# TYPE THEORETICAL FOUNDATIONS FOR DATA STRUCTURES, CLASSES, AND OBJECTS

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

Alexei Pavlovich Kopylov

January 2004

# TYPE THEORETICAL FOUNDATIONS FOR DATA STRUCTURES, CLASSES, AND OBJECTS

Alexei Pavlovich Kopylov, Ph.D.
Cornell University 2004

In this thesis we explore the question of how to represent programming data structures in a constructive type theory. The basic data structures in programing languages are records and objects. Most known papers treat such data structure as primitive. That is, they add new primitive type constructors and supporting axioms for records and objects. This approach is not satisfactory. First of all it complicates a type theory a lot. Second, the validity of the new axioms is not easily established. As we will see the naive choice of axioms can lead to contradiction even in the simplest cases.

We will show that records and objects can be *defined* in a powerful enough type theory. We will also show how to use these type constructors to define abstract data structure.

## BIOGRAPHICAL SKETCH

Alexei Kopylov was born in Moscow State University on April 2, 1974. His parents were students in the Department of Mathematics and Mechanics there. First year of his life Alexei lived in a student dormitory in the main building of the Moscow State University. Then his parents moved to Chernogolovka, a cozy scientific town near Moscow.

Alexei returned to Moscow State University as a student in 1991. Five years later he graduated from the Department of Mathematics and Mechanics and entered the graduate school of the same Department. He passed all qualifying exam and almost finish his thesis there, but in 1998 he dropped the graduate school in Moscow and enrolled in the PhD program at Cornell University.

Now in January 2004 he is looking forward to move to Caltech as a post doctoral fellow.

## ACKNOWLEDGEMENTS

I would like to thank here my teachers I had in my life. First of all, I am thankful to my parents Pavel Kopylov and Ekaterina Gamazova. They inculcated in me a taste for mathematics in my early ages. For example, when I was a little boy, my father brought me a pair of sand-glasses. I played with them and came up with different puzzles, like *how to measure eleven minutes using sand-glasses for three and ten minutes?* My father also taught me programming in Pascal when we did not have a computer.

I am also thankful to Raymond Smullyan, although I never met him in person. My early interest in mathematics is partially due to his great book — *What Is the Name of This Book?* — with logical puzzles about knights and knaves. My father read me the problems from this book (I could hardly read at that time) and I competed with my mother trying to solve the problems first.

My special thanks are due to my school math teacher Alexandr Nikolaevich Zemlyakov "Zemmm". I admire his mathematical taste and his teaching style.

Unfortunately some people who had great influence on me are already passed away. I am very grateful to my grandfather Andrei Konstantinovich Gamazov, who was a great teacher, I am very proud of him. I am also very grateful to my other grandfather Nikolai Georgievivh Kopylov, who taught me chess. My schoolfriend Ivan Soloviev had a big influence on me. He was one year older than me and was always a step ahead of me in mathematical Olympics.

I am thankful to my Moscow adviser Sergei Artemov. He helped me a **lot** both in Moscow and at Cornell. Thanks to him I am here. I owe many thanks to my Cornell adviser Robert Constable for his guidance and many useful discussions.

I was very pleased to work with my colleague and namesake (although he spells his name differently) Aleksey Nogin. Part of the thesis is a joint work with Aleksey. Many thanks are due to my other colleague Jason Hickey for his discussions and early appreciation of my work.

I am also thankful to Christoph Kreitz and Stuart Allen for reading and reviewing my work. I want to thank many other Cornellians included: Anil Nerode and Jon Kleinberg for serving on my committee, Dexter Kozen who said that "a computer scientist is a mathematician with a job", Evan Moran for his comments on my work during PRL seminars, Mark Bickford, working with him was a pleasure, Pavel Naumov and Lena Safirova for their help during my first year at Cornell, Alexandre Evfimievski for his sharp criticism, and many others.

I also want to thank the PRL seminar for giving me a forum for presenting my ideas and helping me refine them — especially the long series on objects.

I also acknowledge support from the DoD Multidisciplinary University Research Initiative (MURI) program administered by the Office of Naval Research (ONR) under Grant N00014-01-1-0765, the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-98-2-0198, and by NSF Grant CCR 0204193.

TABLE OF CONTENTS

# LIST OF TABLES

# Chapter 1
# Introduction

This thesis is done in the framework of a certain constructive type theory, which is an extension of Martin-Löf type theory. Type theory is powerful tool for formalizing programming languages. It already contains the functional programming language ($\lambda$-calculus) and typing systems. The typing system is powerful enough to represent any program specification. In this thesis we research the question of expanding type theory with more programming tools.

## 1.1 Records

One of the important tools of any programming languages is the record type. We also will consider a dependent record type, that is, a record type where the types of components may depend of previous components (like $\{\mathtt{x} : A; \mathtt{y} : B[\mathtt{x}]\}$). Records and especially dependent records are a powerful tool for programming, representing mathematical concepts and data structures. In the last decade several type systems with records as primitive types were proposed. We will see that the record type is too complex a type to be primitive, and naive axiomatization leads to contradiction (see Section 4.2). The question arose: whether it is possible to define the record type in existent type theories using standard types without introducing new primitives.

It was known that *independent* records can be defined in type theories with dependent functions or intersection. On the other hand *dependent* records cannot be formed using standard types [5]. Hickey [20] introduced a complex notion of *very dependent functions* to represent dependent records. Here we extend the constructive type theory with a simpler type constructor *dependent intersection*, i.e., the intersection of *two* types, where the second type may depend on elements of the first one (not to be confused with the intersection of a family of types). This new type constructor allows us to define dependent records in a very simple way.

Dependent intersection is very simple and natural type constructor. It also allows us to define the set type constructor (which is primitive in the original theory), thus it simplifies the overall type theory.

Also it turns out that natural join operator ($\bowtie$) is just an intersection of sets of records.

## 1.2 Objects

Another important concept in programming languages is object-oriented programming. Unfortunately object-oriented languages are hard to represent in the type theories due to self-application. (See [1, 17].)

In the last decade several encodings of objects in type theory were proposed. See a comparison among the most basic ones in [7]. Almost every existing encoding uses an extension of system $F$ [14] as a target type theory.

We show how to embed object types in the constructive type theory using intersection and union. The object encoding in this system has its own specific characters.

Objects may have recursive methods. In our system we have total functions. That is, we allow recursive functions as soon as we can prove that they terminate. So we are looking for a definition of a type of objects, such that it allows recursive methods and at the same time allows for a type of objects with a certain method, application of this method to any object of this type should always terminate. Note that in $F$-like systems application of a method does not necessary terminate. Therefore we can not simply follow the encoding of objects in $F$-like systems. It also shows that there is no simple way to define objects as primitives.

We will also see similarities with the existing encodings. Most of the known encodings of the type of object use an existential type in $F$-like type theories. In our type theory, the union type (Section 2.3.5) could be used instead of an existential quantifier. That is, we could use $\bigcup_{X:\mathbb{U}_i} A[X]$ instead of $\exists X.A[X]$, where $\mathbb{U}_i$ is the universe (a type of types, Section 2.1.3) of level $i$. On the one hand, the union type is more powerful: we can take a union over types satisfying some condition. This feature allows us to find a simpler encoding of objects. Also the union type does not require packing/unpacking its elements as does an existential type. On the other hand, the unions type has its own restrictions. We cannot take union over all types, but only over types of a particular level $i$. This union will be a type of level $i+1$ (i.e., $\bigcup_{X:\mathbb{U}_i} A[X] \in \mathbb{U}_{i+1}$). That means we are not allowed to substitute this type in place of $X$. That is, for example, we cannot prove that $A[\bigcup_{X:\mathbb{U}_i} A[X]] \subseteq \bigcup_{X:\mathbb{U}_i} A[X]$. This problem significantly complicates our theory of objects. In particular, it requires that types of methods should depend continuously on the $\mathit{Self}$ type.

Our encoding of object types has most of the standard object-oriented features such as polymorphism, inheritance, method abstraction, method overriding and so on. Also our object type allows full abstraction. That is, users do not have access to abstract fields. So two different implementations of an object may be equal from the interface point of view. Moreover, this can be formally proved inside system itself. We do not allow binary methods on objects, since it would contradict full abstraction.

## 1.3 Organization of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 gives an overview of Martin-Löf type theory and the constructive type theory extension of it implemented in MetaPRL.

In Chapter 3 we introduce the new type constructor *dependent intersection* and show that record types can be defined using this constructor. Even with this new definition of the record type, finding the right elimination rule a for record calculus is challenging. In Chapter 4 we will show that a naive elimination rule for records is contradictory. We will discuss how functionality affects the elimination rule. We also introduce an idea of functions with limited polymorphism.

In Section 5.1 we show that our dependent intersection can replace the set type constructor. In Section 5.2 we will show the definition of the variant type which is dual to the record type. In Section 5.3 we show that our record calculus could be used to define abstract algebraic structures. In Section 5.4 we show that natural join operator ($\bowtie$) is just an intersection of sets of records.

In Chapter 6 we show an example of an abstract data structure, $Set$, and give a formally correct implementation of this data structure using red-black trees.

In Chapter 7 we encode objects into the type theory.

# Chapter 2
# Constructive Type Theory

Our work is done in the setting of constructive type theory as implemented in the MetaPRL logical framework [22, 19, 23]. Our type theory is an extension of the constructive type theory implemented in NuPRL [8, 9], which is an extension of Martin-Löf's type theory [30].

In this chapter we give a short overview of our type theory.

## 2.1 Martin-Löf Type Theory

First let us give an overview of the original Martin-Löf Type Theory [30].

### 2.1.1 Types

The basic notion in the theory is *type*. Type is a primitive notion. Two main judgments about types are $A\,\text{Type}$ meaning that $A$ is a type and $a \in A$ meaning that $a$ has type $A$. Each type $A$ is associated with an equality relation on elements of this type, $a = b \in A$. There is also the equivalence relation on types: $A = B$. So, Martin-Löf's type theory has the following four forms of judgments:

| | |
|---|---|
| $A\,\text{Type}$ | $A$ is a well-formed type |
| $A = B$ | $A$ and $B$ are equal types |
| $a \in A$ | $a$ has type $A$ |
| $a = b \in A$ | $a$ and $b$ are equal as elements of type $A$ |

The examples of types include simple types like $\mathbb{Z}$ for integers $0$, $1$, $-1$ and $\mathbb{B}$ for booleans. We can also construct new types using basic type constructors, like product $A \times B$ for the type of pairs $\langle a, b\rangle$ and function type $A \to B$ for the type of functions $\lambda x.b[x]$.

Some notations: we will use $T[x_1, \ldots, x_n]$ for expressions that may contain free variables $x_1, \ldots, x_n$ (and probably some other free variables), and $T[t_1, \ldots, t_n]$ for the substitution of terms $t_i$'s for *all* free occurrences of $x_i$'s. We call such variables that stands for terms *second order variables*. If a second-order variable is in scope of a bound variable we will always write all variables it may contain. For example we will write $\lambda x.f[x]$ for general $\lambda$-expressions. The expression $\lambda x.f$ means that $f$ does not contain free variables.

Functions types represent *total* computable functions. For example, $\lambda x.b[x]$ has type $\mathbb{Z} \to \mathbb{Z}$ if for any integer $a$ evaluation of $b[a]$ terminates and returns an integer. Thus, we are allowed to have recursive functions as long as we can prove that they terminate on any input from their domain. Of course that makes type-checking undecidable.

Membership and equality in a type is extensional. In particular it means that two functions $f$ and $g$ are equal in the type $A \to B$ if $f(a) = g(a) \in B$ for any $a \in A$.

Our type theory uses the proposition-as-types principle. That is, we will consider any type as a proposition which is true when this type is non-empty.

### 2.1.2 Dependent Types

Martin-Löf's type theory also has dependent types, namely dependent product and dependent function type.

Suppose, we have a type expression $B[x]$ that contains a free variable $x$ ranging over a type $A$. For example, $B[x]$ may be $[0..x]$ which represents an initial sequence of natural numbers. This expression is a type when $x \in \mathbb{N}$.

Then we can form a dependent product type $x : A \times B[x]$ (also known as a $\Sigma$-type) which is a type of all pairs $\langle a, b\rangle$ where $a \in A$ and $b \in B[a]$. For example, if $A = \mathbb{N}$ and $B[x] = [0..x]$ then $x : A \times B[x]$ is a type of pairs of natural numbers $\langle n, m\rangle$, where $m \leq n$.

We can also form a dependent function type $x : A \to B[x]$ (also known as a $\Pi$-type) which is a type of all functions $\lambda x.b[x]$ where $b[a] \in B[a]$ for any $a \in A$. For example, if $A = \mathbb{N}$ and $B[x] = [0..x]$ then $x : A \to B[x]$ is a type of functions $f(n)$, s.t. $0 \leq f(n) \leq n$.

Dependent types make the theory powerful enough to represent any mathematical statement.

### 2.1.3 Universe Types

Introduction of a type of all types leads to contradiction (Girard's paradox [15]). But we can introduce a sequence of universe types $\mathbb{U}_1$, $\mathbb{U}_2$, $\ldots$. Where $\mathbb{U}_1$ is the universe of the first level, a type of all types constructed without using universes. $\mathbb{U}_2$ is the universe of the second level, a type of all types constructed without using universes of any level above 1. And so on.

In this thesis we will assume that we fix some universe level $\mathbb{U} = \mathbb{U}_i$, and we will write $\mathbb{U}'$ for $\mathbb{U}_{i+1}$ and $\mathbb{U}''$ for $\mathbb{U}_{i+2}$.

## 2.2 Functionality

In our type theory we derive *sequents*. Each sequent has a form:

$$x_1 : H_1; x_2 : H_2[x_1]; \ldots ; x_n : H_n[x_1; \ldots ; x_{n-1}] \vdash C[x_1; \ldots ; x_n] \tag{2.1}$$

Here $x_i$'s are declared variables, $H_i$'s are hypotheses and $C$ is a conclusion. The $i$-th hypothesis may depend on the variables declared before it, and conclusion may depend on all variables.

Roughly speaking the sequent (2.1) is true when $C[x_1; \ldots ; x_n]$ is true (i.e. non-empty) for all $x_i$'s from $H_i[x_1; \ldots ; x_{i-1}]$. The formal definition of the truth of the sequent deals with functionality. Basically, we say that a type $C[x]$ is functional over $x : T$ if $t_1 = t_2 \in T$ implies $C[t_1] = C[t_2]$.

There are different nonequivalent approaches to define what it means for a sequent to be true. Originally Martin-Löf required that any hypothesis $H_i$ must be functional over previous hypotheses and the conclusion must be functional over all hypotheses.

The type theory implemented in NuPRL uses a weaker requirement that allows one to formulate stronger rules (for example a rule for induction over natural numbers). This approach is called *pointwise functionality* and was discovered by Stuart Allen in [3].

Another version of functionality was used in [31]. Aleksey Nogin later independently rediscovered it and called it *pairwise functionality*.

In the thesis we will consider pairwise and pointwise functionality.

### 2.2.1 Pointwise Functionality

Pointwise functionality is fairly complicated notion. We will use Aleksey Nogin's presentation of it.

Let we are given a list of hypotheses $\Gamma$:

$$x_1 : H_1; x_2 : H_2[x_1]; \ldots ; x_n : H_n[x_1; \ldots ; x_{n-1}].$$

Then we write $\vec{t}$ for a list of terms $t_1, t_2, \ldots , t_n$. We will also write

$$\begin{array}{ll} \vec{t} \in \Gamma[\vec{t}\,] & \text{for } \forall i \in [1..n].\ t_i \in H_i[t_1; \ldots ; t_{i-1}]; \\ \vec{t} = \vec{t'} \in \Gamma[\vec{t}\,] & \text{for } \forall i \in [1..n].\ t_i = t'_i \in H_i[t_1; \ldots ; t_{i-1}]; \\ \Gamma[\vec{t}\,] = \Gamma[\vec{t'}] & \text{for } \forall i \in [1..n].\ H_i[t_1; \ldots ; t_{i-1}] = H_i[t'_1; \ldots ; t'_{i-1}]. \end{array}$$

Then a sequent $\Gamma \vdash C[\vec{x}\,]$ said to be *true in pointwise functionality* iff

$$\begin{array}{c} \forall \vec{t}.\ \big(\vec{t} \in \Gamma[\vec{t}\,] \ \wedge\ \forall \vec{t'}.\ (\vec{t} = \vec{t'} \in \Gamma[\vec{t}\,] \Rightarrow \Gamma[\vec{t}\,] = \Gamma[\vec{t'}]) \Rightarrow \\ \forall \vec{t'}. (\vec{t} = \vec{t'} \in \Gamma[\vec{t}\,] \Rightarrow C[\vec{t}\,] \wedge C[\vec{t}\,] = C[\vec{t'}])\big) \end{array}$$

### 2.2.2 Pairwise Functionality

The alternative definition of the truth of a sequent is pairwise functionality. Using the above notation, we say that a sequent $\Gamma \vdash C[\vec{x}\,]$ is *true in pairwise functionality* iff

$$\forall \vec{t}. \forall \vec{t'}.\ (\Gamma[\vec{t}\,] = \Gamma[\vec{t'}] \ \wedge\ \vec{t} = \vec{t'} \in \Gamma[\vec{t}\,]) \Rightarrow (C[\vec{t}\,] \wedge C[\vec{t}\,] = C[\vec{t'}])$$

### 2.2.3 Comparing

Most of the rules are true in both functionalities. But some rules are true only in pointwise functionality, and some rules are true only in pairwise functionality.

The most important rule that holds only in pairwise functionality is the $Let$ rule (a form of the $Cut$ rule):

$$\frac{\Gamma; x : A; \Delta[x] \vdash C[x] \qquad \Gamma \vdash a \in A}{\Gamma; \Delta[a] \vdash C[a]} \quad (Let\ x = a \in A)$$

In pointwise functionality this rule is invalid, only a weaker form of this rule (when $\Delta$ does not depend on $x$) is valid:

$$\frac{\Gamma; x : A; \Delta \vdash C[x] \qquad \Gamma \vdash a \in A}{\Gamma; \Delta \vdash C[a]}$$

(Note that according to our notations the above rule means that $\Delta$ does not contain free occurrences of $x$.)

The following corollary of the $Let$ rule is also invalid in pointwise functionality (but of course holds in pairwise functionality):

$$\frac{\Gamma; x : B; \Delta[x] \vdash C[x] \qquad \Gamma \vdash A \subseteq B}{\Gamma; x : A; \Delta[x] \vdash C[x]}$$

On the other hand, the following rule is true in pointwise, but not in pairwise:

$$\frac{\begin{array}{c}\Gamma_1; t : T; \Gamma_2[t]; x : A[t]; \Delta[x; t] \vdash t = t' \in T \\ \Gamma_1; t : T; \Gamma_2[t]; x : A[t']; \Delta[x; t] \vdash C[t; x]\end{array}}{\Gamma_1; t : T; \Gamma_2[t]; x : A[t]; \Delta[x; t] \vdash C[t; x]} \qquad (PointwiseSubstitute)$$

This rule states that we can replace a variable declared as $t : T$ by a term $t'$ if we know that $t = t' \in T$. Sometimes this rule is stronger than a general substitution rule (which is true in both functionalities). The later rule requires that type $A[t]$ is functional over $t : T$:

$$\frac{\begin{array}{c}\Gamma; x : A[t]; \Delta[x]; z : T \vdash A[z]\ \text{Type} \\ \Gamma; x : A[t]; \Delta[x] \vdash t = t' \in T \\ \Gamma; x : A[t']; \Delta[x] \vdash C[x]\end{array}}{\Gamma; x : A[t]; \Delta[x] \vdash C[x]}$$

Rules stated in this thesis are true for both functionalities, unless otherwise mentioned.

## 2.3 Additional Types

The constructive type theory implemented in MetaPRL has some additional type constructors, some of them inherited form the NuPRL type theory.

### 2.3.1 Squiggle Equality

The squiggle equality on terms $a \equiv b$ is defined as the symmetric transitive closure of the reduction relation. Howe showed that it is a congruence [24].

For example we can prove that for any element $p$ of type $A \times B$

$$p \equiv \langle \pi_1 p, \pi_2 p \rangle\ .$$

Also we have $\eta$-reduction for any $f \in A \to B$:

$$f \equiv \lambda x.(f x)$$

### 2.3.2 The Set Type Constructor

Our type theory has a primitive type constructor for set types [9]. By definition, the set type $\{x : T \mid P[x]\}$ is a subtype of $T$, which contains only such elements $x$ of $T$ that satisfy property $P[x]$ (see [9]).

**Example 2.1** *The type of natural numbers is defined as* $\mathbb{N} = \{n : \mathbb{Z} \mid n \geq 0\}$. *Without set types we would have to define* $\mathbb{N}$ *as* $n : \mathbb{Z} \times (n \geq 0)$. *In this case we would not have the subtyping property* $\mathbb{N} \subseteq \mathbb{Z}$.

Later in the thesis (Section 5.1) we will replace this primitive type constructor by more fundamental primitive type, thus simplifying the type theory.

