# A Polymorphic λ-calculus with Type:Type

*Luca Cardelli*

Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue, Palo Alto, CA 94301

SRC Research Report 10, May 1, 1986.

# 1. Introduction

Ever since the notion of *type* was introduced in computer science, there have been people claiming that Type (the collection of all types) should be a type, and hence be a member of itself. This was intended to permit computations yielding types as results, and it seemed to be a straightforward extension of the (otherwise) successful principle that, in programming languages, one should be able to operate uniformly on every entity in the domain of discourse.

The notion of Type being a type (written Type:Type) is however in apparent contrast with the requirements of static typechecking, philosophically dubious, and often (when formalized) mathematically inconsistent. All the existing programming languages with Type:Type have been designed on shaky grounds, and built without much investigation of the really fundamental difficulties that have to be resolved.

Pebble [Burstall 84] is probably the first serious attempt at defining a programming language with a consistent type structure based on dependent types and the notion of Type:Type. Pebble's semantics and typechecking strategy are defined operationally, and leave open some semantic questions. This paper describes a type-theoretical and denotational semantics foundation for Pebble-like languages.

To make sense of Type:Type it seems necessary to abandon at least two very familiar ideas: first, the notion that a type is a set of values and second, the notion that typechecking should be decidable, or, in pragmatic terms, that program compilation should always terminate. The problem with the first notion is that we would need a set which contains itself as an element - a concept not supported by ordinary set theory. However, we shall see that we can use less intuitive meanings of *type* which are not inconsistent with Type:Type. The second notion is questionable in the light of modern requirements of software engineering and data base languages.

To deal with Type:Type it is useful to introduce a relatively unfamiliar idea: the notion of *dependent type*. This allows one to assign useful types to computations operating on types, and hence to perform static typechecking even in very dynamic situations where types are being computed. We have to be careful with the meaning of *static* typechecking in this context: although computations do not require run-time typing, typechecking requires arbitrary computations; it is still possible to identify a first phase of *static* (although possibly not terminating) typechecking, followed by a second execution phase which does not require any typechecking.

All the key ideas presented here are fairly well known. Semantic domains where Type:Type holds were defined by Scott [Scott 76]. The basic language semantics problems were solved by McCracken [McCracken 79]. Dependent types come from intuitionistic type theory [Martin-Löf 80], and their denotational semantics is studied by Barendregt and Rezus [Barendregt 83] [Rezus 85]. Relevant languages have been proposed such as Russell [Boehm 80] and Pebble [Burstall 84]. Relevant formal systems have been widely studied; they include intuitionistic logic and type theory [Scott 70] [Martin-Löf 80]; second-order lambda calculus [Girard 72] [Reynolds 74] [Fortune 83] [Bruce 84]; Automath [de Bruijn 80] [Barendregt 83]; the theory of constructions [Coquand 85a, 85b]); the foundations of Russell [Hook 84] [Donahue 85]; and a calculus with Type:Type [Meyer 86].

Somehow, a clear connection between these ideas, from the programming language point of view, was missing. For example, Girard discovered that a dependent type system where Type:Type holds is inconsistent as a logic system (but it is not inconsistent as a computation system); Meyer and Reinhold concluded that a language where Type:Type holds is computationally consistent but pathological and dangerous for programming; Burstall and Lampson concluded that such a language is useful and necessary for large-scale programming, but did not investigate a denotational semantics; McCracken developed semantic techniques more general than the language she applied them to; and Girard and Reynolds developed systems where Type:Type does not hold but which have all the complications, from a denotational point of view, of systems where it does holds.

The purpose of this paper is to present a polymorphic language with Type:Type, where types are values. Such a language has a formal type inference system and formal denotational semantics, and can be used to model second-order λ-calculus and the basic features of Pebble. The type system is very expressive, possibly the most expressive type system known to date, as it embodies the power of intuitionistic type theory. However, this expressive power comes at the cost of decidability: there are no typechecking algorithms for the full language. In practice a whole range of partial typecheckers can be written, from very simple to very complex; a larger number of correct programs will be recognized as legal by more complex typecheckers.

Languages with the Type:Type property will be useful in areas such as software engineering, to express computations involving collections of types and values (like parametric modules and linking), and data bases, to express computations parameterized on data schemas. Although there may still be philosophical objections to its use, it is now clear that the Type:Type property makes perfect sense and can be incorporated in useful and semantically understood tools.

## 2. Syntax

An expression can be a variable x, the constant Type (the "type of all types"), a typed λ-expression, an application, a *universal* type expression (also called *dependent product*), a pair, a rip-expression (for taking pairs apart), an *existential* type expression (also called *dependent sum*), or a μ-expression for recursion.

Functions have universal types, which in simple cases degenerate to ordinary function types, and pairs have existential types, which degenerate to cartesian product types. Universal types are dependent because the *type* of the result of a function may depend on the *value* of the argument. Existential types are dependent, as the *type* of the second component of a pair may depend on the *value* of the first component.

Instead of having two primitives, fst and snd, for breaking up pairs, we have a single primitive: let x,y = c in d (where we always assume x≠y). Here c evaluates to a pair whose first and second components are bound to x and y in the scope d.

In the following syntax, ι are identifiers and ε are expressions:

ε ::=
ι |
Type |
λι: ε. ε | ε(ε) | ∀ι: ε. ε |

$<\varepsilon, \varepsilon>$ | $\mathsf{let}\ \iota,\iota = \varepsilon\ \mathsf{in}\ \varepsilon$ | $\exists\iota{:}\ \varepsilon.\ \varepsilon$ |

$\mu\iota{:}\ \varepsilon.\ \varepsilon$

There is no distinction between type-expressions and value-expressions: they are both expressions and can be intermixed. If we add a constant Int for the integer type (which can be defined, as we shall see), then type-expressions and integer-expressions have similar status: they denote disjoint classes of well-typed expressions. This is in contrast to most languages, where integer-expressions can be *computed*, while type-expressions are used only in typechecking.

The recursion operator $\mu$ can be used to define both recursive types (in the form $\mu x{:}\ \mathsf{Type}.\ \varepsilon$) and recursive values.

**Notation**: we shall use any of the letters a, b, c, A, B, C, as metavariables ranging over expressions, and x, y, z as metavariables ranging over identifiers. The upper case letters will normally be used for expressions which are type expressions.

# 3. Scoping and Substitution

The scoping of identifiers is determined by the following definition of the set of free variables (FV) of an expression:

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(Type) & = & \emptyset \\
FV(\lambda x{:}\ A.\ b) & = & FV(A) \cup (FV(b)\text{-}\{x\}) \\
FV(b(c)) & = & FV(b) \cup FV(c) \\
FV(\forall x{:}\ A.\ B) & = & FV(A) \cup (FV(B)\text{-}\{x\}) \\
FV(<b,c>) & = & FV(b) \cup FV(c) \\
FV(\mathrm{let}\ x,y = b\ \mathrm{in}\ c) & = & FV(b) \cup (FV(c)\text{-}\{x,y\}) \\
FV(\exists x{:}\ A.\ B) & = & FV(A) \cup (FV(B)\text{-}\{x\}) \\
FV(\mu x{:}\ A.\ b) & = & FV(A) \cup (FV(b)\text{-}\{x\})
\end{array}
$$

Note that in expressions like $\lambda x{:}x.\ x$ the second occurrence of x is not bound by the first one, so that the previous expression is equivalent to $\lambda y{:}x.\ y$ (in general, we identify expressions up to renaming of bound variables). The following definition of substitution makes this clear, where $b\{x\leftarrow a\}$ is the result of substituting the expression a for all the free occurrences of the variable x in the expression b.

$$
\begin{array}{lcll}
x\{x\leftarrow a\} & = & a & \\
y\{x\leftarrow a\} & = & y & y \neq x \\
Type\{x\leftarrow a\} & = & Type & \\
(\lambda x{:}\ A.\ b)\{x\leftarrow a\} & = & \lambda x{:}\ A\{x\leftarrow a\}.\ b & \\
(\lambda y{:}\ A.\ b)\{x\leftarrow a\} & = & \lambda y'{:}\ A\{x\leftarrow a\}.\ b\{y\leftarrow y'\}\{x\leftarrow a\} & y,y' \neq x;\quad y' \notin FV(b) \\
(b(c))\{x\leftarrow a\} & = & b\{x\leftarrow a\}(c\{x\leftarrow a\}) & \\
(\forall x{:}\ A.\ B)\{x\leftarrow a\} & = & \forall x{:}\ A\{x\leftarrow a\}.\ B & \\
(\forall y{:}\ A.\ B)\{x\leftarrow a\} & = & \forall y'{:}\ A\{x\leftarrow a\}.\ B\{y\leftarrow y'\}\{x\leftarrow a\} & y,y' \neq x;\quad y' \notin FV(B) \\
<b, c>\{x\leftarrow a\} & = & <b\{x\leftarrow a\}, c\{x\leftarrow a\}> &
\end{array}
$$

$$
\begin{array}{lcll}
(\text{let } x,y = b \text{ in } c)\{x \leftarrow a\} & = & \text{let } x,y = b\{x \leftarrow a\} \text{ in } c & \\
(\text{let } y,x = b \text{ in } c)\{x \leftarrow a\} & = & \text{let } y,x = b\{x \leftarrow a\} \text{ in } c & \\
(\text{let } y,z = b \text{ in } c)\{x \leftarrow a\} & = & \text{let } y',z' = b\{x \leftarrow a\} \text{ in } c\{y \leftarrow y'\}\{z \leftarrow z'\}\{x \leftarrow a\} & \\
 & & & y,z,y',z' \neq x;\ y',z' \notin FV(c) \\
(\exists x{:}\ A.\ B)\{x \leftarrow a\} & = & \exists x{:}\ A\{x \leftarrow a\}.\ B & \\
(\exists y{:}\ A.\ B)\{x \leftarrow a\} & = & \exists y'{:}\ A\{x \leftarrow a\}.\ B\{y \leftarrow y'\}\{x \leftarrow a\} & y,y' \neq x;\ \ y' \notin FV(B) \\
(\mu x{:}\ A.\ b)\{x \leftarrow a\} & = & \mu x{:}\ A\{x \leftarrow a\}.\ b & \\
(\mu y{:}\ A.\ b)\{x \leftarrow a\} & = & \mu y'{:}\ A\{x \leftarrow a\}.\ b\{y \leftarrow y'\}\{x \leftarrow a\} & y,y' \neq x;\ \ y' \notin FV(b)
\end{array}
$$

