

**Intersection Types
and Bounded Polymorphism**

by

Benjamin C Pierce

Intersection Types and Bounded Polymorphism

LFCS Report Series

ECS-LFCS-92-200

LFCS

March 1992

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1992, LFCS

**Copyright © 1992, Laboratory for Foundations of Computer Science
University of Edinburgh. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

Intersection Types and Bounded Polymorphism

Benjamin C. Pierce*

March 6, 1992

Abstract

Intersection types and bounded quantification are complementary mechanisms for extending the expressive power of statically typed programming languages. They begin with a common framework: a simple, typed language with higher-order functions and a notion of subtyping. Intersection types extend this framework by giving every pair of types σ and τ a greatest lower bound, $\sigma \wedge \tau$, corresponding intuitively to the intersection of the sets of values described by σ and τ . Bounded quantification extends the basic framework along a different axis by adding polymorphic functions that operate uniformly on all the subtypes of a given type. We describe a new type system incorporating both, illustrate some of its properties, and present a typechecking algorithm.

1 Introduction

During the past decade, research in static type systems has succeeded in developing formal accounts of various notions of *subtyping*. In 1984, Cardelli [12] suggested that the basic concepts of object-oriented programming could be understood type-theoretically by treating objects as records and modeling inheritance using subtyping. Some parts of this picture have been filled in during the intervening years, whereas accounting for other aspects of object-oriented programming — method inheritance, for example — remains the subject of active research. But whatever the ultimate success of this research program, the basic framework has proven to be a fruitful source of innovation in language design.

Intersection types were developed in the late 1970s by Coppo and Dezani-Ciancaglini [27], and independently by Sallé [33, 67] and Pottinger [59]. Since then,

*Author's address: LFCS, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, U.K., bcp@dcsl.ed.ac.uk. This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597; in part by the Office of Naval Research under Contract N00013-84-K-0415; in part by the National Science Foundation under Contract CCR-8922109; and in part by Siemens.

they have been studied extensively by members of the group at Turin and others [2, 22, 28, 29, 30, 31, 32, 38, 39, 46, 47, 63, 64, 65, 66, 68, 69]. Various extensions of the original intersection type discipline have also been explored. These include the notion of infinite intersections [50], the dual notion of union types [1, 45, 56], and the relationship between intersection types and models of polymorphism [49, 54, 70]. Some related extensions to ML-style type inference systems are the notions of refinement types [41, 44, 56] and soft typing [23, 40], as well as Coppo and Giannini's recent work on type inference for a decidable restriction of intersection types [34]. Reynolds provided the first demonstration that intersection types can be used as the basis for practical programming languages [61, 63].

The most intriguing property of intersection types is their ability to express an essentially unbounded (though of course finite) amount of information about a program. For example, the addition function $+$ can be given the type $Int \rightarrow Int \rightarrow Int \wedge Real \rightarrow Real \rightarrow Real$, capturing both the general fact that the sum of two real numbers is always a real and the more specialized observation that the sum of two integers is always an integer. A compiler for a language with intersection types might even provide two different object-code sequences for the different versions of $+$, one using a floating point addition instruction and one using integer addition. For each instance of $+$ in a program, the compiler can decide whether both arguments are integers and generate the more efficient object code sequence in this case. This kind of *finitary polymorphism* or *coherent overloading* is so expressive, that (in a sense that can be made theoretically precise; c.f. [22]) the set of all valid typings for an untyped λ -term amounts to a complete characterization of its behavior.

The notion of *bounded quantification* was introduced by Cardelli and Wegner [20] in the language Fun. Based on informal ideas by Cardelli and formalized using techniques developed by Mitchell [52], Fun integrated Girard-Reynolds polymorphism [43, 60] with Cardelli's first-order calculus of subtyping [11, 12]. The original Fun was simplified and slightly generalized by Bruce and Longo [5], and again by Curien and Ghelli [35]. (Curien and Ghelli's formulation, called *minimal Bounded Fun* or $F_{\leq}$ (" F -sub"), is the one considered here.) Cardelli and Wegner's survey paper gives the first programming examples using bounded quantification; more are developed in Cardelli's study of power kinds [13]. Curien and Ghelli [35, 42] address a number of syntactic properties of $F_{\leq}$. Semantic aspects of closely related systems have been studied by Bruce and Longo [5], Martini [51], Breazu-Tannen, Coquand, Gunter, and Scedrov [3], Cardone [21], Cardelli and Longo [17], Cardelli, Martini, Mitchell, and Scedrov [18], Curien and Ghelli [35, 36], and Bruce and Mitchell [6]. $F_{\leq}$ has been extended to include record types and richer notions of inheritance by Cardelli and Mitchell [19], Bruce [4], Cardelli [15], and Canning, Cook, Hill, Olthoff, and Mitchell [9]. Bounded quantification also plays a key role in Cardelli's programming language Quest [14, 17] and in the Abel language developed at HP Labs [8, 9, 10, 26].

Bounded quantification is an extension of ordinary second-order quantification, which captures the intuitive concept of a function that takes a type as a parameter. For instance, the "polymorphic reverse" function, which accepts a type τ and returns the *monomorphic* reverse function that reverses lists whose elements are all of type τ , has the quantified type $\forall \alpha. List(\alpha) \rightarrow List(\alpha)$. Cardelli and Wegner integrated this mechanism with the notion of subtyping by allowing a quantified type to give a *bound* for its parameter; for example, $\forall \alpha \leq Student. List(\alpha) \rightarrow List(\alpha)$ takes as its first parameter an

arbitrary subtype of the type *Student* and returns a function on lists of this type. This *universal* or *parametric polymorphism* is both broader (since the number of possible instantiations of a polymorphic type is infinite) and more rigid (since all instances must have the same basic shape) than the finitary polymorphism provided by intersection types. Its main practical advantages are brevity and compile-time efficiency: it allows polymorphic expressions to be written, typechecked, and compiled just once.

Intersection types and bounded quantification thus complement and, in a sense, complete each another: quantifiers can be used to compactly capture the common cases where all the desired types of a term are structurally similar, which lead to long and unwieldy typings when expressed only in terms of intersections, while intersections allow a degree of refinement that is impossible with pure polymorphism. Moreover, intersections can be used in the bounds of quantifiers to express a simple form of “multiple inheritance”; for example, the type $\forall \alpha \leq \textit{Student} \wedge \textit{Employee}. \textit{List}(\alpha) \rightarrow \textit{List}(\alpha)$ describes polymorphic functions applicable only to the common subtypes of *Student* and *Employee*.

Following a more detailed discussion of the pure systems of intersections and bounded quantification (Section 2), we describe (Section 3) a typed λ -calculus called $F_\wedge$ (“F-meet”) — a “core type system” integrating the features of both. Section 4 gives examples illustrating $F_\wedge$ ’s expressive power. Sections 5 and 6 present algorithms for checking $F_\wedge$ ’s subtype relation and synthesizing minimal types for terms. Section 7 discusses semantic aspects of the calculus, and Section 8 offers concluding remarks. A more detailed presentation of these results, including complete proofs, may be found in [55].

