Programming With Intersection Types, Union Types, and Polymorphism

Benjamin C. Pierce

February 5, 1991
CMU-CS-91-106

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Type systems based on *intersection types* have been studied extensively in recent years, both as tools for the analysis of the pure λ-calculus and, more recently, as the basis for practical programming languages. The dual notion, *union types*, also appears to have practical interest. For example, by refining types ordinarily considered as atomic, union types allow a restricted form of abstract interpretation to be performed during typechecking. The addition of second-order *polymorphic types* further increases the power of the type system, allowing interesting variants of many common datatypes to be encoded in the "pure" fragment with no type or term constants.

This report summarizes a preliminary investigation of the expressiveness of a programming language combining intersection types, union types, and polymorphism.

This research was supported in part by the Office of Naval Research and in part by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract N00014-84-K-0415, ARPA Order No. 5404.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, DARPA or the U.S. government.

Keywords: Lambda calculus and related systems, Language theory, Programming, Type structure, Data types and structures, Polymorphism, Intersection types, Union types.

1 Introduction

This report describes the preliminary results of an investigation of a typed λ-calculus combining intersection types, union types, and second-order polymorphism. Although the proof-theoretic and model-theoretic properties of this calculus have not yet been investigated in detail, a number of interesting observations can be made about its expressive power. Specifically, the report presents:

1. A formulation of the notion of union types in a programming-language setting.
2. Encodings of several common inductive datatypes showing how intersections, unions, and polymorphic types together allow a restricted form of abstract interpretation to be performed by the typechecker.
3. A novel treatment of finitary polymorphism in terms of an explicit finitary quantifier.
4. A comparison of two different ways of combining polymorphism and intersection types, with one formulation, based on ordinary universal quantification, worked out in detail. The principal alternative, based on bounded quantification, is discussed in Section 5.2 and in more detail in [33].

The remainder of the report is organized as follows. Section 2 sketches some background to the research described here and outlines related work by others. Section 3 defines a calculus $\lambda(\forall, \wedge, \vee)$ with intersection types, union types, and polymorphic types, gives typing rules for expressions, and illustrates these rules with some small examples. Section 4 presents several larger examples. Section 5 discusses alternatives in the definition of the calculus. Section 6 outlines paths for future research.

2 Background and Related Work

The work described in this report builds on previous studies of two well-known calculi: the intersection type discipline and the polymorphic λ-calculus.

2.1 Intersection Types

Intersection types in the pure λ-calculus have been extensively studied by researchers at the university of Turin and elsewhere [14, 13, 15, 2, 39, 26, 10]. More recently, Reynolds has showed how intersection types can be used as the basis for the type system of a practical programming language, called Forsythe [38]. The core Forsythe type system can be viewed as consisting of the following components:

1. a collection of primitive types and, for each pair of types σ and `tau`, a function type $\sigma \rightarrow \tau$, as in the simply typed lambda calculus [12];
2. a preorder (the "subtype relation") on the primitive types, which is extended to a preorder on the entire set of type expressions;
3. a rule of "subsumption" stating that the type of a term may be promoted to any supertype;
4. a type constructor $\wedge$ (intersection) yielding meets in the subtype ordering, along with appropriate subtyping laws for distributing intersections and arrows and a typing rule for introducing intersection types.

The calculi discussed in this report retain this basic structure, extending it with

1. a type constructor $\vee$ (union) yielding joins in the type preorder, with appropriate distributive laws and typing rules; and
2. second-order polymorphism.

A few ideas mentioned in this report are directly applicable to Forsythe itself: the notion of finitary polymorphism (Sections 2.4 and 4.6) and the observation that the Forsythe type system can express types for procedures with default parameters (Section 4.7).

2.2 Union Types

The dual notion, union types, is a very natural one, and appears to have occurred to a number of researchers working independently in surprisingly disparate contexts. The idea arises in several different ways:

1. as the dual of intersection types [1, 18, 38, and this report];
2. from logical or semantic considerations [25];
3. as a generalization of disjoint unions or variant records [11, 17].

Intuitively, union types stand in the same relation to disjoint union types (also called sum types or variant records) as ordinary set-theoretic union does to set-theoretic disjoint union.[1] Operationally:

1. The injections $\mathtt{inl} : \tau_1 \to (\tau_1 + \tau_2)$ and $\mathtt{inr} : \tau_2 \to (\tau_1 + \tau_2)$ are replaced by *implicit* coercions represented by the subtyping laws $\tau_1 \leq (\tau_1 \vee \tau_2)$ and $\tau_2 \leq (\tau_1 \vee \tau_2)$.
2. Whereas each element of $\tau_1 + \tau_2$ contains a tag indicating which of the two summands it comes from, elements of the union type $\tau_1 \vee \tau_2$ are untagged: the only operations that can be applied to values of type $\tau_1 \vee \tau_2$ are those that make sense for *both* τ_1 and τ_2.

These differences can also be seen by comparing the usual elimination rule for disjoint union types with the rule proposed here for union types:

$$\frac{\Gamma \vdash \mathtt{e} \ \mathtt{in} \ \tau_1 + \tau_2 \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}_i \in \tau}{\Gamma \vdash (\mathtt{case\ e\ of\ inl(x)} \Rightarrow \mathtt{e}_1 \mid \mathtt{inr(x)} \Rightarrow \mathtt{e}_2) \in \tau} \qquad \text{(Disj-Union-E)}$$

$$\frac{\Gamma \vdash \mathtt{e} \ \mathtt{in} \ \tau_1 \vee \tau_2 \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash (\mathtt{case\ e\ of\ x} \Rightarrow \mathtt{e}') \in \tau} \qquad \text{(Union-E)}$$

The formulation of union types used in this report is described in detail in Section 3. The rest of this subsection describes related formulations by other researchers.

In the report on the Forsythe language [38], Reynolds describes an attempt to add ordinary sum types (disjoint unions) to the language. The attempt was eventualy abandoned because sums interacted badly with the "generalized conditional" construct to produce a failure of confluence for the operational semantics.

[1] More formally, unions should probably be interpreted in a category-theoretic model as pushouts, by analogy with Reynolds' interpretation of intersections in Forsythe as pullbacks [38]. The details of the construction are problematic.

MacQueen, Plotkin, and Sethi, in their work on ideal models of polymorphic types [28], introduce a notion of union types with a rule for union elimination that is essentially the same as the alternative form of the UNION-E rule given in Section 5.3:

$$\frac{\Gamma \vdash \mathtt{e\ in}\ \tau_1 \vee \tau_2 \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{e}' \in \tau} \qquad \text{(UNION-E')}$$

(The metanotation $[\mathtt{e}/\mathtt{x}]\mathtt{e}'$ denotes the capture-avoiding substitution of $\mathtt{e}$ for free occurrences of $\mathtt{x}$ in $\mathtt{e}'$.) Their system does not include an explicit notion of subtyping or a rule of subsumption, so its proof-theoretic properties involving intersections and unions are simpler than those for the other systems discussed here.

