Introduction to Simply Typed Lambda Calculus with Subtyping

Adriana Compagnoni
abc@@dcs.ed.ac.uk

Department of Computer Science, University of Edinburgh
The King's Buildings, Edinburgh, EH9 3JZ, United Kingdom
Tel: (+44) (131) 650-1000 Fax: (+44) (131) 667-7209

1 Introduction

1.1 Types and programs

One of the basic ideas in programming is the notion of algorithm. An algorithm is a description of the rules one must follow to accomplish a task. But for a machine to be able to perform such a task, this description must be expressed in a formal language, and in particular programming languages serve this purpose.

Alan Turing introduced a formal language for describing computable functions, now called Turing machines, from which imperative programming arose. The lambda calculus was invented by Alonso Church to define the notion of computable functions [Chu36]. Since then a variety of lambda calculi have been defined and used in the study of programming languages. Lambda calculi can be seen as simple programming languages, since they are formal and describe computations. In that sense, a program is a term of the lambda calculus. From a different perspective, lambda calculi constitute metalanguages for analyzing other programming languages. Although λ-calculi are particularly well-suited to studying functional programming languages, they have also been used to study imperative programming disciplines [Lan65].

Types are an important tool in programming languages and logic which serve to classify terms according to basic properties, such as being a number or being a function. For example, if we think of the integer number 5, the term is 5 and its type is integer. The addition function has a type expressing that it takes two integer numbers as arguments and returns an integer number as result, which we write as follows.

$$+ : (\mathrm{Int} \times \mathrm{Int}) \to \mathrm{Int}$$

One immediate advantage of types is that nonsensical expressions can be considered *illegal*. For example, the expression

$$3 + \text{“good morning”}$$

will not be part of the language because “good morning” is not an integer number, but a string of characters. Although this is a rather coarse example, this sort of mistake is very frequent in the development of programs.

Further in that direction, AUTOMATH [dB80], Martin-Löf Type Theory [Mar73], Coquand and Huet's Calculus of Constructions [CH88], and Luo's Extended Calculus of Constructions [Luo90] are rich type systems in which a type can not only prevent the formation of nonsensical expressions but can also state properties of terms. For example, in the case of a term corresponding to a sorting algorithm for lists, the type can express the fact that the output is an ordered list.

Type structures help organizing ideas and structuring programs in such a way that disciplines of programming and type systems walk hand in hand. The development of ideas about programming motivates the design of type systems that encourage programming in a particular style. Ideally, one would like to have tailor-made type systems for each particular style of programming, so that bad programming style results in illegal terms.

We refer the reader to [Bar90, Mit90b] for a more detailed analysis of the relation between lambda calculi and programming languages.

But types also have drawbacks. The programmer may need to provide type annotations. ML gives a nice solution to this problem for functional programming. Types can also be too restrictive. Imagine that the variable x is defined having type *Nat* (for natural numbers). Then

$$3 + x$$

generates a compile time error, because + expected two integers as arguments. An elegant solution to this problem is to use subtyping.

1.2 Subtyping

Subtyping is a primitive relation that uniformly captures concepts from diverse areas of computer science. If S and T are sets, then $S \leq T$ (*S is a subtype of T*) means that elements of S are also elements of T. If S and T are specifications, then elements satisfying specification S also satisfy T. In object-oriented programming, if S and T are object descriptions, then $S \leq T$ states that where an object with interface T is expected, it is safe to use an object with interface S. When S and T are module interfaces in a software system, an implementation of S is also an implementation of T. If S and T are theorems, then a proof of S is also a proof of T. Understanding the essence, subtleties, and general properties of subtyping illuminates a wide area.

The idea of subtypes appears quite naturally in programming languages. If we think of types as sets, we can easily picture what a subtype could be. Informally, we can say that a type S is a subtype of T if any element of S can be seen as an element of T. We say *can be seen as* and not directly *is* because the act of considering an element of type S as an element of type T might hide some transformation. Consider for example the types *Int* and *real* of integers and real numbers respectively. Usually, on a computer, integers are represented in a different way than real numbers are; even if we might think of the integers as a subset of the real numbers, there is a translation to be performed. The act of considering an element of type S as an element of type T will be called *coercion*. In other words, we say that an element of type S is coerced into an element of type T. Somehow an element of type S has enough information to be seen as an element of type T.

The archetypal rule for subtyping is:

$$\frac{M : S \qquad S \leq T}{M : T} \qquad \text{(SUBSUMPTION)}$$

Note that the coercion from S to T is left implicit. An implicit coercion is motivated by the fact that the same term can be considered as belonging to different types.

Explicit coercions have been used as a way of giving semantics to systems with implicit coercions in [CG92]. In [CL91], PER models for Quest, a higher order lambda calculus with subsumption, and Quest_C, a higher order lambda calculus with coercion, are studied.

One of the first applications of subtyping in λ-calculi was modeling the refinement of interfaces in object-oriented languages [Car88]. The formal subtype relation $S \leq T$ models the assertion that the objects in some collection S provide more services than those in T, so that it is safe to use a member of S in any context where a member of T is expected.

1.3 Type reconstruction and type checking

We can say that if a term has a type, it is, to a certain extent, correct. This correctness can be as simple as guaranteeing that computations will not fail by mismatch of the expected argument of a function, for example, and as elaborate as ensuring that certain specification or safety property is satisfied. With these ideas in mind, a first question one may ask is whether a given expression e is legal or correct, which in our framework means whether there exists a type T such that $e : T$. This problem is traditionally called *type reconstruction* or *type inference* depending on whether there is explicit type information in the terms. In this notes we will consider type reconstruction. A related question, given a term e and a type T, is whether $e : T$, known as *type checking.* In the presence of subtyping both problems, type checking and type reconstruction, become more complicated because typing is defined in terms of subtyping. It is clear in the SUBSUMPTION rule that in order to answer a question of the form $e : T$ we should be able to answer questions of the form $S \leq T$. This shows that at the heart of the decidability of typing lies the question of whether the subtyping relation is decidable. In a system with the SUBSUMPTION rule, each term may be assigned more than one type. Then to answer the type checking and type reconstruction questions we need a way to identify all possible types: for example, by finding some kind of representative of the types of each term. One plausible candidate is a *minimal type* with respect to the subtyping relation. Then type reconstruction consists of finding a minimal type, and type checking whether $e : T$ consists of finding a minimal type S such that $e : S$ and checking if $S \leq T$. Without the minimal type property, type checking becomes algorithmically intractable. Imagine that instead of having just one representative we have a (finite) set of them, say $S_1, .., S_n$ such that for each type T of e there exists j such that $S_j \leq T$. Imagine that e is an application, say $e_1\, e_2$, then, to find the set of representative types of e, we need to match each representative of e_1 against each representative of e_2, which produces a combinatorial explosion.