2 Background

We begin by establishing some notational conventions and describing the precursors to $F_\wedge$ in greater detail.

2.1 Notational Preliminaries

Definition: A finite sequence with elements x_1 through x_n is written $[x_1..x_n]$. Concatenation of finite sequences is written $X_1 * X_2$. Single elements are adjoined to the right or left of sequences with a comma: $[x_a, x_b]$ or $[X_a, x_b]$. Sequences are sometimes written in a “comprehension” notation: $[x \mid \dots]$. For example, if $T \equiv [\sigma \rightarrow \tau, \psi \rightarrow \phi, \sigma \rightarrow \theta]$, then the comprehension $[\zeta_2 \mid \zeta_1 \rightarrow \zeta_2 \in T \text{ and } \zeta_1 \equiv \sigma]$ stands for the finite sequence $[\tau, \theta]$.

Notation: The metavariables α and β range over type variables; $\sigma, \tau, \theta, \phi$, and ψ range over types; e and f range over terms; and x and y range over term variables.

Definition: A *context* Γ is a finite sequence of typing assumptions $x:\tau$ and subtyping assumptions $\alpha \leq \tau$, with no variable or type variable listed twice. The empty context is written $\{\}$.

Definition: A *subtyping statement* is a phrase of the form $\Gamma \vdash \sigma \leq \tau$, where σ and τ are types. A *typing statement* is a phrase of the form $\Gamma \vdash e \in \tau$, where e is a term and τ is a type.

Convention: Types, terms, contexts, and statements that differ only in the names of bound variables are considered identical. (It is formally clearer to think of variables not as names but, as suggested by deBruijn [37], as pointers into the surrounding context.)

Definition: A type τ is *closed* with respect to a context Γ if $FTV(\tau) \subseteq \text{dom}(\Gamma)$. A term e is closed with respect to Γ if $FTV(e) \cup FV(e) \subseteq \text{dom}(\Gamma)$. A context Γ is closed if

1. $\Gamma \equiv \{\}$, or
2. $\Gamma \equiv \Gamma_1, \alpha \leq \tau$, with Γ_1 closed and τ closed with respect to Γ_1 , or
3. $\Gamma \equiv \Gamma_1, x : \tau$, with Γ_1 closed and τ closed with respect to Γ_1 .

A subtyping statement $\Gamma \vdash \sigma \leq \tau$ is closed if Γ is closed and σ and τ are closed with respect to Γ ; a typing statement $\Gamma \vdash e \in \tau$ is closed if Γ is closed and e and τ are closed with respect to Γ .

Convention: In the following, we assume that all statements under discussion are closed. In particular, we allow only closed statements in instances of inference rules.

Definition: The capture-avoiding substitution of e for x in f is written $\{e/x\}f$. The capture-avoiding substitution of σ for α in τ or e is written $\{\sigma/\alpha\}\tau$ or $\{\sigma/\alpha\}e$. The capture-avoiding substitution of σ for α in the range of Γ is written $\{\sigma/\alpha\}\Gamma$.

Notation: Examples illustrating sessions with a prototype typechecker for F_λ are set in a typewriter font. The mathematical symbols used in the λ -calculus notation are transliterated as follows: $\top$ is written as `T`, λ as `\`, Λ as `\`, $\forall$ as `All`, and $\leq$ as `<`. Lines of input to the typechecker are prefixed with a `>` character and followed by the system's response.

Convention: The type constructors $\rightarrow$ and $\forall$ bind more tightly than $\wedge$, allowing most parentheses to be dropped. Also, $\rightarrow$ associates to the right and $\forall$ obeys the usual “dot rule” where the body τ of a quantified type $\forall \alpha \leq \sigma. \tau$ is taken to extend to the right as far as possible.

2.2 Simply Typed λ -Calculus with Subtyping

F_λ may be viewed as a “least upper bound” of two calculi: a first-order λ -calculus with intersection types (called $\lambda_\wedge$ here) and a second-order λ -calculus with bounded quantification (generally called $F_\leq$). These, in turn, are both extensions of the simply typed λ -calculus enriched with a subtyping relation ($\lambda_\leq$). The latter system was proposed by Cardelli [12] as a “core calculus of subtyping” in a foundational framework for object-oriented programming languages.

Definition: The types of $\lambda_\leq$ consist of a set of *primitive types* (ranged over by the metavariable ρ) closed under the *function space* type constructor $\rightarrow$:

$$\tau ::= \rho \mid \tau_1 \rightarrow \tau_2$$

Definition: The terms of $\lambda_\leq$ consist of a countable set of variables, together with all the phrases that can be built from these by functional abstraction and application:

$$e ::= x \mid \lambda x : \tau. e \mid e_1 e_2$$

The typing relation of $\lambda_\leq$ is formalized as a collection of inference rules for deriving typing statements of the form $\Gamma \vdash e \in \tau$ (“under assumptions Γ , expression e has type

τ''), where Γ contains a typing assumption for each of the free variables of e . The rules for variables, abstractions, and applications are exactly the same as in the ordinary simply typed λ -calculus [24]. In addition, we introduce a rule of *subsumption* stating that whenever a term e has a type σ and σ is a subtype of another type τ , the type of e may be promoted to τ .

Definition: The $\lambda_{\leq}$ typing relation $\Gamma \vdash e \in \tau$ is the least three-place relation closed under the following rules:

$$\begin{array}{c}
\Gamma \vdash x \in \Gamma(x) \quad (\text{VAR}) \\
\\
\frac{\Gamma, x:\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x:\tau_1. e \in \tau_1 \rightarrow \tau_2} \quad (\text{ARROW-I}) \\
\\
\frac{\Gamma \vdash e_1 \in \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1 e_2 \in \tau_2} \quad (\text{ARROW-E}) \\
\\
\frac{\Gamma \vdash e \in \tau_1 \quad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \quad (\text{SUB})
\end{array}$$

Intuitively, a subtyping statement $\Gamma \vdash \sigma \leq \tau$ corresponds to the assertion that σ is a *refinement* of τ , in the sense that every element of σ contains enough information to meaningfully be regarded as an element of τ . In some models this means simply that σ is a subset of τ ; more generally, it implies the existence of a distinguished *coercion function* from σ to τ .

These considerations immediately entail that the subtype relation should be both reflexive and transitive — i.e., that it should be a preorder. We assume that the subtype relation on primitive types is given in advance by some preorder $\leq_P$. This relation is extended to the smallest preorder closed under the following subtyping rule for function types: $\sigma_1 \rightarrow \sigma_2$ is a subtype of $\tau_1 \rightarrow \tau_2$ iff τ_1 is a subtype of σ_1 and σ_2 is a subtype of τ_2 . Notice that, as usual, this relation is *covariant* in the right-hand side and *contravariant* in the left-hand side of the $\rightarrow$ constructor: a collection of functions can be refined either by narrowing the range into which their results must fall or by enlarging the domain over which they must behave properly.

