; they had a natural notion of total functions, but had difficulty dealing with partial functions. The lack of partial functions seriously limited the scope of those theorem provers, because it made them unable to reason about programs in real programming languages where recursion does not always necessarily terminate.

This problem may be addressed in type theory by the partial type discussed in Chapter 4. In a partial type theory, recursively defined objects may be typed using the *fixpoint principle:* if f has type $\overline{T} \rightarrow \overline{T}$ then $\mathit{fix}(f)$ has type $\overline{T}$. However, the fixpoint principle is not valid for every type T; it is only valid for types that are *admissible.* This phenomenon was not unknown to LCF; LCF used the related device of fixpoint induction, which was valid only for admissible predicates. When the user attempted to invoke fixpoint induction, the system would automatically check that the goal was admissible using a set of syntactic rules [55].

Despite their obvious uses in program analysis, partial types have seen little use in theorem proving systems [25, 9, 7]. This is due in large part to two problems: partial type extensions have been developed only for fragments of type theory that do not include equality, and too few types have been known to be admissible. I addressed the former problem in Chapter 4; in this chapter I address the latter.

Smith [92] gave a significant class of admissible types for a Nuprl-like theory, but his class required product types to be non-dependent. The type $\Sigma x{:}A.B$ (where x appears free in B) was explicitly excluded. Later, Smith [91] extended his class to include some dependent products $\Sigma x{:}A.B$, but disallowed any free occurrences of x to the left of an arrow in B. Partial type extensions to Coq [7] were also restrictive, assuming function spaces to be the only type constructor. These restrictions are quite strong; dependent products are used in encodings of modules [65], objects [83], algebras [57], and even such simple devices as variant records. Furthermore, ruling out dependent products disallows reasoning using fixpoint induction as in LCF. (This is explained further in Section 5.1.) Finally, the restriction is particularly unsatisfying since most types used in practice do turn out to be admissible, and may be shown so by metatheoretical reasoning.

In this chapter I present a very wide class of admissible types using two devices, a condition called *predicate-admissibility* and a monotonicity condition. In particular, many dependent products may be shown to be admissible. Predicate-admissibility relates to when the limit of a chain of type approximations contains certain terms, whereas admissibility relates to the membership of a single type. Monotonicity is a simpler condition that will be useful for showing types admissible that do not involve partiality.

5.1 The Fixpoint Principle

The central issue of this chapter is the *fixpoint principle:*

$$f \in \overline{T} \to \overline{T} \Rightarrow \mathit{fix}(f) \in \overline{T}$$

The fixpoint principle allows us to type recursively defined objects, such as recursive functions. Unfortunately, unlike in programming languages, where the principle can usually be invoked on arbitrary types, expressive type theories such as the one in this thesis contain types for which the fixpoint principle is not valid. I shall informally say that a type is *admissible* if the fixpoint principle is valid for that type and give a formal definition in Section 5.3. To make maximum use of a partial type theory, one wants as large a class of admissible types as possible.

In Section 5.3 I will explore two wide classes of admissible types, one derived from a *predicate-admissibility* condition and another derived from a monotonicity condition. But first, it is worthwhile to note that there are indeed inadmissible types:

Theorem 5.1 *There exist inadmissible types.*

Proof Sketch

This example is due to Smith [92]. Recall that $\mathbb{N} = \{n : \mathbb{Z} \mid 0 \leq n\}$. Let T be the type of functions that do not halt for all inputs, and let f be the function that halts on zero, and on any other n immediately recurses with $n-1$. This is formalized as follows:

$$\begin{array}{rcl} T & \stackrel{\text{def}}{=} & \Sigma h{:}(\mathbb{N} \to \overline{\mathbb{N}}).\,((\Pi x{:}\mathbb{N}.\, h\,x\ \mathit{in!}\ \mathbb{N}) \to \mathit{Void}) \\ f & \stackrel{\text{def}}{=} & \lambda p.\langle \lambda x.\, \mathit{if}\ x \leq_Z 0\ \mathit{then}\ 0\ \mathit{else}\ \pi_1(p)(x-1), \lambda y.\star\rangle \end{array}$$

Intuitively, any finite approximation of $\mathit{fix}(f)$ will recurse some limited number of times and then give up, placing it in T, but $\mathit{fix}(f)$ will halt for every input, excluding it from T. Formally, the function f has type $\overline{T} \to \overline{T}$, but $\mathit{fix}(f) \notin \overline{T}$. (The proof of these two facts appears in Appendix C.) Therefore T is not admissible.

5.2 Computational Lemmas

Before presenting my main results in Section 5.3, I first require some lemmas about the computational behavior of the fixpoint operator. The central result is that $\mathit{fix}(f)$ is the least upper bound of the finite approximations $\bot, f(\bot), f(f(\bot)), \ldots$ with regard to a computational approximation relation defined below. The compactness of fix (if $\mathit{fix}(f)$ halts then one of its finite approximations halts) will be a simple corollary of this result. However, the proof of the least upper bound theorem is considerably more elegant than most proofs of compactness.

5.2.1 Computational Approximation

For convenience, throughout this section we will frequently consider terms using a unified representation scheme for terms: A term is either a variable or a compound term $\theta(x_{11} \cdots x_{1k_1}.t_1, \ldots, x_{n1} \cdots x_{nk_n}.t_n)$ where the variables $x_{i1}, \ldots, x_{ik_i}$ are bound in the subterm t_i. For example, the term $\Pi x{:}T_1.T_2$ is represented $\Pi(T_1, x.T_2)$ and the term $\langle t_1, t_2 \rangle$ is represented $\langle\rangle(t_1, t_2)$.

Informally speaking, a term t_1 approximates the term t_2 when: if t_1 converges to a canonical form then t_2 converges to a canonical form with the same outermost operator, and the subterms of t_1's canonical form approximate the corresponding subterms of t_2's canonical form. The formal definition appears below and is due to Howe [53].[1] Following Howe, when R is a binary relation on closed terms, I adopt the convention extending R to possibly open terms that if t and t' are possibly open then $t \mathrel{R} t'$ if and only if $\sigma(t) \mathrel{R} \sigma(t')$ for every substitution σ such that $\sigma(t)$ and $\sigma(t')$ are closed.