1.4 Background

The formal study of subtyping in programming languages was begun by Reynolds [Rey80] and Cardelli [Car88], who used a lambda-calculus with subtyping to model the refinement of interfaces in object oriented languages. This led to a considerable body of work, covering an increasing range of object-oriented features by combining subtyping with other type-theoretic constructs, including polymorphic functions [CW85, CG92, BCGS91]; records with update and extension operators [Car88, CM91]; recursive types [AC93, BM92], and higher-order polymorphism [Car90, Mit90a, Com95, SP94, CG97].

Type systems with subtyping have also arisen from the study of lambda-calculi with intersection types at the University of Torino [CD80, BCD83]. Most of this work has been carried out in the setting of pure lambda-calculi, but it has also been applied to programming language design by Reynolds [Rey88]. Some work has begun on combining intersections with other typing features [Pie91, CDdL93].

The contribution of this thesis is to weave together these two threads by combining higher-order subtyping, which forms the cornerstone of several recent models of typed object-oriented programming [CHC90, Bru94, PT94], with intersection types, leading to an extended object model with multiple inheritance [CP96].

CONVENTION 1.1 Throughout this chapter the metavariables X , Y and Z will range over type variables, S, T, and U will range over types, M, N, and P will range over terms, x, y, and z over term variables, Γ will range over contexts.

We start by reviewing the untyped λ-calculus and the Church-Rosser property.

2 Untyped λ-calculus

This section is based on [Bar92, Tak95]. The syntax of the λ-calculus is defined by the following grammar.

$$\Lambda ::= V \mid (\lambda V.\Lambda) \mid (\Lambda\Lambda)$$

where V is a denumerable set of variables.

Parenthesis are often omitted following the convention that

1. $MN_1 \cdots M_n$ stands for $(\cdots(MN_1)\cdots M_n)$.

2. $\lambda x_1 \cdots x_n.M$ stands for $(\lambda x_1(\lambda x_2(\cdots(\lambda x_n(M))\cdots)))$.

3. outermost parenthesis are not written.

DEFINITION 2.1 (*Free and bound variables*)

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(MN) & = & \mathrm{FV}(M) \cup \mathrm{FV}(N) \\ \mathrm{FV}(\lambda x.M) & = & \mathrm{FV}(M) - \{x\} \end{array}$$

x is *bound* in M if it is not free. Terms that only differ in the names of bound variables are called α-equivalent. ($M \equiv N$).

DEFINITION 2.2 (*Substitution*) The result of substituting N for (the free occurrences of) x in M is defined as follows.

$$\begin{array}{rcll} x[x{\leftarrow}N] & \equiv & N & \\ y[x{\leftarrow}N] & \equiv & y & \\ ML[x{\leftarrow}N] & \equiv & (M[x{\leftarrow}N])(L[x{\leftarrow}N]) & \\ \lambda y.M[x{\leftarrow}N] & \equiv & \lambda y.(M[x{\leftarrow}N]) & \text{if } y \notin \mathrm{FV}(N) \\ \lambda y.M[x{\leftarrow}N] & \equiv & \lambda z.(M[y{\leftarrow}z])[x{\leftarrow}N]) & \text{if } y \in \mathrm{FV}(N) \text{ and } z \notin \mathrm{FV}(N) \cup \mathrm{FV}(M) \\ \lambda x.M[x{\leftarrow}N] & \equiv & \lambda x.M & \end{array}$$

We now present the λ-calculus as a formal theory of equations between λ-terms.

DEFINITION 2.3 (*Equational Theory*)

$$\begin{array}{rcll} (\lambda x.M)N & = & M[x{\leftarrow}N] & \beta \\ M & = & M & \\ M = N & \Longrightarrow & N = M & \\ M = N, N = L & \Longrightarrow & M = L & \\ M = N & \Longrightarrow & ML = NL & \\ M = N & \Longrightarrow & LM = LN & \\ M = N & \Longrightarrow & \lambda x.M = \lambda x.N & \end{array}$$

The formal way to identify terms that only differ on their bound variables is adding the following rule called α.

$$\lambda x.M \quad = \quad \lambda y.M[x{\leftarrow}y] \qquad \text{if } y \notin \mathrm{FV}(M).$$

We write $\lambda \vdash M = N$ when $M = N$ is derived using these rules.

EXERCISE 2.4 Let $w \equiv \lambda x.xx$ and $l \equiv \lambda xy.xy$. Prove using the rules above that $wl = \lambda zz'.zz'$

2.1 Reduction

Reduction is useful to study the formal theory of equations between λ-terms. The Church-Rosser Theorem says that if two terms are convertible then there is a term to which both reduce.

DEFINITION 2.5 1. A binary relation R on Λ is called *compatible* (w.r.t. operations) if

$$\begin{array}{rcl} M\,R\,N & \text{implies} & (ZM)\,R\,(ZN), \\ & & (MZ)\,R\,(NZ), \text{ and} \\ & & (\lambda x.M)\,R\,(\lambda x.N). \end{array}$$

2. A *congruence* relation on Λ is a compatible equivalence relation.

3. A *reduction* relation on Λ is a compatible, reflexive and transitive relation.

Definition 2.6 The binary relations $\to_\beta$, $\twoheadrightarrow_\beta$, and $=_\beta$ are defined inductively as follows.

1. (a) $(\lambda x.M)N \to_\beta M[x\leftarrow N]$
   (b) If $M \to_\beta N$ then $ZM \to_\beta ZN$
   (c) If $M \to_\beta N$ then $MZ \to_\beta NZ$
   (d) If $M \to_\beta N$ then $\lambda x.M \to_\beta \lambda x.N$
2. (a) If $M \to_\beta N$ then $M \twoheadrightarrow_\beta N$.
   (b) $M \twoheadrightarrow_\beta M$.
   (c) If $M \twoheadrightarrow_\beta N$ and $N \twoheadrightarrow_\beta L$ then $M \twoheadrightarrow_\beta L$.
3. (a) If $M \twoheadrightarrow_\beta N$ then $M =_\beta N$.
   (b) If $M =_\beta N$ then $N =_\beta M$.
   (c) If $M =_\beta N$ and $N =_\beta L$ then $M =_\beta L$.

By definition $\to_\beta$ is compatible; $\twoheadrightarrow_\beta$ is the reflexive and transitive closure of $\to_\beta$ and therefore it is a reduction relation; the relation $=_\beta$ is a congruence relation.

We can now relate the equational theory and the reduction of the λ-calculus.