Definition: The $\lambda_{\leq}$ subtyping relation $\Gamma \vdash \sigma \leq \tau$ is the least three-place relation closed under the following rules:

$$\begin{array}{c}
\Gamma \vdash \tau \leq \tau \quad (\text{SUB-REFL}) \\
\\
\frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} \quad (\text{SUB-TRANS}) \\
\\
\frac{\Gamma \vdash \rho_1 \leq_P \rho_2}{\Gamma \vdash \rho_1 \leq \rho_2} \quad (\text{SUB-PRIM}) \\
\\
\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \quad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 \leq \tau_1 \rightarrow \tau_2} \quad (\text{SUB-ARROW})
\end{array}$$

Remark: The context Γ plays no part in these rules and could be dropped without changing the system. We include it here for notational compatibility with $F_{\leq}$ and F_{λ} , in which contexts also contain subtyping assumptions about type variables.

2.3 Intersection Types

Definition: The first-order calculus of intersection types, $\lambda_\wedge$, is formed from $\lambda_\leq$ by adding intersections to the language of types:

$$\tau ::= \rho \mid \tau_1 \rightarrow \tau_2 \mid \wedge[\tau_1.. \tau_n]$$

Definition: In the examples, we use the following abbreviations:

$$\begin{aligned} \sigma \wedge \tau &\stackrel{\text{def}}{=} \wedge[\sigma, \tau] \\ \top &\stackrel{\text{def}}{=} \wedge[.] \end{aligned}$$

Two new subtyping rules capture the order-theoretic properties of the $\wedge$ operator:

$$\frac{\text{for all } i, \Gamma \vdash \sigma \leq \tau_i}{\Gamma \vdash \sigma \leq \wedge[\tau_1.. \tau_n]} \quad (\text{SUB-INTER-G})$$

$$\Gamma \vdash \wedge[\tau_1.. \tau_n] \leq \tau_i \quad (\text{SUB-INTER-LB})$$

One additional subtyping rule captures the relation between intersections and function spaces, allowing the two constructors to “distribute” when an intersection appears on the right-hand side of an arrow:

$$\Gamma \vdash \wedge[\sigma \rightarrow \tau_1 .. \sigma \rightarrow \tau_n] \leq \sigma \rightarrow \wedge[\tau_1.. \tau_n] \quad (\text{SUB-DIST-IA})$$

(This inclusion is actually an equivalence, since the other direction may be proved from the rules for meets and arrows.) This rule has a strong effect on both syntactic and semantic properties of the language. For example, it implies that $\top \leq \sigma \rightarrow \top$ for any σ .

The typing rules must also be extended slightly. As for any type constructor, we expect to find a pair of an introduction rule, by which terms can be shown to possess intersection types, and an elimination rule, by which this fact may later be exploited. The introduction rule allows an intersection type to be derived for a term whenever each of the elements of the intersection can be derived for it separately:

$$\frac{\text{for all } i, \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \wedge[\tau_1.. \tau_n]} \quad (\text{INTER-I})$$

The corresponding elimination rule would allow us to infer, on the basis of a derivation of a statement like $\Gamma \vdash e \in \wedge[\tau_1.. \tau_n]$, that e possesses every τ_i individually. But this follows already from the rule SUB-INTER-G and the rule of subsumption; we need not add the elimination rule explicitly to the calculus.

The nullary case of this rule is worth particular notice, since it allows the type $\top$ to be derived for *every* term of the calculus, including terms whose evaluation intuitively encounters a run time error

```
> bad1 = 5 true;
bad1 : T

or fails to terminate:

> bad2 = (\x:T. x x) (\x:T. x x);
bad2 : T
```

The system as we have described it so far supports the use of intersection types in programming only to a limited degree. Suppose, for example, that the primitive subtype relation has $Int \leq Real$ and the addition function in the pervasive context is overloaded to operate on both integers and reals:

$$\Gamma_p(+) = Int \rightarrow Int \rightarrow Int \wedge Real \rightarrow Real \rightarrow Real.$$

Using just the constructs introduced so far, there is no way of writing our own functions that “inherit” the finitary polymorphism of $+$. For example, the doubling function $\lambda x:?. x + x$ cannot be given the type $Int \rightarrow Int \wedge Real \rightarrow Real$, since replacing the $?$ with either Int or $Real$ (or $Int \wedge Real$, or indeed — even assuming we could write it — $Int \wedge Real$) gives a typing that is too restrictive:

```
> double1 = \x:Int. plus x x;
double1 : Int -> Int

> double2 = \x:Real. plus x x;
double2 : Real -> Real

> double3 = \x:Int/\Real. plus x x;
double3 : Int -> Int
```

This led Reynolds [63] to introduce a generalized form of λ -abstraction allowing explicit programmer-controlled generation of alternative typings for terms:

$$e ::= \dots \mid \lambda x:\tau_1..\tau_n. e$$

The typing rule for this form allows the typechecker to make a choice of any of the σ 's as the type of x in the body:

$$\frac{\Gamma, x:\sigma_i \vdash e \in \tau_i}{\Gamma \vdash \lambda x:\sigma_1..\sigma_n. e \in \sigma_i \rightarrow \tau_i} \quad (\text{ARROW-I}')$$

This rule can be used together with INTER-I to generate a set of n alternative typings for the body and then form their intersection as the type of the whole λ -abstraction:

```
> double = \x:Int,Real. plus x x;
double : Real->Real /\ Int->Int
```

2.4 Bounded Polymorphism

Like other second-order λ -calculi, the terms of $F_{\leq}$ include the variables, abstractions, and applications of $\lambda_{\leq}$ plus type abstractions and type applications — slightly refined here to take account of the subtype relation: each type abstraction gives a *bound* for the type variable it introduces and each type application must satisfy the constraint that the argument type is a subtype of the bound of the polymorphic function being applied. Also, like that of $\lambda_{\wedge}$, the $F_{\leq}$ subtype ordering includes a maximal element. (Since the two are not exactly the same (c.f. Section 3), the maximal $F_{\leq}$ type is called here by its conventional name, *Top*, instead of $\top$.)

$$\begin{aligned} \tau &::= Top \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha \leq \tau_1. \tau_2 \\ e &::= x \mid \lambda x:\tau. e \mid e_1 e_2 \mid \Lambda \alpha \leq \tau. e \mid e[\tau] \end{aligned}$$

Type abstractions have almost the same typing rule as in other second-order λ -calculi; they are checked by moving their stated bound for the type variable they

introduce into the context and checking the body of the abstraction under the enriched set of assumptions:

$$\frac{\Gamma, \alpha \leq \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda \alpha \leq \tau_1. e \in \forall \alpha \leq \tau_1. \tau_2} \quad (\text{ALL-I})$$

The rule for type applications must check that the type being passed as a parameter is indeed a subtype of the bound of the corresponding quantifier:

$$\frac{\Gamma \vdash e \in \forall \alpha \leq \tau_1. \tau_2 \quad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2} \quad (\text{ALL-E})$$