Barbanera and Dezani-Ciancaglini [1], working in a Curry-style type assignment system, have studied a formulation of union types much more similar to the one presented here. However, their system includes two distributivity axioms not present in the one described in this report:

$$\sigma \wedge (\tau_1 \vee \tau_2) \leq (\sigma \wedge \tau_1) \vee (\sigma \wedge \tau_2)$$

$$(\sigma \vee \tau_1) \wedge (\sigma \vee \tau_2) \leq \sigma \vee (\tau_1 \wedge \tau_2)$$

Formal properties of this system are proved using a notion of *large basis* and a different formulation of the union elimination rule:

$$\frac{\Gamma, \mathtt{e} : \sigma \vdash \mathtt{e}' \in \tau \qquad \Gamma \vdash \mathtt{e} \in \sigma}{\Gamma \vdash \mathtt{e}' \in \tau}$$

Barbanera and Dezani-Ciancaglini's long-term interests in union types are directed toward an investigaton of infinitary union types, along the lines suggested by Leivant's work on infinitary intersections [27]. In their paper they present several intruguing examples of encodings of algebraic datatypes, including the *Berarducci numerals*, which can be shown to have no uniform type in the second-order polymorphic λ-calculus.

Fagan and Cartwright [11, 17] have developed an extension of the ML type system [20, 30] that includes both recursive types and a notion of "true union" of disjoint types. This is not quite the same as the union types described in this report, since for Fagan and Cartwright $\sigma \vee \tau$ is defined only when σ and τ have different outermost type constructors. Their system has a decidable type reconstruction problem but lacks principal types.

Freeman and Pfenning [18] describe a type system incorporating a variant of intersection and union types with the restriction that all conjuncts and disjuncts must have the same functionality. They propose a variant of Standard ML where an ML type is derived for an expression $\mathtt{e}$ and then *refined* by performing an abstract interpretation of $\mathtt{e}$ with respect to the finite lattice of restricted intersection and union types lying beneath it.[2]

Hayashi and Takayama [23, 25] propose a logic with a new existential quantifier whose realizability interpretation has the same form as a union type. The elimination form for this quantifier is rather limited, however [24].

[2]The notion of union types used by Freeman and Pfenning was inspired by an early draft of the present report. Conversely, the idea of using intersection and union types to perform a restricted form of abstract interpretation during typechecking, used in several of the examples in Section 4, is based on early descriptions of their work.

2.3 Polymorphism

The second-order polymorphic λ-calculus was originally developed by Girard [19] and Reynolds [36] and has since been a topic of significant research from both theoretical and pragmatic perspectives. For present purposes, two lines of work are particularly relevant:

1. Methods for encoding inductive datatype definitions as types in the pure polymorphic λ-calculus [4, 31, 32, 35, 36, 37]. These encodings form the basis for most of the examples presented here.

2. Type systems combining polymorphism with an order structure on the set of types.

The first calculus to combine polymorphism and subtyping was Cardelli and Wegner's Bounded Fun [8]. Bounded Fun has been studied fairly extensively in its own right [3, 9] and has been incorporated into various proposals for programming languages with subtyping and polymorphism [5, 6].

Curien and Ghelli [16] have studied a minimal formulation of a second-order λ-calculus with bounded quantification called $\mathtt{F}_{\leq}$ ("F-sub"), containing only pure type and term constructors. Cardelli and others have extended the usual encodings of common inductive dataypes to encodings in $\mathtt{F}_{\leq}$ [7] that take account of the order structure on types in interesting ways; these encodings have much in common with the examples shown in Section 4.

One of the goals of this report is to formulate a type system combining polymorphism with intersection types. As far as I know, this combination has not yet been satisfactorily achieved in a programming-language context, though some preliminary work along these lines appears in [33, 34]; a more theoretical analysis of the combination has been carried out by Jacobs, Margaria, and Zacchi [26].

2.4 Finitary Polymorphism

Because the type reconstruction problem for the usual formulation of intersection types is undecidable, a programming language incorporating intersections in its type system must use explicit type annotations to make the typechecking problem tractable. The solution adopted in Forsythe is to require type annotations on λ-abstractions as usual in explicitly typed λ-calculi, but to allow any finite number of types, rather than just one, to be mentioned as possible domains for the function described by the abstraction. The body of the abstraction is typechecked once for each given type and the results conjoined to form the final type of the abstraction. This maintains the decidability of typechecking, while allowing "finitely polymorphic" types to be derived for λ-abstractions.

The present report proposes a refinement of this scheme, where λ-abstractions are annotated with exactly one type and a new `for` construct is introduced to provide finitary polymorphism. Separating the two mechanisms (functional abstraction vs. typechecking an under a finite set of assumptions) has two advantages:

1. In calculi with (ordinary or bounded) universal polymorphism in addition to intersection types, a typing assumption may need to be mentioned explicitly as part of a type argument to a polymorphic function. The `for` construct provides a name (a type variable) for the assumptions it introduces. See Section 4.4.

2. Even in the fragment without the universal quantifier, the explicit `for` construct may improve typechecking efficiency. See Section 4.6.

3 Syntax

This section introduces the notational conventions used in the rest of the report and defines the concrete syntax and typing rules of $\lambda(\forall, \wedge, \vee)$, a polymorphic λ-calculus with intersection and union types.

3.1 Notational Conventions

The metavariables σ, τ, θ, ϕ, and ψ range over types; α and β range over a denumerable set of type variables; $\mathtt{e}$, $\mathtt{f}$, and $\mathtt{b}$ range over terms; $\mathtt{x}$ ranges over a denumerable set of variables.

The notation $\tau_1, \tau_2, \ldots, \tau_n$ represents a finite sequence of types indexed by the set $\{1, \ldots, n\}$, where $n \geq 0$. To save space in formulas, $\tau_1, \tau_2, \ldots, \tau_n$ is normally shortened to $\tau_1..\tau_n$. When the set that an index variable ranges over is clear from context, it is usually omitted. It is occasionally convenient to use a "sequence comprehension" notation to denote a finite sequence of types. For example,

$$\bigwedge[\bigvee[\sigma_i, \tau_i] \mid 1 \leq i \leq 3] \;=\; \bigwedge[\bigvee[\sigma_1, \tau_1],\, \bigvee[\sigma_2, \tau_2],\, \bigvee[\sigma_3, \tau_3]].$$

The sets of types and typed terms are defined by the following abstract grammars:[3]

$$\begin{array}{lcl}
\tau & ::= & \alpha \\
& \mid & \tau \to \tau' \\
& \mid & \bigwedge[\tau_1..\tau_n] \\
& \mid & \bigvee[\tau_1..\tau_n] \\
& \mid & \forall\alpha.\,\tau
\end{array}$$

$$\begin{array}{lcl}
\mathtt{e} & ::= & \mathtt{x} \\
& \mid & \lambda\mathtt{x}{:}\tau.\,\mathtt{e} \\
& \mid & \mathtt{e}\ \mathtt{e}' \\
& \mid & \Lambda\alpha.\,\mathtt{e} \\
& \mid & \mathtt{e}[\tau] \\
& \mid & \mathtt{for}\ \alpha\ \mathtt{in}\ \tau_1..\tau_n.\,\mathtt{e} \\
& \mid & \mathtt{case}\ \mathtt{x} = \mathtt{e}_1\ \mathtt{of}\ \mathtt{e}_2
\end{array}$$

A *context* is a finite sequence of pairs $\mathtt{x} : \tau$ of a variable and a type, with no variable mentioned twice. The metavariable Γ ranges over contexts. The set of variables in Γ is written $\mathrm{dom}(\Gamma)$.