Definition 5.2 (Computational Approximation)

- *Let R be a binary relation on closed terms and suppose e and e' are closed. Then $e \mathrel{C(R)} e'$ exactly when if $e \Downarrow \theta(\vec{x}_1.t_1, \ldots, \vec{x}_n.t_n)$ then there exists some closed $e'' = \theta(\vec{x}_1.t'_1, \ldots, \vec{x}_n.t'_n)$ such that $e' \Downarrow e''$ and $t_i \mathrel{R} t'_i$.*
- *$e \leq_0 e'$ whenever e and e' are closed.*
- *$e \leq_{i+1} e'$ if and only if $e \mathrel{C(\leq_i)} e'$*
- *$e \leq e'$ if and only if $e \leq_i e'$ for every i*

Definition 5.3 (Computational Equivalence) *The terms e and e' are computationally equivalent ($e \sim e'$) if and only if $e \leq e'$ and $e' \leq e$.*

The following are facts about computational approximation that will be used without explicit reference. The first two follow immediately from the definition, the third is easy using determinism (Proposition 4.2) and the last is proven using Howe's method [53].

Proposition 5.4

- *$\leq$ and $\leq_i$ are reflexive and transitive.*
- *If $t \mapsto t'$ then $t' \leq t$ and $t' \leq_i t$.*
- *If $t \mapsto t'$ then $t \leq t'$ and $t \leq_i t'$.*

Lemma 5.5 (Congruence) *If $e \leq e'$ and $t \leq t'$ then $e[t/x] \leq e'[t'/x]$.*

[1]Howe's definition actually differs slightly from the one here; he defines $\leq$ as the greatest fixed point of the operator C. It is not difficult to show that the two definitions are equivalent, as long as the computation system is deterministic (Proposition 4.2). If the computation system is nondeterministic, the definition here fails to be a fixed point, and the more complicated greatest fixed point definition must be employed.

5.2.2 Finite Approximations

With this notion of computational approximation in hand, we may now show that the terms $\bot, f\,\bot, f(f\,\bot), \ldots$ form a chain of approximations to the term $\mathit{fix}(f)$. Let $\bot$ be the divergent term $\mathit{fix}(\lambda x.x)$. Since $\bot$ never converges, $\bot \leq t$ for any term t. Let f^i be defined as follows:

$$\begin{aligned} f^0 &\overset{\text{def}}{=} \bot \\ f^{i+1} &\overset{\text{def}}{=} f(f^i) \end{aligned}$$

Certainly $f^0 \leq f^1$, since $f^0 \equiv \bot$. By congruence, $f(f^0) \leq f(f^1)$, and thus $f^1 \leq f^2$. Similarly, $f^i \leq f^{i+1}$ for all i. Thus $f^0, f^1, f^2, \ldots$ forms a chain; I now wish to show that $\mathit{fix}(f)$ is an upper bound of the chain. Certainly $f^0 \leq \mathit{fix}(f)$. Suppose $f^i \leq \mathit{fix}(f)$. By congruence $f(f^i) \leq f(\mathit{fix}(f))$. Thus, since $\mathit{fix}(f) \mapsto f(\mathit{fix}(f))$, it follows that $f^{i+1} \equiv f(f^i) \leq f(\mathit{fix}(f)) \leq \mathit{fix}(f)$. By induction it follows that $\mathit{fix}(f)$ is an upper bound of the chain. The following corollary follows from congruence and the definition of approximation:

Corollary 5.6 *If there exists j such that $e[f^j/x]\!\downarrow$ then $e[\mathit{fix}(f)/x]\!\downarrow$. Moreover, the canonical forms of $e[f^j/x]$ and $e[\mathit{fix}(f)/x]$ must have the same outermost operator.*

5.2.3 Least Upper Bound Theorem

In this section I summarize the proof of the least upper bound theorem. To begin, we need a lemma stating a general property of evaluation. Lemma 5.7 captures the intuition that closed, noncanonical terms that lie within a term being evaluated are not destructed; they either are moved around unchanged (the lemma's first case) or are evaluated in place with the surrounding term left unchanged (the lemma's second case). The variable x indicates positions where the term of interest is found and, in the second case, the variable y indicates which of those positions, if any, is about to be evaluated.

Lemma 5.7 *If $e_1[t/x] \mapsto e_2$, and $e_1[t/x]$ is closed, and t is closed and noncanonical, then either*

- *there exists e_2' such that for any closed t', $e_1[t'/x] \mapsto e_2'[t'/x]$, or*
- *there exist e_1' and t' such that $e_1 \equiv e_1'[x/y]$, $t \mapsto t'$ and for any closed t'', $e_1'[t'', t/x, y] \mapsto e_1'[t'', t'/x, y]$.*

It is worthwhile to note that Proposition 4.2 and Lemmas 5.5 and 5.7 are the only properties of evaluation used in the proof of the least upper bound theorem, and that these properties are true in computational systems with considerable generality. Consequently, the theorem may be used in a variety of applications beyond the computational system of this thesis.

Lemma 5.8 shows that *fix* terms may be effectively simulated in any particular computation by sufficiently large finite approximations. The lemma is simplified by using computational approximation instead of evaluation for the simulation, which makes it unnecessary to track which of the approximations are unfolded and which are not, an issue that often complicates compactness proofs.

Lemma 5.8 (Simulation) *For all f, e_1 and e_2 (where f is closed and x is the only free variable of e_1), there exist j and e_2' such that if $e_1[\mathit{fix}(f)/x] \mapsto^* e_2$ then $e_2 \equiv e_2'[\mathit{fix}(f)/x]$ and for all $k \geq j$, $e_2'[f^{k-j}/x] \leq e_1[f^k/x]$.*

Theorem 5.9 (Least Upper Bound) *For all f, t and e (where f is closed), if $\forall j.\, e[f^j/x] \leq t$, then $e[\mathit{fix}(f)/x] \leq t$.*

Proof Sketch

By induction on l that $e[\mathit{fix}(f)/x] \leq_l t$. (The complete proof in Appendix C addresses free variables.) Suppose $e[\mathit{fix}(f)/x]$ evaluates to some canonical form $e'[\mathit{fix}(f)/x]$ (where e' is chosen by Lemma 5.8). Let e' be of the form $\theta(\vec{x}_1.t_1, \ldots, \vec{x}_n.t_n)$. Using Lemma 5.8, the assumption $\forall k.\, e[f^k/x] \leq t$, and transitivity, we may show that $e'[f^j/x] \leq t$ for all j. Therefore $t \Downarrow \theta(\vec{x}_1.t'_1, \ldots, \vec{x}_n.t'_n)$ and $t_i[f^j/x] \leq t'_i$ for all j. Now, by induction, $t_i[\mathit{fix}(f)/x] \leq_l t'_i$. Thus $e[\mathit{fix}(f)/x] \leq_{l+1} t$.