The subtype relation of $\lambda_{\leq}$ is also extended with several rules. First, we stipulate that *Top* is a maximal element of the subtype order:

$$\Gamma \vdash \sigma \leq \text{Top} \quad (\text{SUB-TOP})$$

Type variables are subtypes of the bounds given for them in the prevailing context:

$$\Gamma \vdash \alpha \leq \Gamma(\alpha) \quad (\text{SUB-TVAR})$$

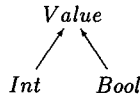
Like arrow types, subtyping of quantified types is contravariant in their bounds and covariant in their bodies:

$$\frac{\Gamma \vdash \tau_1 \leq \sigma_1 \quad \Gamma, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall \alpha \leq \sigma_1. \sigma_2 \leq \forall \alpha \leq \tau_1. \tau_2} \quad (\text{SUB-ALL})$$

This rule deserves a closer look, since it causes considerable difficulties in the proof theory of $F_{\leq}$ [42, 55, 57]. Intuitively, it reads as follows:

A type $\tau \equiv \forall \alpha \leq \tau_1. \tau_2$ describes a collection of polymorphic values (functions from types to values), each mapping subtypes of τ_1 to instances of τ_2 . If τ_1 is a subtype of σ_1 , then the domain of τ is smaller than that of $\sigma \equiv \forall \alpha \leq \sigma_1. \sigma_2$, so σ is a stronger constraint and describes a smaller collection of polymorphic values. Moreover, if, for each type θ that is an acceptable argument to the functions in both collections (i.e., one that satisfies the more stringent requirement $\theta \leq \tau_1$), the θ -instance of σ_2 is a subtype of the θ -instance of τ_2 , then σ is a “pointwise stronger” constraint and again describes a smaller collection of polymorphic values.

Remark: Since contexts now include assumptions about free type variables, we can drop the separate notion of “primitive types.” For each primitive type of $\lambda_{\leq}$, we add a variable with the same name to the pervasive context Γ_p . The bounds given to these variables encode the subtype relation $\leq_p$. For example, the primitive subtype relation



is represented by the pervasive context $\Gamma_p \equiv \text{Value} \leq \text{Top}, \text{Int} \leq \text{Value}, \text{Bool} \leq \text{Value}$.

3 The $F_\wedge$ Calculus

$F_\wedge$ can roughly be characterized as the union of the systems $\lambda_\wedge$ and $F_\leq$. To achieve a compact and symmetric calculus, however, a few modifications and extensions are needed. Since $F_\leq$ allows primitive types to be encoded as elements of the pervasive context, we drop the primitive types of $\lambda_\leq$ and $\lambda_\wedge$ and the rule SUB-PRIM. Since $\top$ and *Top* both function as maximal elements of their respective subtype orderings, we drop *Top* and let $\top$ take over its job. Since $\forall$ behaves like a kind of function space constructor, we add a new law SUB-DIST-IQ, analogous to SUB-DIST-IA, allowing intersections to be distributed over quantifiers on the right-hand side. Finally, we use the ordinary form of λ -abstraction from $F_\leq$ rather than the generalized one introduced by Reynolds for $\lambda_\wedge$; the notion of deriving alternative typings for subphrases under different sets of assumptions is captured by a new syntactic form, *for*.

3.1 Explicit Alternation: The *for* Construct

The notions of type variables and type substitution inherited from $F_\leq$ can be used to define an elegant further generalization of $\lambda_\wedge$'s form of λ -abstraction. We extend the syntax of terms with a new form

$$e ::= \dots \mid \text{for } \alpha \text{ in } \sigma_1.. \sigma_n. e$$

whose typing rule allows a choice of any of the σ 's as a replacement for α in the body:

$$\frac{\Gamma \vdash \{\sigma_i/\alpha\} e \in \tau_i}{\Gamma \vdash \text{for } \alpha \text{ in } \sigma_1.. \sigma_n. e \in \tau_i} \quad (\text{FOR})$$

This rule, like the generalized arrow introduction rule ARROW-I' of $\lambda_\wedge$, can be used together with INTER-I to generate a set of n alternative typings for the body and then form their intersection as the type of the whole *for* expression:

```
> double = for A in Int,Real. \x:A. plus x x;
double : Real->Real /\ Int->Int
```

Indeed, $\lambda_\wedge$'s generalized λ -abstraction may be reintroduced as a simple syntactic abbreviation:

Notation: $\lambda x:\sigma_1.. \sigma_n. e \stackrel{\text{def}}{=} \text{for } \alpha \text{ in } \sigma_1.. \sigma_n. \lambda x:\alpha. e$, where α is fresh.

Besides separating the mechanisms of functional abstraction and alternation, the introduction of the *for* construct extends the expressive power of the language by providing a *name* for the “current choice” being made by the type checker. For example, the explicit *for* construct may be used to improve the efficiency of typechecking even for first-order languages with intersections:

```
> poly =
>   \w:Int,Real. \x:Int,Real. \y:Int,Real. \z:Int,Real.
>     plus (double x) (plus (plus w y) z);
poly : Real->Real->Real->Real->Real /\ Int->Int->Int->Int->Int

> poly =
>   for A in Int,Real.
>     \w:A. \x:A. \y:A. \z:A.
>       plus (double x) (plus (plus w y) z);
poly : Real->Real->Real->Real->Real /\ Int->Int->Int->Int->Int
```

(The second version of `poly` requires that the body be checked only twice, as compared to sixteen times for the first version.)

3.2 Syntax, Subtyping, and Typing

We can now state the definition of F_λ in full.

Definition: The set of F_λ types is defined by the following abstract grammar:

$$\begin{array}{lcl} \tau & ::= & \alpha \\ & & | \tau_1 \rightarrow \tau_2 \\ & & | \forall \alpha \leq \tau_1. \tau_2 \\ & & | \bigwedge [\tau_1.. \tau_n] \end{array}$$

Definition: The set of F_λ terms is defined by the following abstract grammar:

$$\begin{array}{lcl} e & ::= & x \\ & & | \lambda x : \tau. e \\ & & | e_1 e_2 \\ & & | \Lambda \alpha \leq \tau. e \\ & & | e[\tau] \\ & & | \text{for } \alpha \text{ in } \tau_1.. \tau_n. e \end{array}$$

Definition: The three-place F_λ *subtype relation* $\Gamma \vdash \sigma \leq \tau$ is the least relation closed under the following rules:

$$\begin{array}{lcl} \Gamma \vdash \tau \leq \tau & & (\text{SUB-REFL}) \\ \frac{\Gamma \vdash \tau_1 \leq \tau_2 \quad \Gamma \vdash \tau_2 \leq \tau_3}{\Gamma \vdash \tau_1 \leq \tau_3} & & (\text{SUB-TRANS}) \\ \Gamma \vdash \alpha \leq \Gamma(\alpha) & & (\text{SUB-TVAR}) \\ \frac{\Gamma \vdash \tau_1 \leq \sigma_1 \quad \Gamma \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 \leq \tau_1 \rightarrow \tau_2} & & (\text{SUB-ARROW}) \\ \frac{\Gamma \vdash \tau_1 \leq \sigma_1 \quad \Gamma, \alpha \leq \tau_1 \vdash \sigma_2 \leq \tau_2}{\Gamma \vdash \forall \alpha \leq \sigma_1. \sigma_2 \leq \forall \alpha \leq \tau_1. \tau_2} & & (\text{SUB-ALL}) \\ \frac{\text{for all } i, \Gamma \vdash \sigma \leq \tau_i}{\Gamma \vdash \sigma \leq \bigwedge [\tau_1.. \tau_n]} & & (\text{SUB-INTER-G}) \\ \Gamma \vdash \bigwedge [\tau_1.. \tau_n] \leq \tau_i & & (\text{SUB-INTER-LB}) \\ \Gamma \vdash \bigwedge [\sigma \rightarrow \tau_1.. \sigma \rightarrow \tau_n] \leq \sigma \rightarrow \bigwedge [\tau_1.. \tau_n] & & (\text{SUB-DIST-IA}) \\ \Gamma \vdash \bigwedge [\forall \alpha \leq \sigma. \tau_1.. \forall \alpha \leq \sigma. \tau_n] \leq \forall \alpha \leq \sigma. \bigwedge [\tau_1.. \tau_n] & & (\text{SUB-DIST-IQ}) \end{array}$$