$\mathrm{FV}(\mathtt{e})$ is the set of variables free in $\mathtt{e}$. $\mathrm{FTV}(\tau)$ is the set of type variables free in τ. $\mathrm{FTV}(\Gamma)$ is the set of type variables free in Γ, that is,

$$\mathrm{FTV}(\Gamma) = \bigcup_{\mathtt{x}_i:\tau_i \in \Gamma} \mathrm{FTV}(\tau_i).$$

Terms and types are identified up to renaming of bound (term and type) variables. When $\mathtt{e}$ and $\mathtt{e}'$ are the same modulo renaming of bound variables, we write $\mathtt{e} \equiv \mathtt{e}'$.

When a term $\mathtt{e}$ is substituted for a variable $\mathtt{x}$ in another term $\mathtt{e}'$, written $[\mathtt{e}/\mathtt{x}]\mathtt{e}'$, the bound variables of $\mathtt{e}'$ are first renamed to be different from the free variables of $\mathtt{e}$. Similarly, when a type τ is substituted for a type variable α in τ', the bound variables of τ' are first renamed to be different from the free variables of τ.

[3] To reduce the number of cases in the grammar and inference rules, intersection and union are formulated as n-ary type constructors rather than giving a binary and a nullary constructor for each.

The following abbreviations for common intersection and union types are useful for making complicated expressions more readable:

$$\begin{array}{lcl} \texttt{NS} & \stackrel{\text{def}}{=} & \bigwedge[\,] \\ \sigma\wedge\tau & \stackrel{\text{def}}{=} & \bigwedge[\sigma,\,\tau] \\ \texttt{VOID} & \stackrel{\text{def}}{=} & \bigvee[\,] \\ \sigma\vee\tau & \stackrel{\text{def}}{=} & \bigvee[\sigma,\,\tau]. \end{array}$$

The "nonsense" type `NS` is a maximal element in the type preorder; `VOID` is a minimal element. The expression "$\sigma\wedge\tau$" may be read as "σ intersect τ," "σ meet τ," or "σ and τ." The expression "$\sigma\vee\tau$" may be read as "σ union τ," "σ join τ," or "σ or τ."

Sessions with the prototype compiler for $\lambda(\forall,\wedge,\vee)$ are set in a typewriter font using only Ascii symbols. The mathematical symbols used in the more formal λ-calculus notation are transliterated as follows:

Ascii	TEX
`s, t`	$\sigma,\ \tau$
`'a, 'b`	$\alpha,\ \beta$
`->`	$\rightarrow$
`/\`	$\bigwedge,\ \wedge$
`\/`	$\bigvee,\ \vee$
`All 'a. t`	$\forall\alpha.\ \tau$
`\x:s. e`	λ`x:`σ`. e`
`\\'a. e`	$\Lambda\alpha.$ `e`
`<=`	$\leq$

3.2 The Subtype Relation

We define a preorder $\leq$ on the set of type expressions, where $\sigma \leq \tau$ asserts that every value in type σ is also in type τ, or, operationally, that a term of type σ may safely be used in any context where a term of type τ is expected. When $\sigma \leq \tau$ we say that σ is a *subtype* of τ and that τ is a supertype of σ. When σ and τ inhabit the same equivalence class in the subtype preorder ($\sigma \leq \tau$ and $\tau \leq \sigma$), we say that σ and τ are equivalent and write $\sigma\sim\tau$.

The rules in this section and the following one are summarized in Appendix A.

The first two rules state that $\leq$ is a preorder.

$$\tau \leq \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\sigma \leq \theta \qquad \theta \leq \tau}{\sigma \leq \tau} \qquad \text{(Sub-Trans)}$$

The subtyping rule for arrow types is covariant on the right hand side and contravariant on the left, as usual.

$$\frac{\tau_1 \leq \sigma_1 \qquad \sigma_2 \leq \tau_2}{\sigma_1\rightarrow\sigma_2 \ \leq\ \tau_1\rightarrow\tau_2} \qquad \text{(Sub-Arrow)}$$

$\forall$ is covariant in its body type.

$$\frac{\sigma \leq \tau}{\forall\alpha.\sigma \ \leq\ \forall\alpha.\tau} \qquad \text{(Sub-All)}$$

$\bigwedge[\tau_1..\tau_n]$ is a greatest lower bound of $\tau_1..\tau_n$.

$$\frac{\text{for all } i,\ \sigma \le \tau_i}{\sigma \ \le\ \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\bigwedge[\tau_1..\tau_n] \ \le\ \tau_i \qquad \text{(Sub-Inter-LB)}$$

$\bigvee[\tau_1..\tau_n]$ is a least upper bound of $\tau_1..\tau_n$.

$$\frac{\text{for all } i,\ \sigma_i \le \tau}{\bigvee[\sigma_1..\sigma_n] \ \le\ \tau} \qquad \text{(Sub-Union-L)}$$

$$\tau_i \ \le\ \bigvee[\tau_1..\tau_n] \qquad \text{(Sub-Union-UB)}$$

Arrows may be distributed over intersections on the right hand side and over unions of the left hand side. When an arrow is distributed over a union, the union changes to an intersection. Similarly, quantifiers distribute over intersections. (These rules and the two distributive laws below are actually equivalences; the other directions may be derived from the laws for $\wedge$ and $\vee$.)

$$\bigwedge[\sigma\to\tau_1 \,..\, \sigma\to\tau_n] \ \le\ \sigma\to\bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-AI)}$$

$$\bigwedge[\sigma_1\to\tau \,..\, \sigma_n\to\tau] \ \le\ \bigvee[\sigma_1..\sigma_n]\to\tau \qquad \text{(Sub-Dist-AU)}$$

$$\bigwedge[\forall\alpha.\,\tau_1 \,..\, \forall\alpha.\,\tau_n] \ \le\ \forall\alpha.\,\bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-QI)}$$

Finally,[4] intersections and unions distribute with each other.

$$\frac{\mathtt{U} \ \equiv\ [\bigvee[\tau_{1,1} \,..\, \tau_{1,m_1}] .. \bigvee[\tau_{n,1} \,..\, \tau_{n,m_n}]] \quad \mathtt{I} \ \equiv\ [\bigwedge[\tau_{1,j_1} \,..\, \tau_{n,j_n}] \mid 1 \le j_i \le m_i]}{\bigvee\mathtt{I} \ \le\ \bigwedge\mathtt{U}} \qquad \text{(Sub-Dist-IU)}$$

$$\frac{\mathtt{I} \ \equiv\ [\bigwedge[\tau_{1,1} \,..\, \tau_{1,m_1}] .. \bigwedge[\tau_{n,1} \,..\, \tau_{n,m_n}]] \quad \mathtt{U} \ \equiv\ [\bigvee[\tau_{1,j_1} \,..\, \tau_{n,j_n}] \mid 1 \le j_i \le m_i]}{\bigwedge\mathtt{U} \ \le\ \bigvee\mathtt{I}} \qquad \text{(Sub-Dist-UI)}$$

These laws may be used to transform type expressions into a "canonical form" where arrows and quantifiers have been pushed inward as far as possible, intersections distributed over unions, and most redundant conjuncts and disjuncts dropped. (For efficiency in the normalization procedure, redundant conjuncts and disjuncts are not *guaranteed* to be dropped; a simple heuristic discovers most situations where they may be.) For example:[5]

---

[4] There is one other plausible distributive law, relating quantifiers and arrows:

$$\frac{\alpha \notin \mathrm{FTV}(\tau_1)}{\forall\alpha.\ \tau_1\to\tau_2 \ \sim\ \tau_1 \to (\forall\alpha.\ \tau_2)} \qquad \text{(Sub-Dist-QA)}$$

Since its status is less clear than the other rules and it appears somewhat tricky to implement, it is omitted from the present formulation.