### 2.3.3 Subtyping

Our type theory also has a subtyping relation [31]. The subtyping relation as well as the membership relation are extensional. That means that $A \subseteq B$ does not say anything about structure of these types, but only means that all elements of type $A$ are also elements of type $B$ and if two elements are equal in $A$ then they are also equal in $B$. As a result the subtyping relation is undecidable (as well as type checking).

**Example 2.2** *If* $A \subseteq B$ *then* $(B \to C) \subseteq (A \to C)$. *It may seem strange at a first: suppose* $A$, $B$ *and* $C$ *are finite types and* $a$, $b$ *and* $c$ *are the number of elements in these types correspondingly, then* $B \to C$ *has* $c^b$ *elements and* $A \to C$ *has* $c^a < c^b$ *elements. This example shows that a subtype may have more elements than a supertype!*

*Remark* Of course, when we say that a type $A$ has $n$ elements, we mean that type $A$ has $n$ *different* elements. Actually this type may have many elements that are equal from the point of view of this type.

After the subtyping is defined, the natural question arises: what is the biggest (w.r.t. subtyping) common subtype of two or more types and what is the smallest supertype of two or more types?

### 2.3.4 Intersection

#### Binary Intersection

It is easy to see that $t$ can be in a common subtype of $A$ and $B$ only if $t \in A$ and $t \in B$. Also, $t_1$ may be equal to $t_2$ in a common subtype only if they are equal in both $A$ and $B$. Since the more elements the type has and the more elements are equal in a type, the "greater" the type is (in the sense of subtyping), in order to get the biggest common subtype of $A$ and $B$, we need to take all the objects that are both in $A$ and in $B$ and make all elements that are equal in both $A$ and $B$ equal in our type. In other words, the biggest common subtype of two types is a type whose set of members is an intersection of sets of members of those types and whose equivalence relation is an intersection (as sets of pairs) of equivalence relations of those two types. We call such type *an intersection* of $A$ and $B$, written: $A \cap B$.

**Example 2.3** $\lambda x.x + 1$ *is an element of the type* $(\mathbb{Z} \to \mathbb{Z}) \cap (\mathbb{N} \to \mathbb{N})$.

**Example 2.4** *Let* $A = \mathbb{N} \to \mathbb{N}$ *and* $B = \mathbb{Z}^- \to \mathbb{Z}$ *(where* $\mathbb{Z}^-$ *is a type of negative integers). Let* $id = \lambda x.x$ *and* $abs = \lambda x.|x|$. *Then* $id$ *and* $abs$ *are both elements of the type* $A \cap B$. *Although* $id$ *and* $abs$ *are equal as elements of the type* $\mathbb{N} \to \mathbb{N}$ *(because these two functions do not differ on* $\mathbb{N}$*),* $id$ *and* $abs$ *are different as elements of* $\mathbb{Z}^- \to \mathbb{Z}$. *Therefore,* $id \neq abs \in A \cap B$.

**Example 2.5** *Let* $A = \{0\} \to \mathbb{B}$, *where* $\{0\}$ *is a singleton subset of* $\mathbb{Z}$. *Then* $A$ *is a type of functions that maps* $0$ *to a boolean value. Obviously, this type has two elements. Now let* $B = \{1\} \to \mathbb{B}$. *This type also has two elements. But their intersection is* $A \cap B = \{0, 1\} \to \mathbb{B}$ *has four elements!*

The inference rules for the intersection type are presented in Table 2.1.

Table 2.1: Inference rules for the binary intersection type

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma \vdash B \text{ Type}}{\Gamma \vdash A \cap B \text{ Type}} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma \vdash B = B'}{\Gamma \vdash A \cap B = A' \cap B'} \qquad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash a \in B}{\Gamma \vdash a \in A \cap B} \qquad (Introduction)$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash a = a' \in B}{\Gamma \vdash a = a' \in A \cap B} \qquad (Equality)$$

$$\frac{\Gamma \vdash x \in A \cap B}{\Gamma \vdash x \in A} \qquad \frac{\Gamma \vdash x \in A \cap B}{\Gamma \vdash x \in B} \qquad (Elimination)^1$$

[1]See also Section 2.3.6

**Intersection of a Family of Types**

It is easy to see that the same is true if we take the largest common subtype of more than two types or if we take a largest common subtype of a whole family of types. We call the biggest common subtype of several types or of a family of types *an intersection type* of those types.

**Example 2.6** *$\lambda x.x$ has type $A \rightarrow A$ for any type $A$. Therefore*

$$\lambda x.x \in \bigcap_{A:\mathbb{U}} A \rightarrow A$$

**Example 2.7** *Let* $\text{Top} = \bigcap_{x:\text{Void}} \text{Void}$. *This type contains anything, and any two element of this type are equal. This is similar to the type* Void $\rightarrow$ Void, *but the later type contains only $\lambda$-terms. Again any two elements are equal in* Void $\rightarrow$ Void.

*It seems very strange that* $\text{Top} \in \text{Top}$, *and any* $\mathbb{U}_i \in \text{Top}$, *even* $\mathbb{U}_i \subseteq \text{Top}$, *whenever* $\text{Top} \in \mathbb{U}_1$. *But it does not contradict anything. The reason is similar to the reason why* $\lambda x.\mathbb{U}_i \in \text{Void} \rightarrow \text{Void}$ *does not lead to a contradiction. Although* Top *is a supertype of any type it is very trivial, because it has the trivial equality. So, we can not define something like "the type of all types" using* Top.

The inference rules for the family intersection type are presented in Table 2.2.

Note that we can define binary intersection as a partial case of of family intersection:

$$A \cap B = \bigcap_{b:\mathbb{B}} \texttt{if } b \texttt{ then } A \texttt{ else } B$$

### 2.3.5 Union

**Binary Union**

A similar argument shows that whenever either $t \in A$ or $t \in B$, $t$ should also be in common supertype of $A$ and $B$, and whenever $t_1 = t_2$ in either $A$ or $B$, $t_1$ should be equal to $t_2$ in any common supertype. Similarly, for the intersection type, the the set of all members of the smallest common supertype of two

Table 2.2: Inference rules for the family intersection type

$$\frac{\Gamma \vdash A \,\text{Type} \qquad \Gamma; x : A \vdash B[x] \,\text{Type}}{\Gamma \vdash \bigcap_{x:A} B[x] \,\text{Type}} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A \vdash B[x] = B'[x]}{\Gamma \vdash \bigcap_{x:A} B[x] = \bigcap_{x:A'} B'[x]} \qquad (TypeEquality)$$

$$\frac{\Gamma; x : A \vdash b \in B[x]}{\Gamma \vdash b \in \bigcap_{x:A} B[x]} \qquad (Introduction)$$

$$\frac{\Gamma; x : A \vdash b = b' \in B[x]}{\Gamma \vdash b = b' \in \bigcap_{x:A} B[x]} \qquad (Equality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash b \in \bigcap_{x:A} B[x]}{\Gamma \vdash b \in B[a]} \qquad (Elimination)^1$$

[1]See also Section 2.3.6

types is just a union of the sets of members of those types. However the union of two equivalence relations is not necessary an equivalence relation (it is not necessarily transitive). So the equivalence relation of the smallest common supertype is the smallest equivalence relation containing the union of the equivalence relations of the two types — the transitive closure of that union of the equivalence relations. We call this type the *union* of $A$ and $B$ and denote it by $A \cup B$.

The union considered as a proposition is a disjunction: $A \cup B$ is true iff $A$ is true or $B$ is true. But unlike the standard disjunction, union is not constructive. Knowing $A \cup B$ we cannot always say what is true: $A$ or $B$. Therefore the rule

$$\frac{x : A \vdash C \qquad x : B \vdash C}{x : A \cup B \vdash C}$$

is not constructively true. Indeed if a witness of $C$ is constructed differently in case when $x \in A$ and in case when $x \in B$ then we have no way to construct a witness if we now only that $x \in A \cup B$. But in case when $C$ does not have the computational context, like membership, this rule would be true.

The inference rules for the union type are presented in Table 2.3.

The following holds for union. If $f \in A \to C$ and $f \in B \to C$ then
$f \in A \cup B \to C$.

## Union of a Family of Types

Similarly we can define the union of a family of types.

The inference rules for the family union type are presented in Table 2.4.

**Example 2.8** *Let $P[x]$ be a predicate on some $x \in A$. Then $\bigcup_{x:A} P[x]$ is a true proposition (i.e., non empty type) if there is an element $a \in A$, s.t. $P[a]$. Therefore union could be considered as an existential quantifier. The difference between union type and standard existential quantifier $\exists x : A.P[x] = x : A \times P[x]$ is that union type "hides" the* first *component of the existential quantifier. That is, the witness of the union type is just a witness of $P[x]$ for some $x \in A$, but it does not contain $x$ itself. Compare with the set type: $\{x : A \mid P[x]\}$. The set type hides the second component of the existential quantifier. The witness of this type is just $x$ from $A$, s.t. $P[x]$.*

Table 2.3: Inference rules for the union type

$$\frac{\Gamma \vdash A\,\text{Type} \qquad \Gamma \vdash B\,\text{Type}}{\Gamma \vdash A \cup B\,\text{Type}} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma \vdash B = B'}{\Gamma \vdash A \cup B = A' \cup B'} \qquad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash B\,\text{Type}}{\Gamma \vdash a \in A \cup B} \qquad \frac{\Gamma \vdash b \in B \qquad \Gamma \vdash A\,\text{Type}}{\Gamma \vdash b \in A \cup B} \qquad (Introduction)$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash B\,\text{Type}}{\Gamma \vdash a = a' \in A \cup B} \qquad \frac{\Gamma \vdash b = b' \in B \qquad \Gamma \vdash A\,\text{Type}}{\Gamma \vdash b = b' \in A \cup B} \qquad (Equality)$$

$$\frac{\Gamma; u : A, \Delta \vdash c[u] \in C[u] \qquad \Gamma; u : B, \Delta \vdash c[u] \in C[u]}{\Gamma; u : (A \cup B); \Delta \vdash c[u] \in C[u]} \qquad (Elimination)^1$$

[1]See also Section 2.3.6

Table 2.4: Inference rules for the family union type

$$\frac{\Gamma \vdash A\,\text{Type} \qquad \Gamma; x : A \vdash B[x]\,\text{Type}}{\Gamma \vdash \bigcup_{x:A} B[x]\,\text{Type}} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A \vdash B[x] = B'[x]}{\Gamma \vdash \bigcup_{x:A} B[x] = \bigcup_{x:A'} B'[x]} \qquad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash b \in B[a] \qquad \Gamma; x : A \vdash B[x]\,\text{Type}}{\Gamma \vdash b \in \bigcup_{x:A} B[x]} \qquad (Introduction)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash b = b' \in B[a] \qquad \Gamma; x : A \vdash B[x]\,\text{Type}}{\Gamma \vdash b = b' \in \bigcup_{x:A} B[x]} \qquad (Equality)$$

$$\frac{\Gamma; x : A, u : B[x], \Delta \vdash c[u] \in C[u]}{\Gamma; u : \bigcup_{x:A} B[x], \Delta \vdash c[u] \in C[u]} \qquad (Elimination)^1$$

[1]See also Section 2.3.6

By analogy with intersection we can define binary union as a partial case of of family union:

$$A \cup B = \bigcup_{b:\mathbb{B}} \texttt{if } b \texttt{ then } A \texttt{ else } B$$

### 2.3.6 Elimination Rules for Intersections and Unions in Different Functionalities

All of the above rules for union and intersection hold in both functionalities. In pairwise functionality we can prove a stronger elimination rule for intersections and in pointwise functionality we can prove a stronger elimination rule for unions.

In pairwise functionality we have the $Let$ rule (Section 2.2.3). Using this rule and the elimination rules for intersection from Tables 2.1 and 2.2 we can prove stronger elimination rules:

$$\frac{\Gamma; x : A; y : B; \Delta[x; y] \vdash C[x; y]}{\Gamma; u : (x : A \cap B); \Delta[u; u] \vdash C[u; u]}$$

$$\frac{\Gamma; u{:} \bigcap_{x:A} B[x]; \Delta[u; u] \vdash a \in A \quad \Gamma; u{:} \bigcap_{x:A} B[x]; v : B[a]; \Delta[u; v]; u{=}v{\in}B[a] \vdash C[u; v]}{\Gamma; u{:} \bigcap_{x:A} B[x]; \Delta[u; v] \vdash C[u; v]}$$

In pointwise functionality using the weak $Let$ rule, we can only prove weak versions of the above rules where $\Delta$ does not depend on $u$:

$$\frac{\Gamma; x : A; y : B; \Delta \vdash C[x; y]}{\Gamma; u : (x : A \cap B); \Delta \vdash C[u; u]}$$

$$\frac{\Gamma; u : \bigcap_{x:A} B[x]; \Delta \vdash a \in A \qquad \Gamma; u : \bigcap_{x:A} B[x]; v : B[a]; \Delta; u = v \in B[a] \vdash C[u; v]}{\Gamma; u : \bigcap_{x:A} B[x]; \Delta \vdash C[u; v]}$$

Oppositely, the elimination rules for union type are stronger in pointwise functionality. In the elimination rules from Tables 2.3 and 2.4 $\Delta$ does not depend on $u$. In the pointwise functionality using $PointwiseSubstitute$ rule (Section 2.2.3) we can make these rules stronger by allowing $\Delta$ to depend on $u$:

$$\frac{\Gamma; u : A; \Delta[u] \vdash c[u] \in C[u] \qquad \Gamma; u : B; \Delta[u] \vdash c[u] \in C[u]}{\Gamma; u : (A \cup B); \Delta[u] \vdash c[u] \in C[u]}$$

$$\frac{\Gamma; x : A, u : B[x]; \Delta[u] \vdash c[u] \in C[u]}{\Gamma; u : \bigcup_{x:A} B[x]; \Delta[u] \vdash c[u] \in C[u]}$$

These rules are invalid in pairwise functionality.

**Remark 2.9** *Intersection of types was introduced in [11] and [37]. Our interpretation of intersection and union is most close to [34]. The understanding of semantics and rules for intersection and union is our join work with Aleksey Nogin.*

# Chapter 3
# Record Type and Dependent Intersection

In general, records are tuples of labeled fields, where each field may have its own type. In dependent records (or more formally, dependently typed records) the type of some components may depend on values of the other components. Since we have the type of types $\mathbb{U}$, values of record components may be types. This makes the notion of dependent records very powerful. Dependent records may be used to represent algebraic structures (such as groups) and modules in programming languages like SML or Haskell (see for example [4, 18]).

**Example 3.1** *One can define the signature for an ordered set as a dependent record type:*

$$OrdSetSig \triangleq \{\mathtt{t} : \mathbb{U}; \mathtt{less} : \mathtt{t} \to \mathtt{t} \to \mathbb{B}\}$$

*This definition can be understood as an algebraic structure as well as an interface of a module in a programing language.*

**Example 3.2** *The proposition-as-type principle allows us to add the property of order as a new component:*

$$OrdSet \triangleq \{\mathtt{t}{:}\mathbb{U}; \mathtt{less}{:}\mathtt{t} \to \mathtt{t} \to \mathbb{B}; \mathtt{axm} : Ord(\mathtt{t}, \mathtt{less})\}$$

*where $Ord(\mathtt{t}, \mathtt{less})$ is a predicate stating that* `less` *is a transitive irreflexive relation on* `t`*. Here* `axm` *is a new field that defines the axiom of the algebraic structure of ordered sets (or specification of the module type $OrdSet$).*

**Example 3.3** *In type theories with equality, manifested fields ([28]) may be also represented in the specification.*

$$IntOrdSetSig \triangleq \{\mathtt{t}{:}\mathbb{U}; \mathtt{less}{:}\mathtt{t}{\to}\mathtt{t}{\to}\mathbb{B}; \mathtt{mnf}{:}\mathtt{t}{=}\mathbb{Z}\}$$

*is a signature where* `t` *is bound to be the type of integers.*

From a mathematical point of view the record type is similar to the product type. The essential difference is the subtyping property: we can extend a record type with new fields and get a subtype of the original record type. E.g. $OrdSet$ and $IntOrdSetSig$ defined above are subtypes of $OrdSetSig$. The subtyping property is important in mathematics: we can apply all theorems about monoids to included types such as groups. It is also essential in programing for inheritance and abstractions.

Different type theories with records were proposed both for proof systems as well as for programming languages ([18, 28, 13, 4, 5, 36] and others). These systems treat the record type as a new primitive. In the current thesis we are interested in the following natural question: *is it possible to express the notion of records in usual type theories without the record type as primitive?* This question is especially interesting for pure mathematical proof systems. As we saw, the record type is a handy tool for representing algebraic structures. On the other hand records do not seem to be the basic mathematical concept that should be included in the foundation of mathematics. Rather records should be defined in terms of more abstract mathematical concepts.

It is known that it is possible to define *independent records* in a sufficiently powerful type theory that has dependent functions [20] or intersection [38]. On the other hand, there is no known way to form dependent records in standard Martin-Löf's type theory [5]. However, Hickey [20] showed that *dependent records* can be formed in an extension of Martin-Löf's type theory. Namely, he introduced a new type of *very dependent function types*. This type is powerful enough to express dependent records in a type theory and provides a mathematical foundation of dependent records. Unfortunately the type of very dependent functions is very complex itself. The rules and the semantics are probably more complicated for this type than for dependent records. The question is whether there is a simpler way to add dependent records to a type theory.

In this thesis we extend the NuPRL type theory with a simpler and easier to understand primitive type constructor, *dependent intersection*. This is a natural generalization of the standard intersection introduced in [11] and [37]. Dependent intersection is an intersection of *two* types, where the second type may depend on elements of the first one. This type constructor is built by analogy to dependent products: elements of dependent products are pairs where the type of the second component may depend on the first component. We will show that dependent intersection allows us to define the record type in a very simple way. Our definition of records is extensionally equal to Hickey's, but is far simpler. Moreover our constructors (unlike Hickey's) allow us to extend record types. For example, having a definition of monoids we can define groups by extending this definition rather than repeating the definition of monoid.

## 3.1 Dependent Intersection

We extend the definition of intersection $A \cap B$ to a case when the type $B$ can depend on elements of the type $A$. Let $A$ be a type and $B[x]$ be a type for all $x$ of type $A$. We define a new type, *dependent intersection* $x{:}A \cap B[x]$. This type contains all elements $a$ from $A$ such that $a$ is also in $B[a]$ (see below for equality).

**Remark 3.4** *Do not confuse* the dependent intersection *with* the intersection of a family of types $\bigcap_{x:A} B[x]$. *The latter refers to an intersection of types $B[x]$ for all $x$ in $A$. The difference between these two type constructors is similar to the difference between dependent products $x{:}A \times B[x] = \Sigma_{x:A} B[x]$ and the product of a family of types $\Pi_{x:A} B[x] = x : A \to B[x]$.*

**Example 3.5** *The ordinary binary intersection is just a special case of a dependent intersection with a constant second argument: $A \cap B = x : A \cap B$.*

**Example 3.6** *Let $A = \mathbb{Z}$ and $B[x] = [0 \mathrel{..} x^2{-}5]$. Then $x : A \cap B[x]$ is a set of all integers, such that $0 \le x \le x^2 - 5$.*

Two elements $a$ and $a'$ are equal in the dependent intersection $x{:}A \cap B[x]$ when they are equal both in $A$ and $B[a]$.

**Example 3.7** *Let $A$ be $\{0\} \to \mathbb{N}$ and $B[f]$ be $\{1\} \to [0 \mathrel{..} f(0)]$, where $\{0\}$ and $\{1\}$ are types that contain only one element ($0$ and $1$ respectively). Then $x{:}A \cap B[x]$ is a type of functions $f$ that map $0$ to a natural number $n_0$ and map $1$ to a natural number $n_1 \in [0 \mathrel{..} n_0]$. Two such functions $f$ and $f'$ are equal in this type, when first, $f = f' \in \{0\} \to \mathbb{N}$, i.e. $f(0) = f'(0)$, and second, $f = f' \in \{1\} \to [0 \mathrel{..} f(0)]$, i.e. $f(1) = f'(1) \le f(0)$.*

### 3.1.1 Semantics

We are going to give the formal semantics for dependent intersection types based on the predicative PER semantics, for the NuPRL type theory [2, 3]. In the PER semantics types are interpreted as partial equivalence relations (PERs) over terms. Partial equivalence relations are relations that are transitive and symmetric, but not necessary reflexive.

According to [3], to give the semantics for a type expression $A$ we need to determine when this expression is a well-formed type, define elements of this type, and specify the partial equivalence relation on terms for this type ($a = b \in A$). We should also give an equivalence relation on types, i.e. determine when two types are equal. See [3] for details.

**Extension of the Semantics** We introduce a new term constructor for dependent intersection $x : A \cap B[x]$. This constructor bounds the variable $x$ in $B[x]$. We extend the semantics of [3] as follows.

- The expression $x : A \cap B[x]$ is a well-formed type if and only if $A$ is a type and $B[x]$ is a functional type over $x : A$. That is, for any $x$ from $A$ the expression $B[x]$ should be a type and if $x = x' \in A$ then $B[x] = B[x']$.
- The elements of the well-formed type $x : A \cap B[x]$ are such terms $a$ that $a$ is an element of both types $A$ and $B[a]$.