## 4. Type Assignments

A type assignment $\Upsilon$ is a partial function from identifiers to terms, and it can be written as $x_1{:}A_1,\ldots,x_n{:}A_n,\ldots$ where each $x_i$ is a variable and each $A_i$ is a type ($\varnothing$ is the empty assignment). The assignment $\Upsilon.x{:}A$ is the same as $\Upsilon$, except that the variable $x$ is now associated with $A$. We shall allow $\Upsilon.x{:}A$ only in situations where $x \notin dom(\Upsilon)$, where $dom(\Upsilon)$ is the domain of $\Upsilon$, i.e., it is the set of variables which are defined in $\Upsilon$.

Assignments are used in the next sections to derive typing relations ($\Upsilon \vdash a{:}A$) and equivalence relations ($\Upsilon \vdash a \leftrightarrow b$). In each of these cases, when something holds relative to an assignment, it also holds for larger assignments:

$$
[\text{Extend 1}] \qquad \frac{\Upsilon \vdash A{:}Type \qquad \Upsilon \vdash b{:}B}{\Upsilon,\, x{:}A \vdash b{:}B}
$$

$$
[\text{Extend 2}] \qquad \frac{\Upsilon \vdash A{:}Type \qquad \Upsilon \vdash a \leftrightarrow b}{\Upsilon,\, x{:}A \vdash a \leftrightarrow b}
$$

The relations $\Upsilon \vdash a{:}A$ and $\Upsilon \vdash a \leftrightarrow b$ are defined in the next section.

## 5. Type Inference and Reduction Rules

Although types and values are mixed, all expressions, whether denoting types or other values, must be *well-typed*. In particular we must be able to determine types for expressions denoting types and computations over types.

This section defines a set of typing and reduction rules. If an expression can be typed according to the typing rules, then it is well-typed. Evaluation (i.e., reduction) may be required during the process of assigning types to expressions. In general it is undecidable whether an expression is well-typed.

The presentation of the rules follows and extends that of Meyer and Reinhold [Meyer 86]. The rules are divided into groups; the *typing rules* describe the typing relations between values and types. There is exactly one typing rule for each syntactic class of expressions, plus one rule which is responsible for introducing computation during typing. The *conversions* group adapts the usual conversion rules for untyped $\lambda$-calculus to the typed case. The remaining two groups are lengthy but uninteresting; they extend the basic conversion relation to a substitutive equivalence relation on terms.

The first line of assumptions (if any) in each of the rules states the well-formedness of the expressions in the second line of assumptions (if any) and in the conclusions. Such well-formedness assumptions are often omitted in the presentation of similar inference systems such as Martin-Löf's.

**Typing Rules**

[Assumption]
$$\frac{\Upsilon \vdash A{:}Type}{\Upsilon, x{:}A \vdash x{:}A}$$

[Type Formation] $\varnothing \vdash Type{:}\ Type$

[∀ Formation]
$$\frac{\begin{array}{c}\Upsilon \vdash A{:}Type \\ \Upsilon, x{:}A \vdash B{:}Type\end{array}}{\Upsilon \vdash (\forall x{:}A.B){:}\ Type}$$

[∀ Introduction]
$$\frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon, x{:}A \vdash b{:}B\end{array}}{\Upsilon \vdash (\lambda x{:}A.\ b){:}\ (\forall x{:}A.B)}$$

[∀ Elimination]
$$\frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon \vdash a{:}A \qquad \Upsilon \vdash b{:}\ \forall x{:}A.\ B\end{array}}{\Upsilon \vdash b(a){:}\ B\{x \leftarrow a\}}$$

[∃ Formation]
$$\frac{\begin{array}{c}\Upsilon \vdash A{:}Type \\ \Upsilon, x{:}A \vdash B{:}Type\end{array}}{\Upsilon \vdash (\exists x{:}A.B){:}\ Type}$$

[∃ Introduction]
$$\frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon \vdash a{:}A \qquad \Upsilon \vdash b{:}\ B\{x \leftarrow a\}\end{array}}{\Upsilon \vdash \langle a, b\rangle{:}\ (\exists x{:}A.B)}$$

[∃ Elimination]
$$\frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \quad \Upsilon, z{:}(\exists x{:}A.B) \vdash C{:}Type \\ \Upsilon \vdash c{:}\ \exists x{:}A.B \quad \Upsilon, x{:}A, y{:}B \vdash d{:}\ C\{z \leftarrow \langle x, y\rangle\}\end{array}}{\Upsilon \vdash \text{let } x,y = c \text{ in } d{:}\ C\{z \leftarrow c\}}$$

[μ Formation]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \qquad \Upsilon, x{:}A \vdash a{:}A}{\Upsilon \vdash (\mu x{:}A.a) : A}$$

[Reduction]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon \vdash B{:}\mathrm{Type} \quad \Upsilon \vdash a{:}A \quad \Upsilon \vdash A \leftrightarrow B}{\Upsilon \vdash a{:}B}$$

**Conversion Rules**

[β Conversion]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon, x{:}A \vdash B{:}\mathrm{Type} \quad \Upsilon \vdash a{:}A \quad \Upsilon, x{:}A \vdash b{:}B}{\Upsilon \vdash (\lambda x{:}A.\ b)(a) \leftrightarrow b\{x \leftarrow a\}}$$

[η Conversion]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon, x{:}A \vdash B{:}\mathrm{Type} \quad \Upsilon \vdash b{:}\ \forall x{:}A.\ B}{\Upsilon \vdash (\lambda x{:}A.\ b(x)) \leftrightarrow b}$$

[σ Conversion]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon, x{:}A \vdash B{:}\mathrm{Type} \quad \Upsilon, z{:}(\exists x{:}A.B) \vdash C{:}\mathrm{Type} \quad \Upsilon \vdash a{:}A \quad \Upsilon \vdash b{:}\ B\{x \leftarrow a\} \quad \Upsilon, x{:}A, y{:}B \vdash d{:}\ C\{z \leftarrow \langle x, y\rangle\}}{\Upsilon \vdash \mathrm{let}\ x,y = \langle a,b\rangle\ \mathrm{in}\ d \leftrightarrow d\{x \leftarrow a, y \leftarrow b\}}$$

[π Conversion]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon, x{:}A \vdash B{:}\mathrm{Type} \quad \Upsilon \vdash c{:}\ \exists x{:}A.B}{\Upsilon \vdash \langle \mathrm{let}\ x,y = c\ \mathrm{in}\ x,\ \mathrm{let}\ x,y = c\ \mathrm{in}\ y\rangle \leftrightarrow c}$$

[μ Conversion]

$$\frac{\Upsilon \vdash A{:}\mathrm{Type} \qquad \Upsilon, x{:}A \vdash a{:}A}{\Upsilon \vdash (\mu x{:}A.\ a) \leftrightarrow a\{x \leftarrow (\mu x{:}A.\ a)\}}$$

**Equality Rules**

[Reflexivity]

$$\frac{\Upsilon \vdash a{:}A}{\Upsilon \vdash a \leftrightarrow a}$$

[Symmetry]

$$\frac{\Upsilon \vdash a \leftrightarrow b}{\Upsilon \vdash b \leftrightarrow a}$$

$$\text{[Transitivity]} \qquad \frac{\Upsilon \vdash a \leftrightarrow b \qquad \Upsilon \vdash b \leftrightarrow c}{\Upsilon \vdash a \leftrightarrow c}$$