[5] In sessions with the prototype compiler, lines of input typed by the user are preceded by an angle bracket.

```
> normalize t1/\NS/\t1;
Normal form: t1

> normalize (s1\/s2) -> (t1/\t2);
Normal form: s1->t1 /\ s2->t1 /\ s1->t2 /\ s2->t2

> normalize t1\/(t2/\t3);
Normal form: (t1\/t2) /\ (t1\/t3)

> normalize s -> (t1/\t2);
Normal form: s->t1 /\ s->t2

> normalize s -> NS;
Normal form: NS

> normalize VOID -> t;
Normal form: NS

> normalize All 'a. 'a->(t1/\t2);
Normal form: (All 'a. 'a->t1) /\ (All 'a. 'a->t2)

> normalize All 'a. NS;
Normal form: NS

> normalize s -> (t1\/(t2/\t3));
Normal form: s->(t1\/t2) /\ s->(t1\/t3)

> normalize (s1/\(s2\/s3)) -> t;
Normal form: (s1/\s2)->t /\ (s1/\s3)->t
```

3.3 Well-Typed Terms

We can now define the set of well typed $\lambda(\forall, \wedge, \vee)$ terms.

The SUBSUMPTION rule states that a type that has been derived for a term may be promoted, as desired, to any supertype.

$$\frac{\Gamma \vdash \mathtt{e} \in \sigma \qquad \sigma \leq \tau}{\Gamma \vdash \mathtt{e} \in \tau} \qquad \text{(SUBSUMPTION)}$$

(The claim that this rule is valid essentially amounts to asserting that the subtype relation is correctly defined.)

The intersection introduction rule states that if all of the types $\tau_1..\tau_n$ have been derived for an expression $\mathtt{e}$, then $\bigwedge[\tau_1..\tau_n]$ may also be derived.

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

No elimination rule will be needed for intersection types; its effect is already provided by the SUBSUMPTION rule. For the same reason, no introduction rule is needed for union types.

The rules for typing variables, abstractions, and applications are the usual ones.

$$\Gamma_1, \mathtt{x} : \tau, \Gamma_2 \vdash \mathtt{x} \in \tau \qquad \text{(VAR)}$$

$$\frac{\Gamma, \mathtt{x} : \sigma \vdash \mathtt{e} \in \tau}{\Gamma \vdash \lambda \mathtt{x}{:}\sigma.\, \mathtt{e} \in \sigma \to \tau} \qquad \text{(ARROW-I)}$$

$$\frac{\Gamma \vdash \mathtt{f} \in \sigma \to \tau \qquad \Gamma \vdash \mathtt{e} \in \sigma}{\Gamma \vdash \mathtt{f}\ \mathtt{e} \in \tau} \qquad \text{(ARROW-E)}$$

To apply the ARROW-E rule, the rule of SUBSUMPTION may need to be applied to the types of the function `f` and/or the argument `e` so that the type of `e` precisely matches the domain type of `f`:[6]

```
> \f:s1->t. \x:s1/\s2. f x;
it : (s1->t) -> (s1/\s2) -> t
```

If `f` has an intersection type, ARROW-E may be applied more than once to give different types for (`f x`), which may then be conjoined using INTER-I:[7]

```
> \f:s1->t1/\s2->t2. \x:s1/\s2. f x;
it :
     (s1->t1/\s2->t2)->(s1/\s2)->t1
  /\ (s1->t1/\s2->t2)->(s1/\s2)->t2
```

If `f` has an intersection type, some of whose conjuncts are not arrow types, then the application rule is not used for these conjuncts. In the result type of the application (`f x`), these applications simply disappear:

```
> \f:s1/\s2->t. \x:s2. f x;
it : (s1/\s2->t) -> s2 -> t
```

Similarly, if `x` has an intersection type, some of whose conjuncts are not subtypes of any domain type of `f`, the ARROW-E rule fails to apply for these conjuncts and they disappear in the result type:

```
> \f:s1->t. \x:s1/\s2. f x;
it : (s1->t) -> (s1/\s2) -> t
```

Together, these observations illustrate some of the power of the conjunctive typing discipline by allowing a restricted form of self-application:

```
> \f:s/\s->t. f f;
it : (s/\s->t) -> t
```

If `f` and `x` fail to have any supertypes to which the UNION-E rule is applicable, the minimal type of the application is `NS`:

---

[6] To reduce clutter in some of the examples, primitive ("built-in") types like `s1` are used in place of type variables. Their formal treatment is discussed in Section 5.1.

By convention, when an "anonymous" term is presented to the compiler's read-eval-print loop, it is assigned the name `it` so that it can be referred to again.

[7] Arrow binds more tightly than intersection or union.

```
> \f:s1. \x:s2. f x;
it : NS
```

Indeed, *every* $\lambda(\forall, \wedge, \vee)$ term has type `NS`. The `NS` type in this calculus (as in Forsythe) corresponds both to complete lack of information about a term's runtime behavior and to typechecking failure.

The rules for type abstractions and applications are standard.

$$\frac{\Gamma \vdash \mathtt{e} \in \tau \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash \Lambda\alpha.\mathtt{e} \in \forall\alpha.\tau} \qquad \text{(All-I)}$$

$$\frac{\Gamma \vdash \mathtt{e} \in \forall\alpha.\tau}{\Gamma \vdash \mathtt{e}[\sigma] \in [\sigma/\alpha]\tau} \qquad \text{(All-E)}$$

Similar observations to the ones above for the Union-E also apply to All-E:

```
> (\\'a. \x:'a. x) [s];
it : s -> s

> \f: (All 'a. 'a->'a). \x:s. f [s] x;
it : (All 'a. 'a->'a) -> s -> s

> f : (All 'a. 'a->(t1/\t2));
f : (All 'a. 'a->t1) /\ (All 'a. 'a->t2)

> f [s];
it : s->t1 /\ s->t2

> f : (All 'a. 'a->(t1/\t2)) /\ s->t;
f : (All 'a. 'a->t1) /\ (All 'a. 'a->t2) /\ s->t

> f [u];
it : u->t1 /\ u->t2

> \x:s. f x;
it : s -> t
```

The union elimination rule is probably the most novel element of the calculi investigated in this report.

$$\frac{\Gamma \vdash \mathtt{e\ in} \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{case\ e\ of\ x} \Rightarrow \mathtt{e}' \in \tau} \qquad \text{(Union-E)}$$

Operationally, the rule reads as follows: if an expression `e` has type $\bigvee[\tau_1..\tau_n]$, and if `e'` is an expression with a free variable `x`, such that `e'` has type τ under the assumption that `x` has type τ_i for *every* i, then `case x = e of e'` has type τ as well.