There are two easy corollaries to the least upper bound theorem. One is that $\mathit{fix}(f)$ is the least fixed point of f, and the other is compactness.

Corollary 5.10 (Least Fixed Point) *For all closed f and t, if $f(t) \leq t$ then $\mathit{fix}(f) \leq t$.*

Proof

Certainly $f^0 \equiv \bot \leq t$. Then $f^1 \equiv f(f^0) \leq f(t) \leq t$. Similarly, by induction, $f^j \leq t$ for any j. Therefore $\mathit{fix}(f) \leq t$ by Theorem 5.9. □

Corollary 5.11 (Compactness) *If f is closed and $e[\mathit{fix}(f)/x]\downarrow$ then there exists some j such that $e[f^j/x]\downarrow$. Moreover, the canonical forms of $e[\mathit{fix}(f)/x]$ and $e[f^j/x]$ must have the same outermost operator.*

Proof

Suppose there does not exist j such that $e[f^j/x]\downarrow$. Then $e[f^j/x] \leq \bot$ for all j. By Theorem 5.9, $e[\mathit{fix}(f)/x] \leq \bot$. Therefore $e[\mathit{fix}(f)/x]$ does not converge, but this contradicts the assumption,[2] so there must exist j such that $e[f^j/x]\downarrow$. Since $e[f^j/x] \leq e[\mathit{fix}(f)/x]$, the canonical forms of $e[f^j/x]$ and $e[\mathit{fix}(f)/x]$ must have the same outermost operator. □

5.3 Admissibility

I am now ready to begin specifying some wide classes of types for which the fixpoint principle is valid. First we define admissibility. The simple property of validating the fixpoint principle is too specific to allow any good closure conditions to be shown easily, so we generalize a bit to define admissibility. A type is *admissible* if the upper bound $t[\mathit{fix}(f)]$ of an approximation chain $t[f^0], t[f^1], t[f^2], \ldots$ belongs to the type whenever a cofinite subset of the chain belongs to the type. This is formalized as Definition 5.13, but first I define some convenient notation.

Notation 5.12 *For any natural number j, the notation $t^{[j]_f}$ means $t[f^j/w]$, and the notation $t^{[\omega]_f}$ means $t[\mathit{fix}(f)/w]$. Also, the f subscript is dropped when the intended term f is unambiguously clear.*

Definition 5.13 *A type T is* admissible *(abbreviated* Adm(T)*) if:*

$$\forall f, t, t'.\, (\exists j. \forall k \geq j.\, t^{[k]} = t'^{[k]} \in T) \Rightarrow t^{[\omega]} = t'^{[\omega]} \in T$$

[2]Although this proof is non-constructive, a slightly less elegant constructive proof may be derived directly from Lemma 5.8.

As expected, admissibility is sufficient to guarantee applicability of the fixpoint principle:

Theorem 5.14 *For any T and f, if T is admissible and $f = f' \in \overline{T} \to \overline{T}$ then $\mathit{fix}(f) = \mathit{fix}(f') \in \overline{T}$.*

Proof

$\overline{T}$ type since $\overline{T} \to \overline{T}$ type. Note that $f^j = f'^j \in \overline{T}$ for every j. Suppose $\mathit{fix}(f)\downarrow$. By compactness, $f^j\downarrow$ for some j. Since $f^j = f'^j \in \overline{T}$, it follows that $f'^j\downarrow$ and thus $\mathit{fix}(f')\downarrow$ by Corollary 5.6. Similarly $\mathit{fix}(f')\downarrow$ implies $\mathit{fix}(f)\downarrow$. It remains to show that $\mathit{fix}(f) = \mathit{fix}(f') \in T$ when $\mathit{fix}(f)\downarrow$. Suppose again that $\mathit{fix}(f)\downarrow$. As before, there exists j such that $f^j\downarrow$ by compactness. Hence $f^j = f'^j \in T$. Since T is admissible, $\mathit{fix}(f) = \mathit{fix}(f') \in T$. □

A number of closure conditions exist on admissible types and are given in Lemma 5.15. Informally, basic compound types other than dependent products are admissible so long as their component types in positive positions are admissible. Base types—natural numbers, convergence types, and (for this lemma only) equality types—are always admissible. These are essentially the admissible types of Smith [92], except that for a function type to be admissible Smith required that its domain type be admissible.

Lemma 5.15

- $\mathrm{Adm}(A + B)$ *if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B)$
- $\mathrm{Adm}(\Pi x{:}A.B)$ *if* $\forall a \in A.\, \mathrm{Adm}(B[a/x])$
- $\mathrm{Adm}(A \times B)$ *if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B)$
- $\mathrm{Adm}(\mathit{Void})$, $\mathrm{Adm}(\mathit{Atom})$, $\mathrm{Adm}(\mathbb{Z})$ *and* $\mathrm{Adm}(\mathbb{E})$
- $\mathrm{Adm}(a = a' \mathit{\ in\ } A)$
- $\mathrm{Adm}(a \leq a')$
- $\mathrm{Adm}(\overline{A})$ *if* $\mathrm{Adm}(A)$
- $\mathrm{Adm}(a \mathit{\ in!\ } A)$

Proof

The proof follows the same lines as Smith's proof, except that handling equality adds a small amount of complication to the proof. I show the function case by way of example.

Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \Pi x{:}A.B$. I need to show that $t^{[\omega]} = t'^{[\omega]} \in \Pi x{:}A.B$. Since $\Pi x{:}A.B$ is inhabited it is a type. Both $t^{[j]}$ and $t'^{[j]}$ converge to lambda abstractions, so, by Corollary 5.6, $t^{[\omega]} \Downarrow \lambda x.b$ and $t'^{[\omega]} \Downarrow \lambda x.b'$ for some terms b and b'. Suppose $a = a' \in A$. To get that $b[a/x] = b'[a'/x] \in B[a/x]$ it suffices to show that $t^{[\omega]}a = t'^{[\omega]}a' \in B[a/x]$. Since $\mathrm{Adm}(B[a/x])$, it suffices to show that $\forall k \geq j.\, t^{[k]}a = t'^{[k]}a' \in B[a/x]$, which follows from the supposition. □

Unfortunately, Lemma 5.15 can show the admissibility of a product space only if it is *non-dependent.* Dependent products do not have an admissibility condition similar to that of dependent functions. This reason for this is as follows: Admissibility states that a *single fixed type* contains the limit of an approximation chain if it contains a cofinite subset of that chain. For functions, disjoint union, partial types, and non-dependent products it is possible to decompose prospective members in such a way that admissibility may be applied to a single type (such as the type $B[a/x]$ used in the proof of Lemma 5.15). In contrast, for a dependent product, the right-hand term's desired type depends upon the left-hand term, which is changing at the same time as the right-hand term. Consequently, there is no single type into which to place the right-hand term.

However, understanding the problem with dependent products suggests a solution, to generalize the definition of admissibility to allow the type to vary. This leads to the notion of *predicate-admissibility* that I discuss in the next section.

5.3.1 Predicate-Admissibility

Definition 5.16 *A type T is* predicate-admissible *for x in S (abbreviated* $\mathrm{Adm}(T \mid x : S)$*) if:*

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j.\, \forall k \geq j.\, e^{[k]} \in S \wedge t^{[k]} = t'^{[k]} \in T[e^{[k]}/x]) \Rightarrow t^{[\omega]} = t'^{[\omega]} \in T[e^{[\omega]}/x]$$

The term "predicate-admissibility" stems from its similarity to the notion of admissibility of predicates in domain theory (and LCF). If one ignores the inhabiting terms t and t', which may be seen as evidences of the truth of the predicate $T[\,]$, then predicate-admissibility is saying $T[e^{[\omega]}]$ if $T[e^{[k]}]$ for all k greater than some j. This is precisely the notion of admissibility of predicates in domain theory. Indeed, the results here were influenced by the work of Igarashi [55], who established conditions on admissibility of domain-theoretic predicates.

To show the admissibility of a dependent product type, it is sufficient to show predicate-admissibility of the right-hand side (along with admissibility of the left):

Lemma 5.17 *The type $\Sigma x{:}A.B$ is admissible if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B \mid x : A)$.

Proof

Let f, t and t' be arbitrary. Suppose j is such that $\forall k \geq j.\, t^{[k]} = t'^{[k]} \in \Sigma x{:}A.B$. It is necessary to show that $t^{[\omega]} = t'^{[\omega]} \in \Sigma x{:}A.B$. Since $\Sigma x{:}A.B$ is inhabited it is a type. Both $t^{[j]}$ and $t'^{[j]}$ converge to pairs, so, by Corollary 5.6, $t^{[\omega]} \Downarrow \langle a, b\rangle$ and $t'^{[\omega]} \Downarrow \langle a', b'\rangle$ for some terms a, b, a' and b'. To get that $a = a' \in A$ it suffices to show that $\pi_1(t^{[\omega]}) = \pi_1(t'^{[\omega]}) \in A$. Since $\mathrm{Adm}(A)$, it suffices to show that $\forall k \geq j.\, \pi_1(t^{[k]}) = \pi_1(t'^{[k]}) \in A$, which follows from the supposition.

To get that $b = b' \in B[a/x]$ (the interesting part), it suffices to show that $\pi_2(t^{[\omega]}) = \pi_2(t'^{[\omega]}) \in B[\pi_1(t^{[\omega]})/x]$. Since $\mathrm{Adm}(B \mid x : A)$, it suffices to show that $\pi_1(t^{[\omega]}) \in A$, which has already been shown, and $\forall k \geq j.\, \pi_1(t^{[k]}) \in A \wedge \pi_2(t^{[k]}) = \pi_2(t'^{[k]}) \in B[\pi_1(t^{[k]})/x]$, which follows from the supposition. □

The conditions for predicate-admissibility are more elaborate, but also more general. I may immediately state conditions for basic types other than functions. Informally, basic compound types other than functions are predicate-admissible so long as their component types are predicate-admissible, and base types are always predicate-admissible.

Lemma 5.18

- $\mathrm{Adm}(A + B \mid y : S)$ *if* $\forall s \in S.\, (A + B)[s/y]$ type *and* $\mathrm{Adm}(A \mid y : S)$ *and* $\mathrm{Adm}(B \mid y : S)$.
- $\mathrm{Adm}(\Sigma x{:}A.B \mid y : S)$ *if* $\forall s \in S.\, (\Sigma x{:}A.B)[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{Adm}(A \mid y : S)$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathrm{Adm}(\mathit{Void} \mid y : S)$, $\mathrm{Adm}(\mathit{Atom} \mid y : S)$, $\mathrm{Adm}(\mathbb{Z} \mid y : S)$ *and* $\mathrm{Adm}(\mathbb{E} \mid y : S)$

- $\mathrm{Adm}(a_1 = a_2\ in\ A \mid y : S)$ *if* $\forall s \in S. (a_1 = a_2\ in\ A)[s/y]$ type *and* $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a_1 \leq a_2 \mid y : S)$
- $\mathrm{Adm}(\overline{A} \mid y : S)$ *if* $\forall s \in S. \overline{A}[s/y]$ type *and* $\mathrm{Adm}(A \mid y : S)$
- $\mathrm{Adm}(a\ in!\ A \mid y : S)$ *if* $\forall s \in S. (a\ in!\ A)[s/y]$ type

Predicate-admissibility of a function type is more complicated because a function argument with the type $A[e^{[\omega]}/x]$ does not necessarily belong to any of the finite approximations $A[e^{[j]}/x]$. To settle this, it is necessary to require a *coadmissibility* condition on the domain type. Then a function type will be predicate-admissible if the domain is weakly coadmissible and the codomain is predicate-admissible.