Table 3.1: Rules for dependent intersection

$$\frac{\Gamma \vdash A \text{ Type} \qquad \Gamma; x : A \vdash B[x] \text{ Type}}{\Gamma \vdash (x : A \cap B[x]) \text{ Type}} \qquad (TypeFormation)$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma; x : A \vdash B[x] = B'[x]}{\Gamma \vdash (x : A \cap B[x]) = (x : A' \cap B'[x])} \qquad (TypeEquality)$$

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash a \in B[a] \qquad \Gamma \vdash x : A \cap B[x] \text{ Type}}{\Gamma \vdash a \in (x : A \cap B[x])} \qquad (Introduction)$$

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash a = a' \in B[a] \qquad \Gamma \vdash x : A \cap B[x] \text{ Type}}{\Gamma \vdash a = a' \in (x : A \cap B[x])} \qquad (Equality)$$

$$\frac{\Gamma; u : (x : A \cap B[x]); \Delta; x : A; y : B[x] \vdash C[x, y]}{\Gamma; u : (x : A \cap B[x]); \Delta \vdash C[u, u]} \qquad (Elimination)^1$$

[1]In pairwise functionality we can make this rule stronger, cf. Section 2.3.6

- Two elements $a$ and $a'$ are equal in the well-formed type $x : A \cap B[x]$ iff $a = a' \in A$ and $a = a' \in B[a]$.
- Two types $x : A \cap B[x]$ and $x : A' \cap B'[x]$ are equal when $A$ and $A'$ are equal types and for all $x$ and $y$ from $A$ if $x = y \in A$ then $B[x] = B'[y]$.

### 3.1.2 The Inference Rules

The corresponding inference rules are shown in Table 3.1.

**Theorem 3.8** *All rules of Table 3.1 are valid in the semantics given above.*

This theorem is proved by straightforward application of the semantics definition.

**Theorem 3.9** *The following rules can be derived from the primitive rules of Table 3.1 in a type theory with the appropriate cut rule.*

$$\frac{\Gamma \vdash a = a' \in (x : A \cap B[x])}{\Gamma \vdash a = a' \in A}$$

$$\frac{\Gamma \vdash a = a' \in (x : A \cap B[x])}{\Gamma \vdash a = a' \in B[a]}$$

**Theorem 3.10** *Dependent intersection is associative, i.e.*

$$x : A \cap (y : B[x] \cap C[x, y]) =_e z : (x : A \cap B[x]) \cap C[z, z]$$

The formal proof is checked by the MetaPRL system. We show here a sketch of a proof. An element $x$ has type $a : A \cap (b : B[a] \cap C[a, b])$ iff it has types $A$ and $b : B[x] \cap C[x, b]$. The latter is a case iff $x \in B[x]$ and $x \in C[x, x]$. On the other hand, $x$ has type $ab : (a : A \cap B[a]) \cap C[ab, ab]$ iff $x \in (a : A \cap B[a])$ and $x \in C[x, x]$. The former means that $x \in A$ and $x \in B[x]$. Therefore $x \in a : A \cap (b : B[a] \cap C[a, b])$ iff $x \in A$ and $x \in B[x]$ and $x \in C[x, x]$ iff $x \in ab : (a : A \cap B[a]) \cap C[ab, ab]$.

## 3.2 Records

We are going to define record types using dependent intersection. In this section we informally describe what properties we are expecting from records. The formal definitions are presented in Section 3.3.

### 3.2.1 Plain Records

Records are collections of labeled fields. We use the following notations for records:

$$\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\} \tag{3.1}$$

where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are *labels* and $a_1, \ldots a_n$ are corresponding field values. Usually labels have a string type, but generally speaking labels can be of any fixed type $Label$ with a decidable equality. We will use the `true type` font for labels.

The selection operator $r.\mathtt{x}$ is used to access record fields. If $r$ is a record then $r.\mathtt{x}$ is a field of this record labeled $\mathtt{x}$. That is we expect the following reduction rule:

$$\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}.\mathtt{x}_i \longrightarrow a_i \tag{3.2}$$

Fields may have different types. If each $a_i$ has type $A_i$ then the whole record (3.1) has the type

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}. \tag{3.3}$$

Also we want the natural typing rule for the field selection: for any record $r$ of the type (3.3) we should be able to conclude that $r.\mathtt{x}_i \in A_i$.

The main difference between record types and products $A_1 \times \cdots \times A_n$ is the that record type has the *subtyping property*. Given two records $R_1$ and $R_2$, if any label declared in $R_1$ as a field of type $A$ is also declared in $R_2$ as a field of type $B$, such that $B \subseteq A$, then $R_2$ is subtype of $R_1$. In particular,

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} \subseteq \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_m : A_m\} \tag{3.4}$$

where $m < n$.

**Example 3.11** *Let*

$$Point = \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\} \text{ and } ColorPoint = \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}; \mathtt{color} : Color\}.$$

*Then the record* $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\}$ *is not only a* $ColorPoint$*, but it is also a* $Point$*, so we can use this record whenever* $Point$ *is expected. For example, we can use it as an argument of the function of the type* $Point \to T$*. Further the result of this function does not depend whether we use* $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\}$ *or* $\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = green\}$*. That is, these two records are equal as elements of the type* $Point$*, i.e.*

$$\begin{array}{l}\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = red\} = \\ \{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = green\} \in \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\}\end{array}$$

Using subtyping one can model the private fields. Consider a record $r$ that has one "private" field $\mathtt{x}$ of the type $A$ and one "public" field $\mathtt{y}$ of the type $B$. This record has the type $\{\mathtt{x} : A; \mathtt{y} : B\}$. Using the subtyping property we can conclude that it also has type $\{\mathtt{y} : B\}$. Now we can consider type $\{\mathtt{y} : B\}$ as a public interface for this record. A user knows only that $r \in \{\mathtt{y} : B\}$. Therefore the user has access to field $\mathtt{y}$, but access to field $\mathtt{x}$ would be type invalid (i.e. untyped). Formally it means that a function of the type $\{\mathtt{y} : B\} \to T$ can access only the field $\mathtt{y}$ on its argument (although an argument of this function can have other fields).

Further, records' equality does not depend on field ordering. For example, $\{\mathtt{x} = 0; \mathtt{y} = 1\}$ should be equal to $\{\mathtt{y} = 1; \mathtt{x} = 0\}$, moreover $\{\mathtt{x} : A; \mathtt{y} : B\}$ and $\{\mathtt{y} : B; \mathtt{x} : A\}$ should define the same type.

**Records as Dependent Functions**

Records may be considered as mappings from labels to the corresponding fields. Therefore it is natural to define a record type as a function type with the domain $Label$ (cf. [8]). Since the types of each field may vary, one should use dependent function type (i.e., $\Pi$ type). Let $Field[l]$ be a type of a field labeled $l$. For example, for the record type (3.3) take

$$\begin{aligned} Field[l] \triangleq\ & \texttt{if}\ l = \mathtt{x}_1\ \texttt{then}\ A_1\ \texttt{else} \\ & \ldots \\ & \texttt{if}\ l = \mathtt{x}_n\ \texttt{then}\ A_n\ \texttt{else}\ \mathrm{Top} \end{aligned}$$

Then define the record type as the dependent function type:

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} \triangleq l : Label \to Field[l]. \tag{3.5}$$

Now records may be defined as functions:

$$\begin{aligned} & \{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\} \triangleq \\ & \qquad \lambda l. \texttt{if}\ l = \mathtt{x}_1\ \texttt{then}\ a_1\ \texttt{else} \\ & \qquad\quad \ldots \\ & \qquad\quad \texttt{if}\ l = \mathtt{x}_n\ \texttt{then}\ a_n \end{aligned} \tag{3.6}$$

And selection is defined as application:

$$r.l \triangleq r\, l \tag{3.7}$$

One can see that these definitions meet the expected properties mentioned above including the subtyping property.

**Records as Intersections**

Using the above definitions we can prove that in the case when all $\mathtt{x}_i$'s are distinct labels

$$\{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\} =_e \{\mathtt{x}_1 : A_1\} \cap \cdots \cap \{\mathtt{x}_n : A_n\}. \tag{3.8}$$

This property provides us a simpler way to define records. First, let us define the type of records with only one field. We define it as a function type like we did it in the last section, but for single-field records we do not need dependent functions, so we may simplify the definition:

$$\{\mathtt{x} : A\} \triangleq \{\mathtt{x}\} \to A \tag{3.9}$$

where $\{\mathtt{x}\}$ is the singleton subset of type $Label$. Now we may take (3.8) and (3.9) as a definition of an arbitrary record type instead of (3.5) and keep definitions (3.6) and (3.7). This way was used in [38] where $\{\mathtt{x} : A\}$ was a primitive type.

**Example 3.12** *The record $\{\mathtt{x} = 1; \mathtt{y} = 2\}$ by definition (3.6) is a function that maps $\mathtt{x}$ to $1$ and $\mathtt{y}$ to $2$. Therefore it has type $\{\mathtt{x}\} \to \mathbb{Z} = \{\mathtt{x} : \mathbb{Z}\}$ and also has type $\{\mathtt{y}\} \to \mathbb{Z} = \{\mathtt{y} : \mathbb{Z}\}$. Hence it has type $\{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\} = \{\mathtt{x} : \mathbb{Z}\} \cap \{\mathtt{y} : \mathbb{Z}\}$.*

One can see that when all labels are distinct, definitions (3.5) and (3.8)+(3.9) are equivalent. That is, for any record expression $\{x_1 : A_1; \ldots; x_n : A_n\}$ where $x_i \neq x_j$, these two definitions define two extensionally equal types.

However, definitions (3.8)+(3.9) differ from the traditional ones in the case when labels coincide. Most record calculi prohibit repeating labels in the declaration of record types, e.g., they do not recognize the expression $\{\mathtt{x} : A; \mathtt{x} : B\}$ as a valid type. On the other hand, in [20] in the case when labels coincide the last field overlaps the previous ones, e.g., $\{\mathtt{x} : A; \mathtt{x} : B\}$ is equal to $\{\mathtt{x} : B\}$. In both these cases many typing rules of the record calculus need some additional conditions that prohibit coincident labels. For example, the subtyping relation (3.4) would be true only when all labels $\mathtt{x}_i$ are distinct.

We will follow the definition (3.8) and allow repeated labels and assume that

$$\{\mathtt{x} : A; \mathtt{x} : B\} = \{\mathtt{x} : A \cap B\}. \tag{3.10}$$

This may look unusual, but this notation significantly simplifies the rules of the record calculus, because we do not need to worry about coincident labels. Moreover, this allows us to have multiple inheritance (see Section 3.3.3 for an example). Note that the equation (3.10) holds also in [10].

### 3.2.2 Dependent Records

We want to be able to represent abstract data types and algebraic structures as records. For example, a semigroup may be considered as a record with the fields `car` (representing a carrier) and `product` (representing a binary operation). The type of `car` is the universe $\mathbb{U}$. The type of `product` should be $\mathtt{car} \times \mathtt{car} \to \mathtt{car}$. The problem is that the type of `product` depends on the value of the field `car`. Therefore we cannot use plain record types to represent such structures.

We need dependent records [5, 20, 36]. In general a dependent record type has the following form

$$\{\mathtt{x} : A; \mathtt{y} : B[\mathtt{x}]; \mathtt{z} : C[\mathtt{x}, \mathtt{y}]; \ldots\} \tag{3.11}$$

That is, the type of a field in such records can depend on the values of the previous fields.

The following main property shows the intended meaning of this type.

The record $\{\mathtt{x} = a; \mathtt{y} = b; \mathtt{z} = c; \ldots\}$ has type (3.11) if and only if

$$a \in A, \quad b \in B[a], \quad c \in C[a, b], \quad \ldots$$

**Example 3.13** *Let $SemigroupSig$ be the record type that represents the signature of semigroups:*

$$SemigroupSig \triangleq \{\mathtt{car} : \mathbb{U}; \mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car}\}.$$

*Semigroups are elements of $SemigroupSig$ satisfying the associativity axiom. This axiom may be represented as an additional field:*

$$\begin{aligned} Semigroup \triangleq \{&\mathtt{car} : \mathbb{U}; \\ &\mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car}; \\ &\mathtt{axm} : \forall x, y, z : \mathtt{car}.\ (x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z)\} \end{aligned}$$

*where $x \cdot y$ stands for* $\mathtt{product}(x, y)$.

#### Dependent Records as Very Dependent Functions

We cannot define the dependent record type using the ordinary dependent function type, because the type of the fields depends not only on labels, but also on values of other fields.

To represent dependent records Hickey [20] introduced the *very dependent function* type constructor:

$$\{f \mid x : A \to B[f, x]\} \tag{3.12}$$

Here $A$ is the domain of the function type and the range $B[f, x]$ can depend on the argument $x$ and the function $f$ itself. That is, type (3.12) refers to the type of all functions $g$ with the domain $A$ and the range $B[g, a]$ on any argument $a \in A$.

For instance, $SemigroupSig$ can be represented as a very dependent function type

$$SemigroupSig \triangleq \{r \mid l : Label \to Field[r, l]\} \tag{3.13}$$

where $Field[r,l] \triangleq$

$$\begin{array}{l} \texttt{if } l = \texttt{car} \texttt{ then } \mathbb{U} \texttt{ else} \\ \texttt{if } l = \texttt{product} \texttt{ then } r.\texttt{car} \times r.\texttt{car} \to r.\texttt{car} \\ \ \texttt{else } \mathrm{Top} \end{array}$$

Not every very dependent function type has a meaning. For example the range of the function on argument $a$ cannot depend on $f(a)$ itself. For instance, the expression

$$\{f \mid x : A \to f(x)\}$$

is not a well-formed type.

The type (3.12) is well-formed if there is some well-founded order $<$ on the domain $A$, and the range type $B[x,f]$ on $x = a$ depends only on values $f(b)$, where $b < a$. The requirement of well-founded order makes the definition of very-dependent functions very complex. See [20] for more details.

**Dependent Records as Dependent Intersection**

By using dependent intersection we can avoid the complex concept of very dependent functions. For example, we may define

$$\begin{array}{l} SemigroupSig \triangleq \mathit{self} : \{\texttt{car} : \mathbb{U}\} \cap \\ \{\texttt{product} : \mathit{self}.\texttt{car} \times \mathit{self}.\texttt{car} \to \mathit{self}.\texttt{car}\} \end{array}$$

Here $\mathit{self}$ is a bound variable that is used to refer to the record itself considered as a record of the type $\{\texttt{car} : \mathbb{U}\}$. This definition can be read as follows:

> $r$ has type $SemigroupSig$, when first, $r$ is a record with a field `car` of the type $\mathbb{U}$, and second, $r$ is a record with a field `product` of the type $r.\texttt{car} \times r.\texttt{car} \to r.\texttt{car}$.

This definition of the $SemigroupSig$ type is extensionally equal to (3.13), but it has two advantages. First, it is much simpler. Second, dependent intersection allows us to extend the $SemigroupSig$ type to the $Semigroup$ type by adding an extra field `axm`:

$$\begin{array}{l} Semigroup \triangleq \mathit{self} : SemigroupSig \cap \\ \{\texttt{axm} : \forall x,y,z : \mathit{self}.\texttt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\} \end{array}$$

where $x \cdot y$ stands for $\mathit{self}.\texttt{product}(x,y)$.

We can define a dependent record type of an arbitrary length in this fashion as a dependent intersection of single-field records associated to the left.

Note that $Semigroup$ can be also defined as an intersection associated to the right: $Semigroup =$

$$\begin{array}{rl} r_c : & \{\texttt{car} : \mathbb{U}\} \cap \\ \big(r_p : & \{\texttt{product} : r_c.\texttt{car} \times r_c.\texttt{car} \to r_c.\texttt{car}\} \cap \\ & \{\texttt{axm} : \forall x,y,z : r_c.\texttt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\}\big) \end{array}$$

where $x \cdot y$ stands for $r_p.\texttt{product}(x,y)$. Here $r_c$ and $r_p$ are bound variables. Both of them refer to the record itself, but $r_c$ has type $\{\texttt{car} : \mathbb{U}\}$ and $r_p$ has type $\{\texttt{product} : \dots\}$. These two definitions are equal, because of associativity of dependent intersection (Theorem 3.10).

Note that Pollack [36] considered two types of dependent records: left associating records and right associating records. However, in our framework left and right association are just two different ways of building the same type. We will allow using both of them. Which one to choose is the matter of taste.

## 3.3 The Record Calculus

### 3.3.1 The Formal Definition

Now we are going to give the formal definition of records using dependent intersection.

**Records**

Elements of record types are defined as functions from labels to the corresponding fields. We need three primitive operations:

1. Empty record: $\{\} \triangleq \lambda l.l$
   (We could pick any function as a definition of an empty record.)

2. Field update/extension:

$$r.(\mathtt{x} := a) \triangleq (\lambda l.\mathtt{if}\ l = \mathtt{x}\ \mathtt{then}\ a\ \mathtt{else}\ r\ l)$$

3. Field selection: $r.\mathtt{x} \triangleq r\ x$

We can construct any record by these operations: we define $\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}$ as

$$\{\}.(\mathtt{x}_1 := a_1).(\mathtt{x}_2 := a_2). \ \ldots \ .(\mathtt{x}_n := a_n)$$

**Record Types**

**Single-field record type** is defined as

$$\{\mathtt{x} : A\} \triangleq \{\mathtt{x}\} \to A$$

where $\{\mathtt{x}\} \triangleq \{l : \mathit{Label} \mid l = \mathtt{x} \in \mathit{Label}\}$ is a singleton set.

**Independent concatenation** of record types is defined as

$$\{R_1; R_2\} \triangleq R_1 \cap R_2 \tag{3.14}$$

This definition is a partial case of the below definition of left associating records when $R_2$ does not depend on $\mathit{self}$.

**Left associating dependent concatenation** of record types is defined as

$$\{\mathit{self} : R_1; R_2[\mathit{self}]\} \triangleq \mathit{self} : R_1 \cap R_2[\mathit{self}] \tag{3.15}$$

*Syntactical Remarks* Here variable $\mathit{self}$ is bounded in $R_2$. When we use the name "self" for this variable, we can use the shortening $\{R_1; R_2[\mathit{self}]\}$ for this type. Further, we will omit "$\mathit{self}$." in the body of $R_2$, e.g. we will write just $\mathtt{x}$ for $\mathit{self}.\mathtt{x}$, when such notation does not lead to misunderstanding. We assume that this concatenation is a left associative operation and we will omit inner braces. For example, we will write $\{\mathtt{x} : A;\mathtt{y} : B[\mathit{self}];\mathtt{z} : C[\mathit{self}]\}$ instead of $\{\{\{\mathtt{x} : A\}; \{\mathtt{y} : B[\mathit{self}]\}\}; \{\mathtt{z} : C[\mathit{self}]\}\}$. Note that in this expression there are two distinct bound variables $\mathit{self}$. The first one is bound in $B$ and refers to the record itself as a record of the type $\{\mathtt{x} : A\}$. The second $\mathit{self}$ is bound in $C$; it also refers to the same record, but it has type $\{\mathtt{x} : A; \mathtt{y} : B[\mathit{self}]\}$.

**Right associating dependent concatenation.** The above definitions are enough to form any record type, but to complete the picture we give the definition of right associating record constructor:

$$\{x : \mathtt{x} : A; R[x]\} \triangleq \mathit{self} : \{\mathtt{x} : A\} \cap R[\mathit{self}.\mathtt{x}] \tag{3.16}$$

Table 3.2: Inference rules for records

**Reduction rules**

$(r\text{.}\mathtt{x} := a)\text{.}\mathtt{x} \longrightarrow a$

$(r\text{.}\mathtt{y} := b)\text{.}\mathtt{x} \longrightarrow r\text{.}\mathtt{x}$ when $\mathtt{x} \neq \mathtt{y}$.

In particular: $\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}\text{.}\mathtt{x}_i \longrightarrow a_i$ when all $\mathtt{x}_i$'s are distinct.