For example, if

```
> f : s1->t/\s2->t;
f : s1->t /\ s2->t
```

and

```
> e : s1\/s2;
e : s1 \/ s2
```

then the value of `e` will certainly have either type `s1` or type `s2`; in both cases, supplying it as an argument to `f` will yield a a result of type `t`:

```
> case x=e of f x;
it : t
```

To make typechecking more efficient, the syntactic marker "`case`" is used to indicate where union elimination may be applied:

```
> \f:s1->s1->t/\s2->s2->t. \e:s1\/s2. case x=e of f x x;
it :
     (s1->s1->t/\s2->s2->t)->s1->t
  /\ (s1->s1->t/\s2->s2->t)->s2->t

> \f:s1->s1->t/\s2->s2->t. \e:s1\/s2. f e e;
it : NS
```

The rule can also be formulated without the `case`, placing on the typechecker the burden of choosing the correct points to apply union elimination. (See Section 5.3.)

Since λ-abstractions may be annotated only with a single domain type, the constructs described so far do not allow the "finitary polymorphism" of functions with intersection types to be inherited by function definitions that use them. For example, assume we have a type `int` and a type `real` such that `int` $\leq$ `real`[8]

```
> prim int <= real;
```

and a `plus` function that maps pairs of integers to an integer and pairs of reals to a real:

```
> plus : int->int->int/\real->real->real;
plus : int->int->int /\ real->real->real
```

Then

```
> \x:int. plus x x;
it : int -> int

> \x:real. plus x x;
it : real -> real

> \x:int/\real. plus x x;
it : int -> int

> \x:int\/real. plus x x;
it : int->real /\ real->real
```

[8] Again, the specification of a subtype relation on primitive types falls outside the calculus defined in this section, though it is supported by the prototype compiler. See Section 5.1.

To allow the type `int`→`int` ∧ `real`→`real` to be derived for this function, it is necessary to typecheck the body *twice* — once under the assumption that `x` : `int` and once under the assumption that `x` : `real`. The `for` construct is used for this purpose:

$$\frac{\Gamma \vdash [\sigma_i/\alpha]\mathtt{e} \in \tau_i}{\Gamma \vdash \mathtt{for}\ \alpha\ \mathtt{in}\ \sigma_1..\sigma_n.\ \mathtt{e} \in \tau_i} \qquad (\textsc{For})$$

Using For, our function definition can be typechecked twice and the results conjoined using Inter-I to obtain the desired type:

```
> for 'a in int,real. \x:'a. plus x x;
it : int->int /\ real->real
```

For notational convenience, when $\alpha \notin \mathrm{FTV}(\mathtt{e})$ we write

$$\lambda\mathtt{x}{:}\tau_1..\tau_n.\,\mathtt{e} \stackrel{\text{def}}{=} \mathtt{for}\ \alpha\ \mathtt{in}\ \tau_1..\tau_n.\ \lambda\mathtt{x}{:}\alpha.\,\mathtt{e}.$$

as in Forsythe:

```
> \x:int,real. plus x x;
it : int->int /\ real->real
```

4 Examples

To make the foregoing definitions more concrete and demonstrate some of the expressive power of $\lambda(\forall, \wedge, \vee)$, we now develop several larger examples. The interactions shown are reproduced exactly from sessions with a prototype compiler implemented in Standard ML.

The compiler itself is not discussed in detail in this report, beyond a rough sketch of its internals in Section 4.1.1. The focus in the work described here has been on the examples that could be expressed rather than the compiler technology necessary to support their translation. In particular, the completeness of the algorithms used in the typechecker have not been proved complete.

4.1 The Compiler

We begin by sketching the architecture and some of the features of the compiler.

4.1.1 Internals

The high-level flow of information through the compiler is as follows:

1. An expression typed by the user is parsed into an abstract syntax tree.
2. The abstract syntax tree is passed to the typechecker, which returns a data structure representing a proof of a (purportedly) minimal typing for this term in the inference system described in Section 3.
3. Because it was coded for simplicity rather than efficiency, the typing derivations returned by the typechecker can be quite large. A simple proof-normalization procedure transforms the original derivation into an equivalent but more compact derivation.

4. Object code for the expression is produced by walking over the optimized typing derivation and generating a Standard ML program, which is then passed to the Standard ML compiler to be translated to native code and executed. (Standard ML makes a convenient target language in this case because the prototype compiler itself is implemented in SML. The only hitch is that, since the type system of $\lambda(\forall, \wedge, \vee)$ is more flexible than that of SML, instances of an unsafe "typecast" operator must be sprinkled through the SML object code to discourage the SML typechecker.)

   The present compilation scheme translates values of intersection types into expressions whose result is a list with the same number of elements as the original type had conjuncts. Union types are translated into variant records.

5. The result is printed, along with its type.

4.1.2 Type abbreviations

The keyword `type` introduces an abbreviation for a common type expression.

```
> type PolyId = All 'a. 'a -> 'a;
type PolyId = All 'a. 'a -> 'a
```

When a type involving the abbreviation is presented to the parser, it is immediately expanded. All internal processing is in terms of the expansion:

```
> \pid:PolyId. pid [int];
it : PolyId -> int -> int
val it = <fn>
```

Finally, when the top-level read-eval-print loop is about to print a type, it checks whether any subexpressions are equal (modulo renaming of bound type variables) to the right hand side of previously introduced type abbreviations. If so, it prints the names of the abbreviations in place of the subexpressions themselves:

```
> \\'a. \x:'a. x;
it : PolyId
val it = <typefn>
```

4.1.3 Observing results

The final feature of the compiler that is important for present purposes is the facility for printing the results of computations on encodings of algebraic datatypes.[9]

For example, the usual encoding of booleans in the polymorphic λ-calculus is:

```
> type Bool = All 't. 't -> 't -> 't;
type Bool = All 't. 't -> 't -> 't
```

The boolean values `tt` and `ff` are then defined as:

[9] A similar facility is discussed by Michaylov and Pfenning [29].

```
> tt = \\'t. \x:'t. \y:'t. x;
tt : Bool
val tt = <typefn>

> ff = \\'t. \x:'t. \y:'t. y;
ff : Bool
val ff = <typefn>
```

The default result-printing mechanism is only able to display the fact that these values are both type functions. However, we can install a special-purpose printing function for a specified type, providing an SML function to be applied to a runtime result of this type to coerce it into a string:[10]

```
> install `fun b_to_str b = b () "tt" "ff";`;

> observe Bool == `b_to_str`;

> (\x:Bool. x) tt;
it : Bool
val it = tt
```

In some of the examples below, where the results that would be printed are uninteresting, the code generation and execution phases are omitted.