Definition 5.19 *A type T is* weakly coadmissible *for x in S (abbreviated* $\mathrm{WCoAdm}(T \mid x : S)$*) if:*

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j. \forall k \geq j.\, e^{[k]} \in S) \wedge t = t' \in T[e^{[\omega]}/x] \Rightarrow \\ (\exists j. \forall k \geq j.\, t = t' \in T[e^{[k]}/x])$$

A type T is coadmissible *for x in S (abbreviated* $\mathrm{CoAdm}(T \mid x : S)$*) if:*

$$\forall f, t, t', e.\, e^{[\omega]} \in S \wedge (\exists j. \forall k \geq j.\, e^{[k]} \in S) \wedge t^{[\omega]} = t'^{[\omega]} \in T[e^{[\omega]}/x] \Rightarrow \\ (\exists j. \forall k \geq j.\, t^{[k]} = t'^{[k]} \in T[e^{[k]}/x])$$

Lemma 5.20 $\mathrm{Adm}(\Pi x{:}A.B \mid y : S)$ *if* $\forall s \in S. (\Sigma x{:}A.B)[s/y]$ type *and* $\mathrm{WCoAdm}(A \mid y : S)$ *and* $\forall s \in S, a \in A[s/y].\, \mathrm{Adm}(B[a/x] \mid y : S)$

Clearly coadmissibility implies weak coadmissibility. A general set of conditions listed in Lemma 5.21 establish weak and full coadmissibility for various types. Weak and full coadmissibility are closed under disjoint union and dependent sum formation, and full coadmissibility is additionally closed under equality-type formation. I use both notions of coadmissibility, rather than just adopting one or the other, because full coadmissibility is needed for equality types but under certain circumstances weak coadmissibility is easier to show (Proposition 5.22 below).

Lemma 5.21

- *$A + B$ is (weakly) coadmissible for y in S if* $\forall s \in S. (A + B)[s/y]$ type *and A and B are (weakly) coadmissible for y in S*
- $\mathrm{WCoAdm}(\Sigma x{:}A.B \mid y : S)$ *if* $\forall s \in S. (\Sigma x{:}A.B)[s/y]$ type *and* $\mathrm{WCoAdm}(A \mid y : S)$ *and* $\forall s \in S, a \in A[s/y].\, \mathrm{WCoAdm}(B[a/x] \mid y : S)$
- $\mathrm{CoAdm}(\Sigma x{:}A.B \mid y : S)$ *if* $\forall s \in S. (\Sigma x{:}A.B)[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{CoAdm}(A \mid y : S)$ *and* $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- *Void, Atom, $\mathbb{Z}$ and $\mathbb{E}$ are strongly or weakly coadmissible for y in any S*
- $\mathrm{CoAdm}(a_1 = a_2\ in\ A \mid y : S)$ *if* $\forall s \in S. (a_1 = a_2\ in\ A)[s/y]$ type *and* $\mathrm{CoAdm}(A \mid y : S)$
- *$a_1 \leq a_2$ is strongly or weakly coadmissible for y in any S*
- *$\overline{A}$ is (weakly) coadmissible for y in S if* $\forall s \in S. \overline{A}[s/y]$ type *and A is (weakly) coadmissible for y is S*

- *a in! A is strongly or weakly coadmissible for y in S if* $\forall s \in S. (a \text{ in! } A)[s/y]$ type

When T does not depend upon S, predicate-admissibility and weak coadmissibility become easier to show:

Proposition 5.22 *Suppose x does not appear free in T. Then:*

- $\mathrm{Adm}(T)$ *if* $\mathrm{Adm}(T \mid x : S)$ *and S is inhabited*
- $\mathrm{Adm}(T \mid x : S)$ *if* $\mathrm{Adm}(T)$
- $\mathrm{WCoAdm}(T \mid x : S)$

There remains one more result related to predicate-admissibility. Suppose one wishes to show $\mathrm{Adm}(T \mid x : S)$ where T depends upon x. There are two ways that x may be used in T. First, T might contain an equality type where x appears in one or both of the equands. In that case, predicate-admissibility can be shown with the tools discussed above. Second, T may be an expression that computes a type from x. In this case, T can be simplified using direct computation (Section 4.1.7), but another tool will be needed if T performs any case analysis (as in the embedding of the λ^K disjoint union type in Section 3.3.1).

Lemma 5.23 *Consider a type $case(d, x.A, x.B)$ that depends upon y from S. Suppose there exist T_1 and T_2 such that:*

- $\forall s \in S.\, d[s/y] \in (T_1 + T_2)[s/y]$
- $\forall s \in S, t \in T_1[s/y].\, A[s, t/y, x]$ type
- $\forall s \in S, t \in T_2[s/y].\, B[s, t/y, x]$ type
- $\Sigma y{:}S.T_1$ type *and* $\Sigma y{:}S.T_2$ type

Then the following are the case:

- $\mathrm{Adm}(case(d, x.A, x.B) \mid y : S)$ *if* $\mathrm{Adm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$
- $\mathrm{WCoAdm}(case(d, x.A, x.B) \mid y : S)$ *if* $\mathrm{WCoAdm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ *and* $\mathrm{WCoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$
- $\mathrm{CoAdm}(case(d, x.A, x.B) \mid y : S)$ *if* $\mathrm{CoAdm}(A[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_1))$ *and* $\mathrm{CoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.T_2))$

5.3.2 Monotonicity

In some cases a very simple device may be used to show admissibility. We say that a type is monotone if it respects computational approximation, and it is easy to show that all monotone types are admissible.

Definition 5.24 *A type T is* monotone *(abbreviated* $\mathrm{Mono}(t)$*) if $t = t' \in T$ whenever $t \in T$ and $t \leq t'$.*

Lemma 5.25 *All monotone types are admissible.*

Proof

Let f, t and t' be arbitrary and suppose there exists j such that $t^{[j]} = t'^{[j]} \in T$. Since $t^{[j]} \leq t^{[\omega]}$ and $t'^{[j]} \leq t'^{[\omega]}$, it follows that $t^{[j]} = t^{[\omega]} \in T$ and $t'^{[j]} = t'^{[\omega]} \in T$. The result follows directly. □

All type constructors are monotone except universes and partial types, which are never monotone. The proof of this fact is easy [53].