**Type formation**

*Single-field record:*

$$\frac{\Gamma \vdash A \text{ Type} \quad \Gamma \vdash \mathtt{x} \in \mathit{Label}}{\Gamma \vdash \{\mathtt{x} : A\} \text{ Type}}$$

*Dependent record:*

$$\frac{\Gamma \vdash R_1 \text{ Type} \qquad \Gamma; \mathit{self} : R_1 \vdash R_2[\mathit{self}] \text{ Type}}{\Gamma \vdash \{R_1; R_2[\mathit{self}]\} \text{ Type}}$$

*Independent record:*

$$\frac{\Gamma \vdash R_1 \text{ Type} \quad \Gamma \vdash R_2 \text{ Type}}{\Gamma \vdash \{R_1; R_2\} \text{ Type}}$$

*Right associating record:*

$$\frac{\Gamma \vdash \{\mathtt{x} : A\} \text{ Type} \qquad \Gamma; x : A \vdash R[x] \text{ Type}}{\Gamma \vdash \{x : \mathtt{x} : A; R[x]\} \text{ Type}}$$

**Introduction (membership rules)**

*Single-field record:*

$$\frac{\Gamma \vdash a \in A \qquad \Gamma \vdash \mathtt{x} \in \mathit{Label}}{\Gamma \vdash r\text{.}\mathtt{x} := a \in \{\mathtt{x} : A\}} \qquad \frac{\Gamma \vdash r \in \{\mathtt{x} : A\} \qquad \Gamma \vdash \mathtt{x} \neq \mathtt{y} \in \mathit{Label}}{\Gamma \vdash (r\text{.}\mathtt{y} := b) = r \in \{\mathtt{x} : A\}}$$

*Independent record:*

$$\frac{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2}{\Gamma \vdash r \in \{R_1; R_2\}}$$

*Dependent record:*

$$\frac{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2[r] \qquad \Gamma \vdash \{R_1; R_2[\mathit{self}]\} \text{ Type}}{\Gamma \vdash r \in \{R_1; R_2[\mathit{self}]\}}$$

*Right associating record:*

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A\} \qquad \Gamma \vdash r \in R[r\text{.}\mathtt{x}] \qquad \Gamma \vdash \{x : \mathtt{x} : A; R[x]\} \text{ Type}}{\Gamma \vdash r \in \{x : \mathtt{x} : A; R[x]\}}$$

**Elimination (inverse typing rules)**[1]

*Single-field record:*

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A\}}{\Gamma \vdash r\text{.}\mathtt{x} \in A}$$

*Dependent record:*

$$\frac{\Gamma \vdash r \in \{R_1; R_2[\mathit{self}]\}}{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2[r]}$$

*Independent record:*

$$\frac{\Gamma \vdash r \in \{R_1; R_2\}}{\Gamma \vdash r \in R_1 \qquad \Gamma \vdash r \in R_2}$$

*Right associating record:*

$$\frac{\Gamma \vdash r \in \{x : \mathtt{x} : A; R[x]\}}{\Gamma \vdash r\text{.}\mathtt{x} \in A \qquad \Gamma \vdash r \in R[r\text{.}\mathtt{x}]}$$

[1]See also Chapter 4

*Syntactical Remarks* Here $x$ is a variable bound in $R$ that represents a field $\mathtt{x}$. Note that we may $\alpha$-convert the variable $x$, but not a label $\mathtt{x}$, e.g., $\{x : \mathtt{x} : A; R[x]\} = \{y : \mathtt{x} : A; R[y]\}$, but $\{x : \mathtt{x} : A; R[x]\} \neq \{y : \mathtt{y} : A; R[y]\}$. We will usually use the same name for labels and corresponding bound variables. This connection is right associative, e.g., $\{x : \mathtt{x} : A; y : \mathtt{y} : B[x]; \mathtt{z} : C[x, y]\}$ stands for $\{x : \mathtt{x} : A; \{y : \mathtt{y} : B[x]; \{\mathtt{z} : C[x, y]\}\}\}$.

### 3.3.2 The Rules

The basic rules of our record calculus are shown in Table 3.2. The elimination rules in this table are weak. We will discuss stronger rule in Chapter 4.

**Theorem 3.14** *All the rules of Table 3.2 are derivable from the definitions given above.*

From the reduction rules we get:

$$\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}.\mathtt{x}_i \longrightarrow a_i$$

when all $\mathtt{x}_i$'s are distinct.

We do not show the equality rules here, because in fact, these rules repeat rules in Table 3.2 and can be derived from them using substitution rules in our type theory. For example, we can prove the following rules

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash \mathtt{x} = \mathtt{x}' \in \mathit{Label}}{\Gamma \vdash (r.\mathtt{x} := a) = (r'.\mathtt{x}' := a') \in \{\mathtt{x} : A\}}$$

$$\frac{\Gamma \vdash r = r' \in R_1 \qquad \Gamma \vdash r = r' \in R_2}{\Gamma \vdash r = r' \in \{R_1; R_2\}}$$

In particular, we can prove that

$$\begin{gathered}\{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = \mathit{red}\} = \\ \{\mathtt{x} = 0; \mathtt{y} = 0; \mathtt{color} = \mathit{green}\} \in \{\mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z}\}\end{gathered}$$

We can also derive the following subtyping properties:

$$\begin{gathered}\{R_1; R_2\} \subseteq R_1 \\ \{R_1; R_2\} \subseteq R_2 \\ \{R_1; R_2[\mathit{self}]\} \subseteq R_1 \\ \{x : \mathtt{x} : A; R[x]\} \subseteq \{\mathtt{x} : A\}\end{gathered}$$

$$\frac{\vdash R_1 \subseteq R_1' \qquad \mathit{self} : R_1 \vdash R_2[\mathit{self}] \subseteq R_2'[\mathit{self}]}{\vdash \{R_1; R_2[\mathit{self}]\} \subseteq \{R_1'; R_2'[\mathit{self}]\}}$$

$$\frac{\vdash A \subseteq A' \qquad x : A \vdash R[x] \subseteq R'[x]}{\vdash \{x : \mathtt{x} : A; R[x]\} \subseteq \{x : \mathtt{x} : A'; R'[x]\}}$$

Further, we can establish two facts that state the equality of left and right associating records.
$\{x : \mathtt{x} : A; R[x]\} =_e \{\mathtt{x} : A; R[\mathit{self}.\mathtt{x}]\}$,
and
$\{R_1; \{x : \mathtt{x} : A[\mathit{self}]; R_2[\mathit{self}, x]\}\} =_e$
$\{\{R_1; \mathtt{x} : A[\mathit{self}]\}; R_2[\mathit{self}, \mathit{self}.\mathtt{x}]\}$.
For example, using these two equalities we can prove that
$\{\mathtt{x} : A; \mathtt{y} : B[\mathit{self}.\mathtt{x}]; \mathtt{z} : C[\mathit{self}.\mathtt{x}; \mathit{self}.\mathtt{y}]\} =_e$
$\{x : \mathtt{x} : A; y : \mathtt{y} : B[x]; \mathtt{z} : C[x; y]\}$.

### 3.3.3 Examples

#### Semigroup Example

Now we can define the $SemigroupSig$ type in two ways:

$$\{\texttt{car} : \mathbb{U}; \texttt{product} : \texttt{car} \times \texttt{car} \to \texttt{car}\} \quad \text{or}$$
$$\{car : \texttt{car} : \mathbb{U}; \texttt{product} : car \times car \to car\}$$

Note that in the first definition `car` in the declaration of `product` stands for *self*.`car`, and in the second definition $car$ is just a bound variable.

We can define $Semigroup$ by extending $SemigroupSig$:

$$\{SemigroupSig; \texttt{axm} : \forall x, y, z : \texttt{car} \quad (x{\cdot}y){\cdot}z = x{\cdot}(y{\cdot}z)\}$$

or as a right associating record:

$$\begin{aligned}\{&car : \texttt{car} : \mathbb{U};\\ &product : \texttt{product} : car \times car \to car;\\ &\texttt{axm} : \forall x, y, z : car \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\}\end{aligned}$$

In the first case $x \cdot y$ stands for *self*.`product`$(x, y)$ and in the second case for just $product(x, y)$.

#### Multiply Inheriting Example

A monoid is a semigroup with a unit. So,

$$MonoidSig \triangleq \{SemigroupSig; \texttt{unit} : \texttt{car}\}$$

A monoid is an element of $MonoidSig$ which satisfies the axiom of semigroups and an additional property of the unit. That is, $Monoid$ inherits fields from both $MonoidSig$ and $Semigroup$. We can define the $Monoid$ type as follows:

$$\begin{aligned}Monoid \triangleq \{\{\ &MonoidSig; Semigroup;\\ &\texttt{unit\_axm} : \forall x : \texttt{car} \quad x \cdot \texttt{unit} = x\}\end{aligned}$$

Note that since $MonoidSig$ and $Semigroup$ share the fields `car` and `product`, these two fields are present in the definition of $Monoid$ twice. This does not create problems, since we allow repeating labels (Section 3.2.1).

Now we have the following subtyping relations:

$$\begin{array}{ccc} SemigroupSig & \supset & MonoidSig \\ \cup & & \cup \\ Semigroup & \supset & Monoid \end{array}$$

#### Abstract Data Type

We can also represent abstract data types as dependent records. Consider for example a data structure *collection of elements of a type* $T$. This data structure consists of an abstract type `car` for collections of elements of the type $T$, a constant of this type `empty` to construct an empty collection, and functions `member` $s$ $a$ to inquire if element $a$ is in collection $s$, and `insert` $s$ $a$ to add element $a$ into collection $s$. These functions should satisfy certain properties that guarantee their intended behavior:

1. The empty collection does not have elements.
2. `insert` $s$ $a$ has all elements that $s$ has and element $a$ and nothing more.

A formal definition of the data structure of collections could be written as a record:

$$
\begin{array}{l}
Collection(T) \ \triangleq \\
\{\texttt{car} : \mathbb{U}; \\
\ \texttt{empty} : \texttt{car}; \\
\ \texttt{member} : \texttt{car} \rightarrow T \rightarrow \mathbb{B}; \\
\ \texttt{insert} : \texttt{car} \rightarrow T \rightarrow \texttt{car}; \\
\ \texttt{emp\_axm} : \forall a : T \quad a \notin \texttt{empty} \\
\ \texttt{ins\_axm} : \forall s : \texttt{car} \quad \forall a, b : T \quad (\texttt{member}\ (\texttt{insert}\ s\ a)\ b) \\
\qquad \iff \ (\texttt{member}\ s\ b) \vee (a = b \in T)\}
\end{array}
$$

It Section 6 we will show an example of an implementation of this data structure.

# Chapter 4
# Elimination Rule for Independent Records

In this chapter we consider the question what should be the right elimination rule for the record type. As we will see this question is not as simple as it looks. While the introduction rule for records is very natural and simple, the right elimination rule is not obvious.

In this chapter we will consider independent records for the sake of simplicity.

We will use the following notations: in the inference rules we will use $\Phi[x]$ for $\Delta[x] \vdash C[x]$. For example instead of the rule:

$$\frac{\Gamma; a_1 : A_1; a_2 : A_2; \Delta[\langle a_1, a_2\rangle] \vdash C[\langle a_1, a_2\rangle]}{\Gamma; a : A_1 \times A_2; \Delta[a] \vdash C[a]}$$

we would just write:

$$\frac{\Gamma; a_1 : A_1; a_2 : A_2; \Phi[\langle a_1, a_2\rangle]}{\Gamma; a : A_1 \times A_2; \Phi[a]}$$

## 4.1 Weak Elimination Rule

In Table 3.2 we showed a weak elimination rule for records:

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A; R\}}{\Gamma \vdash r.\mathtt{x} \in A} \qquad \text{(Weak Elimination)}$$

It just said that if $r \in \{\mathtt{x} : A; R\}$ then $r.\mathtt{x} \in A$. This rule is valid and easy to prove, but it turns out that it is too weak in practice.

The correct elimination rule should have a conclusion of the form

$$\Gamma; r : \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}; \Phi[r]$$

## 4.2 Naive Elimination Rule

The elimination rule for records should be dual to the introduction rule. Let us look at the introduction rule. It follows from the rules of Table 3.2 that

$$\frac{\Gamma \vdash a_1 \in A_1 \qquad \ldots \qquad \Gamma \vdash a_n \in A_n}{\Gamma \vdash \{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\} \in \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}} \qquad (4.1)$$

This rule is just an analog of the introduction rule for products:

$$\frac{\Gamma \vdash a_1 \in A_1 \qquad \Gamma \vdash a_2 \in A_2}{\Gamma \vdash \langle a_1, a_2\rangle \in A_1 \times A_2}$$

The elimination rule for products is

$$\frac{\Gamma; a_1 : A_1; a_2 : A_2; \Phi[\langle a_1, a_2\rangle]}{\Gamma; a : A_1 \times A_2; \Phi[a]}$$

One can expect the following elimination rule for records:

$$\frac{\Gamma; a_1 : A_1; \ldots; a_n : A_n; \Phi[\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n\}]}{\Gamma; r : \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}; \Phi[r]} \qquad \text{(Naive Elimination)}$$

But this rule is not valid! Moreover this rule contradicts other basic rules of records. Indeed, consider the simple case when $n = 1$. In this case this rule says that any record of the type $\{\mathtt{x} : A\}$ has a form $\{\mathtt{x} = a\}$. But this is clearly not true. For example, the record $\{\mathtt{x} = a; \mathtt{y} = b\}$ also has this type. So the above elimination rule would be invalid if $\Phi[r]$ refers to fields of $r$ other than $\mathtt{x}$. For example, there is a proposition $C[z]$ such that $C[z]$ is true when $z$ is $\{\mathtt{x} = a\}\bullet\mathrm{y}$, but is not true for all $z$. E.g. let $C[z] = (z \equiv \{\}\bullet\mathrm{y})$. Then the sequent

$$a : A \vdash C[\{\mathtt{x} = a\}\bullet\mathrm{y}]$$

would be true. Applying the Naive Elimination rule, we get:

$$r : \{\mathtt{x} : A\} \vdash C[r\bullet\mathrm{y}]$$

Therefore, since $\{\mathtt{x} = a; \mathrm{y} = b\} \in \{\mathtt{x} : A\}$, we get that $C[b]$ for any $b$. Contradiction.

This example shows us that one should be careful when choosing elimination rules for records. It also shows why it is important to define records and prove all rules, rather than take them as a primitive type with a bunch of new axioms.

## 4.3 Strong Elimination Rule

The mistake made in the last section is that (4.1) does not actually capture the whole introduction rule. It does not say that records of type $\{x_i : A_i\}$ could have additional fields. The complete introduction rule (derived from the rules of Table 3.2) is the following:

$$\frac{\Gamma \vdash a_1 \in A_1 \qquad \ldots \qquad \Gamma \vdash a_n \in A_n \qquad r \in \{\}}{\Gamma \vdash \{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n; r\} \in \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}}$$

where $\{\}$ is the record type with empty declaration (it contains all records). The dual rule would be:

$$\frac{\Gamma; a_1 : A_1; \ldots; a_n : A_n; r : \{\}; \Phi[\{\mathtt{x}_1 = a_1; \ldots; \mathtt{x}_n = a_n; r\}]}{\Gamma; r : \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}; \Phi[r]} \qquad \text{(Strong Elimination)}$$

The Strong Elimination rule captures our intuition of record types. We can also state it as two rules:

$$\frac{\Gamma; a : A; r : R; \Phi[\{\mathtt{x} = a; r\}]}{\Gamma; r : \{\mathtt{x} : A; R\}; \Phi[r]} \qquad \text{(Strong Elimination}_1\text{)}$$

and

$$\frac{\Gamma; a : A; r : \{\}; \Phi[\{\mathtt{x} = a; r\}]}{\Gamma; r : \{\mathtt{x} : A\}; \Phi[r]} \qquad \text{(Strong Elimination}_2\text{)}$$

It follows from this rule that if $r \in \{\mathtt{x} : A\}$ then

$$r \equiv \{\mathtt{x} = r\bullet\mathrm{x}; r\}$$

We will call this $\eta$-reduction for records. We will see that this reduction is actually equivalent to the Strong Elimination rule.

Unfortunately the $\eta$-reduction (and therefore the Strong Elimination rule) is invalid when records are defined as functions (definition (3.9)) and

$$\{\mathtt{x} = a; r\} \triangleq (\lambda l.\mathtt{if}\ \ l = \mathtt{x}\ \ \mathtt{then}\ \ a\ \ \mathtt{else}\ \ r\ \ l)$$

Indeed, the $\eta$-reduction says that any element of a record type has the form $\{\mathtt{x} = r\bullet\mathrm{x}; r\}$. But this is not true for all functions with domain $Label$. For example, if $a \in A$ then by definition (3.9) $\lambda l.a \in \{\mathtt{x} : A\}$. Note that this function could be applied to any argument $l$, not only to labels. On the other hand, function $\lambda l.\mathtt{if}\ \ l = \mathtt{x}\ \ \mathtt{then}\ \ a\ \ \mathtt{else}\ \ r\ \ l$ could be applied only to $l$ from the type $Label$, because if $l \notin Label$ then the expression $l = \mathtt{x}$ would be undefined, therefore the application would be undefined. Therefore $r \not\equiv \{\mathtt{x} = r\bullet\mathtt{x}; r\}$ for $r = \lambda l.a$. Contradiction.

Note that the Naive Elimination rule contradicts the basic introduction rule of records. Therefore it is not valid for any possible definition of records. On the other hand, the Strong Elimination Rule contradicts only our definition of records. Therefore there is still a hope that we can find a better definition to satisfy this rule.

## 4.4 Functions with Limited Polymorphism

Let us consider the problem with the Strong Elimination rule more closely. We have $\eta$-reduction rule for functions: if $f$ is a function then

$$f \equiv \lambda z.(fz)\ .$$

That means that any function is a $\lambda$-expression. The $\eta$-reduction for records says that if $r$ is a record of the type $\{\mathtt{x} : A\}$ then

$$r \equiv \lambda l.\mathtt{if}\ \ l = \mathtt{x}\ \ \mathtt{then}\ \ r\ \mathtt{x}\ \ \mathtt{else}\ \ r\ l.$$

So, we would like to have the following reduction:

$$\lambda l.(rl) \equiv \lambda l.\mathtt{if}\ \ l = \mathtt{x}\ \ \mathtt{then}\ \ r\ \mathtt{x}\ \ \mathtt{else}\ \ r\ \ l\ \ . \tag{4.2}$$

We can prove only that for any $l$ *from type* $Label$:

$$r\ \ l \equiv \mathtt{if}\ \ l = \mathtt{x}\ \ \mathtt{then}\ \ r\ \mathtt{x}\ \ \mathtt{else}\ \ r\ \ l\ \ . \tag{4.3}$$

Unfortunately, (4.3) does not hold for any $l$ and therefore (4.2) is not true.

The problem is that our definition uses *polymorphic* functions. As a result we may potentially apply the function $r$ to any argument, not only to labels. On the other hand, we never apply it to anything other than labels. We need to have some form of type of functions with *limited polymorphism*. That is, we need a type of functions that can be applied only to elements of a particular type (in our case $Label$).

There is no such type in our type theory. The interesting questions are whether we can add such type, what would be the semantics for it and what would be inference rules for this type. We will not discuss these questions here. But we can *define* such a type in current type theory for some particular cases, e.g., when $Label$ is the type of natural numbers. Informally speaking we can define "integer functions" as long tuples:

$$f = \langle f_0, \langle f_1, \langle f_2, \ldots \rangle\rangle\rangle$$

and applications as taking $n$-th element of the tuple. That is,

$f(0) \triangleq \pi_1 f$

$f(1) \triangleq \pi_1(\pi_2 f)$

$f(2) \triangleq \pi_1(\pi_2(\pi_2 f))$ and so on. We will not give the formal definition, but rather just use the idea of non-polymorphic functions. We are going define records as tuples. It may help intuition to view these tuples as "integer functions".

### 4.4.1 Non-polymorphic Definition of Record Type

Without loss of generality we can assume that labels are natural numbers, i.e., $Label = \mathbb{N}$ (or we can assume that there is a given injection of the label type into $\mathbb{N}$).

We will give a new definition of the type $\{n : A\}$ for any natural number $n$ and any type $A$. Then we define an arbitrary record type (dependent or not dependent) using intersection as in Section 3.3.1.

#### New definition of records

The type $\{n : A\}$ is a type of tuples where the $n$-th element has the type $A$. We define it by induction:

$\{0 : A\} \triangleq A \times \mathrm{Top}$;

$\{n+1 : A\} \triangleq \mathrm{Top} \times \{n : A\}$.

That is, $\{1 : A\} = \mathrm{Top} \times A \times \mathrm{Top}$, $\{2 : A\} = \mathrm{Top} \times \mathrm{Top} \times A \times \mathrm{Top}$, and so on.

Note that Top contain everything. So for example if $a \in A$ then $\langle t_0, \langle a, t_2\rangle\rangle$ is in $\{1 : A\}$ as well as $\langle t_0, \langle a, \langle t_2, \langle t_3, t_4\rangle\rangle\rangle\rangle$

Then we define application (field selection) $r.n$ as the $n$-th element of tuple $r$. We define it by induction:

$r.0 \triangleq \pi_1 r$

$r.(n+1) \triangleq (\pi_2 r).n$

Finally, we define record extension/update $r.n := a$ as updating the $n$-th component to be $a$.

$r.0 := a \triangleq \langle a, \pi_2 r\rangle$

$r.(n+1) := a \triangleq \langle \pi_1 r, (\pi_2 r).n := a\rangle$

These definitions with the definitions (3.14), (3.15), and (3.16) of an arbitrary record type as an intersection of single record types provide the formal account of record types in our theory.

**Old rules are still valid**

The reductions for records from Table 3.2 could be easily proved by induction for our new definitions:

$(r.\mathtt{x} := a).\mathtt{x} \longrightarrow a$ for any $\mathtt{x} \in \mathit{Label}$.

$(r.\mathtt{y} := b).\mathtt{x} \longrightarrow r.\mathtt{x}$ for any $\mathtt{x}, \mathtt{y} \in \mathit{Label}$ when $\mathtt{x} \neq \mathtt{y}$.