**Congruence Rules**

$$\text{[}\forall\text{ Formation-C]} \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}Type \\ \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon \vdash A \leftrightarrow A' \qquad \Upsilon, x{:}A \vdash B \leftrightarrow B'\end{array}}{\Upsilon \vdash (\forall x{:}A.\ B) \leftrightarrow (\forall x{:}A'.\ B')}$$

$$\text{[}\forall\text{ Introduction-C]} \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon, x{:}A \vdash b{:}\ B \\ \Upsilon \vdash A \leftrightarrow A' \qquad \Upsilon, x{:}A \vdash b \leftrightarrow b'\end{array}}{\Upsilon \vdash (\lambda x{:}A.\ b) \leftrightarrow (\lambda x{:}A'.\ b')}$$

$$\text{[}\forall\text{ Elimination-C]} \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon \vdash a{:}A \qquad \Upsilon \vdash b{:}\ \forall x{:}A.\ B \\ \Upsilon \vdash a \leftrightarrow a' \qquad \Upsilon \vdash b \leftrightarrow b'\end{array}}{\Upsilon \vdash b(a) \leftrightarrow b'\ (a')}$$

$$\text{[}\exists\text{ Formation-C]} \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}Type \\ \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon \vdash A \leftrightarrow A' \qquad \Upsilon, x{:}A \vdash B \leftrightarrow B'\end{array}}{\Upsilon \vdash (\exists x{:}A.\ B) \leftrightarrow (\exists x{:}A'.\ B')}$$

$$\text{[}\exists\text{ Introduction-C]} \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \\ \Upsilon \vdash a{:}A \qquad \Upsilon \vdash b{:}\ B\{x \leftarrow a\} \\ \Upsilon \vdash a \leftrightarrow a' \qquad \Upsilon \vdash b \leftrightarrow b'\end{array}}{\Upsilon \vdash \langle a, b\rangle \leftrightarrow \langle a', b'\rangle}$$

$$\text{[}\exists\text{ Elimination-C]} \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash B{:}Type \quad \Upsilon, z{:}(\exists x{:}A.B) \vdash C{:}Type \\ \Upsilon \vdash c{:}\ \exists x{:}A.B \qquad \Upsilon, x{:}A, y{:}B \vdash d{:}\ C\{z \leftarrow \langle x, y\rangle\} \\ \Upsilon \vdash c \leftrightarrow c' \qquad \Upsilon, x{:}A, y{:}B \vdash d \leftrightarrow d'\end{array}}{\Upsilon \vdash \text{let } x,y = c \text{ in } d \leftrightarrow \text{let } x,y = c' \text{ in } d'}$$

$$[\mu \text{ Formation-C}] \quad \frac{\Upsilon \vdash A{:}Type \quad \Upsilon, x{:}A \vdash a{:}A \quad \Upsilon \vdash A \leftrightarrow A' \quad \Upsilon, x{:}A \vdash a \leftrightarrow a'}{\Upsilon \vdash (\mu x{:}A.\ a) \leftrightarrow (\mu x{:}A'.\ a')}$$

**Prop** (Conversion preserves types)

If $\Upsilon \vdash a{:}A$ and $\Upsilon \vdash a \leftrightarrow b$ then $\Upsilon \vdash b{:}A$

# 6. Examples

The best way to understand the typing rules is to look at examples and special cases, and this is what we are going to do in this section.

In many situations the constant Type will be omitted, according to the following abbreviations:

$$\begin{aligned} \forall x.\ A &\cong \forall x{:}Type.\ A \\ \exists x.\ A &\cong \exists x{:}Type.\ A \\ \mu x.\ A &\cong \mu x{:}Type.\ A \\ \lambda x.\ A &\cong \lambda x{:}Type.\ A \end{aligned}$$

***6.1 Functions***

All the common type constants and operators can be defined. We start with the function space operator which can be defined as a special case of universal types. The $\forall$-introduction rule says that a function $\lambda x{:}A.\ b$ has a universal type $\forall x{:}A.B$, where in general $x$ may occur in $B$ (for example, $\lambda A{:}Type.\ \lambda x{:}A.\ x$ has type $\forall A{:}Type.\forall x{:}A.\ A$, where the variable $A$ occurs in the body of the outer quantifier). However, if $x$ does not occur in $B$, then $x$ is useless, and we can take $A \rightarrow B$ as an abbreviation for $\forall x{:}A.B$. Now, according to the $\forall$-introduction rule, some functions can have function types instead of the more general universal types; for example, $\lambda x{:}A.\ x$ has type $A \rightarrow A$ (which is the same as $\forall x{:}A.\ A$). Even better, $\rightarrow$ can be defined as a term, as opposed to a metasyntactic abbreviation: it is a function which takes two types $A$ and $B$ and returns $\forall x{:}A.\ B$ (note that $x$ is not free in $B$ as $B$ is a variable). The type operator $\rightarrow$ is then immediately used to describe its own type, and the Type:Type property is put into action:

$$\rightarrow\ = \lambda A.\ \lambda B.\ \forall x{:}A.\ B \quad : \ Type \rightarrow Type \rightarrow Type$$

The following inference rules can then be derived from the $\forall$ rules:

$$[\rightarrow \text{ Formation}] \quad \frac{\Upsilon \vdash A{:}Type \quad \Upsilon \vdash B{:}Type}{\Upsilon \vdash A \rightarrow B{:}\ Type}$$

$$\text{[}\rightarrow\text{ Introduction]}\quad \frac{\Upsilon \vdash A{:}Type \qquad \Upsilon \vdash B{:}Type \qquad \Upsilon, x{:}A \vdash b{:}B}{\Upsilon \vdash (\lambda x{:}A.\ b){:}\ (A\rightarrow B)}$$

$$\text{[}\rightarrow\text{ Elimination]}\quad \frac{\Upsilon \vdash A{:}Type \quad \Upsilon \vdash B{:}Type \quad \Upsilon \vdash a{:}A \qquad \Upsilon \vdash b{:}\ A\rightarrow B}{\Upsilon \vdash b(a){:}\ B}$$

### *6.2 Basic types*

The type $\forall A.\ A$ is called Void as there are no normal-form (n.f.) terms of this type:

$$Void = \forall A.\ A \quad : \ Type$$
$$\bot = \lambda A.\ \mu x{:}A.\ x \quad : \quad Void$$

Given a value $v{:}Void$, we can obtain a value of *any* type $A$ as $v(A)$, by [$\forall$ Elimination]; for example $\bot(Type){:}Type$, and $\bot(\bot(Type)){:}\ \bot(Type)$. The term $\bot(A)$ represents the divergent computation of type $A$ (i.e., a computation which was trying to deliver something of type $A$, but diverged in the process).

Some useful type constants are Unit (there is a single n.f. term of this type) and Bool (there are two n.f. terms of this type):

$$Unit = \forall A.\ \forall a{:}A.\ A \quad : \ Type$$
$$unity = \lambda A.\ \lambda a{:}A.\ a \quad : \ Unit$$

$$Bool = \forall A.\ \forall a{:}A.\ \forall b{:}A.\ A \quad : \ Type$$
$$true = \lambda A.\ \lambda a{:}A.\ \lambda b{:}A.\ a \quad : \ Bool$$
$$false = \lambda A.\ \lambda a{:}A.\ \lambda b{:}A.\ b \quad : \ Bool$$
$$cond = \lambda c{:}Bool.\ \lambda A.\ \lambda a{:}A.\ \lambda b{:}A.\ c(A)(a)(b) \quad : \ Bool \rightarrow Bool$$

We can use the following syntactic sugar for conditionals, where we may want to omit the "both A" type information whenever possible.

$$\text{if } c \text{ then } a \text{ else } b \text{ both } A \ \cong \ cond(c)(A)(a)(b)$$

### *6.3 Pairs*

A cartesian product operator can be defined as a special case of existential types, in the same way in which we defined $\rightarrow$ as a special case of universal types. An object of type $\exists x{:}A.\ B$ (where in general $x$ may occur in $B$) is a pair $<a,b>$ where $a$ has type $A$ and $b$ has type $B\{x\leftarrow a\}$. If $x$ does not occur in $B$ then an object of type $\exists x{:}A.\ B$ is simply a pair $<a,b>$ with $a$ in $A$ and $b$ in $B$, and we can abbreviate $\exists x{:}A.\ B$ as $A \times B$. Again, we can define $\times$ as a term:

$$
\begin{aligned}
&\times = \lambda A.\ \lambda B.\ \exists x{:}A.\ B \quad : \ \mathrm{Type}\to\mathrm{Type}\to\mathrm{Type}\\
&\quad \mathrm{pair} = \lambda A.\ \lambda B.\ \lambda a{:}A.\ \lambda b{:}B.\ <a,b> \quad : \ \forall A.\ \forall B.\ A\to B\to(A\times B)\\
&\quad \mathrm{fst} = \lambda A.\ \lambda B.\ \lambda c{:}\ A\times B.\ \mathrm{let}\ x,y = c\ \mathrm{in}\ x \quad : \ \forall A.\ \forall B.\ (A\times B)\to A\\
&\quad \mathrm{snd} = \lambda A.\ \lambda B.\ \lambda c{:}\ A\times B.\ \mathrm{let}\ x,y = c\ \mathrm{in}\ y \quad : \ \forall A.\ \forall B.\ (A\times B)\to B\\
&\quad \mathrm{split} = \lambda A.\ \lambda B.\ \lambda C{:}(A\times B\to\mathrm{Type}).\ \lambda c{:}\ A\times B.\ \lambda d{:}\ (\forall x{:}A.\ \forall y{:}B.\ C(<x,y>)).\ \mathrm{let}\ x,y = c\ \mathrm{in}\ d(x)(y)\\
&\qquad\qquad : \forall A.\ \forall B.\ \forall C{:}(A\times B\to\mathrm{Type}).\ \forall c{:}\ A\times B.\ \forall d{:}\ (\forall x{:}A.\ \forall y{:}B.\ C(<x,y>)).\ C(c)
\end{aligned}
$$