Proposition 2.7 $M =_\beta N$ if and only if $\lambda \vdash M = N$

Exercise 2.8 Prove Proposition 2.7.

Definition 2.9 A *β-redex* is a term of the form $(\lambda x.M)N$ and $M[x\leftarrow N]$ is its *contractum*. A λ-term M is in *β-normal form* (β-nf) if it does not have a β-redex as subexpression. A term M has a β-normal form if $M =_\beta N$ and N is a β-nf, for some N.

An immediate property of nf's is the following.

Lemma 2.10 1. If M is a β-nf, then $M \twoheadrightarrow_\beta N$ implies $M \equiv N$.

2. If $M \to_\beta M'$ then $M[x\leftarrow N] \to_\beta M'[x\leftarrow N]$.

Exercise 2.11 Find M and N such that $N \to_\beta N'$, but it is not the case that $M[x\leftarrow N] \to_\beta M[x\leftarrow N']$.

2.2 The Church-Rosser Theorem

The most elegant proof of the Church-Rosser property for the λ-calculus published to date is a refinement of the proof of Tait and Martin-Löf using parallel reductions produced by Takahashi[Tak95].

Definition 2.12 1. $x \Rightarrow x$.

2. If $M \Rightarrow N$ then $\lambda x.M \Rightarrow \lambda x.N$.

3. If $M \Rightarrow M'$ and $N \Rightarrow N'$ then $MN \Rightarrow M'N'$.

4. If $M \Rightarrow M'$ and $N \Rightarrow N'$ then $(\lambda x.M)N \Rightarrow M'[x\leftarrow N]'$.
   Curry called this reduction *Minimal Complete Development*.

Lemma 2.13 1. If $M \to_\beta N$ then $M \Rightarrow N$.

2. If $M \Rightarrow N$ then $M \twoheadrightarrow_\beta N$.

3. If $M \Rightarrow M'$ and $N \Rightarrow N'$ then $M[x \leftarrow N] \Rightarrow M'[x \leftarrow N']$.

Compare Lemma 2.13 part 3 with exercise 2.11. Note that $\Rightarrow$ satisfies a substitution property that $\rightarrow_\beta$ does not satisfy.

NOTATION 2.14 We write $R^\star$ to denote the transitive closure of R.

EXERCISE 2.15
1. Prove that $\twoheadrightarrow_\beta$ is the transitive closure of $\Rightarrow$. In other words, $M \Rightarrow^\star N$ if and only if $M \twoheadrightarrow_\beta N$.
2. Prove that $\Rightarrow$ is reflexive.

DEFINITION 2.16 (*diamond property*)
1. A binary relation $\rhd$ on Λ satisfies the *diamond property* if for all M, M_1 and M_2 if $M \rhd M_1$ and $M \rhd M_2$ then there exists M_3 such that $M_1 \rhd M_3$ and $M_2 \rhd M_3$.
2. A notion of reduction is said to be Church-Rosser if its reflexive and transitive closure satisfies the diamond property.

To prove that $\rightarrow_\beta$ is Church-Rosser we need to prove that $\twoheadrightarrow_\beta$ satisfies the diamond property.

LEMMA 2.17 If $\rhd$ satisfies the diamond property then $\rhd^\star$ satisfies the diamond property.

Hence to establish that $\rightarrow_\beta$ is Church-Rosser it is enough to prove that $\rhd$ satisfies the diamond property.

THEOREM 2.18 (*Church-Rosser*) For all M, M_1 and M_2 if $M \twoheadrightarrow_\beta M_1$ and $M \twoheadrightarrow_\beta M_2$ then there exists M_3 such that $M_1 \twoheadrightarrow_\beta M_3$ and $M_2 \twoheadrightarrow_\beta M_3$.

PROOF: By Takahashi[Tak95].

We continue by reviewing the simply typed λ-calculus and its properties.

3 Simply typed λ-calculus

DEFINITION 3.1 The typed λ-calculus $\lambda\!\rightarrow$, is defined as follows.

1. The set of types $\mathbb{T} = \mathrm{Type}(\lambda\!\rightarrow)$ is defined by

$$\mathbb{T} ::= \mathbb{V} \mid \mathbb{T} \rightarrow \mathbb{T},$$

where $\mathbb{V}$ is a denumerable set of (type) variables.

2. The set of pseudo-terms $\Lambda = \Lambda(\lambda\!\rightarrow)$ is defined as follows.

$$\Lambda ::= V \mid \lambda V{:}\mathbb{T}.\Lambda \mid \Lambda\Lambda,$$

where V is a denumerable set of (term) variables.

3. A *statement* is of the form $M : S$ (M is of type S) with $M \in \Lambda$ and $S \in \mathbb{T}$. The term M is called the *subject* of the statement. A typing assumption is an expression of the form $x{:}S$. The variable x is called the subject of the typing assumption.

4. A *context* is a set of typing assumptions with distinct variables as subjects.

Reduction is defined as in the untyped case, the only difference being that the β-reduction rule is.

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

DEFINITION 3.2 (*Typing rules*)

$$\Gamma \vdash_{\lambda\rightarrow} x : S, \quad \text{if } x{:}S \in \Gamma \tag{T-Var}$$

$$\frac{\Gamma, x{:}S \vdash_{\lambda\rightarrow} M : T}{\Gamma \vdash_{\lambda\rightarrow} \lambda x{:}S.M : S \rightarrow T} \tag{T-Abs}$$

$$\frac{\Gamma \vdash_{\lambda\rightarrow} M : S \rightarrow T \qquad \Gamma \vdash_{\lambda\rightarrow} N : S}{\Gamma \vdash_{\lambda\rightarrow} MN : T} \tag{T-App}$$

3.1 Properties of $\lambda\!\rightarrow$

Let $FV(M)$ denote, as usual, the set of free variables of M and $Dom(\Gamma)$ the domain of Γ, i.e. the set of subjects of Γ.

LEMMA 3.3 (*Free Variable*)
If $\Gamma \vdash_{\lambda\rightarrow} M : S$, then $FV(M) \subset Dom(\Gamma)$.

If one tries to prove Weakening directly by induction on derivations the induction hypothesis is too weak in the case of T-ABS.This problem was identified by McKinna and Pollack. We use their idea of *renaming* [MP93].

DEFINITION 3.4 Let Γ and Δ be contexts. A *renaming* for Γ in Δ is a parallel substitution γ from variables to variables such that for every $x{:}A$ in Γ, $\gamma(x){:}A$ is in Δ. We write $J[\gamma]$ for the result of performing the substitution γ in J. The renaming $\gamma\{x \mapsto y\}$ maps x to y and behaves like γ elsewhere.