We can also prove by induction the rules for single-record types from Table 3.2:

$$\frac{\Gamma \vdash A\ \text{Type} \quad \Gamma \vdash \mathtt{x} \in \mathit{Label}}{\Gamma \vdash \{\mathtt{x} : A\}\ \text{Type}}$$

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash \mathtt{x} \in \mathit{Label}}{\Gamma \vdash r.\mathtt{x} := a \in \{\mathtt{x} : A\}}$$

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A\} \quad \Gamma \vdash \mathtt{x} \neq \mathtt{y} \in \mathit{Label}}{\Gamma \vdash (r.\mathtt{y} := b) = r \in \{\mathtt{x} : A\}}$$

$$\frac{\Gamma \vdash r \in \{\mathtt{x} : A\}}{\Gamma \vdash r.\mathtt{x} \in A}$$

All these rules were proven by induction on $\mathtt{x}$ (and $\mathtt{y}$) and checked in MetaPRL.

All remaining rules from Table 3.2 are still valid, because we have not changed the definition of the record type as an intersection of single record types.

**The $\eta$-reduction for records**

The $\eta$-reduction that was invalid for the old definition, could be easily proven for the new definition:

For any $\mathtt{x} \in \mathit{Label}$ if $r \in \{\mathtt{x} : A\}$ then $r \equiv \{\mathtt{x} = r.\mathtt{x}; r\}$

The proof is based on the fact that

If $p \in A \times B$ then $p \equiv \langle \pi_1 p, \pi_2 p\rangle$

The proof was checked in MetaPRL.

**New equalities**

Another advantage of out new definitions is that now we can exchange record fields. That is, we can prove the following squiggle equality:

$\{\mathtt{x} = a; \mathtt{y} = b; r\} \equiv \{\mathtt{y} = a; \mathtt{x} = b; r\}$ for any $\mathtt{x}, \mathtt{y} \in \mathit{Label}$ when $\mathtt{x} \neq \mathtt{y}$

We can also prove that

$\{\mathtt{x} = a; \mathtt{x} = b; r\} \equiv \{\mathtt{x} = a; r\}$ for any $\mathtt{x} \in \mathit{Label}$

These equalities were proved by induction on $\mathtt{x}$ and $\mathtt{y}$ in MetaPRL.

Note that these equalities were invalid for the old definitions. We could only prove the equalities in a record type. The squiggle equalities gives us more freedom in using them: we can change the order of fields of a record without worrying about its type.

**Efficiency**

Note that our new definition of records assumes that there is an injection (coding function) of type $Label$ into $\mathbb{N}$. It may seems to be very inefficient. Indeed, assume that `car` is a label with a huge number, say 333148. Then it means that record $\{\texttt{car} = A\}$ is a huge tuple with at least 333148 elements. And $\{\texttt{car} = A\}\texttt{.car}$ is reduced to $A$ in 333148 steps. Fortunately we do not need to unfold the definition and does all these steps, since we have proven the rule $(r\texttt{.x} := a)\texttt{.x} \longrightarrow a$ for any label `x`. MetaPRL uses this rule and do the reduction $\{\texttt{car} = A\}\texttt{.car} \longrightarrow A$ in just one step. Therefore we do not need to worry about these huge numbers, there is no difference in the efficiency between old and new definitions.

## 4.5 Functionality

Now let us come back to the record calculus. In the Section 4.4.1 we gave the new definition of records that satisfies the $\eta$-reduction. Our goal was the Strong Elimination rule:

$$\frac{\Gamma; a : A; r : R; \Phi[\{\texttt{x} = a; r\}]}{\Gamma; r : \{\texttt{x} : A; R\}; \Phi[r]}$$

The question is: can we prove this rule from the $\eta$-reduction rule? It turns out that the answer depends on functionality.

### 4.5.1 Elimination Rule in Pairwise Functionality

It is very easy to prove the Strong Elimination rule using the $Let$ rule (Section 2.2.3) in pairwise functionality. Indeed, we need to prove:

$$\Gamma; r : \{\texttt{x} : A; R\}; \Phi[r].$$

Using $\eta$-reduction to replace $r$ by $\{\texttt{x} = r\texttt{.x}; r\}$ we get

$$\Gamma; r : \{\texttt{x} : A; R\}; \Phi[\{\texttt{x} = r\texttt{.x}; r\}].$$

Then noting that $r\texttt{.x} \in A$ and $r \in R$ we can apply rules $Let\ a = r\texttt{.x} \in A$ and $Let\ r' = r \in R$. Then we get

$$\Gamma; r : R; a : A; r' : R; \Phi[\{\texttt{x} = a; r'\}]$$

Then thinning the $r : R$ hypothesis and renaming $r'$ to $r$ we get the original assumption:

$$\Gamma; a : A; r : R; \Phi[\{\texttt{x} = a; r\}]$$

### 4.5.2 Elimination Rule in Pointwise Functionality

The above reasoning does not hold in pointwise functionality. We can prove the weak form of the Strong Elimination rule:

$$\frac{\Gamma; a : A; r : R; \Delta \vdash C[\{\texttt{x} = a; r\}]}{\Gamma; r : \{\texttt{x} : A; R\}; \Delta \vdash C[r]}$$

where $\Delta$ does not depend on $r$.

The original Strong Elimination rule is invalid in pointwise functionality. But we can get almost Strong Elimination rule in pointwise functionality if we introduce a new notion of orthogonality.

**Orthogonality**

Basically we say that a record type $R$ is orthogonal to $\{\texttt{x} = a\}$ if the declaration of $R$ does not contain `x`. Formally, for any type $R$, for any label `x` and for any element $a$ we define a predicate:

$$\{\texttt{x} = a\} \perp R \stackrel{\triangle}{=} \forall r : R.\, r = (r\texttt{.x} := a) \in R$$

It is clear that if $R = \{\mathtt{x}_1 : A_1; \ldots; \mathtt{x}_n : A_n\}$ and all $\mathtt{x}_i$'s differ from $\mathtt{x}$ then $\{\mathtt{x} = a\} \perp R$.
In pointwise functionality we can prove that

$$\frac{\Gamma; a : A \vdash \{\mathtt{x} = a\} \perp R \qquad \Gamma; a : A; r : R; \Phi[\{\mathtt{x} = a; r\}]}{\Gamma; r : \{\mathtt{x} : A; R\}; \Phi[r]}$$

This is the closest version of Strong Elimination rule valid in pointwise functionality. The proof is fairly complicated and uses the rule $PointwiseSubstitute$ (Section 2.2.3). It was checked by MetaPRL.

# Chapter 5
# Other Possible Applications

## 5.1 Sets and Dependent Intersections

The set type constructor allows us to hide part of a witness.

**Example 5.1** *Instead of defining $Semigroup$ type as an extension of $SemigroupSig$ type with an additional field* `axm`, *we could define the $Semigroup$ type as a subset of $SemigroupSig$:*

$$Semigroup \triangleq \{S : SemigroupSig \mid \forall x, y, z : S.\mathtt{car} \ldots\}$$

Now we will show that the set type constructor (which is primitive in our original type theory) may be defined as a dependent intersection as well.

Now consider the following type (squash operator):

$$[P] \triangleq \{x : \text{Top} \mid P\}$$

$[P]$ is an empty type when $P$ is false, and is equal to $Top$ when $P$ is true.

**Theorem 5.2**

$$\{x : T \mid P[x]\} =_e x : T \cap [P[x]] \tag{5.1}$$

We can not take (5.1) as a definition of sets yet, because we defined the squash operator as a set. But actually the squash operator is defined in our type theory as a primitive constructor and rules for the set type depend on the squash operator. (See [32] for the rules for the squash type and explanations why this is a primitive type.) Thus, we can take (5.1) as a definition.

Moreover, the squash operator could be defined using other primitives. For example, one can define the squash type using union:

$$[P] \triangleq \bigcup_{x:P} \text{Top}.$$

*Remark* In is interesting to note that in the presence of Markov's principle [27] there is an alternative way to define $[P]$:

$$[P] \triangleq ((P \equiv> \text{Void}) \equiv> \text{Void})$$

where $A \equiv> B \triangleq \bigcap_{x:A} B$. We will not give any details here, since it is beyond the scope of the thesis.

We can also define sets without $Top$ and squash type. First, define *independent* sets:

$$\{A \mid B\} \triangleq \bigcup_{x:B} A.$$

Then define the set type:

$$\{x : A \mid B[x]\} \triangleq x : A \cap \{A \mid B[x]\}.$$

**The Mystery of Notations** It is very surprising that braces $\{\ldots\}$ were used for sets and for records independently for a long time. But now it turns out that sets and records are almost the same thing, namely, dependent intersection! Compare the definitions for sets and records:

$$\begin{array}{rlcrl} \{x : T \mid & P[x]\} & \triangleq & x : T & \cap\, [P[x]] \\ \{self : R_1; & R_2[self]\} & \triangleq & self : R_1 & \cap\, R_2[self] \end{array}$$

The only differences between them are that we use squash in the first definition and write "|" for sets and ";" for records.

So, we will use the following definitions for records:

$\{self : R_1 \mid R_2[self]\} \triangleq \{self : R_1; [\![R_2[self]]\!]\} = self : R_1 \cap [\![R_2[self]]\!]$

$\{x : \mathtt{x} : A \mid R[x]\} \triangleq \{x : \mathtt{x} : A; [\![R[x]]\!]\} =$
$self : \{\mathtt{x} : A\} \cap [\![R[self.\mathtt{x}]]\!]$

This gives us the right to use the shortening notations as in Section 3.3.1 to omit inner braces and "*self*". For example, we can rewrite the definition of the $Semigroup$ type as

$$\begin{aligned} Semigroup \triangleq \{&\mathtt{car} : \mathbb{U}; \\ &\mathtt{product} : \mathtt{car} \times \mathtt{car} \to \mathtt{car} \mid \\ &\forall x, y, z : \mathtt{car} \quad (x \cdot y) \cdot z = x \cdot (y \cdot z)\} \end{aligned}$$

**Remark** Note that we cannot define dependent intersection as a set:

$$x : A \cap B[x] \triangleq \{x : A \mid x \in B[x]\}. \qquad \text{(wrong!)}$$

First of all, this set is not well-formed in our type theory (this set would be a well-formed type, only when $x \in B[x]$ is a type for all $x \in A$, but the membership is a well-formed type in the our type theory, only when it is true). Second, this set type does not have the expected equivalence relation. Two elements are equal in this set type when they are equal just in $A$, but to be equal in the intersection they must be equal in both types $A$ and $B$ (see Example 2.4).

## 5.2 Variant Type

In the same way that the union type is dual to the intersection type, there exists a type dual to the records type — the variant type. The variant type is an expression of the form $(\mathtt{x}_1 \;\mathtt{of}\; A_1 | \mathtt{x}_2 \;\mathtt{of}\; A_2 | \ldots | \mathtt{x}_n \;\mathtt{of}\; A_n)$, where $\mathtt{x}_i$ are labels and $A_i$ are types. The elements of this type are expressions of the form $\mathtt{x}_i(a)$ where $a \in A_i$.

**Example 5.3** *We can define the type of binary trees*

$$BinTree(A) \triangleq \mu T.(\mathtt{node} \;\mathtt{of}\; T \times T \times A \mid \mathtt{emptytree} \;\mathtt{of}\; Unit\}$$

*Here $\mu$-operator is an inductive recursive type constructor, i.e. the least fixpoint [31], and $Unit$ is a type that contains only one element $\bullet$.*

We will abbreviate $\mathtt{x}_i(\bullet)$ as $\mathtt{x}_i$ and $\mathtt{x}_i(\langle a_1, a_2, \ldots, a_n\rangle)$ as $\mathtt{x}_i(a_1, a_2, \ldots, a_n)$. For example, the type $BinTree(A)$ includes $\mathtt{emptytree}, \mathtt{tree}(\mathtt{emptytree}, \mathtt{emptytree}, a_0), \mathtt{tree}(\mathtt{tree}(\mathtt{emptytree}, \mathtt{emptytree}, a_1), \mathtt{tree}(\mathtt{emp}$
where $a_i$'s are of type $A$.

### 5.2.1 Definitions

We can define the variant type as a dependent product, e.g. $(\mathtt{x} \;\mathtt{of}\; A \mid \mathtt{y} \;\mathtt{of}\; B) \triangleq$

$$l : Label \times (\mathtt{if}\; l = \mathtt{x} \;\mathtt{then}\; A \;\mathtt{else}\; \mathtt{if}\; l = \mathtt{y} \;\mathtt{then}\; B \;\mathtt{else}\; \mathrm{Void})$$

Or we can first define $(\mathtt{x} \;\mathtt{of}\; A) \triangleq \{\mathtt{x}\} \times A$, and then define

$$(\mathtt{x} \;\mathtt{of}\; A \mid \mathtt{y} \;\mathtt{of}\; B) \triangleq (\mathtt{x} \;\mathtt{of}\; A) \cup (\mathtt{y} \;\mathtt{of}\; B)$$

In any case the constructor for this type is defined as a pair:

$$\mathtt{x}(a) \triangleq \langle \mathtt{x}, a\rangle$$

We also need to define a destructor:

$$\begin{array}{l}\texttt{match}\ t\ \texttt{with}\\ \quad \mathtt{x}_1(a_1) \texttt{=>} f_1[a_1]\ |\\ \quad \mathtt{x}_2(a_2) \texttt{=>} f_2[a_3]\ |\\ \quad \ldots\\ \quad \mathtt{x}_n(a_n) \texttt{=>} f_n[a_n]\ |\end{array}$$

as

$$\begin{array}{l}\texttt{let}\ \langle l, a\rangle = t\ \texttt{in}\\ \quad \texttt{if}\ l = \mathtt{x}_1\ \texttt{then}\ f_1[a]\\ \quad \texttt{if}\ l = \mathtt{x}_2\ \texttt{then}\ f_2[a]\\ \quad \ldots\\ \quad \texttt{if}\ l = \mathtt{x}_n\ \texttt{then}\ f_n[a]\end{array}$$

### 5.2.2 Properties

The variant type has a subtyping property which is dual to the subtyping property of record types:

$$(\mathtt{x}_i\ \texttt{of}\ A_i)|_{i\in I} \subseteq (\mathtt{x}_i\ \texttt{of}\ A'_i)|_{i\in J}$$

when $I \subseteq J$ and $A_i \subseteq A'_i$ for any $i \in I$.

**Example 5.4** *Let*

$$\begin{array}{ll} Week \triangleq ( & \texttt{Sunday of}\ Unit \,|\, \texttt{Monday of}\ Unit \,|\, \texttt{Tuesday of}\ Unit \,|\\ & \texttt{Wednesday of}\ Unit \,|\, \texttt{Thursday of}\ Unit \,|\\ & \texttt{Friday of}\ Unit \,|\, \texttt{Saturday of}\ Unit)\end{array}$$

*Then* $Weekend \triangleq (\texttt{Sunday of}\ Unit \,|\, \texttt{Saturday of}\ Unit \,|\,)$ *is a subtype of* $Week$.

There is a general formula about variant types and union that is dual to the formula about records and intersection:

$$\begin{array}{l}(\mathtt{x}_1\ \texttt{of}\ A_1 \,|\, \ldots \,|\, \mathtt{x}_k\ \texttt{of}\ A_k \,|\, \mathtt{y}_1\ \texttt{of}\ B_1 \,|\, \ldots \,|\, \mathtt{y}_n\ \texttt{of}\ B_n) \cup\\ (\mathtt{x}_1\ \texttt{of}\ A'_1 \,|\, \ldots \,|\, \mathtt{x}_k\ \texttt{of}\ A'_k \,|\, \mathtt{z}_1\ \texttt{of}\ C_1 \,|\, \ldots \,|\, \mathtt{z}_m\ \texttt{of}\ C_m) =\\ (\mathtt{x}_1\ \texttt{of}\ A_1 \cup A'_1 \,|\, \ldots \,|\, \mathtt{x}_k\ \texttt{of}\ A_k \cup A'_k \,|\\ \mathtt{y}_1\ \texttt{of}\ B_1 \,|\, \ldots \,|\, \mathtt{y}_n\ \texttt{of}\ B_n \,|\, \mathtt{z}_1\ \texttt{of}\ C_1 \,|\, \ldots \,|\, \mathtt{z}_m\ \texttt{of}\ C_m)\end{array}$$

So, the intersection of two record types is alway a record type, and the union of two variant types is always a variant type.

## 5.3 Abstract Algebra

In this section we outline a way how one can define general abstract algebraic structures using our record type.

Our encoding of records uses the type $Label$ for names of the fields. In all of the above examples names were constants. But we are allowed to use variables over type $Label$. In fact, we may even use arbitrary terms of the type $Label$ as the name of the fields. It could be useful to define an algebraic structure of an arbitrary signature.

A signature is a list of operations with their arity:

$$Signature \triangleq (Label \times \mathbb{N})\ \mathrm{List}$$

We can define an algebraic structure of any signature:

$$Algebra(op_1, n_1 :: \ldots :: op_k, n_k) \triangleq \{\texttt{car}{:}\mathbb{U}; op_1{:}\texttt{car}^{n_1}{\rightarrow}\texttt{car}; \ldots; op_k{:}\texttt{car}^{n_k}{\rightarrow}\texttt{car}\}$$

Now we can define standard notions from abstract algebra. For example, homomorphism between two algebraic structures $A$ and $B$ of the same signature $Sig$ is defined as:

$$\begin{array}{ll} Hom & (A, B, Sig) \triangleq \\ \quad\{ & f : A\mathtt{.car} \to B\mathtt{.car} \mid \\ & \forall \langle op, n\rangle \in Sig.\ \forall x \in A\mathtt{.car}^n.\ f(A.op(x)) = B.op(f_n(x)) \in B\mathtt{.car}\ \} \end{array}$$

where $f_n(\langle x_1, \ldots, x_n\rangle) = \langle f(x_1), \ldots, f(x_n)\rangle$.

We can prove some general properties about homomorphisms, like composition of two homomorphisms is a homomorphism. Then we can apply this theorem to concrete algebraic structures.

## 5.4 Join Operator

In this section we outline possible applications of records and intersections to databases. One of the basic operation for relation databases is a join operator.

We can represent a relation with attributes $\mathtt{A}_1$, $\mathtt{A}_2$, ..., $\mathtt{A}_n$ as a finite subset of a type $\{\mathtt{A}_1 : T_1; \mathtt{A}_2 : T_2; \ldots \mathtt{A}_n : T_n\}$, where $T_i$ is a type of an attribute $\mathtt{A}_i$. That means that a relation $R$ is represented by a set of records that has fields $\mathtt{A}_1$, ..., $\mathtt{A}_n$ that coincide with one of the tuples in $R$, and probably other fields. Then one can easily see that the intersection of two relations $R_1$ and $R_2$ is exactly the natural join of these relations! That is, we can very easily define the natural join for the relations:

$$R_1 \bowtie R_2 = R_1 \cap R_2 \ \ !$$

# Chapter 6
# Red–Black Trees

In this section we will show an example of how one can define an abstract data structure in the constructive type theory, and formally prove the correctness of the concrete implementation. We will consider red–black trees [16], one of the most popular implementation of a data structure of collections of elements of a certain type.

## 6.1 Introduction

In the end of Section 3.3.3 we gave a definition of the data structure $Collection(T)$, a collection of elements of the type $T$. Here we repeat the definition using set type (using notations of Section 5.1):

$$\begin{array}{l} Collection(T) \triangleq \\ \{\texttt{car} : \mathbb{U}; \\ \ \texttt{empty} : \texttt{car}; \\ \ \texttt{member} : \texttt{car} \to T \to \mathbb{B}; \\ \ \texttt{insert} : \texttt{car} \to T \to \texttt{car} \mid \\ \forall a : T \quad a \notin \texttt{empty} \mid \\ \forall s : \texttt{car} \quad \forall a, b : T \quad (\texttt{member}\ (\texttt{insert}\ s\ a)\ b) \\ \qquad \iff (\texttt{member}\ s\ b) \vee (a = b \in T)\} \end{array}$$

We can implement this data structure in several ways. The simplest but inefficient implementation of sets uses lists. Each set is represented by an unordered list. Formally we take `car` to be $T$ List, `empty` to be $nil$ and define operations `insert` and `member` correspondingly. In this implementation, functions `insert` and `member` take $O(n)$ time, where $n$ is a number of elements of the set.

A more efficient implementation of sets is binary search trees. Each set is represented by a binary tree, where elements are stored at the nodes, such that the element at any given node is greater than each element in its left subtree and less than each element in its right subtree. In this implementation, functions `insert` and `member` take $O(d)$ time, where $d$ is a depth of the tree. On random data the heights of the tree is $log(n)$. But in the worst case the tree will be imbalanced, and an individual operation will take up to $O(n)$ time.

The solution to this problem is to use *balanced* binary trees. The most popular balanced binary search trees are red–black trees [16]. We will show how the implementation of red–black trees could be written as a term in type theory.

Red–black trees could be defined only on an ordered set. We have defined ordered structures in Example 3.2. Thus the implementation of red–black trees should be a *functor* (i.e. a function from one data structure to another) that takes an ordered set and returns a data structure of collections of elements of this set. That is, it has the following type:

$$ord : OrdSet \to Collection(ord\mathbf{.}\texttt{car}).$$

The implementation of red-black trees in a functional programming setting is a little bit different (and simpler) than the typical presentation in imperative programming languages (as for example in [12]). We will follow the presentation of red–black trees in functional languages from [33].