Proposition 5.26

- $\text{Mono}(A + B)$ *if* $\text{Mono}(A)$ *and* $\text{Mono}(B)$
- $\text{Mono}(\Pi x{:}A.B)$ *if* $\text{Mono}(A)$ *and* $\forall a \in A. \text{Mono}(B[a/x])$
- $\text{Mono}(\Sigma x{:}A.B)$ *if* $\text{Mono}(A)$ *and* $\forall a \in A. \text{Mono}(B[a/x])$
- $\text{Mono}(\mathit{Void})$, $\text{Mono}(\mathit{Atom})$, $\text{Mono}(\mathbb{Z})$, $\text{Mono}(\mathbb{E})$, $\text{Mono}(a_1 = a_2 \in A)$, $\text{Mono}(a_1 \leq a_2)$ *and* $\text{Mono}(a\ \mathit{in!}\ A)$

5.3.3 Set and Quotient Types

Given the parallel between the dependent product and set types, it is natural to expect that set types would have a similar admissibility rule: that $\{x : A \mid B\}$ if A is admissible and B is predicate-admissible for x in A. Surprisingly, this turns out not to be the case. Suppose that $t^{[k]} \in \{x : A \mid B\}$ for all $k \geq j$. Then for every $k \geq j$, there exists some term $b_k \in B[t^{[k]}/x]$. We would like it to follow by predicate-admissibility that there exists $b_\omega \in B[t^{[\omega]}/x]$, but it does not. The problem is that each b_k can be a completely different term, and predicate-admissibility applies only when each b_k is of the form $b[f^k/w]$ for a single fixed b.

Intuitively, the desired rule fails because the set type $\{x : A|B\}$ suppresses the computational content of B and therefore B can be inhabited *non-uniformly*, by unrelated terms for related members of A. In contrast, if the chain $t^{[j]}, t^{[j+1]}, t^{[j+2]}, \ldots$ belongs to $\Sigma x{:}A.B$, then the chain $\pi_2(t)^{[j]}, \pi_2(t)^{[j+1]}, \pi_2(t)^{[j+2]}, \ldots$ uniformly inhabits B.

For a concrete example, consider:

$$\begin{aligned} T &\overset{\text{def}}{=} \{g : \mathbb{Z} \to \overline{\mathit{Unit}} \mid \exists n{:}\mathbb{Z}. \neg(gn\ \mathit{in!}\ \mathbb{Z})\} \\ f &\overset{\text{def}}{=} \lambda h. \lambda x.\ \mathit{if}\ x \leq_Z 0\ \mathit{then}\ \star\ \mathit{else}\ h(x-1) \\ t &\overset{\text{def}}{=} \lambda y. wy \end{aligned}$$

The type T is not admissible: For all k, $(t^{[k]_f})k$ diverges, so $t^{[k]_f} \in T$; but $t^{[\omega]_f}$ converges for all arguments, so $t^{[\omega]_f} \notin T$. However, $\exists n{:}\mathbb{Z}. \neg(gn\ \mathit{in!}\ \mathbb{Z})$ is predicate-admissible for g in $\mathbb{Z} \to \overline{\mathit{Unit}}$. The problem is that the inhabiting integers are not related by computational approximation; that is, they are not uniform.

To show a set type admissible, we need to be able to show that the selection predicate can be inhabited uniformly:

Lemma 5.27 *The type* $\{x : A \mid B\}$ *is admissible if:*

- $\text{Adm}(A)$, *and*

- $\mathrm{Adm}(B \mid x : A)$, *and*
- *there exists* b *such that* $b[a/x] \in B[a/x]$ *whenever* $a \in A$ *and* $\exists b'.\, b' \in B[a/x]$.

Another way to state the uniformity condition (the third clause) is that the computational content of B should be recoverable from knowing it exists (*i.e.,* there exists a function with type $\Pi x{:}A.\, \downarrow B \rightarrow B$. In fact, the two versions are equivalent modulo a few functionality requirements:

Proposition 5.28

- *If* $f \in \Pi x{:}A.\, \downarrow B \rightarrow B$ *then* $b[a/x] \in B[a/x]$ *whenever* $a \in A$ *and* $\exists b'.\, b' \in B[a/x]$, *where* $b = f\, x\, \star$.
- *If* $b[a/x] = b[a'/x] \in B[a/x]$ *whenever* $a = a' \in A$ *and* $\exists b'.\, B[a/x]$ *and if* $B[a/x] = B[a'/x]$ *whenever* $a = a' \in A$ *then* $\lambda x.\lambda y.\, b \in \Pi x{:}A.\, \downarrow B \rightarrow B$.

We may give a similar predicate-admissibility condition:

Lemma 5.29 $\mathrm{Adm}(\{x : A \mid B\} \mid y : S)$ *if* $\forall s \in S.\, (\{x : A \mid B\})[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{Adm}(A \mid y : S)$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : \Sigma y{:}S.A)$, *and there exists* b *such that* $b[a, s/x, y] \in B[a, s/x, y]$ *whenever* $s \in S$ *and* $a \in A[s/y]$ *and* $\exists b'.\, b' \in B[a, s/x, y]$

Coadmissibility and monotonicity work on single terms, not chains, so the uniformity issue does not arise, resulting in conditions fairly similar to those for dependent products:

Lemma 5.30

- $\mathrm{WCoAdm}(\{x : A \mid B\} \mid y : S)$ *if* $\forall s \in S.\, \{x : A \mid B\}[s/y]$ type *and* $\mathrm{WCoAdm}(A \mid y : S)$ *and* $\forall s \in S, a \in A[s/y].\, \mathrm{WCoAdm}(B[a/x] \mid y : S)$
- $\mathrm{CoAdm}(\{x : A \mid B\} \mid y : S)$ *if* $\forall s \in S.\, \{x : A \mid B\}[s/y]$ type *and* $\Sigma y{:}S.A$ type *and* $\mathrm{CoAdm}(A \mid y : S)$ *and* $\mathrm{WCoAdm}(B[\pi_1(z), \pi_2(z)/y, x] \mid z : (\Sigma y{:}S.A))$
- $\mathrm{Mono}(\{x : A \mid B\})$ *if* $\mathrm{Mono}(A)$

The conditions for quotient types are similar to those for set types:

Lemma 5.31

- $\mathrm{Adm}(xy{:}A/\!/B)$ *if* $\mathrm{Adm}(A)$ *and* $\mathrm{Adm}(B[\pi_1(z), \pi_2(z)/x, y] \mid z : A \times A)$, *and there exists* b *such that* $b[a, a'/x, y] \in B[a, a'/x, y]$ *whenever* $a \in A$ *and* $a' \in A$ *and* $\exists b'.\, b' \in B[a, a'/x, y]$.
- $\mathrm{Adm}(xy{:}A/\!/B \mid z : S)$ *if* $\forall s \in S.(xy{:}A/\!/B)[s/z]$type *and* $\Sigma z{:}S.(A \times A)$type *and* $\mathrm{Adm}(A \mid z : S)$ *and* $\mathrm{Adm}(B[\pi_1(z'), \pi_1(\pi_2(z')), \pi_2(\pi_2(z'))/z, x, y] \mid z' : \Sigma z{:}S.(A \times A))$, *and there exists* b *such that* $b[a, a', s/x, y, z] \in B[a, a', s/x, y, z]$ *whenever* $s \in S$ *and* $a \in A[s/z]$ *and* $a' \in A[s/z]$ *and* $\exists b'.\, b' \in B[a, a', s/x, y, z]$.
- $\mathrm{WCoAdm}(xy{:}A/\!/B \mid z : S)$ *if* $\forall s \in S.\, (xy{:}A/\!/B)[s/z]$ type *and* $\mathrm{WCoAdm}(A \mid z : S)$ *and* $\forall s \in S, a \in A[s/z], a' \in A[s/z].\, \mathrm{WCoAdm}(B[a, a'/x, y] \mid z : S)$
- $\mathrm{CoAdm}(xy{:}A/\!/B \mid z : S)$ *if* $\forall s \in S.\, (xy{:}A/\!/B)[s/z]$ type *and* $\Sigma z{:}S.(A \times A)$ type *and* $\mathrm{CoAdm}(A \mid z : S)$ *and* $\mathrm{CoAdm}(B[\pi_1(z'), \pi_1(\pi_2(z')), \pi_2(\pi_2(z'))/z, x, y] \mid z' : \Sigma z{:}S.(A \times A))$
- $\mathrm{Mono}(xy{:}A/\!/B)$ *if* $\mathrm{Mono}(A)$

5.3.4 Summary

Figure 5.1 provides a summary of the basic admissibility results of this chapter. It is worthwhile to note that all these results are proved constructively, with the exception of (weak and full) coadmissibility of partial types. The following theorem shows that the proofs of coadmissibility of partial types are necessarily classical; if a constructive proof existed then one could extract an algorithm meeting the theorem's specification, which can be used to solve the halting problem.

Theorem 5.32 *There does not exist an algorithm that computes an integer j such that $\forall k \geq j.\, t = t' \in \overline{T}[e^{[k]}/x]$, when given S, T, f, t, t', e and i such that:*

- $\forall s \in S.\, \overline{T}[s/x]$ type
- $\mathrm{CoAdm}(T \mid x : S)$
- $e^{[\omega]} \in S$
- $\forall k \geq i.\, e^{[k]} \in S$
- $t = t' \in \overline{T}[e^{[\omega]}/x]$

Recall the inadmissible type T from Theorem 5.1. That type fails the predicate-admissibility condition because of the negative appearance of a function type, which could not be shown weakly coadmissible, and it fails the monotonicity condition because it contains the partial type $\overline{\mathbb{N}}$.

5.4 Conclusions

An interesting avenue for future investigation would be to find some negative results characterizing inadmissible types. Such negative results would be particularly interesting if they could be given a syntactic character, like the results of this chapter. Along these lines, it would be interesting to find whether the inability to show coadmissibility of function types represents a weakness of this proof technique or an inherent limitation.

The results presented above provide *metatheoretical* justification for the fixpoint principle over many types. In order for these results to be useful in theorem proving, they must be introduced into the logic. One way to do this, and the way it is done here, is to introduce types to represent the assertions $\mathrm{Adm}(T)$, $\mathrm{Adm}(T \mid x : S)$, etc., that are inhabited exactly when the underlying assertion is true (in much that same way as the equality type is inhabited exactly when the equands are equal), and to add rules relating to these types that correspond to the lemmas of Section 5.3. This brings the tools into the system in a semantically justifiable way, but it is unpleasant in that it leads to a proliferation of new types and inference rules stemming from discoveries outside the logic. Of the Nuprl proof rules of Appendix B, 84 rules (not quite half) deal with admissibility. It would be preferable to deal with admissibility within the logic. A theory with intensional reasoning principles, such as the one proposed in Constable and Crary [23], would allow reasoning about computation internally. Then these results could be proved within the theory and the only extra rule that would be required would be a single rule relating admissibility to the the fixpoint principle.

However they are placed into the logic, these results allow for recursive computation on a wide variety of types. This make partial types and fixpoint induction a useful tool in type-theoretic theorem provers. It also makes it possible to study many recursive programs that used to be barred from the logic because they could not be typed.

Bibliography

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[3] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *Twenty-Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 296–409, St. Petersburg, Florida, January 1996.

[4] Peter Aczel. Frege structures revisited. In B. Nordström and J. Smith, editors, *Proceedings of the 1983 Marstrand Workshop*, 1983.

[5] Stuart Allen. A non-type-theoretic definition of Martin-Löf's types. In *Second IEEE Symposium on Logic in Computer Science*, pages 215–221, Ithaca, New York, June 1987.

[6] Stuart Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, 1987.

[7] Philippe Audebaud. Partial objects in the calculus of constructions. In *Sixth IEEE Symposium on Logic in Computer Science*, pages 86–95, Amsterdam, July 1991.

[8] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual.* INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.

[9] David A. Basin. An environment for automated reasoning about partial functions. In *Ninth International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*. Springer-Verlag, 1988.

[10] Michael Beeson. Recursive models for constructive set theories. *Annals of Mathematical Logic*, 23:127–178, 1982.

[11] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting. In *Theoretical Aspects of Computer Software*, 1997.

[12] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[13] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[14] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, Sendai, Japan, September 1997.