Definition: The three-place F_λ *typing relation* $\Gamma \vdash e \in \tau$ is the least relation closed under the following rules:

$$\begin{array}{lcl} \Gamma \vdash x \in \Gamma(x) & & (\text{VAR}) \\ \frac{\Gamma, x : \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x : \tau_1. e \in \tau_1 \rightarrow \tau_2} & & (\text{ARROW-I}) \\ \frac{\Gamma \vdash e_1 \in \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \in \tau_1}{\Gamma \vdash e_1 e_2 \in \tau_2} & & (\text{ARROW-E}) \end{array}$$

$$\begin{array}{c}
\frac{\Gamma, \alpha \leq \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda \alpha \leq \tau_1. e \in \forall \alpha \leq \tau_1. \tau_2} \quad (\text{ALL-I}) \\
\\
\frac{\Gamma \vdash e \in \forall \alpha \leq \tau_1. \tau_2 \quad \Gamma \vdash \tau \leq \tau_1}{\Gamma \vdash e[\tau] \in \{\tau/\alpha\}\tau_2} \quad (\text{ALL-E}) \\
\\
\frac{\Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash \text{for } \alpha \text{ in } \sigma_1.. \sigma_n. e \in \tau_i} \quad (\text{FOR}) \\
\\
\frac{\text{for all } i, \Gamma \vdash e \in \tau_i}{\Gamma \vdash e \in \Lambda[\tau_1.. \tau_n]} \quad (\text{INTER-I}) \\
\\
\frac{\Gamma \vdash e \in \tau_1 \quad \Gamma \vdash \tau_1 \leq \tau_2}{\Gamma \vdash e \in \tau_2} \quad (\text{SUB})
\end{array}$$

Remark: The one point where λ_Λ and F_Λ do not fit together perfectly smoothly is the maximal types $\top$ and Top . We might hope that these would coincide in F_Λ , but this, unfortunately, is not the case. The difference arises from the INTER-I rule of λ_Λ , which, in its nullary form, states that any term whatsoever has type $\top$. $F_\leq$ has no such rule: the only way a term e can be assigned type Top is by the rules SUB and SUB-TOP, which require that the term already have some type σ with $\sigma \leq \tau$. In other words, Top is the type of all *well-typed* terms, whereas $\top$ is the type of *all* terms. Order-theoretically, of course, the two types are equivalent (each is a subtype of the other), since each is a maximal type.

Remark: As in $F_\leq$, both term and type constants are absent from F_Λ , and programs involving them are expressed as terms with free variables whose behavior is declared in a pervasive context Γ_p . The presence of intersection types allows a greater variety of primitive subtype relations to be encoded in this way than was possible in $F_\leq$. For example, the pervasive context $\Gamma_p \equiv \text{Value} \leq \top, \text{Int} \leq \text{Value}, \text{Bool} \leq \text{Value}, \text{Bit} \leq \text{Int} \wedge \text{Bool}$ encodes a primitive relation that cannot be encoded as an $F_\leq$ context.

4 Examples

Perhaps the most useful property of programming languages with intersection types is that they allow extremely *refined* types to be given for expressions — much more refined than is possible in conventional polymorphic languages. Instead of a single description, each expression may be assigned any finite collection of descriptions, each capturing some aspect of its behavior. Since we are working with explicitly typed calculi, this requires effort from the programmer in the form of type assumptions or annotations; in general, as more effort is expended, better typings are obtained.

Many functional languages provide a primitive type `Bool` with two elements, `true` and `false`. Here we can introduce two subtypes of `Bool`, called `True` and `False` and give more exact types for the constants `true` and `false` in terms of these refinements:

```

> Bool < T,
> True < Bool,
> False < Bool;

> true : True,
> false : False;

```

The polymorphic `if` primitive can also be given a more refined type: if we know whether the value of the test lies in the type `True` or the type `False`, we can tell in advance which of the branches will be chosen.

```
> if : All A.      (True -> A -> T -> A)
>                /\ (False -> T -> A -> A)
>                /\ (Bool -> A -> A -> A);
```

(The third typing is needed here because F_λ 's types cannot express the idea that every element of `Bool` is an element of either `True` or `False`. This shortcoming, while not serious in practice, has motivated the investigation of a dual notion of *union types* [1, 45, 56].

The refinement in the types of `true`, `false`, and `if` can now be exploited in typing new functions:

```
> or =
>   \x:True,False,Bool. \y:True,False,Bool.
>   for R in True,False,Bool.
>   if [R] x true y;
or : Bool->(Bool->Bool/\True->True)
    /\ False->False->False
    /\ True->Bool->True
```

Indeed, we can carry out the same construction in the *pure* calculus with no assumptions about predefined types or constants, using a generalization of the familiar Church encoding of booleans in the λ -calculus [25, 7, 53, 42, 17, 16, 62, 58, 55].

```
> True == All B. All TT<B. All FF<B. TT -> T -> TT,
> False == All B. All TT<B. All FF<B. T -> FF -> FF,
> Bool == All B. All TT<B. All FF<B. TT -> FF -> B;

> true = \B. \TT<B. \FF<B. \x:TT. \y:T. x,
> false = \B. \TT<B. \FF<B. \x:T. \y:FF. y;
true : True
false : False

> or = for M in True,False,Bool.
>     for N in True,False,Bool.
>     \m:M. \n:N.
>     m [Bool] [True] [N] true n;
```

5 Subtyping Algorithm

Next we give a semi-decision procedure for the F_λ subtype relation. This relation can be shown to be undecidable [55, 57], so a semi-decision procedure is the best we can hope for; however, the same algorithm forms a decision procedure for a large fragment of F_λ , believed to contain all but some extremely pathological terms. The algorithm presented here generalizes one described by Reynolds for deciding the subtype relation of Forsythe [personal communication, 1988].

Because of the axioms SUB-DIST-IA and SUB-DIST-IQ, we cannot check whether $\Gamma \vdash \sigma \leq \tau$ just by comparing the outermost constructors of σ and τ and making recursive calls. For example,

$$\alpha \leq T, \beta \leq T \vdash \alpha \rightarrow \alpha \leq \beta \rightarrow T,$$

is derivable (using SUB-DIST-IA and SUB-TRANS, with intermediate type T), although

$$\alpha \leq T, \beta \leq T \vdash \beta \leq \alpha$$

is not.