LEMMA 3.5 (*Renaming*) If $\Gamma \vdash_{\lambda\rightarrow} M : S$ and γ is a renaming for Γ in Δ then $\Delta \vdash_{\lambda\rightarrow} M[\gamma] : S$.

Weakening now follows as a corollary of renaming taking γ to be the identity substitution.

COROLLARY 3.6 (*Weakening for* $\lambda\!\rightarrow$) Let Γ and Γ' be contexts such that $\Gamma \subseteq \Gamma'$. Then if $\Gamma \vdash_{\lambda\rightarrow} M : S$, then $\Gamma' \vdash_{\lambda\rightarrow} M : S$.

PROPOSITION 3.7

1. (Generation for $\lambda\!\rightarrow$)
   (a) If $\Gamma \vdash_{\lambda\rightarrow} x : S$, then $x{:}S \in \Gamma$.
   (b) If $\Gamma \vdash_{\lambda\rightarrow} MN : S$, then there exists T such that $\Gamma \vdash_{\lambda\rightarrow} M : T \rightarrow S$ and $\Gamma \vdash_{\lambda\rightarrow} N : T$.
   (c) If $\Gamma \vdash_{\lambda\rightarrow} \lambda x{:}S.M : U$, then there exists T such that $U \equiv S \rightarrow T$ and $\Gamma, x{:}S \vdash_{\lambda\rightarrow} M : T$.
2. (Uniqueness of types for $\lambda\!\rightarrow$) If $\Gamma \vdash_{\lambda\rightarrow} M : S$ and $\Gamma \vdash_{\lambda\rightarrow} M : S'$, then $S \equiv S'$.

Since the system is deterministic the following result is obvious. There is only one rule matching the subject in each case.

A term M is *strongly normalizing*, if all reductions starting from M terminate. In this framework we mean β-reduction.

PROPOSITION 3.8 (*Strong normalization for* $\lambda\!\rightarrow$) If $\Gamma \vdash_{\lambda\rightarrow} M : S$, then M is strongly normalizing.

4 Subtyping

4.1 The relation $\leqslant_C$

In the present section we define the subtyping relation, $\leqslant_C$, and an algorithm, *Subtype*, to check whether two types are in the subtyping relation. In proposition 4.9, we prove the correctness of the algorithm *Subtype* with respect to the definition of $\leqslant_C$, where $C \subseteq \mathbb{V} \times \mathbb{V}$ is a finite set of subtyping axioms, where if $(X, Y) \in C$ then X, Y are different variables.

We will restrict our attention to the particular case when $C \subset \mathbb{V}\times\mathbb{V}$, given that the more general case when $C \subset \mathbb{T}\times\mathbb{T}$ could allow typing non-terminating terms like for example $(\lambda x{:}S.xx)(\lambda x{:}S.xx)$.

poner despue's de lambda sub

EXERCISE 4.1 If we relax the restriction that C is a set of pairs of type variables one could type non-termination expresions. Find a subtyping axiom and write a typing derivation for $(\lambda x{:}S.xx)(\lambda x{:}S.xx)$.

DEFINITION 4.2 (*Subtyping*) The relation $\leqslant_C \subset \mathbb{T} \times \mathbb{T}$ is the smallest relation closed under the following rules.

$$
\begin{array}{rcll}
(X, Y) \in C & \Longrightarrow & X \leqslant_C Y & \text{S-INCL}, \\
& S \leqslant_C S & & \text{S-REFL}, \\
S \leqslant_C T,\ T \leqslant_C U & \Longrightarrow & S \leqslant_C U & \text{S-TRANS}, \\
T_1 \leqslant_C S_1,\ S_2 \leqslant_C T_2 & \Longrightarrow & S_1 \to S_2 \leqslant_C T_1 \to T_2 & \text{S-ARROW}.
\end{array}
$$

The S-ARROW rule deserves a close look. If we consider the relation $\leqslant_C$ as an ordering, then $\to$ is monotonic in the second argument and antimonotonic in the first argument. Intuitively, if every value of type S can be treated as a value of type S', then every function which maps S' to T also maps S to T.

LEMMA 4.3 If $S_1 \to S_2 \leqslant_C T$ then $T \equiv T_1 \to T_2$, for some T_1m and T_2.

EXERCISE 4.4 Prove lemma 4.3.

We would like to have a decision procedure for the subtyping relation, but the rules of $\leqslant_C$ are non-deterministic. For example, the last rule of the derivation of $S_1 \to S_2 \leqslant_C T_1 \to T_2$ can be S-TRANS or S-ARROW. Lemma 4.5 says that there is always a derivation whose last rule is S-ARROW, therefore an algorithm can avoid the non-determinism of the original presentation using this fact.

LEMMA 4.5 (*Inversion of the arrow rule*) Let S_1, S_2, T_1, and $T_2 \in \mathbb{T}$. Then

$$S_1 \to S_2 \ \leqslant_C \ T_1 \to T_2 \ \text{ if and only if } \ T_1 \leqslant_C S_1 \text{ and } S_2 \leqslant_C T_2.$$

PROOF: From right to left, it is just the S-ARROW rule. From left to right, the proof follows by induction on the derivation of $S_1 \to S_2 \ \leqslant_C \ T_1 \to T_2$. Since C only contains pairs of type variables, the S-INCL rule could not have been the last rule of the derivation.

S-REFL $S_1 \to S_2 \equiv T_1 \to T_2$ and this means that $T_1 \equiv S_1$ and $S_2 \equiv T_2$. Hence, by S-REFL, it follows that $T_1 \leqslant_C S_1$ and $S_2 \leqslant_C T_2$.

S-TRANS We are in the case that for some U, $S_1 \to S_2 \leqslant_C U$ and $U \leqslant_C T_1 \to T_2$. By lemma 4.3, U is of the form $U_1 \to U_2$. Then, by the induction hypothesis, $U_1 \leqslant_C S_1$, $S_2 \leqslant_C U_2$, $T_1 \leqslant_C U_1$, and $U_2 \leqslant_C T_2$. Then, by S-TRANS, we conclude that $T_1 \leqslant_C S_1$ and $S_2 \leqslant_C T_2$.

S-ARROW If the last rule was S-ARROW, then the only possibility for the hypothesis is $T_1 \leqslant_C S_1$ and $S_2 \leqslant_C T_2$. □

In what follows we define the algorithm *Subtype*, which is a decision procedure for the $\leqslant_C$ relation; as it is shown in proposition 4.9. But first we need the following definition.