[15] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[16] Luca Cardelli. Phase distinctions in type theory. Unpublished manuscript, January 1988.

[17] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 70–79, San Diego, January 1988.

[18] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts.* Springer-Verlag, 1991.

[19] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[20] Robert L. Constable. Intensional analysis of functions and types. Technical Report CSR-118-82, Department of Computer Science, University of Edinburgh, June 1982.

[21] Robert L. Constable. Constructive mathematics as a programming logic I: Some principles of theory. In *Topics in the Theory of Computation*, volume 24 of *Annals of Discrete Mathematics*, pages 21–37. Elsevier, 1985. Selected papers of the International Conference on Foundations of Computation Theory 1983.

[22] Robert L. Constable. Type theory as a foundation for computer science. In *Theoretical Aspects of Computer Software 1991*, volume 526 of *Lecture Notes in Computer Science*, pages 226–243, Sendai, Japan, 1991. Springer-Verlag.

[23] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. Technical report, Department of Computer Science, Cornell University, 1997.

[24] Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *Second IEEE Symposium on Logic in Computer Science*, pages 183–193, Ithaca, New York, June 1987.

[25] Robert L. Constable and Scott Fraser Smith. Computational foundations of basic recursive function theory. In *Third IEEE Symposium on Logic in Computer Science*, pages 360–371, Edinburgh, Scotland, July 1988.

[26] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[27] Thierry Coquand. An analysis of Girard's paradox. In *First IEEE Symposium on Logic in Computer Science*, pages 227–236, Cambridge, Massachusetts, June 1986.

[28] Thierry Coquand. Metamathematical investigations of a calculus of constructions. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *The APIC Series*, pages 91–122. Academic Press, 1990.

[29] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[30] Karl Crary. Foundations for the implementation of higher-order subtyping. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 125–135, Amsterdam, June 1997.

[31] Karl Crary. Simple, efficient object encoding using intersection types. Technical Report TR98-1675, Department of Computer Science, Cornell University, April 1998.

[32] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[33] Michael Dummett. *Elements of Intuitionism.* Oxford Logic Guides. Clarendon Press, 1977.

[34] Andrzej Filinski. *Controlling Effects.* Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1996.

[35] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[36] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. dissertation, Université Paris VII, 1972.

[37] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge University Press, 1988.

[38] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag, 1979.

[39] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[40] Robert Harper. Constructing type systems over an operational semantics. *Journal of Symbolic Computation*, 14:71–84, 1992.

[41] Robert Harper. Personal communication, 1993.

[42] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 93.

[43] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 206–219, January 1993.

[44] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[45] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[46] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[47] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[48] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998. To appear.

[49] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996.

[50] Jason J. Hickey. A semantics of objects in type theory. Unpublished manuscript, 1997.

[51] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[52] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Second IEEE Symposium on Logic in Computer Science*, pages 205–214, Ithaca, New York, June 1987.

[53] Douglas J. Howe. Equality in lazy computation systems. In *Fourth IEEE Symposium on Logic in Computer Science*, 1989.

[54] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. Technical report, Bell Labs, 1996.

[55] Shigeru Igarashi. Admissibility of fixed-point induction in first-order logic of typed theories. Technical Report AIM-168, Computer Science Department, Stanford University, May 1972.

[56] Paul Jackson. *The Nuprl Proof Development System, Version 4.1.* Department of Computer Science, Cornell University, 1994.

[57] Paul Bernard Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, January 1995.

[58] Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 80–87, San Diego, January 1988.

[59] Christoph Kreitz. Formal reasoning about communications systems I. Technical report, Department of Computer Science, Cornell University, 1997.

[60] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[61] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[62] Xavier Leroy. Manifest types, modules and separate compilation. In *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[63] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Twenty-Second ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[64] Xavier Leroy. *The Objective Caml System, Release 1.00.* Institut National de Recherche en Informatique et Automatique (INRIA), 1996.

[65] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, January 1986.

[66] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *Fifth European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.

[67] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Proceedings of the Logic Colloquium, 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.

[68] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 1982.

[69] Paul Francis Mendler. *Inductive Definition in Type Theory.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[70] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type. In *Thirteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg Beach, Florida, January 1986.

[71] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[72] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Cambridge, Massachusetts, 1997.

[73] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[74] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[75] Greg Morrisett. *Compiling with Types.* Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[76] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second Workshop on Types in Compilation*, March 1998.

[77] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[78] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Sixth International Symposium on Programming*, number 167 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, April 1984.

[79] Chris Okasaki. Catenable double-ended queues. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 66–74, Amsterdam, June 1997.

[80] Erik Palmgren. An information system interpretation of Martin-Löf's partial type theory with universes. *Information and Computation*, 106:26–60, 1993.

[81] Erik Palmgren and Viggo Stoltenberg-Hansen. Domain interpretations of intuitionistic type theory. U.U.D.M. Report 1989:1, Uppsala University, Department of Mathematics, January 1989.

[82] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Twentieth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[83] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

[84] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *Theoretical Aspects of Computer Software 1994*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346, Sendai, Japan, 1994. Springer-Verlag.

[85] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Twenty-Fourth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 40–53, Paris, January 1997.

[86] John C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 345–372, Amsterdam, October 1981. North-Holland.

[87] Adrian Rezus. Semantics of constructive type theory. Technical Report 70, Informatics Department, Faculty of Science, Nijmegen, University, The Netherlands, September 1985.

[88] Dana Scott. Outline of a mathematical theory of computation. In *Fourth Princeton Conference on Information Sciences and Systems*, pages 169–176, 1970.

[89] Dana Scott. Lattice theoretic models for various type-free calculi. In *Fourth International Congress of Logic, Methodology and Philosophy of Science*. North-Holland, 1972.

[90] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, volume 21 of *Microwave Research Institute Symposia Series*. Polytechnic Institute of Brooklyn, 1971.

[91] Scott F. Smith. Hybrid partial-total type theory. *International Journal of Foundations of Computer Science*, 6:235–263, 1995.

[92] Scott Fraser Smith. *Partial Objects in Type Theory.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, January 1989.

[93] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[94] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, volume 2, pages 461–493. Cambridge University Press, 1992.

[95] Philip Wadler. The essence of functional programming. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.

[96] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, 2nd edition, 1910.