## 6.2 Binary Trees

### Definition

We already gave the definition of binary trees in Example 5.3:

$$BinTree(A) \triangleq \mu T.(\texttt{node of } T \times T \times A \mid \texttt{emptytree of } Unit\}$$

We have the following introduction rules about this type:

$$\frac{A \text{ Type}}{\texttt{emptytree} \in BinTree(A)} \qquad \frac{a \in A \qquad l \in BinTree(A) \qquad r \in BinTree(A)}{\texttt{tree}(l, r, a) \in BinTree(A)}$$

The elimination rule is the induction rule:

$$\frac{\Gamma \vdash C[\texttt{emptytree}] \qquad \Gamma; l : BinTree(A); r : BinTree(A) \vdash C[\texttt{tree}(l, r, a)]}{\Gamma; t : BinTree(A) \vdash C[t]}$$

**Operations with trees**

We can define depth and weight (i.e. number of elements) of the tree by induction:

- $weight(\texttt{emptytree}) \triangleq 0$
- $weight(\texttt{tree}(l, r, a)) \triangleq weight(l) + weight(r) + 1$

- $depth(\texttt{emptytree}) \triangleq 1$
- $depth(\texttt{tree}(l, r, a)) \triangleq max(weight(l); weight(r)) + 1$

We can define quantifiers on the nodes of the tree. Let $P[l; r; a]$ be a proposition of nodes $\texttt{tree}(l, r, a)$. Then we define by induction $\forall\texttt{node}(l, r, a) \in t \,.\, P[l; r; a]$ as a proposition that says that $P$ is true for all nodes of the tree $t$, and $\exists\texttt{node}(l, r, a) \in t \,.\, P[l; r; a]$ as a proposition that says that $P$ is true for at least one node of the tree $t$ ($l$, $r$ and $a$ are bound variables). That is,

- $\forall\texttt{node}(l, r, a) \in \texttt{emptytree} \,.\, P[l; r; a] \triangleq True$
- $\forall\texttt{node}(l, r, a) \in tree(l_1, r_1, a_1) \,.\, P[l; r; a] \triangleq$
  $P[l_1; r_1; a_1] \wedge$
  $\forall\texttt{node}(l, r, a) \in l_1 \,.\, P[l; r; a] \wedge$
  $\forall\texttt{node}(l, r, a) \in r_1 \,.\, P[l; r; a]$

- $\exists\texttt{node}(l, r, a) \in \texttt{emptytree} \,.\, P[l; r; a] \triangleq False$
- $\exists\texttt{node}(l, r, a) \in tree(l_1, r_1, a_1) \,.\, P[l; r; a] \triangleq$
  $P[l_1; r_1; a_1] \vee$
  $\exists\texttt{node}(l, r, a) \in l_1 \,.\, P[l; r; a] \vee$
  $\exists\texttt{node}(l, r, a) \in r_1 \,.\, P[l; r; a]$

We will store elements in the nodes of a tree. We define the proposition $in\_tree(a; t; A)$ that states that node $a$ is stored in the tree $t$:

$$in\_tree(a; t; A) \triangleq \exists\texttt{node}(l, r, a') \in t \,.\, a = a' \in A$$

This proposition needs the type $A$ as a parameter because we have different equalities in different types.

Finally, we can define a set of elements stored in a given tree:

$$|t|_A \triangleq \{a : A \mid in\_tree(a; t; A)\}$$

## 6.3 Sorted Trees

Assume we have an ordered set $ord$. Sorted trees are binary trees satisfying the following property: for any node $tree(l, r, a)$ in the tree any element from the left subtree $l$ is less than the root $a$ and any element from the right subtree $r$ is greater than the root $a$. Formally,

$$\begin{array}{l} SortedTree(ord) \triangleq \\ \quad \{t : BinTree(ord.\mathtt{car}) \mid \\ \quad\quad \forall \mathtt{node}(l, r, a) \in t. \\ \quad\quad\quad \forall x : |l|_{ord.\mathtt{car}} \, . \, x <_{ord} a \wedge \\ \quad\quad\quad \forall y : |r|_{ord.\mathtt{car}} \, . \, a <_{ord} y \\ \quad \} \end{array}$$

**Searching in balance trees**

We can find whether an element is in tree by binary search:

```
- $search(a; \mathtt{emptytree}; ord) \triangleq false_{\mathbb{B}}$

- $search(a; \mathtt{tree}(l, r, data); ord) \triangleq$
  if $a <_{ord} data$ then $search(a; l; ord)$
  if $a =_{ord} data$ then $true_{\mathbb{B}}$
  if $a >_{ord} data$ then $search(a; r; ord)$
```

Note that this function returns a boolean value, unlike $is\_in\_tree$, which is a proposition.
Using the transitivity of order we can prove

**Theorem 6.1 (Correctness of Search)** *For any ordered set $ord \in OrdSet$, for any element $a \in ord.\mathtt{car}$ and for any tree $t \in SortedTree(ord)$*

$$search(a; t; ord) \in \mathbb{B}$$

*and*

$$search(a; t; ord) = true_{\mathbb{B}} \iff a \in |t|_{ord.\mathtt{car}}$$

**Insert function**

To insert a new element into the tree we again use binary search to find an appropriate place:

```
- $ins(a; \mathtt{emptytree}; ord) \triangleq \mathtt{tree}(\mathtt{emptytree}, \mathtt{emptytree}, a)$

- $insert(a; \mathtt{tree}(l, r, data); ord) \triangleq$
  if $a <_{ord} data$ then $\mathtt{tree}(insert(a; l; ord), r, data)$
  if $a =_{ord} data$ then $\mathtt{tree}(l, r, a)$
  if $a >_{ord} data$ then $\mathtt{tree}(l, insert(a; r; ord), data)$
```

We can prove the following

**Theorem 6.2 (Invarian of Insert)** *For any ordered set $ord \in OrdSet$ and for any element $a \in ord.\mathtt{car}$ if $t \in SortedTree(ord)$ then $insert(a; t; ord)$ is also in $SortedTree(ord)$.*

**Theorem 6.3 (Correctness of Insert)** *For any ordered set $ord \in OrdSet$, for any element $a \in ord.\mathtt{car}$, for any tree $t \in SortedTree(ord)$*

$$|insert(a; t; ord)|_{ord.\mathtt{car}} =_e |t|_{ord.\mathtt{car}} \cup \{a\}_{ord.\mathtt{car}}.$$

## 6.4 Red–Black Trees

### Definition

In a red–black tree each node is colored either red or black. A red–black tree should satisfy the following invariants:

- Any child of a red color is black
- All paths from the root to any leaf have the same number of black nodes. (We will call this number a *black depth* of a tree).

We will consider trees that satisfy an additional property:

- The root of a tree is black

We start the formal definition with the definition of colors:

$$Color \triangleq (\texttt{red of } unit \mid \texttt{black of } unit)$$

That is, $Color$ has two elements: `red` and `black`. We also define two subtypes of this type:
$Red \triangleq (\texttt{red of } unit)$ has only one element `red`
$Black \triangleq (\texttt{black of } unit)$ has only one element `black`
Then we define $ColoredTree(A)$ as a type of trees with colored nodes:

$$ColoredTree(A) \triangleq BinTree(Color \times A)$$

Then we define three subtypes of $ColoredTree(A)$: $RB_n(A)$ for red–black trees of the black depth $n$, $B_n(A)$ for red–black trees of the black depth $n$ that have a black root, and $R_n(A)$ for red–black trees of the black depth $n$ that have a red root. (For the sake of this definition we assume that empty tree has a black root.) We define these types simultaneously by induction:

- $B_0(A) \triangleq (\texttt{emptytree of } Unit)$ (only the empty tree has black depth 0);

and for any natural $n$

- $B_{n+1}(A) \triangleq (\texttt{tree of } (RB_n(A) \times RB_n(A)) \times (Black \times A))$ (a black tree of the black depth $n+1$ has a black root and two sons of the black depth $n$);
- $R_n(A) \triangleq (\texttt{tree of } (B_n(A) \times B_n(A)) \times (Red \times A))$ (a red tree has a red root and black sons of the same black depth);
- $RB_n(A) \triangleq R_n(A) \cup B_n(A)$ (a red–black tree is either red or black).

We can prove by induction that these definitions are well-formed for any natural $n$:

$$\forall n : \mathbb{N}.\ B_n(A)\ \text{Type} \wedge R_n(A)\ \text{Type} \wedge RB_n(A)\ \text{Type}$$

Finally we define a type of red–black trees as a union of all $B_n(A)$:

$$RedBlackTree(A) = \bigcup_{n:\mathbb{N}} B_n(A)$$

**Insert Function**

The insert function for red–black trees is similar to the insert function for sorted trees, but it maintains the invariants.

When we insert a new node we will color it red. It satisfies the second invariant, but may break the first invariant if the father of the new node is red.

Let us define an auxiliary function:

- $ins(a; \texttt{emptytree}; ord) \triangleq \texttt{tree}(\texttt{emptytree}, \texttt{emptytree}, \texttt{red}, a)$

- $ins(a; \texttt{tree}(l, r, color, data); ord) \triangleq$
  `if` $a <_{ord} data$ `then` $lbalance(ins(a; l; ord); r; color; data)$
  `if` $a =_{ord} data$ `then` $\texttt{tree}(l; r; color, a)$
  `if` $a >_{ord} data$ `then` $rbalance(l; ins(a; r; ord); color; data)$

Where $lbalance$ and $rbalance$ are functions that rebalance a tree without changing the order to enforce invariants. They are defined as follows:

- $lbalance(\texttt{tree}(\texttt{tree}(t_1, t_2, \texttt{red}, a_1), t_3, \texttt{red}, a_2); t_4; color; a_3) \triangleq$
  $\texttt{tree}(\texttt{tree}(t_1, t_2, \texttt{black}, a_1), \texttt{tree}(t_3, t_4, \texttt{black}, a_3), \texttt{red}, a_2)$

- $lbalance(\texttt{tree}(t_1, \texttt{tree}(t_2, t_3, \texttt{red}, a_2), \texttt{red}, a_1); t_4; color; a_3) \triangleq$
  $\texttt{tree}(\texttt{tree}(t_1, t_2, \texttt{black}, a_1), \texttt{tree}(t_3, t_4, \texttt{black}, a_3), \texttt{red}, a_2)$

- For all other cases
  $lbalance(l; r; color; a) \triangleq tree(l; r; color, a)$

- $rbalance(t_1, \texttt{tree}(\texttt{tree}(t_2, t_3, \texttt{red}, a_2), t_4, \texttt{red}, a_3); color; a_1) \triangleq$
  $\texttt{tree}(\texttt{tree}(t_1, t_2, \texttt{black}, a_1), \texttt{tree}(t_3, t_4, \texttt{black}, a_3), \texttt{red}, a_2)$

- $rbalance(t_1, \texttt{tree}(t_2, \texttt{tree}(t_3, t_4, \texttt{red}, a_3), \texttt{red}, a_2); color; a_1) \triangleq$
  $\texttt{tree}(\texttt{tree}(t_1, t_2, \texttt{black}, a_1), \texttt{tree}(t_3, t_4, \texttt{black}, a_3), \texttt{red}, a_2)$

- For all other cases
  $rbalance(l; r; color; a) \triangleq tree(l; r; color, a)$

Function $ins$ may break the first invariant. Namely it may return a tree with *only one* singularity *at the root*: a red root may have a red son. The functions $lbalance$ and $rbalance$ then take care of this singularity.

Formally let us define a type of trees with at most on one singularity at the root:

- $lRRB_n(A) \triangleq (\texttt{tree of } R_n(A) \times B_n(A) \times Red \times A)$ (trees with a red root and a red left child);

- $rRRB_n(A) \triangleq (\texttt{tree of } B_n(A) \times R_n(A) \times Red \times A)$ (trees with a red root and a red right child);

- $RRB_n(A) \triangleq RB_n(A) \cup lRRB_n(A) \cup rRRB_n(A)$ (trees with at most one singularity at the root).

We will see that the $ins$ function may return trees of the type $RRB_n(A)$. Functions $lbalance$ and $rbalance$ deal with such trees.

**Lemma 6.4** *For any natural $n$ and for any type $A$ the following is true:*

$$l : RRB_n(A); r : RB_n(A) \vdash lbalance(l; r; \texttt{black}; a) \in RB_{n+1}(A)$$
$$l : RB_n(A); r : B_n(A) \vdash lbalance(l; r; \texttt{red}; a) \in RRB_n(A)$$
$$l : RB_n(A); r : RRB_n(A) \vdash rbalance(l; r; \texttt{black}; a) \in RB_{n+1}(A)$$
$$l : B_n(A); r : RB_n(A) \vdash rbalance(l; r; \texttt{red}; a) \in RB_n(A)$$

This lemma could be proved by analyzing all possible cases.

**Lemma 6.5** *For any ordered set $ord \in OrdSet$ and for any $a \in ord.\mathtt{car}$*

$$t : R_n(ord.\mathtt{car}) \vdash ins(a;t;ord) \in RRB_n(ord.\mathtt{car})$$

$$t : B_n(ord.\mathtt{car}) \vdash ins(a;t;ord) \in RB_n(ord.\mathtt{car})$$

This lemma could be easily proved by simultaneous induction using the previous lemma.
Finally, we need to correct the singularity in the root. It may be done by just painting the root black:

$$blackroot(tree(l,r,color,a)) \triangleq tree(l,r,\mathtt{black},a)$$

So,

$$rb\_insert(a;t;ord) \triangleq blackroot(ins(a;t;ord))$$

It is easy to prove the following

**Lemma 6.6**
$$t : R_n(A) \vdash blackroot(t) \in RedBlackTree(A).$$

Therefore we have the following

**Theorem 6.7 (Invariant of the insert function)** *For any ordered set $ord \in OrdSet$ for any $a \in ord.\mathtt{car}$ if $t$ is in $RedBlackTree(A)$ then $rb\_insert(a;t;ord)$ is also in $RedBlackTree(A)$.*

**Red–black trees are balanced**

**Lemma 6.8** *The depth of a red–black tree is not more than 2 times its black depth. Formally,*

$$\forall n : \mathbb{N}.\forall t : RB_n(A).depth(t) \leq 2n$$

**Lemma 6.9** *A red–black tree of the black depth $n$ contains at least $2^n - 1$ elements. Formally,*

$$\forall n : \mathbb{N}.\forall t : RB_n(A).weight(t) \geq 2^{n-1}$$

These lemmas are easily proved by induction on $n$. (We need to prove them also for $R_n$ and $B_n$.)

It follows from these lemmas that the depth of any red–black tree is less than or equal to $2\log(n)$, where $n$ is the number of nodes. Therefore searching and inserting in this tree takes $O(\log n)$ time. The last argument is informal. In the current system there is no way to formally prove an upper bound for the working time of an algorithm.

## 6.5 Sorted Red–Black Trees

Now we define the type of sorted red–black trees just as an intersection of the types of sorted trees and red–black trees:

$$SortedRedBlackTree(ord) \triangleq RedBlackTree(ord.\mathtt{car}) \cap SortedTree(\text{Top} * ord)$$

where $Top * ord$ is an ordered set of all pairs $\langle color, a\rangle$ for $a \in ord.\mathtt{car}$ and the order relation ignoring the first component. That is,

$$\text{Top} * ord \triangleq \{\mathtt{car} = \text{Top} \times ord.\mathtt{car};\ \mathtt{less}\ \langle c_1, a_1\rangle\ \langle c_2, a_2\rangle = ord.\mathtt{less}\ a_1\ a_2\}$$

Since $SortedRedBlackTree(ord)$ is a subtype of $SortedTree(\text{Top} * ord)$ we can use the same function for searching:

$$rb\_search(a;t;ord) \triangleq search(a;t;\text{Top} * ord)$$

**Theorem 6.10 (Correctness of Search)** *For any ordered set $ord \in OrdSet$ for any element $a \in ord\mathtt{.car}$ for any tree $t \in SortedRedBlackTree(ord)$*

$$rb\_search(a;t;ord) \in \mathbb{B}$$

*and*

$$rb\_search(a;t;ord) = true_{\mathbb{B}} \iff a \in |t|_{ord\mathtt{.car}}$$

It immediately follows from Theorem 6.1 and the fact that Top $* \, ord \in OrdSet$.

We can prove that $lbalance$ and $rbalance$ do not change the order of elements in Top$*ord$. Therefore we can prove that

**Lemma 6.11** *For any ordered set $ord \in OrdSet$ for any element $a \in ord\mathtt{.car}$ if $t \in SortedTree(\text{Top} * ord)$ then $rb\_insert(a;t;ord)$ is also in $SortedTree(\text{Top} * ord)$ and*

$$|rb\_insert(a;t;ord)|_{\text{Top}\times ord\mathtt{.car}} =_e |t|_{\text{Top}\times ord\mathtt{.car}} \cup \{\bullet, a\}_{\text{Top}\times ord\mathtt{.car}}.$$

Finally, using the fact that if $f \in A_1 \to A_2$ and $f \in B_1 \to B_2$ then $f \in A_1 \cap A_2 \to B_1 \cap B_2$, we get

**Theorem 6.12 (Correctness of Insert)** *For any ordered set $ord \in OrdSet$ for any element $a \in ord\mathtt{.car}$ if $t \in SortedRedBlackTree(ord)$ then $rb\_insert(a;t;ord)$ is also in $SortedRedBlackTree(ord)$ and for any $b \in ord.\mathtt{car}$*

$$rb\_search(b; rb\_insert(a;t;ord); ord) \iff rb\_search(t) \vee a = b \in ord\mathtt{.car}.$$

## Collection

Finally we combine the above functions into the functor of the type $ord : Ord \to Collection(ord\mathtt{.car})$.

$$\begin{array}{l} redblacktree\_collection(ord) \triangleq \\ \{\mathtt{car} = SortedRedBlackTree(ord); \\ \ \mathtt{empty} = \mathtt{emptytree}; \\ \ \mathtt{member}\ t\ a = rb\_search(a;t;ord); \\ \ \mathtt{insert}\ t\ a = rb\_insert(a;t;ord) \\ \} \end{array}$$

**Theorem 6.13 (Main)** *For any ordered set $ord \in OrdSet$ the structure $redblacktree\_collection(ord)$ is a correct structure for collections of elements of the carrier of the ordered set $ord$. Formally,*

$$redblacktree\_collection(ord) \in Collection(ord\mathtt{.car}).$$

Note that this theorem not only tells us that our functions have the right type, but also tells that this function satisfies the specifications stated in the definition of collections.

# Chapter 7
# Objects

Note that the elements of the type $Collection(T)$ defined in the last chapter are not collections, but rather implementations of collections, i.e., a bunches of functions. The actual collections are elements of type $C\mathbf{.}\mathtt{car}$ where $C \in Collection(T)$. If we have a function that need a collection as a parameter, it actually should have two arguments: an implementation and a collection itself. So, it should have a type like:

$$C : Collection(T) \to C\mathbf{.}\mathtt{car} \to A \tag{7.1}$$

Another disadvantage of this data structure is that it is not fully abstract. Functions of the type (7.1) may have access to field `car`, which is supposed to be abstract.

In this chapter we will define a notion of objects that removes these disadvantages. Note that the theory of objects is not yet implemented MetaPRL.

## 7.1 Object instances

In this section we define object instances and basic operations with them. First we describe the intended behavior of these operations and then we give a formal definition. The problem of the typing of these object instances will be considered in the successive sections.

### 7.1.1 The operations with objects

**Methods**

The main difference between objects and records is that objects have methods. Methods can be understood as functions that have a parameter $self$, that represents the object itself. That is, when we evaluate a method of a particular object we substitute this object for the $self$ parameter.

The main operation that we perform with methods is to apply them to an object. We will use circle dot ($obj_\circ l$) for a method extraction (to distinguish it from field selection for records $rec\mathbf{.}l$). Here $obj$ is an object and $l$ is a name of a method. Thus, if $obj$ is an object instance that has a method named $l$ with a body $m(self)$ then $obj_\circ l$ expands to $m(obj)$. (Here $self$ is a variable, and $m(obj)$ stands for the substitution $obj$ for the variable $self$.)

Fields of objects can be represented as methods that do not depend on $self$.

So, object instances are lists of methods (including fields). We will use the following syntax for objects:

$$\mathfrak{o}\, self.\{l_1 = m_1(self); \ldots ; l_n = m_n(self)\}$$

where $self$ is a bound variable, $l_i$'s are names of the methods (fields) and $m_i$'s are bodies of the corresponding methods (values of the fields).

**Example 7.1** *The following is an example of an object $simpleFlea$. The flea lives on an integer line and has a coordinate* `x`*, that can be obtained, by a method* `getX`*. Method* `getNextX` *returns a coordinate where the flea wants to jump next time.*

$$\begin{array}{l}
simpleFlea \triangleq \mathfrak{o}\, self. \\
\quad \{\mathtt{x} = 0; \\
\quad\ \mathtt{getX} = self_\circ\mathtt{x}; \\
\quad\ \mathtt{getNextX} = self_\circ\mathtt{getX} + 1 \\
\quad \}
\end{array}$$

For the object $simpleFlea$ we expect the following reductions:

$$simpleFlea_\circ\texttt{getX} \to simpleFlea_\circ\texttt{x} \to 0$$
$$simpleFlea_\circ\texttt{getNextX} \to flea_\circ\texttt{getX} + 1 \to 0 + 1 \to 1$$

In general, for object

$$object \ = \ \varsigma\, self.\{l_1 = m_1(self); \ldots; l_n = m_n(self)\} \tag{7.2}$$

with distinct $l_i$'s we have the following reduction rule:

$$object_\circ l_i \to m_i(object) \tag{7.3}$$

**Field update**

Another basic operation that we need for objects is a field/method update.