DEFINITION 4.6 (*Transitive Closure of* C)

1. $(X, Y) \in C \implies (X, Y) \in \mathit{Trans}(C)$,
   $(X, Y) \text{ and } (Y, Z) \in \mathit{Trans}(C) \implies (X, Z) \in \mathit{Trans}(C)$.
2. $\mathit{trans}(S, T, C) = \mathsf{true}$ if and only if (S, T) belongs to $\mathit{Trans}(C)$.

We can now write down the algorithm.

DEFINITION 4.7 (*Subtype*) $\mathit{Subtype} : \mathbb{T} \times \mathbb{T} \times 2^{\mathbb{V} \times \mathbb{V}} \to \mathit{Bool}$

$$
\begin{array}{l}
\mathit{Subtype}(S,T,C) = \\
\qquad\qquad \mathit{if}\ S \equiv T \\
\qquad\qquad \mathit{then}\ \mathsf{true} \\
\qquad\qquad \mathit{else}\ \mathit{if}\ S \text{ and } T \text{ are variables} \\
\qquad\qquad\quad \mathit{then}\ \mathit{trans}(S,T,C) \\
\qquad\qquad\quad \mathit{else}\ \mathit{if}\ S \equiv S_1 \to S_2 \text{ and } T \equiv T_1 \to T_2 \\
\qquad\qquad\qquad \mathit{then}\ \mathit{Subtype}(T_1,S_1,C) \text{ and } \mathit{Subtype}(S_2,T_2,C) \\
\qquad\qquad\qquad \mathit{else}\ \mathsf{false}
\end{array}
$$

Where $\equiv$ is the syntactic equality.

Since C is finite, $\mathit{Trans}(C)$ is also finite. Consequently, $\mathit{trans}(S,T,C)$ is decidable. Moreover, the recursive calls have arguments of strictly smaller size. Hence, the algorithm *Subtype* always terminates.

Lemma 4.8 Let $X, Y \in \mathbb{V}$ and $S, T \in \mathbb{T}$. Then,

1. If $\mathit{trans}(X,Y,C) = \mathsf{true}$, then $X \leqslant_C Y$. If $X \leqslant_C Y$, then $\mathit{trans}(X,Y,C) = \mathsf{true}$ or $X \equiv Y$.

Proof:

1. By induction on the definition of $\mathit{Trans}(C)$.
2. By induction on the derivation of $X \leqslant_C Y$. □

We can now show that the algorithm *Subtype* is correct with respect to definition 4.2. The correctness is split into two parts, usually called soundness and completeness. Soundness means that if $\mathit{Subtype}(S,T,C) = \mathsf{true}$, then it is the case that $S \leqslant_C T$. Conversely, completeness means that if $S \leqslant_C T$, then the algorithm outputs true when called with arguments S, T and C.

Proposition 4.9 (*Soundness and completeness of Subtype*)
$\mathit{Subtype}(S,T,C) = \mathsf{true}$ if and only if $S \leqslant_C T$.

Proof:

$\Rightarrow$) By induction on the complexity of S and T.

Case 1. $S, T \in \mathbb{V}$. Then we have to consider the following two cases.

Case 1a. If $S \equiv T$, then, by S-Refl, $S \leqslant_C T$.

Case 1b. If $\mathit{trans}(S,T,C) = \mathsf{true}$, then, by lemma 4.8(1), we know that $S \leqslant_C T$.

Case 2. If either S or T is a variable and the other one is not, then the algorithm never yields true.

Case 3. Neither S nor T is a variable. Again we have to consider two cases.

$S \equiv T$. Then, by S-Refl, $S \leqslant_C T$.

$S \not\equiv T$. Say $S \equiv S_1 \to S_2$ and $T \equiv T_1 \to T_2$. Then it is the case that $\mathit{Subtype}(T_1,S_1,C) = \mathsf{true}$ and $\mathit{Subtype}(S_2,T_2,C) = \mathsf{true}$.
By the induction hypothesis, $T_1 \leqslant_C S_1$ and $S_2 \leqslant_C T_2$. Hence, due to the S-Arrow rule, $S_1 \to S_2 \leqslant_C T_1 \to T_2$.

Exercise 4.10 Prove the second part of the last proposition.
(Hint: the proof goes by induction on the derivation of $S \leqslant_C T$.

A closer look at the algorithm uncovers some proof theoretic properties of the subtyping relation $\leqslant_C$. Observe that in the algorithm the S-Trans rule is considered only at the level of variables, in other words, the S-Trans rule is never used as the last rule of a proof of a statement of the form $S_1 \to S_2 \leqslant_C T_1 \to T_2$. The corollary is then, that, if there exists a proof of $S \leqslant_C T$, then there exists also a proof of $S \leqslant_C T$ in which the applications of the S-Trans rule are only on statements of the form $X \leqslant_C Y$ and $Y \leqslant_C Z$, where X, Y, and Z are type variables. This fact can be read as follows: the system in which the S-Trans rule is replaced by

$$\text{S-Trans'} \quad X \leqslant_C Y,\ Y \leqslant_C Z \quad \Longrightarrow \quad X \leqslant_C Z, \ \text{ where } X, Y, Z \in \mathbb{V}.$$

can prove the same subtyping statements as the original system defined in 4.2.

From the proof theoretic point of view there is another possible refinement that consists of restricting the application of the S-Refl rule to type variables. In other words, we could replace S-Refl by

$$\text{S-Refl'} \quad X \leqslant_C X \ \text{ for all } X \in \mathbb{V}.$$

Exercise 4.11 Define $\leqslant'_C$ by replacing S-Trans and S-Refl by S-Trans' and S-Refl'. Then prove that $S \leqslant_C T$ if and only if $S \leqslant'_C T$.

5 $\lambda_\leq$, a system with implicit coercions

We define the system $\lambda_\leq$ (lambda sub), an extension of the simply typed λ-calculus with subtyping. The difference between the simply typed λ-calculus and $\lambda_\leq$ is the following rule.

$$\frac{\Gamma \vdash M : S \qquad S \leqslant_C T}{\Gamma \vdash M : T} \qquad \text{(T-Subsumption)}$$

An immediate consequence of the addition of this rule is the loss of the uniqueness of types property. Fortunately, the system has instead the minimal type property. Namely, if $\Gamma \vdash M : S$, then there exists T such that $\Gamma \vdash M : T$ and for every S' if $\Gamma \vdash M : S'$ then $T \leqslant_C S'$. That property is relevant in the design of type checking and type inference algorithms. Consider the case of type checking. Knowing that if a term is typeable, then it has a minimal type, we try to identify a fragment of the system in which every typeable term is assigned a minimal type. In our case, we define a subsystem of $\lambda_\leq$, $Alg\lambda_\leq$, which has the uniqueness of types property and is syntax directed, in such a way that that $Alg\lambda_\leq$ defines an algorithm to compute minimal types in $\lambda_\leq$.