So, given Γ , σ , and τ , our algorithm first performs a complete analysis of the structure of τ . Whenever τ has the form $\tau_1 \rightarrow \tau_2$ or $\forall \alpha \leq \tau_1. \tau_2$, it pushes the left-hand side — τ_1 or $\alpha \leq \tau_1$ — onto a queue of pending left-hand sides and proceeds recursively with the analysis of τ_2 . When τ has the form of an intersection, it calls itself recursively on each of the elements. When τ has finally been reduced to a type variable, the algorithm begins analyzing σ , matching left-hand sides of arrow and polymorphic types against the queue of pending left-hand sides from τ . In the base case, when both σ and τ have been reduced to variables, the algorithm first checks whether they are identical; if so, and if the queue of pending left-hand sides is empty, the algorithm immediately returns *true*. Otherwise, the variable σ is replaced by its upper bound from Γ and the analysis continues as before.

Definition: Let X be a finite sequence of elements of the set

$$\{\tau \mid \tau \text{ a type}\} \cup \{\alpha \leq \tau \mid \alpha \text{ a type variable and } \tau \text{ a type}\}.$$

Define the type $X \Rightarrow \tau$ as follows:

$$\begin{aligned} [] \Rightarrow \tau &= \tau \\ [\sigma, X] \Rightarrow \tau &= \sigma \rightarrow (X \Rightarrow \tau) \\ [\alpha \leq \sigma, X] \Rightarrow \tau &= \forall \alpha \leq \sigma. (X \Rightarrow \tau). \end{aligned}$$

Remark: Note that every type τ has either the form $X \Rightarrow \alpha$ or the form $X \Rightarrow \bigwedge[\tau_1.. \tau_n]$, for a unique X .

Definition: The 4-place relation $\Gamma \vdash \sigma \leq X \Rightarrow \tau$ is the least relation closed under the following rules. (Note that these rules are *syntax-directed* — at most one of them can be used to establish a given conclusion.)

$$\begin{aligned} &\frac{\text{for all } i, \Gamma \vdash \sigma \leq X \Rightarrow \tau_i}{\Gamma \vdash \sigma \leq X \Rightarrow \bigwedge[\tau_1.. \tau_n]} && (\text{ASUBR-INTER}) \\ &\frac{\text{for some } i, \Gamma \vdash \sigma_i \leq X \Rightarrow \alpha}{\Gamma \vdash \bigwedge[\sigma_1.. \sigma_n] \leq X \Rightarrow \alpha} && (\text{ASUBL-INTER}) \\ &\frac{\Gamma \vdash \tau_1 \leq [] \Rightarrow \sigma_1 \quad \Gamma \vdash \sigma_2 \leq X_2 \Rightarrow \alpha}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 \leq [\tau_1, X_2] \Rightarrow \alpha} && (\text{ASUBL-ARROW}) \\ &\frac{\Gamma \vdash \tau_1 \leq [] \Rightarrow \sigma_1 \quad \Gamma, \beta \leq \tau_1 \vdash \sigma_2 \leq X_2 \Rightarrow \alpha}{\Gamma \vdash \forall \beta \leq \sigma_1. \sigma_2 \leq [\beta \leq \tau_1, X_2] \Rightarrow \alpha} && (\text{ASUBL-ALL}) \\ &\frac{}{\Gamma \vdash \alpha \leq [] \Rightarrow \alpha} && (\text{ASUBL-REFL}) \\ &\frac{\Gamma \vdash \Gamma(\beta) \leq X \Rightarrow \alpha}{\Gamma \vdash \beta \leq X \Rightarrow \alpha} && (\text{ASUBL-TVAR}) \end{aligned}$$

Definition: The more convenient three-place relation $\Gamma \vdash \sigma \leq \tau$ may be defined as

$$\Gamma \vdash \sigma \leq \tau \quad \text{iff} \quad \Gamma \vdash \sigma \leq X \Rightarrow \phi,$$

where $\tau \equiv X \Rightarrow \phi$ and either $\phi \equiv \wedge[\phi_1.. \phi_n]$ or $\phi \equiv \alpha$.

Theorem: [Equivalence of ordinary and syntax-directed subtyping] A statement $\Gamma \vdash \sigma \leq \tau$ is derivable from the original F_λ subtyping rules iff it is derivable from the rules defining the algorithm.

6 Type Synthesis Algorithm

We can now state an algorithm for synthesizing types for F_λ terms. Given a term e and a context Γ , the algorithm constructs a *minimal type* σ for e under Γ — that is, a type σ such that $\Gamma \vdash e \in \sigma$, and such that any other type that can be derived for e is a supertype of σ .

The algorithm can be explained by separating the typing rules of F_λ into two sets: the *structural* or *syntax-directed* rules VAR, ARROW-E, ALL-I, ALL-E, and FOR, whose applicability depends on the form of e , and the non-structural rules INTER-I and SUB, which can be applied without regard to the form of e . The non-structural rules are then removed from the system and their possible effects accounted for by modifying the structural rules VAR, ARROW-E, ALL-E, and FOR appropriately.

The main novel source of difficulty here is the application rules ARROW-E and ALL-E. An application $(e_1 \ e_2)$ in the original system has *every* type τ_2 such that e_1 can be shown to have some type $\tau_1 \rightarrow \tau_2$ and e_2 can be shown to have type τ_1 , where the rule SUB may be used on both sides to promote the types of e_1 and e_2 to supertypes with appropriate shapes. For example, if

$$\begin{aligned} e_1 &\in (\sigma_1 \rightarrow \sigma_2) \wedge (\forall \alpha \leq \sigma_3. \sigma_4) \wedge (\sigma_5 \rightarrow \sigma_6) \wedge (\sigma_7 \rightarrow \sigma_8) \\ e_2 &\in \sigma_1 \wedge (\forall \alpha \leq \sigma_3. \sigma_4) \wedge \sigma_5, \end{aligned}$$

then

$$(e_1 \ e_2)$$

has both types σ_2 and σ_6 , and hence (by INTER-I) also type $\sigma_2 \wedge \sigma_6$.

To deal with this flexibility deterministically, we observe that the set of supertypes of $(\sigma_1 \rightarrow \sigma_2) \wedge (\forall \alpha \leq \sigma_3. \sigma_4) \wedge (\sigma_5 \rightarrow \sigma_6) \wedge (\sigma_7 \rightarrow \sigma_8)$ that have the appropriate shape to appear as the type of e_1 in an instance of ARROW-E can be characterized finitely:

$$\begin{aligned} &\text{arrowbasis}((\sigma_1 \rightarrow \sigma_2) \wedge (\forall \alpha \leq \sigma_3. \sigma_4) \wedge (\sigma_5 \rightarrow \sigma_6) \wedge (\sigma_7 \rightarrow \sigma_8)) \\ &= [\sigma_1 \rightarrow \sigma_2, \sigma_5 \rightarrow \sigma_6, \sigma_7 \rightarrow \sigma_8]. \end{aligned}$$

It is then a simple matter to characterize the possible types for $(e_1 \ e_2)$ by checking whether the minimal type of e_2 is a subtype of each left-hand side in the arrow basis of the minimal type of e_1 . Type applications are handled similarly.