The usual properties of pair, fst and snd (modulo the heavy typing) hold:

$$
\begin{aligned}
&c = \mathrm{pair}(A)(B)(a)(b)\\
&\mathrm{fst}(A)(B)(c) \quad \leftrightarrow \quad a\\
&\mathrm{snd}(A)(B)(c) \quad \leftrightarrow \quad b\\
&\mathrm{pair}(A)(B)(\mathrm{fst}(A)(B)(c))(\mathrm{snd}(A)(B)(c)) \quad \leftrightarrow \quad c
\end{aligned}
$$

We now have the following rules, derivable from the $\exists$ rules:

$$[\times\ \mathrm{Formation}] \qquad \frac{\Upsilon \vdash A{:}\mathrm{Type} \qquad \Upsilon \vdash B{:}\mathrm{Type}}{\Upsilon \vdash A\times B{:}\ \mathrm{Type}}$$

$$[\times\ \mathrm{Introduction}] \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon \vdash B{:}\mathrm{Type}\\ \Upsilon \vdash a{:}A \qquad \Upsilon \vdash b{:}B\end{array}}{\Upsilon \vdash \mathrm{pair}(A)(B)(a)(b){:}\ A\times B}$$

$$[\times\ \mathrm{Elimination}] \qquad \frac{\begin{array}{c}\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon \vdash B{:}\mathrm{Type} \quad \Upsilon,z{:}\ A\times B \vdash C{:}\mathrm{Type}\\ \Upsilon \vdash c{:}\ A\times B \quad \Upsilon \vdash d{:}\ \forall x{:}A.\ \forall y{:}B.\ C\{z\leftarrow \mathrm{pair}(A)(B)(x)(y)\}\end{array}}{\Upsilon \vdash \mathrm{split}(A)(B)(\lambda z{:}\ A\times B.\ C)(c)(d){:}\ C\{z\leftarrow c\}}$$

Alternatively, the cartesian product can be defined using only universal types. The previous rules still hold and can be derived from the $\forall$ rules, except that we can only obtain a weaker form of [$\times$ Elimination]:

$$\times = \lambda A.\ \lambda B.\ \forall C.\ (A\to B\to C)\to C \quad : \ \mathrm{Type}\to\mathrm{Type}\to\mathrm{Type}$$

$$\text{pair} = \lambda A.\ \lambda B.\ \ \lambda a{:}A.\ \lambda b{:}B.\ \ \lambda C.\ \lambda c{:}\ A\to B\to C.\ c(a)(b) \quad : \ \forall A.\ \forall B.\ A\to B\to (A\times B)$$

$$\text{fst} = \lambda A.\ \lambda B.\ \lambda c{:}\ A\times B.\ c(A)(\lambda a{:}A.\ \lambda b{:}B.\ a) \quad : \ \forall A.\ \forall B.\ (A\times B)\to A$$

$$\text{snd} = \lambda A.\ \lambda B.\ \lambda c{:}\ A\times B.\ c(B)(\lambda a{:}A.\ \lambda b{:}B.\ b) \quad : \ \forall A.\ \forall B.\ (A\times B)\to B$$

$$\begin{aligned}\text{split} = \ & \lambda A.\ \lambda B.\ \lambda C{:}(A\times B\to \text{Type}).\ \lambda c{:}\ A\times B.\\ & \quad \lambda d{:}\ (\forall x{:}A.\ \forall y{:}B.\ C(\text{pair}(A)(B)(x)(y))).\ d(\text{fst}(A)(B)(c))(\text{snd}(A)(B)(c))\\ & \quad : \forall A.\ \forall B.\ \forall C{:}(A\times B\to \text{Type}).\ \forall c{:}\ A\times B.\ \forall d{:}\ (\forall x{:}A.\ \forall y{:}B.\ C(\text{pair}(A)(B)(x)(y))).\\ & \qquad C(\text{pair}(A)(B)(\text{fst}(A)(B)(c))(\text{snd}(A)(B)(c)))\end{aligned}$$

### *6.4 Unions*

Disjoint unions can also be encoded in terms of universal types only:

$$+ = \lambda A.\ \lambda B.\ \forall C.\ (A\to C)\to (B\to C)\to C \quad : \ \text{Type}\to\text{Type}\to\text{Type}$$

$$\text{inl} = \lambda A.\ \lambda B.\ \lambda a{:}A.\ \lambda C.\ \lambda f{:}\ A\to C.\ \lambda g{:}\ B\to C.\ f(a) \quad : \ \forall A.\ \forall B.\ A\to (A+B)$$

$$\text{inr} = \lambda A.\ \lambda B.\ \lambda b{:}B.\ \lambda C.\ \lambda f{:}\ A\to C.\ \lambda g{:}\ B\to C.\ g(b) \quad : \ \forall A.\ \forall B.\ B\to (A+B)$$

$$\text{unioncase} = \lambda A.\ \lambda B.\ \lambda c{:}\ A+B.\ \lambda C.\ \lambda f{:}A\to C.\ \lambda g{:}B\to C.\ c(C)(f)(g) \quad : \ \forall A.\ \forall B.\ (A+B)\to (A+B)$$

The operations inl and inr inject a value in the left or right component of a union. The unioncase operation takes an element c of A+B, and calls one of two functions on the projection of c into A or B, as appropriate. For example, the operation isl (detecting whether something is in the left component of a union) can be programmed as:

$$\begin{aligned}\text{isl} = \ & \lambda A.\ \lambda B.\ \lambda c{:}\ A+B.\\ & \quad \text{unioncase}(A)(B)(c)(\text{Bool})\\ & \qquad (\lambda a{:}A.\ \text{true})\\ & \qquad (\lambda b{:}B.\ \text{false})\\ & \ \ : \ \forall A.\ \forall B.\ (A+B)\to \text{Bool}\end{aligned}$$

Unfortunately, this definition of disjoint unions is not as general as we might want. We would like to have the following inference rules:

$$[\text{+ Formation}] \qquad \frac{\Upsilon \vdash A{:}\text{Type} \qquad \Upsilon \vdash B{:}\text{Type}}{\Upsilon \vdash A+B{:}\text{Type}}$$

$$[\text{+ Introduction}] \qquad \frac{\begin{matrix}\Upsilon \vdash A{:}\text{Type} & \Upsilon \vdash B{:}\text{Type}\\ \Upsilon \vdash a{:}A & \Upsilon \vdash b{:}B\end{matrix}}{\Upsilon \vdash \text{inl}(a){:}\ A+B \qquad \Upsilon \vdash \text{inr}(b){:}\ A+B}$$

$$[+\ \text{Elimination}]\quad \frac{\Upsilon \vdash A{:}\mathrm{Type} \quad \Upsilon \vdash B{:}\mathrm{Type} \quad \Upsilon, z{:}A+B \vdash C{:}\ \mathrm{Type} \qquad \Upsilon \vdash c : A+B \quad \Upsilon, a{:}A \vdash f{:}\ C\{z \leftarrow inl(a)\} \quad \Upsilon, b{:}B \vdash g{:}\ C\{z \leftarrow inr(b)\}}{\Upsilon \vdash (\mathrm{unioncase}\ c\ \mathrm{of}\ l \Rightarrow a.f\ ,\ r \Rightarrow b.g) : C\{z \leftarrow c\}}$$

These rules (in particular, [+ Elimination]) cannot be derived, and have to be taken as primitives. Given that we have to introduce new primitives, it is more convenient to introduce n-ary disjoint unions instead of binary ones. The constructors inl and inr are now replaced by a countable set S of constructors which, for convenience, we take to be identifiers (the syntactic context will allow us to distinguish them from identifiers used as variables). N-ary union types are denoted by $[s_1{:}A_1, \ldots, s_n{:}A_n]$, where here and in the following rules we assume $s_1, \ldots, s_n \in S$. Moreover, we identify union types up to reordering of their components, and case expressions up to reordering of their branches.

$$[[]\ \text{Formation}]\quad \frac{\Upsilon \vdash A_1{:}\mathrm{Type} \quad \ldots \quad \Upsilon \vdash A_n{:}\mathrm{Type}}{\Upsilon \vdash [s_1{:}A_1, \ldots, s_n{:}A_n] : \mathrm{Type}}$$

$$[[]\ \text{Introduction}]\quad \frac{\Upsilon \vdash A_1{:}\mathrm{Type} \quad \ldots \quad \Upsilon \vdash A_n{:}\mathrm{Type} \qquad \Upsilon \vdash a{:}A_i}{\Upsilon \vdash [s_i{=}a] : [s_1{:}A_1, \ldots, s_n{:}A_n]} \qquad i \in \{1, \ldots, n\}$$

$$[[]\ \text{Elimination}]\quad \frac{\begin{array}{c}\Upsilon \vdash A_1{:}\mathrm{Type} \quad \ldots \quad \Upsilon \vdash A_n{:}\mathrm{Type} \quad \Upsilon, z{:}[s_1{:}A_1, \ldots, s_n{:}A_n] \vdash C{:}\ \mathrm{Type} \\ \Upsilon \vdash c{:}\ [s_1{:}A_1, \ldots, s_n{:}A_n] \\ \Upsilon, a_1{:}A_1 \vdash f_1{:}\ C\{z \leftarrow [s_1{=}a_1]\} \quad \ldots \quad \Upsilon, a_n{:}A_n \vdash f_n{:}\ C\{z \leftarrow [s_n{=}a_n]\}\end{array}}{\Upsilon \vdash (\mathrm{case}\ c\ \mathrm{of}\ s_1 \Rightarrow a_1.\ f_1, \ldots, s_n \Rightarrow a_n.\ f_n) : C\{z \leftarrow c\}}$$

The semantics of these types is not treated later in the paper. However, disjoint unions have important applications, and we describe some of them in the following subsections.