We give now the formal definition of a simply typed λ-calculus with implicit coercions, $\lambda_\leq$.

Definition 5.1 The typed λ-calculus with implicit coercions, $\lambda_\leq$, is defined as follows.

1. The set of types $\mathbb{T} = \text{Type}(\lambda_\leq)$ is defined by

   $$\mathbb{T} \ ::= \ \mathbb{V} \mid \mathbb{T} \to \mathbb{T},$$

   where $\mathbb{V}$ is a set of (type) variables.

2. The set of pseudo-terms $\Lambda = \Lambda(\lambda_\leq)$ is defined as follows.

   $$\Lambda ::= V \mid \lambda V{:}\mathbb{T}.\Lambda \mid \Lambda\Lambda,$$

   where V is a set of (term) variables.

3. A *statement* is of the form $M : S$ (M is of type S) with $M \in \Lambda$ and $S \in \mathbb{T}$. The term M is called the *subject* of the statement. A typing assumption is an expression of the form $x{:}S$. The variable x is called the subject of the typing assumption.

4. A *context* is a set of typing assumptions with distinct variables as subjects.

Definition 5.2 (*Typing rules*)

$$\Gamma \vdash_{\lambda_\leq} x : S \quad \text{if } x{:}S \in \Gamma \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}S \vdash_{\lambda_\leq} M : T}{\Gamma \vdash_{\lambda_\leq} \lambda x{:}S.M : S \to T} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash_{\lambda_{\leq}} M : S \to T \qquad \Gamma \vdash_{\lambda_{\leq}} N : S}{\Gamma \vdash_{\lambda_{\leq}} MN : T} \tag{T-App}$$

$$\frac{\Gamma \vdash_{\lambda_{\leq}} M : S \qquad S \leqslant_C T}{\Gamma \vdash_{\lambda_{\leq}} M : T} \tag{T-Subsumption}$$

DEFINITION 5.3 A statement $M : S$ is *derivable* from the context Γ, we write $\Gamma \vdash_{\lambda_{\leq}} M : S$ – Γ yields M of type S –, if $\Gamma \vdash_{\lambda_{\leq}} M : S$ can be obtained using the rules T-VAR, T-ABS, T-APP, and T-SUBSUMPTION in definition 5.2. A $\lambda_{\leq}$-term M is *legal*, if there exist Γ and S such that $\Gamma \vdash_{\lambda_{\leq}} M : S$, In other words, a legal term is the subject of a derivable statements.

The typing rules in definition 5.2, are not syntax directed. In order to describe a type inference algorithm, we need an alternative presentation of the typing rules in which the term to be typed uniquely determines the last rule of the derivation of its typing statement. In the next section, we define $Alg\lambda_{\leq}$, an algorithmic presentation of $\lambda_{\leq}$. This presentation has the property of finding a minimal type for $\lambda_{\leq}$.

DEFINITION 5.4 (*A type inference algorithm*)

$$\Gamma \vdash_{Alg\lambda_{\leq}} x : S \quad \text{if } x{:}S\in\Gamma \tag{T-Var}$$

$$\frac{\Gamma, x{:}S \vdash_{Alg\lambda_{\leq}} M : T}{\Gamma \vdash_{Alg\lambda_{\leq}} \lambda x{:}S.M : S \to T} \tag{T-Abs}$$

$$\frac{\Gamma \vdash_{Alg\lambda_{\leq}} M : S \to T \qquad \Gamma \vdash_{Alg\lambda_{\leq}} N : U \qquad U \leqslant_C S}{\Gamma \vdash_{Alg\lambda_{\leq}} MN : T} \tag{T-App$_\leqslant$}$$

PROPOSITION 5.5 (*Generation for* Alg$\lambda_{\leq}$)

1. If $\Gamma \vdash_{Alg\lambda_{\leq}} x : S$, then $x{:}S\in\Gamma$.
2. If $\Gamma \vdash_{Alg\lambda_{\leq}} \lambda x{:}T.M : S$, then there exists U such that $\Gamma, x{:}T \vdash_{Alg\lambda_{\leq}} M : U$ and $S \equiv T \to U$.
3. If $\Gamma \vdash_{Alg\lambda_{\leq}} MN : S$, then there exist U and U' such that $\Gamma \vdash_{Alg\lambda_{\leq}} M : U \to S$, $U' \leqslant_C U$, and $\Gamma \vdash_{Alg\lambda_{\leq}} N : U'$.

PROOF: Since the system is syntax directed, the form of each judgement uniquely determines the last rule of its derivation. □

PROPOSITION 5.6 (*Uniqueness of types for* Alg$\lambda_{\leq}$) If $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$ and $\Gamma \vdash_{Alg\lambda_{\leq}} M : S'$, then $S \equiv S'$.

PROOF: By induction on the complexity of M, using that the system is syntax directed. □

LEMMA 5.7 (*Well-foundedness of* Alg$\lambda_{\leq}$) The $Alg\lambda_{\leq}$ rules (5.4) define a terminating algorithm.

PROOF: Since Γ is finite T-VAR cannot cause non-termination. In rules T-APP$_\leqslant$ and T-ABS, the size of the subject in each of the premises is strictly smaller than the size of the subject in the corresponding conclusion. Moreover, the relation $\leqslant_C$ is decidable because the algorithm *Subtype*, which is sound and complete with respect to $\leqslant_C$, always terminates. □

To prove the strong normalization property for $Alg\lambda_{\leq}$ we use the corresponding result for $\lambda{\to}$ in section 3. We first provide some definitions that allow us to relate $Alg\lambda_{\leq}$-terms to $\lambda{\to}$-terms in such a way that β-reductions are preserved.

DEFINITION 5.8 Let $W_0\in\mathbb{T}(\lambda{\to})$. $\_^{W_0} : \mathbb{T}(Alg\lambda_{\leq}) \to \mathbb{T}(\lambda{\to})$.

$$\begin{array}{rcll} X^{W_0} & = & W_0 & \text{if } X\in\mathbb{V}. \\ (S \to T)^{W_0} & = & S^{W_0} \to T^{W_0} & \end{array}$$

The homomorphic extension to contexts is defined as follows.

$$\begin{array}{rcl} \{\}^{W_0} & = & \{\} \\ (\Gamma \cup \{x{:}S\})^{W_0} & = & \Gamma^{W_0} \cup \{x{:}S^{W_0}\} \end{array}$$

DEFINITION 5.9 $\_\!\downarrow : \Lambda(Alg\lambda_{\leq}) \to \Lambda(\lambda{\to})$

$$\begin{array}{rcl} x\downarrow & = & x \\ (\lambda x{:}S.M)\downarrow & = & \lambda x{:}S^{W_0}.M\downarrow \\ (MN)\downarrow & = & M\downarrow N\downarrow \end{array}$$

The choice of W_0 is irrelevant, the essential feature is that it is fixed.