We will use the following syntax for this operation: $obj_\circ l := t$, where $obj$ is an object instance, $l$ is a name of a field and $t$ is a new value. Note that we are working in a pure functional language. Field update does not modify an existing object, but rather creates a new objects. For example, $simpleFlea_\circ\texttt{x} := 17$ is a new object that coincides to $simpleFlea$ in all fields except $\texttt{x}$. Field update should obey the following reduction rule:

$$(obj_\circ l := t)_\circ l \to t \tag{7.4}$$

For example, $(flea_\circ\texttt{x} := 17)_\circ\texttt{x} \to 17$. This rule is the same as an analogous rule for records (3.2). On the other hand, the analog of the record reduction rule form Table 3.2

$$(obj_\circ l := t)_\circ l' \to obj_\circ l', \qquad \text{when } l \neq l' \qquad\qquad (wrong!)$$

is wrong for objects. For example, $(simpleFlea_\circ\texttt{x} := 17)_\circ\texttt{getX}$ reduces to 17, not to $simpleFlea_\circ\texttt{getX}$ which is $0$.

The right reduction rule is the following: for $object$ defined in (7.2) let $object'$ be $object_\circ l := t$, then

$$object'_\circ l_i \to m_i(object') \tag{7.5}$$

where $i \in 1..n$ and $l \neq l_i$.

For example,

$$(flea_\circ\texttt{x} := 17)_\circ\texttt{getX} \to (flea_\circ\texttt{x} := 17)_\circ\texttt{x} \to 17$$

**Example 7.2** *Now we can define a method* `move` *that moves a flea by* 1 *step to the right.*

$$\begin{array}{l} movableFlea \triangleq \varsigma\, self. \\ \quad \{\texttt{x} = 0; \\ \quad\ \texttt{getX} = self_\circ\texttt{x}; \\ \quad\ \texttt{getNextX} = self_\circ\texttt{getX} + 1; \\ \quad\ \texttt{move} = (self_\circ\texttt{x} := self_\circ\texttt{getNextX}) \\ \quad \} \end{array}$$

*In this example, $movableFlea_\circ\texttt{move}_\circ\texttt{move}_\circ\texttt{getX}$ evaluates to 2.*

**Method update**

The generalization of the field update is a method update:

$$obj_\circ l := \varsigma\, self.m(self)$$

Here $l$ is a name of a method, $m$ is a new body of this method with a bound variable $self$.

The reduction rules for the method update are analogous to ones for field update. For $object$ defined in (7.2) let $object'$ be $object_\circ l := \varsigma\, self.m(self)$, then

$$object'_\circ l \to m(object') \tag{7.6}$$

and

$$object'{}_{\circ}l_i \to m_i(object') \tag{7.7}$$

where $i \in 1..n$ and $l \neq l_i$.

**Example 7.3** *We can override method* `getNextX` *in the last example:*

$$fastFlea \triangleq movableFlea_{\circ}\texttt{getNextX} := \varsigma\, self.self_{\circ}\texttt{getX} + 2.$$

*Now $fastFlea$ moves twice faster than $movableFlea$. For example,*

$$fastFlea_{\circ}\texttt{move}_{\circ}\texttt{move}_{\circ}\texttt{getX} \longrightarrow 0 + 2 + 2 = 4.$$

The operation method update could be used for extending an object with new methods. That is, we can apply the operation of updating a method to an object that did not contain this method before.

We will use the following alternative syntax for method update. We will write

$$\begin{array}{l} \mathfrak{o}(obj)\ self. \\ \quad \{l_1 = m_1(self); \\ \quad \ldots \\ \quad\ l_n = m_n(self)\} \end{array}$$

instead of

$$\begin{array}{ll} obj & {}_{\circ}l_1 := \varsigma\, self.m_1(self) \\ & \ldots \\ & {}_{\circ}l_n := \varsigma\, self.m_n(self) \end{array}$$

For example we could defined $movableFlea$ from Example 7.2 as an extension of $simpleFlea$:

$$movableFlea\ =\ \mathfrak{o}(simpleFlea)\ self.\{\texttt{move} = (self_{\circ}\texttt{x} := self_{\circ}\texttt{getNextX})\}.$$

Note that field update can be considered as a partial case of method update when $m$ does not depend on $self$.

These operations and the reduction rules are summarized in Table 7.1.

### 7.1.2 Formal definitions

It is relatively easy to define objects and their operations (method application and method update) in lambda-calculus with records. We will define objects as functions that take $self$ as a parameter and return a record:

$$\begin{array}{ll} \mathfrak{o}\, self. & \{\ \ l_1 = m_1(self); \ldots; l_n = m_n(self)\} \triangleq \\ \lambda self. & \{\ \ l_1 = m_1(self); \ldots; l_n = m_n(self)\} \end{array} \tag{7.8}$$

As one would expect, method application is a self application :

$$obj_{\circ}l \triangleq (obj\ obj)\textbf{.}l, \tag{7.9}$$

i.e., we apply an object to itself and then get a record, and extract a field $l$ from this record.

Field update is defined as

$$obj_{\circ}l := t \triangleq \lambda self.\,(obj\ self)\textbf{.}l := t\,. \tag{7.10}$$

By analogy, method update is defined as

$$obj_{\circ}l := \varsigma\, self.m(self) \triangleq \lambda self.((obj\ self)\textbf{.}l := m(self)). \tag{7.11}$$

**Theorem 7.4** *The definitions* (7.8)–(7.11) *satisfy the intended reduction rules from Table 7.1.*

Table 7.1: Reduction rules for object calculus

**Canonical terms:**
$\varsigma\, self.\{l_1 = m_1(self); \dots ; l_n = m_n(self)\}$

**Operations:**
Method application: $obj_\circ l$
Method update/extension: $obj_\circ l := \varsigma\, self.m(self)$
Field update/extension is a partial case of method update:
$obj_\circ l := f \triangleq obj_\circ l := \varsigma\, self.m$

**Reductions:**
If $obj = \varsigma\, self.\{l_1 = m_1(self); \dots ; l_n = m_n(self)\}$ then

$$obj_\circ l_i \to m_i(obj) \quad \text{when } l_i \neq l_{i+1}, \dots l_n$$
$$obj_\circ l := \varsigma\, self.m(self) \to \varsigma\, self.\{l_1 = m_1(self); \dots ; l_n = m_n(self); l = m(self)\}$$

*Remark* An alternative way would be to define an object as a record of methods, where each method is a function that take $self$ as a parameter:

$$\begin{array}{c}\varsigma\, self.\{l_1 = m_1(self); \dots ; l_n = m_n(self)\} \triangleq \\ \{l_1 = \lambda self.m_1(self); \dots ; l_n = \lambda self.m_n(self)\}\end{array}$$

This approach was used by Hickey in [21]. Although the latter definition may seem more natural, we choose the former one, because the typing rules will be more elegant for it.

### 7.1.3 Additional Properties

From the above definitions it is easy to see that we can define any object as an extension of an empty object $\{||\}$. For example, the $object$ defined in (7.2) is equal to

$$\begin{array}{l}\varsigma(\{||\})\ self. \\ \quad \{l_1 = m_1(self); \\ \quad \dots \\ \quad l_n = m_n(self)\}.\end{array}$$

Also if we rewrite a method, then we can forget about the old method, i.e.,

$$\varsigma\, self\{\dots ; l = m; \dots ; l = m'; \dots\} \equiv \varsigma\, self.\{\dots ; \dots ; l = m'; \dots\}$$

and

$$obj_\circ(l := m)_\circ(l := m') \equiv obj_\circ l := m'.$$

The methods with different names commute. That is,

$$\varsigma\, self.\{\dots ; l = m; l' = m'; \dots\} \equiv \varsigma\, self.\{\dots ; l' = m'; l = m; \dots\}$$

and

$$obj_\circ(l := m)_\circ(l' := m') \equiv obj_\circ(l' := m')_\circ(l := m)$$

where $l \neq l'$.

### 7.1.4 Notations

First, let us note that we use three types of dots in the thesis. The simple dot (.) is used for in expressions like $\lambda x.f$, $\mathfrak{o}\,self.\{\mathtt{x} = 0\}$ to show binding variables. The bold dot (**.**) is used for records, e.g., $r\mathbf{.}\mathtt{x}$, $r\mathbf{.}\mathtt{x} := 1$. The circle dot ($_\circ$) is used for objects, e.g., $o_\circ\mathtt{x}$, $o_\circ\mathtt{x} := 1$.

Like in many programming languages, we will usually omit $self$. That is, we will use the following notations:

| instead of writing: | we will write: |
|---|---|
| $self_\circ\mathtt{x}$ | $\mathtt{x}$ |
| $self_\circ\mathtt{x} := m$ | $\mathtt{x} := m$ |
| $\mathfrak{o}\,self.\{\dots\}$ | $\{\!\mid \dots \mid\!\}$ |
| $\mathfrak{o}(obj)\ self.\{\dots\}$ | $\mathfrak{o}(obj)\ \{\!\mid \dots \mid\!\}$ |

For instance, Example 7.2 can be rewritten as follows:

$$\begin{array}{l} movableFlea = \\ \quad \{\!\mid \mathtt{x} = 0; \\ \quad \mathtt{getX} = \mathtt{x}; \\ \quad \mathtt{getNextX} = \mathtt{getX} + 1; \\ \quad \mathtt{move} = (\mathtt{x} := \mathtt{getNextX}); \\ \quad \mid\!\} \end{array}$$

### 7.1.5 Recursion

The above definition allows us to write recursive objects.

**Example 7.5** *We can write a recursive method that moves the flea by $n$ steps.*

$$\begin{array}{l} advanceFlea \triangleq \mathfrak{o}(movableFlea). \\ \quad \{\mathtt{moveBy} = (\lambda n.\mathtt{if}\ \ n = 0\ \ \mathtt{then}\ \ self\ \ \mathtt{else}\ \ \mathtt{move}_\circ\mathtt{moveBy}\ (n-1)) \\ \quad \} \end{array}$$

*Then $advanceFlea_\circ\mathtt{moveBy}\ (17)_\circ\mathtt{getX}$ evaluates to* 17.

**Example 7.6** *We can also write objects with mutual recursion:*

$$\begin{array}{l} feeFoo \triangleq \\ \quad \{\!\mid \mathtt{foo} = \lambda n.\mathtt{if}\ \ n = 0\ \ \mathtt{then}\ \ 0\ \ \mathtt{else}\ \ \mathtt{fee}(n-1); \\ \quad \mathtt{fee} = \lambda n.\mathtt{if}\ \ n = 0\ \ \mathtt{then}\ \ 1\ \ \mathtt{else}\ \ \mathtt{foo}(n-1) \\ \quad \mid\!\} \end{array}$$

*This object has two methods $fee$ and $foo$, which recursively call each other. According to rules of Table 7.1 $feeFoo_\circ\mathtt{foo}(17)$ evaluates to* 1.

## 7.2 Typing

As we saw, object instances can be defined fairly easily in lambda-calculus with records. However, finding the right type for these objects is a difficult task. Indeed, how do we type even a simple object $simplestFlea \triangleq \{\!\mid \mathtt{x} = 1; \mathtt{getX} = \mathtt{x} \mid\!\}$? This object is a function from objects of this type to the record type $\{\mathtt{x} : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}\}$. Intuitively the type of this object $X$ should satisfy an equation $X = X \to \{\mathtt{x} : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}\}$. Unfortunately, this equation is not monotone in $X$. Therefore, we can not use standard fixpoint operations such as the least fixpoint ($\mu$) or the greatest ($\nu$). Moreover, this equation may not have a fixpoint at all!

First let us examine more carefully what we are looking for. We want to define the type of objects of the form

$$\mathfrak{o}\, \mathit{self}.\{l_1 = m_1(\mathit{self}); \ldots; l_n = m_n(\mathit{self})\}$$

where we are given the type of the methods. Let $M_i$ be a type of a method named $l_i$. Let us denote the type of such objects as

$$\mathfrak{O}\{l_1 : M_1; \ldots; l_n : M_n\}$$

For example, $simplestFlea$ should have type $SimplestFleas \triangleq \mathfrak{O}\{\mathtt{x} : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}\}$.

Note that some methods may return objects of the same type (e.g., `move` and `moveBy` methods). In this case we will use a bound variable *Self* that represent the type of the object itself. We will use the following syntax:

$$\mathfrak{O}\, \mathit{Self}.\{l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self})\} \tag{7.12}$$

For example, we expect $advanceFlea$ to be of the following type

$AdvanceFleas \triangleq$
$\mathfrak{O}\, \mathit{Self}.\{\mathtt{x} : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}; \mathtt{getNextX} : \mathbb{Z}; \mathtt{move} : \mathit{Self}; \mathtt{moveBy} : \mathbb{N} \to \mathit{Self}\}$.

We will call the record type $M[Self] = \{l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self})\}$ *a declaration type* of an object type. Our goal is define a constructor $\mathfrak{O}\, Self.M[Self]$ which is an object type of a given declaration. First, let us describe the properties that we expect from this type constructor.

What does it mean that the method of an object has a type $M$? It means that if we apply this method we get an element of type $M$. That is, if $obj$ has type $\mathfrak{O}\{l_1 : M_1; \ldots; l_n : M_n\}$ then $obj_\circ l_i$ must have type $M_i$. More generally, if

$$Object = \mathfrak{O}\, \mathit{Self}.\{l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self})\}$$

then we can apply method $l_i$ to all objects of this type and the result must have type $M_i(Object)$. That is, the following rule is necessary:

$$\frac{obj \in Object}{obj_\circ l_i \in M_i(Object)} \tag{7.13}$$

For example for all $bug \in AdvanceFleas$ we should have $bug_\circ\mathtt{getX} \in \mathbb{Z}$ and $bug_\circ\mathtt{move} \in AdvanceFleas$.

## 7.3 Definition of Object Types

In this section we are going to give a definition of a type of objects satisfying the properties outlined above. We start with

**Definition 7.7** *Let $X$ and $A$ be types, then*

$$X \lhd A \text{ iff } X \subseteq (X \to A)$$

This definition says that if $X \lhd A$ then we can apply elements of type $X$ to themselves. Therefore we have the following

**Lemma 7.8** *If $X \lhd A$ then if $o \in X$ then $o(o) \in A$.*
*In particular, if*

$$X \lhd \{l_1 : M_1; \ldots; l_n : M_n\} \tag{7.14}$$

*then for any $o \in X$ we have that $o_\circ l_i \in M_i$.*

So intuitively, the type $X = \mathfrak{O}\, \mathit{Self}.\{l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self})\}$ should satisfy the property (7.14). Of course the empty type always satisfies (7.14), but we want the object type to contain as many elements as possible. So we define the object type as a union of all types $X$ satisfying (7.14).

**Definition 7.9** *Generally, let $M[X]$ be a type for any type $X$. We define a type $\mathfrak{D}\, X.M[X]$ as a union of all types $X$ that satisfy $X \lhd M[X]$:*

$$\mathfrak{D}\, X.M[X] = \cup\{X : \mathbb{U} \mid X \lhd M[X]\}.$$

We will also use the following abbreviation: $\{\!| l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self}) |\!\}$ for the type $\mathfrak{D}\, \mathit{Self}.\{l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self})\}$.

This definition does not satisfy the property (7.14), but it turns out that we do not need this property. We still have the following lemma:

**Lemma 7.10** *If $M[X]$ is monotone in $X$ (w.r.t. subtyping relation) then for any $o \in \mathfrak{D}\, X.M[X]$ we have that $o(o) \in M[\mathfrak{D}\, X.M[X]]$.*

*In particular, if $M_i[X]$ are monotone in $X$, then if $O = \{\!| l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self}) |\!\}$ and $o \in O$ then $o_\circ l_i \in M_i(O)$.*

**Proof** If $o \in \mathfrak{D}\, X.M[X]$ then there is a type $X \in \mathbb{U}$, such that $o \in X$ and $X \lhd M(X)$. By Lemma 7.8, $o(o) \in M(X)$. Since $X \subseteq \mathfrak{D}\, X.M[X]$ and $M$ is monotone, $M(X) \subseteq M[\mathfrak{D}\, X.M[X]]$. Therefore $o(o) \in M[\mathfrak{D}\, X.M[X]]$.

The second part of the lemma immediately follows form the first part.

This lemma provides us the elimination rule for objects (7.13).

$$\frac{obj \in Object}{obj_\circ l_i \in M_i(Object)}$$

where $Object = \mathfrak{D}\, \mathit{Self}.\{l_1 : M_1(\mathit{Self}); \ldots; l_n : M_n(\mathit{Self})\}$.

The remaining question is how to prove that this type is nonempty? For example, how can one prove that $simplestFlea \in SimplestFleas$? This is nontrivial question. We should find a type $X$ satisfying $o \in X \lhd M(X)$. We will need another constructor.

## 7.4 Extensibility

Definition 7.9 has one important disadvantage: objects of the type $\mathfrak{D}\, \mathit{Self}.M[\mathit{Self}]$ are not extensible, in the sense that we cannot add new methods to them.

**Example 7.11** *Let $a$ be an arbitrary object of the type $\{\!| \mathtt{move} : \mathit{Self} |\!\}$. Consider another object*

$$b = \{\!| \mathtt{move} = a |\!\}.$$

*Then $b$ is also an object of the type $\{\!| \mathtt{move} : \mathit{Self} |\!\}$. The problem with $b$ is that $b$ is not extensible. For instance an extension*

$$b' = \{\!| \mathtt{move} = a; \mathtt{new\_method} = t |\!\}$$

*does not have a type $\{\!| \mathtt{move} : \mathit{Self}; \mathtt{new\_method} : T |\!\}$ because $b'_\circ \mathtt{move}_\circ \mathtt{new\_method}$ is undefined.*

Extensible objects should have type $T$ such that not only $T \lhd M(T)$, but also any extensions (subtype) $X$ of $T$ should meet $X \lhd M(X)$.

## 7.5 Updatable Fields

Another problem with Definition 7.9 is that we can not update fields and methods of the objects of the type $\mathfrak{D}\, X.M[X]$.

**Example 7.12** *Suppose we want to update a field $\mathtt{x}$ of an object $obj$ of the type $\{\!| \mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z} |\!\}$. That is, we want to prove that $obj_\circ \mathtt{x} := 1$ has the same type. We cannot always do that. For example let*

$$\mathfrak{o} = \{\!| \mathtt{x} = 0; \mathtt{y} = \texttt{if } \mathtt{x} = 0 \texttt{ then } 1 \texttt{ else } error |\!\}.$$

*This object has type $\{\!| \mathtt{x} : \mathbb{Z}; \mathtt{y} : \mathbb{Z} |\!\}$, but $obj_\circ \mathtt{x} := 1$ does not have this type.*

So to be able to update fields, we will need some additional restrictions on the object type. To deal with this problem we need

**Definition 7.13** *Let* $\mathtt{x}$ *be a label,* $A$ *and* $T$ *be types. Let us define the following relation on* $x$*,* $A$*,* $T$*:*

$$\{\!|\mathtt{x} : A|\!\} \prec T \textit{ iff } \forall a : A.\forall t : T.(t_\circ\mathtt{x} := a) \in T$$

*Note that this is a ternary relation, not a binary relation between types.*

Informally speaking $\{\!|\mathtt{x} : A|\!\} \prec T$ gives a lower bound for a type of field $\mathtt{x}$ in $T$. It plays the same role as Hickey's $\prec$-relation [21] and Zwanenburg's $\#$-relation [39].

We are going to define a type of *extensible* objects satisfying conditions of the form $P(T) = \{\!|\mathtt{x} : A|\!\} \prec T$. More precisely, for a given declaration $M(X)$ and a given condition $P(X)$ we define a type of extensible objects $\mathfrak{E}_P\, M$. Here $M \in \mathbb{U} \to \mathbb{U}$ and $P(X)$ is a predicate on types. We cannot give this definition for an arbitrary $M$ and $P$. $M$ should be monotone and continuous and $P$ should be closed under intersection (see below).

## 7.6 Topology

Subtyping relation forms a partial order over the types in $\mathbb{U}$. Partial order forms a topology: the topology is formed by intervals

$$[A; B] \triangleq \{X : \mathbb{U} \mid A \subseteq X \subseteq B\}$$

### 7.6.1 Continuous functions

Usually the following definition is used for continuity of monotone operators:

**Definition 7.14** *Monotone type operator* $M$ *is* continuous *iff for any non-empty family of types* $\{X_i\}_{i:I}$

$$M(\bigcap_{i:I} X_i) = \bigcap_{i:I} M(X_i)$$

Most of the monotone type constructors are continuous: $X + Y$, $X \times Y$, and most important $\{\mathtt{x} : X; \mathtt{y} : Y\}$ are continuous.

### 7.6.2 Semicontinuous functions

We will need to iterate $N(X) = X \to M(X)$. Unfortunately, this operator is not monotone and not continuous in any sense. For example, $N(X) = \neg X$ is clearly not continuous. So we will need a less strict definition.