Binary disjoint unions can now be defined by taking:

$$A+B \quad \cong \quad [inl{:}A,\ inr{:}B]$$

### *6.5 Finite sets*

Finite sets (also called enumeration types) can now be defined as a degenerate form of n-ary unions:

$$[s_1, \ldots, s_n] \quad \cong \quad [s_1{:}\mathrm{Unit}, \ldots, s_n{:}\mathrm{Unit}]$$
$$|s_i| \quad \cong \quad [s_i{=}\mathrm{unity}]$$

### *6.6 Dependent conditionals*

A dependent conditional is one in which the types of the *then* and *else* branches may differ. Its type is then dependent on the test value. We want:

$$\text{if x then 3 else true} \quad : \quad \text{if x then Int else Bool}$$

An operator of this kind can only be defined by using disjoint unions:

| | | | |
|---|---|---|---|
| Bool | $\cong$ | [true,false] | |
| true | $\cong$ | \|true\| | |
| false | $\cong$ | \|false\| | |
| if a then b else c | $\cong$ | case a of true $\Rightarrow$ x.b, false $\Rightarrow$ x.c | ($x \notin b,c$) |

It is now possible to check that the dependent conditional example above is well-typed.

### *6.7 Records*

Again using n-ary unions, we can define unordered labeled cartesian products, that is records (here $b,x \notin a_1, \ldots , a_n, A_1, \ldots , A_n$):

$$\{s_1 : A_1, \ldots , s_n : A_n\} \cong \forall b: [s_1, \ldots ,s_n].\ \text{case } b \text{ of } s_1 \Rightarrow x.A_1 , \ldots , s_n \Rightarrow x.A_n$$

$$\{s_1 = a_1, \ldots , s_n = a_n\} \cong \lambda b: [s_1, \ldots ,s_n].\ \text{case } b \text{ of } s_1 \Rightarrow x.a_1 , \ldots , s_n \Rightarrow x.a_n$$

$$a.s \cong a(|s|)$$

It is now possible to deduce that, for example:

$$\{a = 3,\ b = \text{true}\} : \quad \{a : \text{Int},\ b : \text{Bool}\}$$

Note also that these record types are more flexible than ordinary record types in programming languages: field selection can be expressed as $a(b)$ where $a$ is a record and $b$ is a selector that does not have to be statically known.

### *6.8 Recursion*

Types can be defined by recursion. An element of $List(A)$ (lists whose elements have type $A$), is either the empty list (of type $Unit$) or a pair of an $A$ (the head of the list) and, recursively, a $List(A)$ (the tail of the list).

$$List = \lambda A.\ \mu B.\ Unit + (A \times B) \quad : \quad Type \rightarrow Type$$

$$\text{nil} = \lambda A.\ \text{inl(Unit)}(A\times\text{List}(A))(\text{unity}) \quad : \ \forall A.\ \text{List}(A)$$

$$\text{cons} = \lambda A.\ \lambda x{:}A.\ \lambda y{:}\text{List}(A).\ \text{inr(Unit)}(A\times\text{List}(A))(\text{pair}(A)(\text{List}(A))(x)(y))$$

$$: \ \forall A.\ A\to\text{List}(A)\to\text{List}(A)$$

$$\text{listcase} = \lambda A.\ \lambda C.\ \lambda c{:}\ \text{List}(A).\ \lambda f{:}\text{Unit}\to C.\ \lambda g{:}(A\times\text{List}(A))\to C.$$

$$\text{case(Unit)}(A\times\text{List}(A))(C)(c)(f)(g)$$

$$: \ \forall A.\ \forall C.\ \text{List}(A)\to(\text{Unit}\to C)\to(A\times\text{List}(A)\to C)\to C$$

Natural numbers can be defined as $\text{List(Unit)}$, which is the same as $\mu B.\ \text{Unit} + (\text{Unit} \times B)$.

$$\text{Nat} = \text{List(Unit)} \quad : \ \text{Type}$$

$$0 = \text{nil(Unit)} \quad : \ \text{Nat}$$

$$\text{succ} = \lambda n{:}\text{Nat}.\ \text{cons(Unit)(unity)}(n) \quad : \ \text{Nat}\to\text{Nat}$$

$$\text{pred} = \lambda n{:}\text{Nat}.\ \text{listcase(Unit)(Nat)}(n)(\lambda a{:}\text{Unit}.\ 0)(\lambda a{:}\text{Unit}\times\text{Nat}.\ \text{snd(Unit)(Nat)}(a)) \quad : \ \text{Nat}\to\text{Nat}$$

$$\text{zero} = \lambda n{:}\text{Nat}.\ \text{listcase(Unit)(Bool)}(n)\ (\lambda a{:}\text{Unit}.\ \text{true})\ (\lambda a{:}\text{Unit}\times\text{Nat}.\ \text{false}) \quad : \ \text{Nat}\to\text{Bool}$$

Alternatively, the list type can be defined as:

$$\text{List} = \mu B{:}\ \text{Type}\to\text{Type}.\ \lambda A.\ \text{Unit} + (A \times B(A)) \quad : \ \text{Type}\to\text{Type}$$

### *6.9 Self-describing objects*

If we have a type $A$ and an object $a$ of that type, we can form the (dependent) pair $\langle A,a\rangle$. Such a pair has type $\exists A.\ A$, and since *any* object which has a type can be so treated, $\exists A.\ A$ is also called $\text{Any}$:

$$\text{Any} = \exists A.\ A \quad : \ \text{Type}$$

$$\text{any} = \lambda A.\ \lambda a{:}A.\ \langle A,a\rangle \quad : \ \forall A.\ A\to\text{Any}$$

$$\text{typeof} = \lambda a{:}\text{Any}.\ \text{let } x,y = a \text{ in } x{:}\ \text{Any}\to\text{Type}$$

$$\text{valueof} = \lambda a{:}\text{Any}.\ \text{let } x,y = a \text{ in } y\ {:}\forall a{:}\text{Any}.\ \text{typeof}(a)$$

If we have an object $a$ of type $\text{Any}$ we really have no information about it, because it can be anything. But such an object has its own type information with it, and this can be extracted by the $\text{typeof}(a)$ operation. It is also possible to extract the value by $\text{valueof}(a)$, whose type is $\text{typeof}(a)$.

### *6.10 Dependent pairs*

The typeof and valueof operations can be generalized to arbitrary existential types. In their general form they are known as the *left projection* and *right projection* of a dependent product:

$$\mathrm{lft} = \lambda A.\ \lambda B{:}A\to\mathrm{Type}.\ \lambda c{:}\ (\exists x{:}A.\ B(x)).\ \mathrm{let}\ x,y = c\ \mathrm{in}\ x$$
$$:\ \forall A.\ \forall B{:}\ A\to\mathrm{Type}.\ \forall c{:}\ (\exists x{:}A.\ B(x)).\ A$$
$$\mathrm{rht} = \lambda A.\ \lambda B{:}A\to\mathrm{Type}.\ \lambda c{:}\ (\exists x{:}A.\ B(x)).\ \mathrm{let}\ x,y = c\ \mathrm{in}\ y$$
$$:\ \forall A.\ \forall B{:}\ A\to\mathrm{Type}.\ \forall c{:}\ (\exists x{:}A.\ B(x)).\ B(\mathrm{lft}(A)(B)(c))$$

### *6.11 Data abstraction*

Following Mitchell and Plotkin [Mitchell 85], it is possible to treat *abstract types* as existential types, given operators for building and examining objects of existential types [Girard 71]. We consider here a pack operator, which packages an object so that it has an existential type (and hides some type information which is usually interpreted as the *representation* of the abstract type), and an open operator, which allows one to open and use a package without getting access to the representation. See examples in [Cardelli 85].

$$\mathrm{pack} = \lambda A{:}\mathrm{Type}\to\mathrm{Type}.\ \lambda B.\ \lambda a{:}A(B).\ \langle B,a\rangle$$
$$:\ \forall A{:}\mathrm{Type}\to\mathrm{Type}.\ \forall B.\ \forall a{:}A(B).\ \exists C.\ A(C)$$
$$\mathrm{open} = \lambda A{:}\mathrm{Type}\to\mathrm{Type}.\ \lambda B.\ \lambda a{:}(\exists C.\ A(C)).\ \lambda f{:}(\forall D.\ \forall a{:}A(D).\ B).\ \mathrm{let}\ x,y = a\ \mathrm{in}\ f(x)(y)$$
$$:\ \forall A{:}\mathrm{Type}\to\mathrm{Type}.\ \forall B.\ \forall a{:}(\exists C.\ A(C)).\ \forall f{:}(\forall D.\ \forall a{:}A(D).\ B).\ B$$

$$\mathrm{Abs} = \lambda A.\ A\times(A\to\mathrm{Nat}) \quad : \mathrm{Type}\to\mathrm{Type}$$
$$\mathrm{AbsType} = \exists A.\ \mathrm{Abs}(A) \quad : \mathrm{Type}$$
$$a = \mathrm{pack}(\mathrm{Abs})(\mathrm{Nat})(\mathrm{pair}(\mathrm{Nat})(\mathrm{Nat}\to\mathrm{Nat})(3)(\mathrm{succ})) \quad : \mathrm{AbsType}$$
$$\mathrm{open}(\mathrm{Abs})(\mathrm{Bool})(a)(\lambda D.\ \lambda a{:}\mathrm{Abs}(D).\ \mathrm{zero}(\mathrm{snd}(D)(D\to\mathrm{Nat})(a)(\mathrm{fst}(D)(D\to\mathrm{Nat})(a))) \quad \leftrightarrow \quad \mathrm{false}$$