Definition: The functions arrowbasis_Γ and allbasis_Γ are defined as follows:

$$\begin{aligned} \text{arrowbasis}_\Gamma(\alpha) &= \text{arrowbasis}_\Gamma(\Gamma(\alpha)) \\ \text{arrowbasis}_\Gamma(\tau_1 \rightarrow \tau_2) &= [\tau_1 \rightarrow \tau_2] \\ \text{arrowbasis}_\Gamma(\forall \alpha \leq \tau_1. \tau_2) &= [] \\ \text{arrowbasis}_\Gamma(\wedge[\tau_1.. \tau_n]) &= \text{arrowbasis}_\Gamma(\tau_1) * \dots * \text{arrowbasis}_\Gamma(\tau_n) \\ \text{allbasis}_\Gamma(\alpha) &= \text{allbasis}_\Gamma(\Gamma(\alpha)) \\ \text{allbasis}_\Gamma(\tau_1 \rightarrow \tau_2) &= [] \\ \text{allbasis}_\Gamma(\forall \alpha \leq \tau_1. \tau_2) &= [\forall \alpha \leq \tau_1. \tau_2] \\ \text{allbasis}_\Gamma(\wedge[\tau_1.. \tau_n]) &= \text{allbasis}_\Gamma(\tau_1) * \dots * \text{allbasis}_\Gamma(\tau_n). \end{aligned}$$

Definition: The *type synthesis* relation $\Gamma \vdash e \in \tau$ is the least relation closed under the following (syntax-directed) rules:

$$\begin{array}{c}
\Gamma \vdash x \in \Gamma(x) \quad (\text{A-VAR}) \\
\\
\frac{\Gamma, x:\tau_1 \vdash e \in \tau_2}{\Gamma \vdash \lambda x:\tau_1. e \in \tau_1 \rightarrow \tau_2} \quad (\text{A-ARROW-I}) \\
\\
\frac{\Gamma \vdash e_1 \in \sigma_1 \quad \Gamma \vdash e_2 \in \sigma_2}{\Gamma \vdash e_1 e_2 \in \bigwedge[\psi_i \mid (\phi_i \rightarrow \psi_i) \in \text{arrowbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \sigma_2 \leq \phi_i]} \quad (\text{A-ARROW-E}) \\
\\
\frac{\Gamma, \alpha \leq \tau_1 \vdash e \in \tau_2}{\Gamma \vdash \Lambda \alpha \leq \tau_1. e \in \forall \alpha \leq \tau_1. \tau_2} \quad (\text{A-ALL-I}) \\
\\
\frac{\Gamma \vdash e \in \sigma_1}{\Gamma \vdash e[\tau] \in \bigwedge[\{\tau/\alpha\}\psi_i \mid (\forall \alpha \leq \phi_i. \psi_i) \in \text{allbasis}_\Gamma(\sigma_1) \text{ and } \Gamma \vdash \tau \leq \phi_i]} \quad (\text{A-ALL-E}) \\
\\
\frac{\text{for all } i, \Gamma \vdash \{\sigma_i/\alpha\}e \in \tau_i}{\Gamma \vdash \text{for } \alpha \text{ in } \sigma_1.. \sigma_n. e \in \bigwedge[\tau_1.. \tau_n]} \quad (\text{A-FOR})
\end{array}$$

Theorem: [Soundness] If $\Gamma \vdash e \in \tau$ is derived by this algorithm, then ittt can also be derived from the original F_Λ typing rules.

Theorem: [Minimal typing] If $\Gamma \vdash e \in \sigma$ is derived by the algorithm and $\Gamma \vdash e \in \tau$ can be derived from the original typing rules, then $\Gamma \vdash \sigma \leq \tau$.

Remark: The type synthesis algorithm calls the subtyping algorithm, so it, too, may diverge on some inputs. But it is easy to show that this is the *only* source of possible nontermination during typechecking. We can thus be fairly confident that, in practice, the typechecking algorithm is well behaved as to termination.

7 Semantics

In [55], a straightforward untyped semantics is given for F_Λ , based on Bruce and Longo's partial equivalence relation model for $F_\leq$ [5]. The $\bigwedge$ type constructor is interpreted here as intersection in the model, and the bounded quantifier is interpreted as an infinite intersection.

More refined models have been given for intersections (c.f. [63]) interpreting $\bigwedge$ as a *limit* in the semantic category. A partial account of F_Λ can be given along these lines. [55] defines a translation of F_Λ typing derivations into the pure second-order λ -calculus with surjective pairing, system F_x . This style of presentation avoids some of the subtleties involved in giving a direct denotational semantics for F_Λ , since F_x itself has many well-studied models, but it still yields a useful soundness theorem relating the semantics to the F_Λ type system: valid F_Λ typing derivations are translated to well-typed (and hence well-behaved) F_x terms. An equational theory of provable equivalences between terms of pure F_Λ is also given; this theory can be shown to be sound for both the untyped semantics and the translation semantics. However, the details of both constructions — in particular, attempted proofs of the important property of *coherence* — run afoul of an important negative technical result about F_Λ 's subtype relation: the nonexistence of syntactic least upper bounds.

8 Conclusions

F_λ is based on three elegant and appealing notions of typing — subtyping, finitary polymorphism, and bounded parametric polymorphism — and combines them essentially orthogonally so that the programming idioms of all the components are fully supported.

To fully understand F_λ 's role as a core type system for practical programming languages, however, more examples illustrating its application to real programming problems will be required. In particular, none of the known examples (c.f. [55]) offer convincing evidence that bounded polymorphism is more useful than ordinary unbounded polymorphism in the presence of intersection types. Another concern is the efficiency of typechecking for larger examples. Naive implementations of the algorithms given here exhibit exponential behavior — in practice — in both type synthesis (because of the *for* construct) and subtyping (because of rules ASUBR-INTER and ASUBL-INTER). Fortunately, this behavior normally occurs as a result of explicit programmer directives — requests, in effect, for an exponential amount of analysis of the program during typechecking. Still, a serious implementation must find ways to economize — for example, by caching the partial results of previous analysis. A third consideration for any language based on second-order polymorphism is the problem of verbosity. Without some means of abbreviation (partial type reconstruction), even modest programs quickly become overburdened with type annotations. We have ignored this set of issues here, since it is not well understood even for pure F_Σ .

Acknowledgements

I am grateful for illuminating discussions with Luca Cardelli, Tim Freeman, Nico Habermann, Bob Harper, QingMing Ma, Frank Pfenning, and John Reynolds.