LEMMA 5.10 Let $S,T \in \mathbb{T}$. If $S \leqslant_C T$, then $S^{W_0} \equiv T^{W_0}$.

LEMMA 5.11 Let $M \in \Lambda(Alg\lambda_{\leq})$. If $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$, then $\Gamma^{W_0} \vdash_{\lambda\to} M\downarrow : S^{W_0}$.

PROOF: By induction on the derivation of $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$. Consider the case of the T-APP$_{\leqslant}$ rule. Then the situation is as follows.

$$\frac{\Gamma \vdash_{Alg\lambda_{\leq}} P : T \to S \quad \Gamma \vdash_{Alg\lambda_{\leq}} N : U \quad U \leqslant_C T}{\Gamma \vdash_{Alg\lambda_{\leq}} PN : S,}$$

where $M \equiv PN$. By the induction hypothesis, the definition of $\_^{W_0}$, and lemma 5.10, we have that

$$\Gamma^{W_0} \vdash_{\lambda\to} P\downarrow : T^{W_0} \to S^{W_0} \text{ and } \Gamma^{W_0} \vdash_{\lambda\to} N\downarrow : T^{W_0}.$$

Finally, by T-APP and the definition of $\_\!\downarrow$, it follows that $\Gamma^{W_0} \vdash_{\lambda\to} PN\downarrow : S^{W_0}$. □

LEMMA 5.12 (*Substitution*) Let M, N in $\Lambda(Alg\lambda_{\leq})$. Then,

$$(M[x{:=}N])\downarrow \equiv M\downarrow[x{:=}N\downarrow].$$

PROOF: By induction on the complexity of M. □

LEMMA 5.13 (*Reduction preservation*)
Let M, N in $\Lambda(Alg\lambda_{\leq})$. If $M \to_\beta N$, then $M\downarrow \to_\beta N\downarrow$.

PROOF: The proof is by straightforward induction on the complexity of M. In particular, the case when M is a redex is a consequence of lemma 5.12. □

PROPOSITION 5.14 (*Strong normalization for* Alg$\lambda_{\leq}$) If $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$, then M is strongly normalizing.

PROOF: Suppose, towards a contradiction, that M is not strongly normalizing. This means that there exists an infinite reduction chain starting from M. By the reduction preservation lemma (lemma 5.13), we know that there is also an infinite chain of reductions starting from $M^{\to}$, and using lemma 5.11, we know that

$$\Gamma^{W_0} \vdash_{\lambda\to} M\downarrow : S^{W_0}.$$

But, we also know that $\lambda{\to}$ has the strong normalization property which yields a contradiction. Hence, M is strongly normalizing. □

Observe that the strong normalization property of $Alg\lambda_{\leq}$ implies the strong normalization property of $\lambda_{\leq}$ because both systems assign types to the same set of terms.

The system $Alg\lambda_{\leq}$ does not satisfy the subject reduction property. Consider the following simple example.
Let $X \leqslant_C Z$, then $z{:}X \vdash_{Alg\lambda_{\leq}} (\lambda x{:}Z.x)z : Z$ and $(\lambda x{:}Z.x)z \to_\beta z$, but $z{:}X \nvdash_{Alg\lambda_{\leq}} z{:}Z$. This example illustrates the fact that a step of β-reduction of a term may reduce its original type. The reason being that, in order to type $(\lambda x{:}Z.x)z$, $X \leqslant_C Z$ is used, and such information can only be used by the application rule, T-APP$_{\leqslant}$. The β-reduction step "erases" the application. Therefore the subtyping information cannot be used any longer. Nevertheless, the following monotonic subject reduction property holds. The following result is necessary to prove proposition 5.16.

LEMMA 5.15 If $\Gamma, x{:}S \vdash_{Alg\lambda_{\leq}} M : T$, $\Gamma \vdash_{Alg\lambda_{\leq}} N : S'$, and $S' \leqslant_C S$, then there exists T' such that $\Gamma \vdash_{Alg\lambda_{\leq}} M[x{:=}N] : T'$ and $T' \leqslant_C T$.

PROOF: By induction on the complexity of M. □

PROPOSITION 5.16 (*Monotonic subject reduction*)
If $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$ and $M \to_\beta M'$, then there exists T such that $\Gamma \vdash_{Alg\lambda_{\leq}} M' : T$ and $T \leqslant_C S$.

PROOF: By straightforward induction on the complexity of M, using lemma 5.15. Let us consider the case when M is a redex and M' its reduct, that is the only case that demands some work. Then the situation is as follows.

$$\Gamma \vdash_{Alg\lambda_{\leq}} (\lambda x{:}U.M_1)M_2 : S.$$

By generation (proposition 5.5), we have that

$$\Gamma, x{:}U \vdash_{Alg\lambda_{\leq}} M_1 : S,\ \Gamma \vdash_{Alg\lambda_{\leq}} M_2 : U' \text{ and } U' \leqslant_C U.$$

By lemma 5.15, there exists S' such that $\Gamma \vdash_{Alg\lambda_{\leq}} M_1[x{:=}M_2] : S'$ and $S' \leqslant_C S$. □

Observe that because of the uniqueness of types property M' cannot have type S as well.

Here we show that $Alg\lambda_{\leq}$ describes an effective procedure to compute minimal types in $\lambda_{\leq}$.

PROPOSITION 5.17

1. (Soundness) If $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$, then $\Gamma \vdash_{\lambda_{\leq}} M : S$.
2. (Completeness and minimal typing) If $\Gamma \vdash_{\lambda_{\leq}} M : T$, then there exists S such that $S \leqslant_C T$ and $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$.