### *6.12 More on dependent types*

It is interesting to notice that two terms $\Pi$ and $\Sigma$ can be defined which are essentially equivalent to the $\forall$ and $\exists$ constructs respectively.

$$\Pi = \lambda A.\ \lambda B{:}A\to\mathrm{Type}.\ \forall a{:}A.\ B(a) \quad : \forall A.\ \forall B{:}A\to\mathrm{Type}.\ \mathrm{Type}$$

$$\Sigma = \lambda A.\ \lambda B{:}A\to\mathrm{Type}.\ \exists a{:}A.\ B(a) \quad : \forall A.\ \forall B{:}A\to\mathrm{Type}.\ \mathrm{Type}$$
$$\mathrm{pair} = \lambda A.\ \lambda B{:}A\to\mathrm{Type}.\ \lambda a{:}A.\ \lambda b{:}B(a).\ \langle a,b\rangle$$
$$:\ \forall A.\ \forall B{:}A\to\mathrm{Type}.\ \forall a{:}A.B(a)\to \Sigma(A)(B)$$
$$\mathrm{unpair} = \lambda A.\ \lambda B{:}A\to\mathrm{Type}.\ \lambda C{:}(\Sigma(A)(B)\to\mathrm{Type}).\ \lambda c{:}\Sigma(A)(B).\ \lambda d{:}\ (\forall a{:}A.\forall b{:}B(a).\ C(\langle a,b\rangle)).$$
$$\mathrm{let}\ x,y = c\ \mathrm{in}\ d(x)(y)$$
$$:\ \forall A.\ \forall B{:}A\to\mathrm{Type}.\ \forall C{:}(\Sigma(A)(B)\to\mathrm{Type}).\ \forall c{:}\Sigma(A)(B).\ \forall d{:}\ (\forall a{:}A.\forall b{:}B(a).\ C(\langle a,b\rangle)).\ C(c)$$

$$\mathsf{lft} = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda c{:}\ \Sigma(A)(B).\ \mathsf{unpair}(A)(B)(\lambda x{:}\Sigma(A)(B).\ A)(c)(\lambda a{:}A.\ \lambda b{:}B(a).\ a)$$
$$:\ \forall A.\ \forall B{:}\ A\to\mathsf{Type}.\ \forall c{:}\ \Sigma(A)(B).\ A$$
$$\mathsf{rht} = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda c{:}\ \Sigma(A)(B).\ \mathsf{unpair}(A)(B)(\lambda x{:}\Sigma(A)(B).\ B(\mathsf{lft}(A)(B)(x)))(c)(\lambda a{:}A.\ \lambda b{:}B(a).\ b)$$
$$:\ \forall A.\ \forall B{:}\ A\to\mathsf{Type}.\ \forall c{:}\ \Sigma\ (A)(B).\ B(\mathsf{lft}(A)(B)(c))$$

Generalizing the way we defined cartesian product in terms of universal types only, we can attempt to define Σ without using existential types. This plan however fails; here is the best we can do [Martin-Löf 71]:

$$\Sigma' = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \forall C.\ (\forall a{:}A.B(a)\to C)\to C \qquad :\ \forall A.\ \forall B{:}A\to\mathsf{Type}.\ \mathsf{Type}$$
$$\mathsf{pair}' = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda a{:}A.\ \lambda b{:}B(a).\ \lambda C.\ \lambda c{:}\ (\forall a{:}A.B(a)\to C).\ c(a)(b)$$
$$:\ \forall A.\ \forall B{:}A\to\mathsf{Type}.\ \forall a{:}A.B(a)\to \Sigma'(A)(B)$$
$$\mathsf{unpair}' = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda C.\ \lambda p{:}\Sigma'(A)(B).\ \lambda q{:}\ (\forall a{:}A.B(a)\to C).\ p(C)(q)$$
$$:\ \forall A.\ \forall B{:}A\to\mathsf{Type}.\ \forall C.\ \Sigma'(A)(B)\to (\forall a{:}A.B(a)\to C)\to C$$

The problem is that unpair´ is not as flexible as unpair, as its result type (C) cannot be made dependent. Using unpair´ we can define a version of lft, but the corresponding version of rht is not typeable:

$$\mathsf{lft}' = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda c{:}\ \Sigma'(A)(B).\ \mathsf{unpair}'(A)(B)(A)(c)(\lambda a{:}A.\ \lambda b{:}B(a).\ a)$$
$$= \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda c{:}\ \Sigma'(A)(B).\ c(A)(\lambda a{:}A.\ \lambda b{:}B(a).\ a)$$
$$:\ \forall A.\ \forall B{:}\ A\to\mathsf{Type}.\ \forall c{:}\ \Sigma'(A)(B).\ A$$
$$\mathsf{rht}' = \lambda A.\ \lambda B{:}A\to\mathsf{Type}.\ \lambda c{:}\ \Sigma'(A)(B).\ \mathsf{unpair}'(A)(B)(B(\mathsf{lft}(A)(B)(c)))(c)(\lambda a{:}A.\ \lambda b{:}B(a).\ b) \qquad \text{wrong!}$$
$$:\ \forall A.\ \forall B{:}\ A\to\mathsf{Type}.\ \forall c{:}\ \Sigma'(A)(B).\ B(\mathsf{lft}(A)(B)(c))$$

The right projection of a pair is very useful for defining the Any type and the parametric modules operators in [MacQueen 86]. Thus we have existential types as a primitive construct.

## 7. The meaning of Type

A type is, in first approximation, a *retraction* [Scott 76]. A retraction is similar to a coercion, as we can see from the following example of a boolean retraction (in the untyped λ-calculus):

$$\mathsf{Bool} = \lambda x.\ \text{if } x \text{ then true else false}$$

This retraction coerces an arbitrary object x to true or false (or diverges if x diverges). Note that booleans are not affected by this coercion, while non-booleans are mapped to booleans. A retraction maps the whole domain of values to a subdomain (called a *retract*) whose elements are all fixpoints of the retraction (e.g. Bool(true) = true).

Hence, retracting twice is the same as retracting once, for example $\mathsf{Bool}(\mathsf{Bool}(x)) = \mathsf{Bool}(x)$, which can be written $\mathsf{Bool} \circ \mathsf{Bool} = \mathsf{Bool}$. The latter is taken as the defining property of retractions:

r is a *retraction* iff $r \circ r = r$

d is a *type* iff it is a *retract* (the image of a retraction)

a has type r (written a:r), where r is a retraction, iff $r(a) = a$

It is possible to define retractions for function spaces, cartesian products, etc., and the definitions are given in the following section. Consider now the function:

$d = \lambda r.\ r \circ r$

All retracts are fixpoints of d, hence d defines the set of all retracts. Is such a set a retract? Unfortunately not, and d itself is not a retraction (it does not satisfy $d \circ d = d$). Hence the set of all (retractions determining) types is not a type.

Retractions fail to satisfy Type:Type, but not by much. If we consider particular classes of retractions, then we can achieve Type:Type. The basic idea is still valid: a type is determined by a coercion operation, which is itself a *value* which can be manipulated. In this slightly indirect sense types are values, and the indirection avoids the paradoxes connected with Type:Type.

## 8. The $\lambda\tau$-calculus

We axiomatize a general class of models of the type-free λ-calculus with pairing and retractions having the desired properties [Amadio 85]. These are called models of the $\lambda\beta\sigma\pi\tau$-calculus, or simply $\lambda\tau$-calculus. Expressions of the $\lambda\tau$-calculus have the form:

$\varepsilon$ ::=
$\iota$ |
$\tau$ |
$\lambda\iota.\ \varepsilon$ | $\varepsilon(\varepsilon)$ |
$\langle\varepsilon, \varepsilon\rangle$ | let $\iota$ , $\iota$ = $\varepsilon$ in $\varepsilon$

First we need some definitions:

$f \circ g = \lambda x.\ f(g(x))$.

$fst(p) = let\ x,y = p\ in\ x$

$snd(p) = let\ x,y = p\ in\ y$

$\Pi(A)(B) = \lambda f.\ \lambda x.\ B(A(x))(f(A(x)))$

$\Sigma(A)(B) = \lambda p.\ \langle A(fst(p)), B(A(fst(p)))(snd(p))\rangle$

$\mathbf{Y} = \lambda f.\ (\lambda x.\ f(x(x)))(\lambda x.\ f(x(x)))$

The following are the axioms:

| | | | |
|---|---|---|---|
| [β] | $(\lambda x.\ a)(b)$ | = | $a\{x \leftarrow b\}$ |
| [σ] | $let\ x,y = \langle a,b\rangle\ in\ c$ | = | $c\{x \leftarrow a,\ y \leftarrow b\}$ |
| [π] | $\langle fst(c), snd(c)\rangle$ | = | $c$ |
| [τ] | $\tau(\tau)$ | = | $\tau$ |
| [τa] | $\tau(a) \circ \tau(a)$ | = | $\tau(a)$ |
| [τΠ] | $\tau(\Pi(\tau(A))(\tau \circ B))$ | = | $\Pi(\tau(A))(\tau \circ B)$ |
| [τΣ] | $\tau(\Sigma(\tau(A))(\tau \circ B))$ | = | $\Sigma(\tau(A))(\tau \circ B)$ |
| [τ**Y**] | $\tau(A)(\mathbf{Y}(\tau(A) \circ a \circ \tau(A)))$ | = | $\mathbf{Y}(\tau(A) \circ a \circ \tau(A))$ |

From the [τ] and [τa] axioms it follows that $\tau = \tau \circ \tau$.