4.2 Church arithmetic

Our first programming example shows a variant of Church's encoding of natural numbers where the type `Zero`, whose only member is the encoding of the number `zero`, is distinguished from the type `Pos`, whose members are all the positive natural numbers.

4.2.1 Type definitions

For comparison, recall that Church numerals are encoded in the ordinary polymorphic λ-calculus as elements of the following type:

```
> type OrigNat = All 't. ('t->'t) -> 't -> 't;
type OrigNat = All 't. ('t->'t) -> 't -> 't
```

To print an element of `Nat`, we apply it to the integer successor function and the integer 0.

```
> install `fun prt_OrigNat (n:unit->unit->(int->int)->int->int) =
>              makestring (n () () (fn i=>i+1) 0);`;

> observe Orig == `prt_OrigNat`;
```

Then the first few natural numbers are encoded as follows.

[10] The first argument passed to b in the body of the printing function is a placeholder for the type parameter expected by values of type `Bool`. The code generation phase of the compiler erases all type information except the positions of type abstractions and applications.

```
> origzero = \\'t. \s:'t->'t. \z:'t. z;
origzero : OrigNat
val origzero = <typefn>

> origone = \\'t. \s:'t->'t. \z:'t. s z;
origone : OrigNat
val origone = <typefn>

> origtwo = \\'t. \s:'t->'t. \z:'t. s (s z);
origtwo : OrigNat
val origtwo = <typefn>
```

Operationally, the type argument `'t` to an element `e` element of type `OrigNat` names the type of the result of the `n`-fold iteration of the argument `s` over the argument `z`, where `n` is the number coded by `e`.

Since we intend to distinguish zero from all other natural numbers, our encoding here needs to take *two* type arguments — one for the result type of a 0-fold iteration of `s` over `z` (that is, `z` itself) and one for the result type of an `n`-fold iteration of `s` over `z`, for some $\mathtt{n} \geq 1$. Also, the function `s` must map elements of `'z` to elements of `'p` (on the first iteration, when it is applied to the base element `z`) *and* elements of `'p` to elements of `'p` (for successive iterations).

```
> type Zero = All 'z. All 'p. /\['z->'p,'p->'p] -> 'z -> 'z;
type Zero = All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'z)

> type Pos  = All 'z. All 'p. /\['z->'p,'p->'p] -> 'z -> 'p;
type Pos = All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'p)
```

Now, the type of all natural numbers is just the union of `Zero` and `Pos`:

```
> type Nat = Zero\/Pos;
type Nat = Zero \/ Pos
```

Again, to print elements of `Zero` and `Pos`, we apply them to the integer successor function and the integer 0.

```
> install `fun prt_Nat (n:unit->unit->((int->int)list)->int->int) =
>              makestring (n () () [(fn i=>i+1),(fn i=>i+1)] 0);`;

> observe Zero == `prt_Nat`;

> observe Pos  == `prt_Nat`;
```

Aside from their types, elements of `Nat` are precisely the same as the corresponding elements of `OrigNat`.

```
> zero  = \\'z. \\'p. \s:/\['z->'p,'p->'p]. \z:'z. z;
zero : Zero
val zero = 0

> one  = \\'z. \\'p. \s:/\['z->'p,'p->'p]. \z:'z. s z;
```

```
one : Pos
val one = 1

> two  = \\'z. \\'p. \s:/\['z->'p,'p->'p]. \z:'z. s (s z);
two : Pos
val two = 2
```

4.2.2 Basic arithmetic functions

The successor function for ordinary church numerals takes a numeral `n` as argument and returns a new numeral that iterates `n` times and then once more.

```
> origsucc = \n:OrigNat. \\'t. \s:'t->'t. \z:'t. s (n ['t] s z);
origsucc : OrigNat -> OrigNat
```

Successor for our new encoding is exactly the same, except that we explicitly allow for the argument `n` to be either `Zero` or `Pos` and check the body separately for each case.

```
> succ = \n:Zero,Pos.
>            \\'z. \\'p. \s:/\['z->'p,'p->'p]. \z:'z.
>               s (n ['z] ['p] s z);
succ : Zero->Pos /\ Pos->Pos

> one = succ zero;
one : Pos
```

Addition of original-style church numerals `m` and `n` is defined by iterating the successor function `m` times, using `n` as the starting value instead of `zero`.

```
> origplus = \m:OrigNat. \n:OrigNat. m [OrigNat] origsucc n;
origplus : OrigNat -> OrigNat -> OrigNat
```

Again, addition of numerals is exactly the same, except that we need to be more careful about the types. As for `succ`, we allow the types of both `m` and `n` to be either `Zero` or `Pos`, checking the body separately in each case. Here, though, we need to use the `for` construct explicitly so that we have a name for the type of `n`: this type will be passed as the result type of a 0-fold iteration of `succ` over `n` — that is, the result of applying `m` to `succ` and `n` in the case that `m` happens to be `zero`. When `m` has type `Pos`, the result type of the iteration is always `Pos`.

```
> plus = for 'm in Zero,Pos.
>           for 'n in Zero,Pos.
>               \m:'m. \n:'n.
>                   m ['n] [Pos] succ n;
plus : Zero->Zero->Zero /\ Zero->Pos->Pos /\ Pos->Zero->Pos /\ Pos->Pos->Pos
```

Multiplication and exponentiation of our numerals can be defined in the same way.

```
> mult = for 'm in Zero,Pos.
>           for 'n in Zero,Pos.
>               \m:'m. \n:'n.
>                   m [Zero] ['n] (plus n) zero;
mult :
    Zero->Zero->Zero /\ Zero->Pos->Zero /\ Pos->Zero->Zero /\ Pos->Pos->Pos
```

```
> exp = for 'm in Zero,Pos.
>           for 'n in Zero,Pos.
>                \m:'m. \n:'n.
>                    n [Pos] ['m] (mult m) one;
exp : Zero->Zero->Pos /\ Zero->Pos->Zero /\ Pos->Zero->Pos /\ Pos->Pos->Pos
```

4.2.3 Predecessor

Defining the predecessor function on Church's original encoding was a significant feat in the early days of λ-calculus.[11] First, we need pairing functions for numerals:

```
> type OrigNatPair = All 'r. (OrigNat->OrigNat->'r) -> 'r;
type OrigNatPair = All 'r. (OrigNat->OrigNat->'r) -> 'r

> origpair = \p1:OrigNat. \p2:OrigNat.
>               \\'r. \f:(OrigNat->OrigNat->'r).
>                   f p1 p2;
origpair : OrigNat -> OrigNat -> OrigNatPair

> origfst = \p:OrigNatPair. p [OrigNat] (\p1:OrigNat. \p2:OrigNat. p1);
origfst : OrigNatPair -> OrigNat

> origsnd = \p:OrigNatPair. p [OrigNat] (\p1:OrigNat. \p2:OrigNat. p2);
origsnd : OrigNatPair -> OrigNat
```

Then predecessor is defined by an iteration that starts with the pair (`zero`, `zero`) and returns, as the result of the `n`th iteration, the pair (`n`, `pred n`):

```
> origpred = \n:OrigNat.
>                 origsnd (n [OrigNatPair]
>                            (\p:OrigNatPair. origpair
>                                               (origsucc (origfst p))
>                                               (origfst p))
>                            (origpair origzero origzero));
origpred : OrigNat -> OrigNat
```

For our new encoding, we need three separate pair types:[12]

```
> type ZeroZeroPr = All 'r. (Zero->Zero->'r)->'r;
type ZeroZeroPr = All 'r. (Zero->Zero->'r) -> 'r

> type PosZeroPr  = All 'r. (Pos->Zero->'r)->'r;
type PosZeroPr = All 'r. (Pos->Zero->'r) -> 'r
```

[11] Readers unfamiliar with this encoding may find the more expository presentations in [35, 37] helpful.