**Definition 7.15** *A type operator* $N$ *is* (upper) semicontinuous *iff for any non-empty family of types* $\{X_i\}_{i:I}$

$$N(\bigcap_{i:I} X_i) \supseteq \bigcap_{i:I} N(X_i)$$

It is clear that any continuous function is semicontinuous. We can also prove that if $M(X)$ is semicontinuous then $N(X) = X \to M(X)$ is also semicontinuous. It follows from the following two lemmas.

**Lemma 7.16** *If* $F(X, Y)$ *is a function that is anti-monotone in its first argument and semicontinuous in its second argument, then* $N(X) = F(X, Y)$ *is semicontinuous.*

**Lemma 7.17** $X \to Y$ *is a monotone and continuous in* $Y$ *and anti-monotone in* $X$*.*

Note that a monotone function is semicontinuous iff it is continuous.

### 7.6.3 Closed properties and sets

**Definition 7.18** *We will say that a property $P$ of types is* closed (under intersection) *iff for any family of types $\{X_i\}_{i:I}$ if $P$ is true for all $X_i$'s then $P$ is true for intersection of $X_i$'s, i.e.*

$$(\forall i : I. P(X_i)) \Rightarrow P(\bigcap_{i:I} X_i)$$

In other words it means that $P$ is semicontinuous function from $\mathbb{U}$ to propositions.

**Definition 7.19** *We will say that a subtype $V$ of $\mathbb{U}$ is* closed (under intersection) *iff for any family of types $\{X_i\}_{i:I}$ where $X_i \in V$, intersection of all $X_i$'s is also in $V$.*

Note that it follows from this definition that if $P$ is closed then $P$ is true for Top and Top is in any closed set of types (since Top $= \bigcap_{i:\text{Void}} \text{Void}$).

**Example 7.20** $P(T) = \{|x : A|\} \prec T$ *is a closed predicate.*

## 7.7 Extensible objects: Formal definitions

Now we are going to give a formal definition of $\mathfrak{E}_P\, M$.

**Definition 7.21** *Let $P : \mathbb{U} \to \mathbb{P}$ be a property of types. Define $\mathbb{U}_P \triangleq \{X : \mathbb{U} \mid P(X)\}$.*

**Definition 7.22** *Let $V$ be a subtype of $\mathbb{U}$, and $A$ and $B$ be types. Define*

$$V[A; B] \triangleq \{X : V \mid A \subseteq X \subseteq B\}.$$

**Definition 7.23** *Let $M$ be a continuous monotone type operator. Let $P$ be a closed proposition and $T$ be a type. Define a relation*

$$T \propto_P M \triangleq \forall X : \mathbb{U}_P.\, \exists Y : \mathbb{U}_P[X \cap T; X].\, Y \lhd M(Y).$$

*We will refer to such $Y$ as $M^*(X)$.*

**Definition 7.24** *Let $M$ and $P$ be as in definition 7.23. Define*

$$\mathfrak{E}_P\, M \triangleq \bigcup \{T : \mathbb{U}' \mid T \propto_P M\}$$

We cannot prove that for any type $T$ if $T \propto_P M$ then $T \lhd M(T)$. But we can prove the following

**Lemma 7.25** *Let $M$ and $P$ be as in definition 7.23. For any type $T$, if $T \propto_P M$ then there is a type $T'$ in $\mathbb{U}_P$, such that $T \subseteq T'$ and $T' \lhd M(T')$.*

**Proof** Take $T' = M^*(Top)$.

**Corollary 7.26** $\mathfrak{E}_P\, M \subseteq \mathfrak{O}\, M$

**Lemma 7.27** *$T \propto_P M$ is anti-monotone in $T$ and monotone in $M$, i.e.,*

- *$T_1 \subseteq T_2 \propto_P M$ implies $T_1 \propto_P M$ and*
- *If $M_1(X) \subseteq M_2(X)$ for all $X$ then $T \propto_P M_1$ implies $T \propto_P M_2$*

**Corollary 7.28** *$\mathfrak{E}_P\, M$ is monotone in $M$.*

**Lemma 7.29** *$T \propto_P Top$ for any type $T$.*

**Lemma 7.30** *Let $\{M_i\}_{i:I}$ be a family of continuous functions. If $T \propto_P M_i$ for all $i \in I$ then $T \propto_P \bigcap_{i:I} M_i$ (i.e. $T \propto_P M$ is continuous in $M$).*

**Proof** Since $T \propto_P X.M_i(X)$ we have a family of functions $M_i^*$, s.t. $M_i^*(X) \in \mathbb{U}_P[X \cap T; X]$ and $M_i^*(X) \lhd M_i(M_i^*(X))$ for any $X \in \mathbb{U}_P$.

Now, we want to prove $T \propto_P X. \bigcap_{i:I} M_i(X)$. We are given $X \in \mathbb{U}_P$. We want to find $Y \in \mathbb{U}_P[X \cap T; X]$ such that $Y \lhd \bigcap_{i:I} M_i(Y)$.

Let $N_i(X) = X \to M_i(X)$. We now that $N_i$'s are semicontinuous. Note that $Y \lhd \bigcap_{i:I} M_i(Y)$ iff $Y \subseteq \bigcap_{i:I} N_i(Y)$.

Define a family of sequences of types $Y_n^i$ by induction:

- $Y_0^i = M_i^*(X)$
- $Y_{n+1}^i = M_i^*(\bigcap_{j:I} Y_n^j)$

Then we prove the following:

0. $Y_n \in \mathbb{U}_P$

Proof: straightforward induction using facts that $M_i^* : \mathbb{U}_P \to \mathbb{U}_P$ and $\mathbb{U}_P$ is closed under intersection.

1. $Y_{n+1}^i \subseteq Y_n^j$ for any indexes $i$, $j$.

Proof: $Y_{n+1}^i = M_i^*(\bigcap_{j:I} Y_n^j) \subseteq \bigcap_{j:I} Y_n^j \subseteq Y_n^j$

As a corollary we have:

2. $\bigcap_{n:\mathbb{N}} Y_n^i = \bigcap_{n:\mathbb{N}} Y_n^j$ for any indexes $i$, $j$.

Now, define $Y$ as this intersection $Y = \bigcap_{n:\mathbb{N}} Y_n^i$.

3. $Y_n^i \subseteq N_i(Y_n^i)$

Proof: Since $N_i^*(X) \subseteq N_i(N_i^*(X))$ for any $X \in \mathbb{U}_P$.

4. $Y \subseteq N_i(Y)$

Proof: $N_i(Y) = N_i(\bigcap_{n:\mathbb{N}} Y_n^i) \supseteq \bigcap_{n:\mathbb{N}} N_i(Y_n^i) \supseteq \bigcap_{n:\mathbb{N}} Y_n^i = Y$.

5. $Y_n^i \subseteq X$

Proof: By induction.

6. $Y_n^i \supseteq X \cap T$

Proof: By induction.

So we have that $Y \in \mathbb{U}_P[X \cap T; X]$ and $Y \subseteq N_i(Y)$.

**Corollary 7.31** *$\mathfrak{E}_P$ is continuous in $M$.*

In particular,

$$\mathfrak{E}_P(M_1 \cap M_2) = (\mathfrak{E}_P\, M_1) \cap (\mathfrak{E}_P\, M_2).$$

This establishes the following rule

$$\frac{o \in \mathfrak{E}_P\, M_1 \qquad o \in \mathfrak{E}_P\, M_1}{o \in \mathfrak{E}_P(M_1 \cap M_2)}$$

**Lemma 7.32** *Let $P$ be a closed proposition, $M_1$ be a continuous function from $\mathbb{U}_P$ to $\mathbb{U}_P$ and $M_2$ be a continuous monotone function. Let*

$$T = \bigcap_{\substack{X:\mathbb{U}_P \\ X \subseteq M_1(X)}} M_2(X)$$

*Let $N_2(X) = X \to M_2(X)$. If $N_2 \in \{X : \mathbb{U}_P \mid X \lhd M_1(X)\} \to \mathbb{U}_P$, then for any $T'$ such that $T' \propto_P M_1$ we have that $T' \cap T \propto_P M_2$.*

**Proof** Since $T' \propto_P X.M_1(X)$, there is a function $M_1^*$, s.t. $M_1^*(X) \in \mathbb{U}_P[X \cap T'; X]$ and $M_1^*(X) \lhd M_1(M_1^*(X))$ for any $X \in \mathbb{U}_P$.

Let $N_2(X) = X \to M_2(X)$.

Now, we want to prove $T \cap T' \propto_P X.N_2(X)$. That is for any type $X \in \mathbb{U}_P$ we should find a type $Y$ s.t. $Y \in \mathbb{U}_P[X \cap T \cap T'; X]$ and $Y \subseteq N_2(Y)$.

Define the following sequence:

- $Y_0 = M_1^*(\text{Top})$
- $Y_{n+1} = M_1^*(N_2(Y_n) \cap X)$

Define $Y = \bigcap\limits_{n:\mathbb{N}} Y_n$. Then we can prove the following:

1. $Y_n \in \mathbb{U}_P$ and $Y_n \lhd M_1(Y_n)$.
Proof: straightforward simultaneous induction.
2. $T \subseteq N_2(Y_n)$
Proof: By definition of $T$.
3. $T \cap T' \cap X \subseteq X_n$
Proof: Two cases:

$$Y_0 = M_1^*(\text{Top}) \supseteq T' \supseteq T \cap T' \cap X$$

$$Y_{n+1} = M_1^*(N_2(Y_n) \cap X) \supseteq N_2(Y_n) \cap X \cap T' \supseteq T \cap T' \cap X$$

4. $N_2(Y_n) \supseteq Y_{n+1}$
Proof: $Y_{n+1} = M_1^*(N_2(Y_n) \cap X) \subseteq N_2(Y_n)$.
5. $Y \subseteq N_2(Y)$
Proof: $N_2(Y) \supseteq \bigcap\limits_{n:\mathbb{N}} N_2(Y_n) \supseteq \bigcap\limits_{n:\mathbb{N}} Y_{n+1} \supseteq Y$.
6. $Y \subseteq X$
Proof: $Y \subseteq Y_1 = M_1^*(N_2(Y_0) \cap X) \subseteq N_2(Y_0) \cap X \subseteq X$.
7. $Y \in \mathbb{U}_P[X \cap T \cap T'; X]$
Proof: By (1), (3) and (6).
So we are done.

**Corollary 7.33** *If $P$, $M_1$, $M_2$ and $T$ are as in Lemma 7.32 then*

$$T \cap \mathfrak{E}_P\, M_1 \subseteq \mathfrak{E}_P(M_1 \cap M_2)$$

This corollary provides a main introduction rule for objects:

$$\frac{\begin{array}{c}\Gamma \vdash o \in \mathfrak{E}_P\, M_1 \\ \Gamma; X : \mathbb{U}; P(X); X \lhd M_1(X) \vdash o \in M_2(X) \\ \Gamma; X : \mathbb{U}; P(X); X \lhd M_1(X) \vdash P(M_2(X))\end{array}}{\Gamma \vdash o \in \mathfrak{E}_P\, M_1 \cap M_2}$$

## 7.8 Object Calculus

The rules that we proved above are represented in Table7.2.

We can make these rules more concrete substituting record types in place of $M$. We will use the notation

$$\{\!|\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_n : M_n[\mathit{Self}]|\!\}_P$$

for $\mathfrak{E}_P\,.(\lambda \mathit{Self}.\{\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_n : M_n[\mathit{Self}]\})$.

Table 7.2: Basic typing rules of object calculus

$$\frac{\begin{array}{c}\Gamma \vdash o \in \mathfrak{E}_P\, M_1 \\ \Gamma; X : \mathbb{U}; P(X); X \lhd M_1(X) \vdash o \in M_2(X) \\ \Gamma; X : \mathbb{U}; P(X); X \lhd M_1(X) \vdash P(M_2(X))\end{array}}{\Gamma \vdash o \in \mathfrak{E}_P\, M_1 \cap M_2}$$

$$\frac{\Gamma; X : \mathbb{U}; P(X) \vdash M_1(X) \subseteq M_2(X)}{\Gamma \vdash \mathfrak{E}_P\, M_1 \subseteq \mathfrak{E}_P\, M_2}$$

$$\frac{\Gamma; i : I \vdash o \in \mathfrak{E}_P\, M_i}{\Gamma \vdash o \in \mathfrak{E}_P \bigcap_{i:I} M_i}$$

$$\frac{\Gamma \vdash o \in \mathfrak{E}_P\, M}{\Gamma \vdash o \in \mathfrak{D}\, X.M(X)}$$

In these rules $P$ are closed predicates and $M$'s are monotone continuous functions.

Table 7.3: Some derived rules of object calculus

$$\frac{\begin{array}{c}\Gamma \vdash \mathfrak{o}(o)\{\!|\mathtt{x}_n = m_n[\mathit{self}]|\!\} \in \{\!|\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_{n-1} : M_{n-1}[\mathit{Self}]|\!\}_P \\ \Gamma; X : \mathbb{U}; P(X); X \lhd \{\mathtt{x}_1{:}M_1[X]; \ldots; \mathtt{x}_{n-1}{:}M_{n-1}[X]\}; self : X \vdash m_n[self] \in M_n[X] \\ \Gamma; X : \mathbb{U}; P(X); X \lhd \{\mathtt{x}_1{:}M_1[X]; \ldots; \mathtt{x}_{n-1}{:}M_{n-1}[X]\} \vdash P(X \to M_n[X])\end{array}}{\Gamma \vdash \mathfrak{o}(o)\{\!|\mathtt{x}_n = m_n[\mathit{self}]|\!\} \in \{\!|\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_n : M_n[\mathit{Self}]|\!\}_P}$$

$$\frac{\Gamma \vdash X \lhd \{\mathtt{x}_1 : M_1[X]; \ldots; \mathtt{x}_n : M_n[X]\} \qquad \Gamma \vdash o \in X}{\Gamma \vdash o_\circ \mathtt{x}_i \in M_i[X]}$$

$$\frac{\Gamma \vdash o \in \mathit{obj} \in \{\!|\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_n : M_n[\mathit{Self}]|\!\}_P}{\Gamma \vdash o \in \mathit{obj} \in \{\!|\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_n : M_n[\mathit{Self}]|\!\}}$$

$$\frac{\Gamma \vdash o \in \mathit{obj} \in \{\!|\mathtt{x}_1 : M_1[\mathit{Self}]; \ldots; \mathtt{x}_n : M_n[\mathit{Self}]|\!\}}{\Gamma \vdash o_\circ \mathtt{x}_i \in M_i[X]}$$

$$\frac{\Gamma \vdash \{\!|\mathtt{x} : A|\!\} \prec X \qquad \Gamma \vdash o \in X \qquad \Gamma \vdash a \in A}{\Gamma \vdash o_\circ \mathtt{x} := a \in X}$$

$$\frac{\Gamma \vdash \mathtt{x} \neq \mathtt{y}}{\Gamma \vdash \{\!|\mathtt{x} : A|\!\} \prec (X \to \{\mathtt{y} : B\})}$$

$$\frac{\Gamma \vdash \mathtt{A} \subseteq B}{\Gamma \vdash \{\!|\mathtt{x} : A|\!\} \prec (X \to \{\mathtt{x} : B\})}$$

In these rules $P$ is a closed predicates and $M$'s are monotone continuous functions.

## 7.9 Example

Now we show how rules of Table 7.3 works. Let us prove that $movableFlea$ has type

$$\begin{array}{l} MovableFleas = \\ \quad \{\!|\ \mathtt{getX} : \mathbb{Z}; \\ \quad\ \mathtt{getNextX} : \mathbb{Z}; \\ \quad\ \mathtt{move} : \mathit{Self} \\ \quad |\!\} \end{array}$$

Remember

$$\begin{array}{l} movableFlea = \\ \quad \{\!|\mathtt{x} = 0; \\ \quad\ \mathtt{getX} = \mathtt{x}; \\ \quad\ \mathtt{getNextX} = \mathtt{getX} + 1; \\ \quad\ \mathtt{move} = (\mathtt{x} := \mathtt{getNextX}); \\ \quad |\!\} \end{array}$$

Let $P(T) = \{\!|\mathtt{x} : \mathbb{Z}|\!\} \prec T$. It is enough to prove that $movableFlea \in \{\!|x : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}; \mathtt{getNextX} : \mathbb{Z}; \mathtt{move} : \mathit{Self}\,|\!\}_P$. Applying introduction four times we get four main subgoals:

$X : \mathbb{U}; P(X); self : X \vdash 0 \in \mathbb{Z}$

$X : \mathbb{U}; P(X); X \lhd \{\mathtt{x} : \mathbb{Z}\}; \mathit{self} : X \vdash \mathit{self}_\circ\mathtt{x} \in \mathbb{Z}$

$X : \mathbb{U}; P(X); X \lhd \{\mathtt{x} : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}\}; \mathit{self} : X \vdash \mathit{self}_\circ\mathtt{getX} + 1 \in \mathbb{Z}$

$X : \mathbb{U}; P(X); X \lhd \{\mathtt{x} : \mathbb{Z}; \mathtt{getX} : \mathbb{Z}; \mathtt{getNextX} : \mathbb{Z}\}; \mathit{self} : X \vdash \mathit{self}_\circ\mathtt{x} := \mathit{self}_\circ\mathtt{getNextX} \in X$

and four goals with the conclusions: $P(X \to \{x : \mathbb{Z}\})$, $P(X \to \{getX : \mathbb{Z}\})$, and so on. These subgoals are momentary proved by introduction rules for $\prec$.

The first main subgoal is trivial. The second and the third one are proved by elimination rules for $\lhd$. And finally, the last one is proved by the elimination rule for $\prec$.

## BIBLIOGRAPHY

[1] Martín Abadi and Luca Cardelli. A semantics of object types. In *Proceedings of 9$^{th}$ IEEE Symposium on Logic in Computer Science*, pages 332–341, Paris, France, July 1994. IEEE, IEEE Computer Society Press.

[2] Stuart F. Allen. A Non-type-theoretic Definition of Martin-Löf's Types. In D. Gries, editor, *Proceedings of the 2$^{nd}$ IEEE Symposium on Logic in Computer Science*, pages 215–224. IEEE Computer Society Press, June 1987.

[3] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[4] Lennart Augustsson. Cayenne — a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.

[5] Gustavo Betarte and Alvaro Tasistro. Extension of Martin-Löf's type theory with record types and subtyping. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, volume 36 of *Oxford Logic Guides*, pages 21–39, Oxford, 1998. Clarendon Press.

[6] Mark Bickford and Jason J. Hickey. Predicate transformers for infinite-state automata in NuPRL type theory. In *Proceedings of* $3^{rd}$ *Irish Workshop in Formal Methods*, 1999.

[7] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Proceedings of FOOL 3*, 1996.

[8] Robert L. Constable. Types in logic, mathematics and programming. In Sam Buss, editor, *Handbook of Proof Theory*, chapter 10. Elsevier Science, 1998.

[9] Robert L. Constable et al. *Implementing Mathematics with the NuPRL Development System*. Prentice-Hall, NJ, 1986.

[10] Robert L. Constable and Jason Hickey. NuPRL's class theory and its applications. In Friedrich L. Bauer and Ralf Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[11] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the $\lambda$-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill Book Company, Cambridge, Massachusetts, 1994.

[13] Judicaël Courant. An applicative module calculus. In *TAPSOFT*, Lecture Notes in Computer Science, pages 622–636, Lille, France, April 1997. Springer-Verlag.

[14] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, pages 63–69. Springer-Verlag, NY, 1971.

[15] J-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[16] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *IEEE Symposium on Foundations of Computer Science*, pages 8–21, October 1978.

[17] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics and Language Design*. Types, Semantics, and Language Design. MIT Press, Cambridge, MA, 1994.

[18] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[19] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. Accepted to the TPHOLs 2003 Conference, 2003.

[20] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page.

[21] Jason J. Hickey. A predicative type-theoretic interpretation of objects. Unpublished, 1997.

[22] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[23] Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

[24] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE, IEEE Computer Society Press.

[25] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proceedings of the 1st Symposium on Logic in Computing Science*, pages 237–248. IEEE, 1986.

[26] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, 2003.

[27] Alexei Kopylov and Aleksey Nogin. Markov's principle for propositional type theory. In L. Fribourg, editor, *Computer Science Logic, Proceedings of the 10th Annual Conference of the EACSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 570–584. Springer-Verlag, 2001.

[28] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122. ACM Press, 1994.

[29] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[30] Per Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[31] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.

[32] Aleksey Nogin. Quotient types: A modular approach. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 263–280. Springer-Verlag, 2002.

[33] Chris Okasaki. Red-black trees un a functional setting. *Journal of Functional Programming*, 9(4):471–477, May 1999.

[34] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[35] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.

[36] Robert Pollack. Dependently typed records for representing mathematical structure. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13$^{th}$ International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 461–478. Springer-Verlag, 2000.

[37] Garrel Pottinger. A type assignment for the strongly normalizable $\lambda$-terms. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, London, 1980.

[38] John C. Reynolds. Design of the programming language forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[39] Jan Zwanenburg. A type system for record concatenation and subtyping. In Kim Bruce and Giuseppe Longo, editors, *Informal proceedings of Third Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, 1996.