**Def** $a: A$ iff $A(a) = a$

The intended meaning of $\tau$ is obviously to serve as the type of all types, including itself:

**Prop** (Type formation)

$\tau: \tau$

The function space closure operation $A \rightarrow B$ takes any function $f$ and coerces its arguments to $A$ and its results to $B$, hence coercing $f$ to $A \rightarrow B$:

**Def** $A \rightarrow B = \lambda f.\ B \circ f \circ A$

**Prop**

$A: \tau,\ B: \tau \quad \Rightarrow \quad A \rightarrow B: \tau$

The $\rightarrow$ operator is a special case of the dependent product operator $\Pi$:

**Prop** ($\Pi$ formation)

$$A{:}\ \tau,\ B{:}\ A\to\tau \quad \Rightarrow \quad \Pi(A)(B){:}\ \tau$$

The cartesian product of two types $A\times B$ takes any pair $p$ and coerces its left component to $A$ and its right component to $B$, hence coercing $p$ to $A\times B$:

**Def** $\quad A \times B \ = \ \lambda p.\ \langle A(fst(p)),B(snd(p))\rangle$

**Prop**

$$A{:}\ \tau,\ B{:}\ \tau \quad \Rightarrow \quad A \times B{:}\ \tau$$

Cartesian products can be generalized to dependent sums $\Sigma$:

**Prop** ($\Sigma$ formation)

$$A{:}\ \tau,\ B{:}\ A\to \tau \quad \Rightarrow \quad \Sigma(A)(B){:}\ \tau$$

The fixpoint operator satisfies the property:

**Prop** (**Y** formation)

$$A{:}\ \tau,\ a{:}\ A\to A \quad \Rightarrow \quad \mathbf{Y}(a){:}\ A$$

and it can be used for recursive type definitions:

**Corollary**

$$A{:}\ \tau\to\tau \quad \Rightarrow \quad \mathbf{Y}(A){:}\ \tau$$

Several models of the $\lambda\tau$-calculus are known [Scott 76] [McCracken 86] [Amadio 85] .

# 9. Semantics

The denotational semantics of our calculus can now be given easily in any model M of the $\lambda\tau$-calculus. Here $\rho$ is an environment mapping variables to elements of M, and $V$ is a function mapping terms and environments to elements of M. The environment $\rho\{x \leftarrow v\}$ is the same as $\rho$, except that $x$ is associated to $v$; $\rho(x)$ is the element associated to $x$ in $\rho$.

$$V[\![x]\!]\,\rho \quad = \quad \rho(x)$$

$$V[\![Type]\!]\,\rho \quad = \quad \tau$$

$$
\begin{array}{lll}
V[\![\lambda x\colon a.\ b]\!]\,\rho & = & \lambda v.\ V[\![b]\!]\,\rho\{x \leftarrow (V[\![a]\!]\,\rho)(v)\} \\
V[\![a(b)]\!]\,\rho & = & (V[\![a]\!]\,\rho)(V[\![b]\!]\,\rho) \\
V[\![\forall x{:}A.\ B]\!]\,\rho & = & \Pi(V[\![A]\!]\,\rho)(\lambda v.\ V[\![B]\!]\,\rho\{x \leftarrow v\}) \\
V[\![<a,b>]\!]\,\rho & = & <\ V[\![a]\!]\,\rho,\ V[\![b]\!]\,\rho > \\
V[\![\text{let } x,y = a \text{ in } b]\!]\,\rho & = & V[\![b]\!]\,\rho\{x \leftarrow \mathrm{fst}(V[\![a]\!]\,\rho),\ y \leftarrow \mathrm{snd}(V[\![a]\!]\,\rho)\} \\
V[\![\exists x{:}A.\ B]\!]\,\rho & = & \Sigma\,(V[\![A]\!]\,\rho)(\lambda v.\ V[\![B]\!]\,\rho\{x \leftarrow v\}) \\
V[\![\mu x{:}A.\ a]\!]\,\rho & = & \mathbf{Y}(V[\![\lambda x{:}A.\ a]\!]\,\rho)
\end{array}
$$

To show that the semantics is in some sense correct, we want to relate it to the type inference system. In fact we are only interested in the semantics of well-typed terms, while $V$ gives semantics to all terms. The main result is hence a soundness theorem stating that the semantics of a well-typed term is semantically related to the semantics of its inferred type.

First, we prove a set of propositions which are the semantic versions of the introduction and elimination type inference rules:

**Prop** (Π introduction)

$$A\colon \tau,\ B\colon A \rightarrow \tau,\ (\forall x.\ x{:}A \Rightarrow b(x){:}B(x)) \quad \Rightarrow \quad \lambda x.\ b(A(x)) : \Pi(A)(B)$$

**Prop** (Π elimination)

$$A\colon \tau,\ B\colon A \rightarrow \tau,\ a\colon A,\ b\colon \Pi(A)(B) \quad \Rightarrow \quad b(a) : B(a)$$

**Prop** (Σ introduction)

$$A\colon \tau,\ B\colon A \rightarrow \tau,\ a\colon A,\ b\colon B(a) \quad \Rightarrow \quad <a,b>\colon \Sigma(A)(B)$$

**Prop** (Σ elimination)

$$
\begin{array}{l}
A\colon \tau,\ B\colon A \rightarrow \tau,\ C\colon \Sigma(A)(B) \rightarrow \tau, \\
c\colon \Sigma(A)(B),\ \ (\forall x.\ \forall y.\ \ x{:}A\ ,\ y{:}B(x) \Rightarrow d(x)(y){:}C(<x,y>)) \\
\qquad \Rightarrow \quad \text{let } x,y = c \text{ in } d(x)(y) : C(c)
\end{array}
$$

**Lemma** (substitution)

$$V[\![b\{x \leftarrow a\}]\!]\,\rho = V[\![b]\!]\,\rho\{x \leftarrow V[\![a]\!]\,\rho\}$$

**Def**

$\Upsilon$ is *type compatible* with $\rho$ if for all $x$ in the domain of $\Upsilon$, $\Upsilon(x) = A$ implies $\rho[\![x]\!] : V[\![A]\!]\,\rho'$, where $\rho' \subseteq \rho$ and the domain of $\rho'$ includes all the free variables of $A$.

**Theorem** (semantic soundness)

If $\Upsilon$ is type compatible with $\rho$, then:

(i) $\Upsilon \vdash a{:}A \quad \Rightarrow \quad V[\![a]\!]\rho : V[\![A]\!]\rho$

(ii) $\Upsilon \vdash b \leftrightarrow c \quad \Rightarrow \quad V[\![b]\!]\rho = V[\![c]\!]\rho$

Semantic soundness is normally stated as (i) alone. However, type inference is mixed here with reduction, and as a part of showing (i) one must show that reductions are well behaved; moreover, one needs (i) in proving (ii).

A deeper reason for including (ii) in the statement of the theorem (as opposed to having it as a separate theorem) is the following. In a retraction semantics the conclusion of (i) may be true even if the premise is false: for example, $a = (\lambda x{:}Bool.\ x)(3) : Bool$, as the Bool retraction maps 3 to a boolean. Hence (i) alone leaves open the possibility that the type inference system is somehow "incorrectly" defined so that a is well typed and (i) still holds. But, in the example, a reduces to 3, whose denotation (an integer) is generally different from a's (a boolean). Hence (ii) does not hold for such an incorrect type system. This inadequacy of (i) does not arise in Milner's definition of soundness for a semantics not based on retractions [Milner 78]; hence (ii) is incorporated to make the "semantic soundness theorem" closer to its original significance.

# 10. Expressing other type systems

Our calculus can be regarded as an ω-order typed λ-calculus; hence it is very easy to encode the second-order typed λ-calculus, which in turn can be used as a foundation for the type systems of Russell and ML [Milner 84]. Using Reynolds' notation:

| | | | |
|---|---|---|---|
| $\lambda a{:}A.\ b$ | $\cong$ | $\lambda a{:}A.\ b$ | (value abstraction) |
| $a(b)$ | $\cong$ | $a(b)$ | (value application) |
| $\Lambda A.\ a$ | $\cong$ | $\lambda A{:}Type.\ a$ | (type abstraction) |
| $a[A]$ | $\cong$ | $a(A)$ | (type application) |
| $\rightarrow$ | $\cong$ | $\rightarrow$ | (function types) |
| $\Delta A.B$ | $\cong$ | $\forall A{:}Type.\ B$ | (polymorphic types) |

The language used in the ideal model of types [MacQueen 84a], which is the basis for the Standard ML modules and type system [MacQueen 84b], can also be easily expressed:

| | | | |
|---|---|---|---|
| $\lambda a{:}A.\ b$ | $\cong$ | $\lambda a{:}A.\ b$ | (value abstraction) |
| $a(b)$ | $\cong$ | $a(b)$ | (value application) |
| $\rightarrow$ | $\cong$ | $\rightarrow$ | (function types) |
| $\times$ | $\cong$ | $\times$ | (product types) |
| $+$ | $\cong$ | $+$ | (union types) |

| | | | |
|---|---|---|---|
| $\forall$A.B | $\cong$ | $\forall$A:Type. B | (universal types) |
| $\exists$A.B | $\cong$ | $\exists$A:Type. B | (existential types) |
| $\to$ | $\cong$ | $\to$ | (function kinds) |
| $\times$ | $\cong$ | $\times$ | (product kinds) |

where the structure of *kinds* is pushed down at the type level.