References

- [1] Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In Ito and Meyer [48], pages 651–674.
- [2] H. 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.
- [3] Val Breazu-Tannen, Thierry Coquand, Carl Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.
- [4] Kim B. Bruce. The equivalence of two semantic definitions for inheritance in object-oriented languages. In *Proceedings of Mathematical Foundations of Programming Semantics*, Pittsburgh, PA, March 1991. To appear.
- [5] Kim B. Bruce and Giuseppe Longo. A modest model of records, inheritance, and bounded quantification. *Information and Computation*, 87:196–240, 1990. An earlier version appeared in the proceedings of the IEEE Symposium on Logic in Computer Science, 1988.
- [6] Kim 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*, Albuquerque, NM, January 1992. To appear.

- [7] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ -programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
- [8] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Object Oriented Programming: Systems, Languages, and Applications (Conference Proceedings)*, pages 457–467, 1989.
- [9] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.
- [10] Peter Canning, Walt Hill, and Walter Olthoff. A kernel language for object-oriented programming. Technical Report STL-88-21, Hewlett-Packard Labs, 1988.
- [11] Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.
- [12] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
- [13] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988.
- [14] Luca Cardelli. Typeful programming. Research Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1989.
- [15] Luca Cardelli. Extensible records in a pure calculus of subtyping. To appear, 1991.
- [16] Luca Cardelli. F-sub, the system. Unpublished manuscript, July 1991.
- [17] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest: (Extended abstract). In *ACM Conference on Lisp and Functional Programming*, pages 30–43, Nice, France, June 1990. Extended version available as DEC SRC Research Report 55, Feb. 1990.
- [18] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In Ito and Meyer [48], pages 750–770.
- [19] Luca Cardelli and John Mitchell. Operations on records (summary). In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of Fifth International Conference on Mathematical Foundations of Programming Language Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 22–52, Tulane University, New Orleans, March 1989. Springer Verlag. To appear in *Mathematical Structures in Computer Science*; also available as DEC Systems Research Center Research Report #48, August, 1989.
- [20] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.
- [21] Felice Cardone. Relational semantics for recursive types and bounded quantification. In *Proceedings of the Sixteenth International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 164–178, Stresa, Italy, July 1989. Springer-Verlag.

- [22] Felice Cardone and Mario Coppo. Two extensions of Curry's type inference system. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, number 31 in APIC Studies in Data Processing, pages 19–76. Academic Press, 1990.
- [23] Robert Cartwright and Mike Fagan. Soft typing. Submitted to PLDI '91, November 1990.
- [24] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [25] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [26] 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.
- [27] M. Coppo and M. Dezani-Ciancaglini. A new type-assignment for λ -terms. *Archiv. Math. Logik*, 19:139–156, 1978.
- [28] 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.
- [29] M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In G. Lolli, G. Longo, and A. Marja, editors, *Logic Colloquium 82*, pages 241–262, Amsterdam, 1983. North-Holland.
- [30] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and lambda calculus semantics. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 535–560, New York, 1980. Academic Press.
- [31] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 27:45–58, 1981.
- [32] M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type theories, normal forms and D_∞ -lambda-models. *Information and Computation*, 72:85–116, 1987.
- [33] M. Coppo, M. Dezani, and P. Sallé. Functional characterization of some semantic equalities inside λ -calculus. Number 81 in *Lecture Notes in Computer Science*, pages 133–146. Springer-Verlag, 1979.
- [34] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In *ESOP '92?*, 1992.
- [35] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. *Mathematical Structures in Computer Science*, 1991. To appear.
- [36] Pierre-Louis Curien and Giorgio Ghelli. Subtyping + extensionality: Confluence of $\beta\eta$ -reductions in F_ω . In Ito and Meyer [48], pages 731–749.
- [37] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
- [38] M. Dezani-Ciancaglini and I. Margaria. F-semantics for intersection type discipline. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 279–300. Springer-Verlag, 1984.

- [39] Mariangiola Dezani-Ciancaglini and Ines Margaria. A characterisation of F -complete type assignments. *Theoretical Computer Science*, 45:121–157, 1986.
- [40] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages*. PhD thesis, Rice University, December 1990.
- [41] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*. ACM Press, June 1991.
- [42] Giorgio Ghelli. *Proof Theoretic Studies about a Minimal Type System Integrating Inclusion and Parametric Polymorphism*. PhD thesis, Università di Pisa, March 1990. Technical report TD-6/90, Dipartimento di Informatica, Università di Pisa.
- [43] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.
- [44] Susumu Hayashi. Singleton, union and intersection types for program extraction. In Ito and Meyer [48], pages 701–730.
- [45] Susumu Hayashi and Yukihide Takayama. Extended projection method with Kreisel-Troelstra realizability. Submitted to *Information and Computation*, 1990.
- [46] J. R. Hindley. The simple semantics for Coppo-Dezani-Sallé types. In Dezani-Ciancaglini and Montanari, editors, *Proceedings of the International Symposium on Programming*, pages 212–226. Springer-Verlag, 1982. Lecture Notes in Computer Science No. 137.
- [47] J. Roger Hindley. Coppo-Dezani-Sallé types in lambda-calculus, an introduction. Draft manuscript, February 1989.
- [48] T. Ito and A. R. Meyer, editors. *Theoretical Aspects of Computer Software (Sendai, Japan)*, number 526 in Lecture Notes in Computer Science. Springer-Verlag, September 1991.
- [49] Bart Jacobs, Ines Margaria, and Maddalena Zacchi. Expansion and conversion models in the lambda calculus from filters with polymorphic types. Manuscript, March 1989.
- [50] Daniel Leivant. Discrete polymorphism (summary). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 288–297, 1990.
- [51] Simone Martini. Bounded quantifiers have interval models. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 174–183, Snowbird, Utah, July 1988. ACM.
- [52] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.
- [53] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics, Tulane University, New Orleans, Louisiana*, pages 209–228. Springer-Verlag LNCS 442, March 1989. Also available as Ergo Report 88-069, School of Computer Science, Carnegie Mellon University.
- [54] Benjamin Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report CMU-CS-89-169, School of Computer Science, Carnegie Mellon University, September 1989.

- [55] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, December 1991. Available from CMU School of Computer Science as CMU-CS-91-205.
- [56] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, February 1991.
- [57] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, (Albuquerque, New Mexico)*, January 1992. To appear. Extended version available as CMU School of Computer Science technical Report CMU-CS-91-161.
- [58] Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, March 1989.
- [59] Garrell Pottinger. A type assignment for the strongly normalizable λ -terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 561–577. Academic Press, New York, 1980.
- [60] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.
- [61] J. C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.
- [62] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.
- [63] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.
- [64] John C. Reynolds. The coherence of languages with intersection types. In Ito and Meyer [48], pages 675–700.
- [65] Simona Ronchi della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59:181–209, 1988.
- [66] S. Ronchi della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.
- [67] P. Sallé. Une extension de la theorie des types en λ -calcul. pages 398–410. Springer-Verlag, 1982. Lecture Notes in Computer Science No. 62.
- [68] Steffen van Bakel. Principal type schemes for the strict type assignment system. Technical report 91-6, University of Nijmegen, 1991.
- [69] Steffen van Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 99, 1992. To appear.
- [70] Hirofumi Yokouchi. Relationship between polymorphic types and intersection types (extended abstract). Unpublished manuscript, December 1990.