[12] If we were working in a calculus with higher-order polymorphism (see Section 6), `Pair` itself would be a type constructor and we would simply apply it to pairs of types as needed. Here we're forced to write out `Pair`(α, β) for each α and β where it's needed.

```
> type PosPosPr   = All 'r. (Pos->Pos->'r)->'r;
type PosPosPr = All 'r. (Pos->Pos->'r) -> 'r

> pair = for 'p1 in Zero,Pos.
>           for 'p2 in Zero,Pos.
>               \p1:'p1. \p2:'p2.
>                   \\'r. \f:'p1->'p2->'r.
>                       f p1 p2;
pair :
    Zero->Zero->ZeroZeroPr
 /\ Zero->Pos->(All 'r. (Zero->Pos->'r)->'r)
 /\ Pos->Zero->PosZeroPr
 /\ Pos->Pos->PosPosPr

> fst = for 'p1 in Zero,Pos.
>         \p: (All 'r. ('p1->NS->'r)->'r).
>            p ['p1] (\p1:'p1. \p2:NS. p1);
fst : (All 'r. (Zero->NS->'r)->'r)->Zero /\ (All 'r. (Pos->NS->'r)->'r)->Pos

> snd = for 'p2 in Zero,Pos.
>         \p: (All 'r. (NS->'p2->'r)->'r).
>            p ['p2] (\p1:NS. \p2:'p2. p2);
snd : (All 'r. (NS->Zero->'r)->'r)->Zero /\ (All 'r. (NS->Pos->'r)->'r)->Pos
```

Now we can have enough types to express the predecessor function in the same style as above.

```
> pred = \n:Pos.
>            snd (n [ZeroZeroPr] [PosPosPr\/PosZeroPr]
>                   (\p:ZeroZeroPr,PosPosPr\/PosZeroPr.
>                       pair (succ (fst p)) (fst p))
>                   (pair zero zero));
pred : Pos -> (Pos\/Zero)
```

The most interesting feature of this example is that it's the first use we've seen of a union type — one of the few that I know of — where the union is *essential*, that is, where the union cannot be transformed into an intersection by applying the SUB-DIST-AU law and where the function cannot be given an appropriate type without using unions.

4.2.4 Testing for zero

Our `pred` function has intentionally been defined so that applying it to a numeral that is not known to be positive fails.

```
> f = \x:Nat. pred x;
f : NS
```

Using a technique similar to that used for `pred`, we can define a `natcase` function that takes a natural number `n`, a function (like `pred`) that expects only positive arguments, and a default argument to be returned when `n` is `zero` and applies the function to `n` if possible.

```
> type ZeroNatPr  = All 'r. (Zero->Nat->'r)->'r;
type ZeroNatPr = All 'r. (Zero->Zero->'r/\Zero->Pos->'r) -> 'r

> type PosNatPr   = All 'r. (Pos->Nat->'r)->'r;
type PosNatPr = All 'r. (Pos->Zero->'r/\Pos->Pos->'r) -> 'r

> pairn = for 'p1 in Zero,Pos.
>               \p1:'p1. \p2:Nat.
>                   \\'r. \f:'p1->Nat->'r.
>                      f p1 p2;
pairn :
    Zero->Zero->ZeroNatPr
 /\ Zero->Pos->ZeroNatPr
 /\ Pos->Zero->PosNatPr
 /\ Pos->Pos->PosNatPr

> fstn = for 'p1 in Zero,Pos.
>         \p: (All 'r. ('p1->NS->'r)->'r).
>            p ['p1] (\p1:'p1. \p2:NS. p1);
fstn :
    (All 'r. (Zero->NS->'r)->'r)->Zero /\ (All 'r. (Pos->NS->'r)->'r)->Pos

> sndn = \p: (All 'r. (NS->Nat->'r)->'r).
>            p [Nat] (\p1:NS. \p2:Nat. p2);
sndn : (All 'r. (NS->Zero->'r/\NS->Pos->'r)->'r) -> Nat

> natcase = \n:Zero,Pos.
>               \scase:Pos->Nat. \zcase:Nat.
>                  sndn (n [ZeroNatPr] [PosNatPr]
>                          (\p:ZeroNatPr,PosNatPr.
>                                pairn (succ (fstn p))
>                                      (scase (succ (fstn p))))
>                          (pairn zero zcase));
natcase :
    Zero->(Pos->Nat)->Zero->Nat
 /\ Zero->(Pos->Nat)->Pos->Nat
 /\ Pos->(Pos->Nat)->Zero->Nat
 /\ Pos->(Pos->Nat)->Pos->Nat

> f = \x:Nat. natcase x pred zero;
f : Zero->Nat /\ Pos->Nat
```

4.3 Church Arithmetic (Alternate Form)

There is another way of encoding the basic arithmetic functions on Church numerals:

```
> origplus' = \m:OrigNat. \n:OrigNat.
>                \\'t. \s:'t->'t. \z:'t.
```

```
>                   m ['t] s (n ['t] s z);
origplus' : OrigNat -> OrigNat -> OrigNat

> origmult' = \m:OrigNat. \n:OrigNat.
>                \\'t. \s:'t->'t.
>                   m ['t] (n ['t] s);
origmult' : OrigNat -> OrigNat -> OrigNat

> origexp' = \m:OrigNat. \n:OrigNat.
>                \\'t.
>                   n ['t->'t] (m ['t]);
origexp' : OrigNat -> OrigNat -> OrigNat
```

This version of the arithmetic functions is interesting to try to emulate on our new encoding; the solution involves some fairly tricky use of the `for` construct. Also, the exponential function in this encoding requires iteration at higher types, which provides another good test of the limits of this encoding.[13]

We need to use a slightly more refined formulation for `Zero` here.

```
> type Zero = All 'z. All 'p. NS -> 'z -> 'z;
type Zero = All 'z. (All 'p. NS->'z->'z)

> type Pos  = All 'z. All 'p. /\['z->'p,'p->'p] -> 'z -> 'p;
type Pos = All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'p)

> type Nat = Zero\/Pos;
type Nat = Zero \/ Pos

> install `fun prt_Nat (n:unit->unit->((int->int)list)->int->int) =
>              makestring (n () () [(fn i=>i+1),(fn i=>i+1)] 0);`;

> observe Zero == `prt_Nat`;

> observe Pos  == `prt_Nat`;
```