PROOF:

1. By induction on the derivation of $\Gamma \vdash_{Alg\lambda_{\leq}} M : S$.
2. By induction on the derivation of $\Gamma \vdash_{\lambda_{\leq}} M : S$. We consider the case for T-APP, the other cases follow with similar or simpler arguments. We are given that

   $$M \equiv M_1\, M_2,$$
   $$\Gamma \vdash_{\lambda_{\leq}} M_1 : T \to S, \text{ and}$$
   $$\Gamma \vdash_{\lambda_{\leq}} M_2 : T.$$

   By the induction hypothesis, there exist U_1 and U_2 such that

   $$\Gamma \vdash_{Alg\lambda_{\leq}} M_1 : U_1 \text{ where } U_1 \leqslant_C T \to S, \text{ and}$$
   $$\Gamma \vdash_{Alg\lambda_{\leq}} M_2 : U_2 \text{ where } U_2 \leqslant_C T.$$

   By the correctness of the algorithm *Subtype* (4.7), we know that $U_1 \equiv T_1 \to S_1$, and by lemma 4.5, we have that $T \leqslant_C T_1$ and $S_1 \leqslant_C S$. By S-TRANS, $U_2 \leqslant_C T_1$. Finally, by T-APP$_{\leqslant}$, we have that $\Gamma \vdash_{Alg\lambda_{\leq}} M_1\, M_2 : S_1$. □

The last proposition says that $\lambda_{\leq}$ and $Alg\lambda_{\leq}$ type the same set of terms, in other words, the set of legal terms of $\lambda_{\leq}$ is equal to that of $Alg\lambda_{\leq}$. The difference is that they may assign different types to the same term. Note that item 1 says that $Alg\lambda_{\leq}$-typing statements are also $\lambda_{\leq}$-typing statements, the converse is not true. Consider the following simple example.

Let $C = \{(X, Y)\}$. Then $x{:}X \vdash_{\lambda_{\leq}} x : Y$, but $x{:}X \vdash_{Alg\lambda_{\leq}} x : Y$ is not a derivable statement.

Observe that the typing information present in the $\lambda_{\leq}$ terms is essential for the minimal type property. Consider the following example in the system presented in [Mit84], i.e. a simply typed λ-calculus à la Curry with implicit coercions to see that it might be the case that neither the uniqueness of types property nor the minimal type property are satisfied.

Let $C = \{(X, Y)\}$ and let X, Y, and Z be different type variables. Then the identity function, $\lambda x.x$, has as type scheme $S \to T$ where $S \leqslant_C T$. In particular we have that

$$\Gamma \vdash \lambda x.x{:}X \to Y \text{ and } \Gamma \vdash \lambda x.x{:}Z \to Z,$$

but there is no type U such that

$$\Gamma \vdash \lambda x.x{:}U,\ U \leqslant_C X \to Y, \text{ and } U \leqslant_C Z \to Z.$$

Unlike the system $Alg\lambda_{\leq}$ the system $\lambda_{\leq}$ satisfies the subject reduction property.

PROPOSITION 5.18 (*Subject reduction for* $\lambda_{\leq}$)
If $\Gamma \vdash_{\lambda_{\leq}} M : S$ and $M \to_{\beta} M'$, then $\Gamma \vdash_{\lambda_{\leq}} M' : S$.

PROOF: By lemma 5.17(2), we know that there exists T such that $\Gamma \vdash_{Alg\lambda_{\leq}} M : T$ and $T \leqslant_C S$. By monotonic subject reduction (5.16), $\Gamma \vdash_{Alg\lambda_{\leq}} M : T'$, for some $T' \leqslant_C T$. Then, by lemma 5.17(1), it follows that $\Gamma \vdash_{\lambda_{\leq}} M : T'$. Finally, since $T' \leqslant_C S$, by T-SUBSUMPTION, it follows that $\Gamma \vdash_{\lambda_{\leq}} M' : S$. □

PROPOSITION 5.19 (*Decidability for* $\lambda_{\leq}$)

1. Given Γ and M, it is decidable whether there exists S such that $\Gamma \vdash_{\lambda_{\leq}} M : S$.
2. Given Γ, M, and S, it is decidable whether $\Gamma \vdash_{\lambda_{\leq}} M : S$.

PROOF: By the well-foundedness of $Alg\lambda_{\leq}$, Soundness and Completeness.

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118), and as DEC Systems Research Center Research Report number 62, August 1990.

[Bar90] Henk P. Barendregt. Functional programming and lambda calculus. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 7, pages 323–363. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Bar92] H. Barendregt. Lambda calculi with types. In T. S. E. Maibaum S. Abramsky, Dov M. Gabbay, editor, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Clarendon Press. Oxford, 1992.

[BCD83] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[BCGS91] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[BM92] Kim B. Bruce and John Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Proceedings of the Nineteenth ACM Symposium on Principles of Programming Languages*, Albequerque, NM, January 1992.

[Bru94] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language".

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Preliminary version in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

[Car90] Luca Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[CD80] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[CDdL93] Felice Cardone, Mariangiola Dezani-Ciancaglini, and Ugo de' Liguoro. Combining type disciplines, 1993. Manuscript.

[CG92] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption: Minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[CG97] Adriana Compagnoni and Healfdene Goguen. Typed operational semantics for higher order subtyping. Available at http://www.dcs.ed.ac.uk/lfcsreps/export/97/ecs-lfcs-97-361/index.html. Technical report, University of Edinburgh, LFCS, April 1997.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, San Francisco, CA, January 1990. Also in [GM94].

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *Amer. J. Math*, 58:354–363, 1936.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[CL91] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, October 1991. Preliminary version in ACM Conference on Lisp and Functional Programming, June 1990. Also available as DEC SRC Research Report 55, Feb. 1990.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in [GM94], and available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[Com95] Adriana B. Compagnoni. *Higher-Order Subtyping with Intersection Types*. PhD thesis, University of Nijmegen, The Netherlands, January 1995. ISBN 90-9007860-6.

[CP96] Adriana B. Compagnoni and Benjamin C. Pierce. Higher-order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6:469–501, 1996. Preliminary version available under the title *Multiple Inheritance via Intersection Types* as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[dB80] Nicolas G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[GM94] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994.

[Lan65] P. J. Landin. A correspondence between ALGOL–60 and Church's lambda notation. *Communications of the ACM*, 8:89–101, 1965.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.

[Mar73] Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium, '73*, pages 73–118, Amsterdam, 1973. North Holland.

[Mit84] J. C. Mitchell. *Lambda Calculus Models of Typed Programming Languages.* PhD thesis, Massachusetts Institute of Technology, 1984.

[Mit90a] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990.

[Mit90b] John C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 8, pages 365–458. Elsevier Science Publishers B. V. (North-Holland), 1990.

[MP93] J. McKinna and R. Pollack. Pure type systems formalized. In *Proc. of the International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, 1993.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991. Available as School of Computer Science technical report CMU-CS-91-205.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[Rey80] John Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94 in Lecture Notes in Computer Science. Springer-Verlag, January 1980. Also in [GM94].

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[SP94] Martin Steffen and Benjamin Pierce. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1994. An earlier version appeared as University of Edinburgh technical report ECS-LFCS-94-280 and Universität Erlangen-Nürnberg Interner Bericht IMMD7-01/94,.

[Tak95] Masako Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118:120–127, 1995.