The Pebble type system is not as easy to encode, due to notation peculiarities, although it is clear that there are the following rough correspondences:

| | | | |
|---|---|---|---|
| type | $\leftrightarrow$ | Type | (the type of all types) |
| $\twoheadrightarrow$ | $\leftrightarrow$ | $\forall$ | (dependent function types) |
| $\times\times$ | $\leftrightarrow$ | $\exists$ | (dependent product types) |

Moreover, existential types can be used to simulate Pebble's *bindings* and *declarations.* A binding is an association of values to variables, and a declaration is an association of types to variables. The type of a binding is a declaration. Each variable in a binding or declaration can be used in the definition of the following variables:

[A:type~int, a:A~3, b:int~a+1]: (A:type) $\times\times$ (a:A) $\times\times$ (b:int)

Examples will illustrate the translation from bindings to typed terms in our calculus. Bindings are translated to nested pairs, and the variable names are lost in the process; hence we perform substitutions if a variable is used in later bindings. Declarations are translated to existential types, and the variable names are retained.

| | | |
|---|---|---|
| nil : void | $\cong$ | unity : Unit |
| [a:int~3] : (a:int) | $\cong$ | <3,unity> : $\exists$a:Int. Unit |
| [a:int~3, b:int~a+1] : (a:int) $\times\times$ (b:int) | $\cong$ | <3,<3+1,unity>> : $\exists$a:Int. $\exists$b:Int. Unit |
| [A:type~int, a:A~3] : (A:type) $\times\times$ (a:A) | $\cong$ | <Int,<3,unity>> : $\exists$A. $\exists$a:A. Unit |

A binding b can be opened in a scope by a let-in construct. The binding b can be a parameter, and hence unknown, but its type is sufficient to carry out the translation:

let b : (A:type) $\times\times$ (a:A) in ...A...a... $\cong$ ($\lambda$A:Type. $\lambda$a:A. ...A...a... ) ($b_1$) ($b_2$)

where $b_1$ and $b_2$ are the appropriate terms (defined by rip) selecting the first and second components of b.

Finally, the relations with the theory of constructions [Coquand 85a] are very interesting, and are investigated in [Amadio 86].

## 11. Conclusions

Intuitionistic type theory provides powerful proof systems, based on the proposition-as-type (proof-as-program) isomorphism, which are very promising for program verification. Such systems work only under the assumption that programs always terminate, since proofs must terminate. Hence, general recursion and unbounded iteration are initially banned, as well as the Type:Type property which leads to logic inconsistencies in the form of non-terminating proofs.

While it is true that most ordinary programs are total, at the current state of knowledge it is unthinkable to ask programmers to constrain themselves to bounded iterations. Moreover, computer systems *require* possibly divergent programs for routine functioning, hence total programming languages cannot cover interesting aspects of programming. The retrofitting of recursion into type theory is being actively pursued [Backhouse 84, Constable 83, Constable 84, Paulson 84] and these problems may be solved in the future.

If our position is instead to admit divergent computations from the start, then the proposition-as-type paradigm can be recast as a powerful type system (no longer a logic) which is not incompatible with the Type:Type property. Although this position is conceptually divergent from intuitionistic type theory, it is a natural extension of work on programming language type systems, and adds to the impression that logic, program verification, and type systems for practical languages are on a collision course.

## Acknowledgements

Alan Demers and Albert Meyer pointed out mistakes and misconceptions in early drafts.

## References


[Amadio 85] R.Amadio, K.B.Bruce, G.Longo: **The finitary projection model for second order lambda calculus and solutions to higher order domain equations**, Report S12/85, Dipartimento di Informatica, Università di Pisa, 1985.

[Amadio 86] R.Amadio, G.Longo: **A type-free look at types as parameters**, to appear.

[Backhouse 84] R.Backhouse, A-A. Khamiss: **The while-rule in Martin-Löf's theory of types**, University of Essex, Dept of Computer Science, Technical Report CSM-71, 1984.

[Barendregt 83] H.Barendregt, A.Rezus: **Semantics for classical Automath and related systems**, *Information and control*, 59, 127-147, 1983.

[Boehm 80] H.Boehm, A.Demers, J.Donahue: **An informal description of Russell**, Technical Report, Computer Science Dept, Cornell Univ. 1980.

[Bruce 84] K.B.Bruce, R.Meyer: **The semantics of second order polymorphic lambda calculus**, in *Semantics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Burstall 84] R.Burstall, B.Lampson, **A kernel language for abstract data types and modules**, in *Sematics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, 1984.

[Cardelli 85] L.Cardelli, P.Wegner: **On understanding types, data abstraction and polymorphism**, Technical Report No. CS-85-14, Brown University.

[Constable 83] R.L.Constable: **Partial functions in constructive formal theories**, *Proc. of the 6th G.I. conference*, Lecture Notes in Computer Science No. 135, Springer-Verlag, 1983.

[Constable 84] R.L.Constable, D.R.Zlatin: **The type theory of PL/CV3**, *ACM TOPLAS* 6(1), pp 94-112, January 1984.

[Coquand 85a] T.Coquand: **Une théorie des constructions**, Thèse présentée à l'Université Paris VII pour obtenir le Diplôme de Docteur de 3ème cycle.

[Coquand 85b] T.Coquand, G.Huet: **Constructions: a higher order proof system for mechanizing mathematics**, Technical Report 401, INRIA, May 1985.

[de Bruijn 80] N.G.de Bruijn: **A survey of the project Automath**, in *Essays on combinatory logic, lambda calculus and formalism*, pp. 589-606, J.R.Hindley and J.P.Seldin ed., Academic Press, 1980.

[Donahue 85] J.Donahue, A.Demers: **Data types are values**, ACM TOPLAS, 7(3), pp. 426-445, July 1985.

[Fortune 83] S.Fortune, D.Leivant, M.O'Donnell: **The expressiveness of simple and second-order type structures**, *Journal of the ACM* 30(1), pp. 151-185, January 1983.

[Girard 71] J-Y.Girard: **Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types**, *Proceedings of the second Scandinavian logic symposium*, J.E.Fenstad Ed. pp. 63-92, North-Holland, 1971.

[Girard 72] J-Y.Girard: **Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure**, Thèse de doctorat d'Etat, University of Paris, 1972.

[Hook 84] J.G.Hook: **Understanding Russell, a first attempt**, in *Semantics of Data Types*, Lecture Notes in Computer Science 173, pp. 51-67, Springer-Verlag, 1984.

[MacQueen 84a] D.B.MacQueen, R.Sethi, G.D.Plotkin: **An ideal model for recursive polymorphic types**, *Proc. POPL 1984*.

[MacQueen 84b] D.B.MacQueen: **Modules for Standard ML**, *Proc. Symposium on Lisp and Functional Programming*, 1984.

[MacQueen 86] D.B.MacQueen: **Using dependent types to express modular structure** *Proc. POPL*, 1986.

[Martin-Löf 71] P.Martin-Löf, **A theory of types**, Report 71-3, Dept of Mathematics, University of Stockholm, February 1971, revised October 1971.

[Martin-Löf 80] P.Martin-Löf, **Intuitionistic type theory**, Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980.

[McCracken 79] N.McCracken: **An investigation of a programming language with a polymorphic type structure**, Ph.D. Thesis, Syracuse University, June 1979.

[McCracken 86] N.McCracken: **A finitary retract model for the polymorphic lambda- calculus**, to appear in *Information and Control*.

[Meyer 86] A.R.Meyer, M.B.Reinhold: **'Type' is not a type**, *Proc. POPL 1986.*

[Milner 78] R.Milner: **A theory of type polymorphism in programming**, *Journal of Computer and System Science 17*, pp. 348-375, 1978.

[Milner 84] R.Milner: **A proposal for Standard ML**, *Proc. of the 1984 ACM Symposium on Lisp and Functional Programming*, Aug 1984.

[Mitchell 85] J.C.Mitchell, G.D.Plotkin: **Abstract types have existential type**, *Proc. POPL 1985*.

[Paulson 84] L.C.Paulson: **Constructing recursion operators in intuitionistic type theory**, University of Cambridge, Computer Laboratory, Technical Report No. 57, 1984.

[Rezus 85] A.Rezus: **Semantics of constructive type theory**, Report No. 70, Informatics Department, Nijmegen University, The Netherlands, Sep 1985.

[Reynolds 74] J.C.Reynolds: **Towards a theory of type structure**, in *Colloquium sur la programmation* pp. 408-423, Springer-Verlag, Lecture Notes in Computer Science, n.19, 1974.

[Scott 70] D.Scott: **Constructive validity**, *Symposium on Automatic Demonstration*, Lecture Notes in Mathematics No. 125, pp. 237-275, Springer-Verlag, 1970.

[Scott 76] D.Scott: **Data types as lattices**, *SIAM Journal of Computing*, 4, 1976.