The novel feature of the `plus'` function on our new encoding is the use of the `for` construct in the second line from the end to "guess" the result type of the iteration when `m` is `zero`. What we want to write, intuitively, is "If `n` has type `Zero` then apply `m` to `'z`, otherwise apply `m` to `'p`." But, of course, this falls far outside of what can be expressed in a type system of this kind. What is surprising is that it can be simulated by guessing: we simply try applying `m` to *both* `'z` and `'p` in turn. One choice is the one that we "should" have made, and it yields the desired result type. The other choice is wrong, yields an ill-typed application in the last line, and drops out of the result.

```
> plus' = \m:Zero,Pos. \n:Zero,Pos.
>             \\'z. \\'p.
```

[13] It may provide an even better test of the limits of the encoder. It took me several hours to figure out how to type this version of the exponential function.

```
>                \s: ('z\/'p)->'p,NS.
>                   \z:'z.
>                      for 'g in 'z,'p.
>                         m ['g] ['p] s (n ['z] ['p] s z);
plus' :
    Zero->Zero->(All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'z))
 /\ Zero->Zero->Zero
 /\ Zero->Pos->Pos
 /\ Pos->Zero->Pos
 /\ Pos->Pos->Pos
```

The typing that the typechecker discovers for this function is actually slightly better than we need: the first conjunct is a proper subtype of `Zero->Zero->Zero`.[14]

The cases for multiplication and exponentiation are similar, but slightly more complicated.

```
> mult' = \m:Zero,Pos. \n:Zero,Pos.
>             \\'z. \\'p.
>                \s: ('z\/'p)->'p,NS.
>                   for 'g1 in 'z,'p.
>                      m ['z] ['g1]
>                         (for 'g2 in 'z,'p.
>                            (n ['g2] ['p] s));
mult' :
    Zero->Zero->(All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'z))
 /\ Zero->Zero->Zero
 /\ Zero->Pos->(All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'z))
 /\ Zero->Pos->Zero
 /\ Pos->Zero->(All 'z. (All 'p. ('z->'p/\'p->'p)->'z->'z))
 /\ Pos->Zero->Zero
 /\ Pos->Pos->Pos

> exp' = \m:Zero,Pos. \n:Zero,Pos.
>           \\'z. \\'p.
>              for 'n in NS,'z->'p/\'p->'p. for 'm in 'z,'p.
>                 n ['n] ['z->'m/\'m->'m]
>                   (for 'g1,'g2 in 'z,'p.
>                     m ['g1] ['g2]);
exp' :
    Zero->Zero->Pos
 /\ Zero->Zero->(All 'z. (All 'p. ('z->'p/\'p->'p)->'p->'p))
 /\ Zero->Pos->Zero
 /\ Pos->Zero->Pos
 /\ Pos->Zero->(All 'z. (All 'p. ('z->'p/\'p->'p)->'p->'p))
 /\ Pos->Pos->Pos
 /\ Pos->Pos->(All 'z. (All 'p. ('z->'p/\'p->'p)->'p->'p))
```

[14] This means, of course, that the second conjunct can be dropped without changing the type. The type simplification heuristic presently used in the compiler misses this case.

The diagonalization of this formulation of the exponential function is particularly interesting, since it involves a polymorphic self-application.

```
> diag' = \n:Zero,Pos. exp' n n;
diag' :
    Zero->Pos
 /\ Zero->(All 'z. (All 'p. ('z->'p/\'p->'p)->'p->'p))
 /\ Pos->Pos
 /\ Pos->(All 'z. (All 'p. ('z->'p/\'p->'p)->'p->'p))
```

4.4 Church Booleans

We can play a similar game with operations on booleans, using a Church-like encoding that distinguishes "true" booleans from "false" booleans.

```
> type T = All 'a. All 'b. 'a -> NS -> 'a;
type T = All 'a. (All 'b. 'a->NS->'a)

> type F = All 'a. All 'b. NS -> 'b -> 'b;
type F = All 'a. (All 'b. NS->'b->'b)

> type Bool = T\/F;
type Bool = T \/ F

> observe T == `fn b => "tt"`;

> observe F == `fn b => "ff"`;

> tt = \\'a. \\'b. \x:'a. \y:NS. x;
tt : T
val tt = tt

> ff = \\'a. \\'b. \x:NS. \y:'b. y;
ff : F
val ff = ff
```

The type of the boolean-not function is its truth table:

```
> bnot = \m:T,F.
>          \\'a. \\'b.
>              \x:NS,'a. \y:NS,'b.
>                          m ['b] ['a] y x;
bnot : T->F /\ F->T
val bnot = <inter>

> bnot tt;
it : F
val it = ff
```

Without the "for" construct, the best typing we can obtain for boolean-or is:

```
> bor = \m:T,F. \n:T,F.
>          m [T] [Bool] tt n;
bor : T->T->T /\ T->F->T /\ F->T->Bool /\ F->F->Bool
val bor = <inter>
```

Using "for," we obtain a more refined type for boolean-or:

```
> bor = for 'a in T,F.
>          for 'b in T,F.
>              \m:'a. \n:'b.
>                  m [T] ['b] tt n;
bor : T->T->T /\ T->F->T /\ F->T->T /\ F->F->F
val bor = <inter>

> bor ff tt;
it : T
val it = tt

> bor ff ff;
it : F
val it = ff
```

4.5 Bit String Addition

Another example, interesting mainly because it uses iteration at higher types to implement a double recursion, is the addition of binary numerals in standard form. The idea for this example is due to Freeman and Pfenning [18].

Binary numerals (bit strings) may be characterized by the following inductive type definition:

```
indtype Bitstr = e | z(Bitstr) | o(Bitstr)
```

That is, a bit string is either empty, or has a low order bit that is either `z` (zero) or `o` (one) followed by a bit string. For example, `e`, `z(e)`, and `z(z(z(e)))` all represent zero; `z(o(o(e)))` represents six.

Standard-form binary numerals may be defined by the following inductive type definitions:

```
indtype E  = e
indtype NE = o(e) | z(NE) | o(NE)
```

That is, an empty standard-form numeral is just the token `e`. A nonempty standard-form numeral has a low-order bit that is either `z` (zero) or `o` (one), followed by a nonmempty numeral. For example, `z(o(e))` is in standard form, but `z(e)` is not.

These types can be encoded in $\lambda(\forall, \wedge, \vee)$ as follows:

```
> type Bitstr = All 'e. All 'n.
>                 'e -> ('e->'n/\'n->'n) -> ('e->'n/\'n->'n) -> ('e\/'n);
type Bitstr =
      All 'e. (All 'n. 'e->('e->'n/\'n->'n)->('e->'n/\'n->'n)->('e\/'n))
```

```
> type E  = All 'e. All 'n. 'e -> ('n->'n) -> ('e->'n/\'n->'n) -> 'e;
type E = All 'e. (All 'n. 'e->('n->'n)->('e->'n/\'n->'n)->'e)

> type NE = All 'e. All 'n. 'e -> ('n->'n) -> ('e->'n/\'n->'n) -> 'n;
type NE = All 'e. (All 'n. 'e->('n->'n)->('e->'n/\'n->'n)->'n)

> type Std = E\/NE;
type Std = E \/ NE

> check Std <= Bitstr;
Yes.

> install `fun bitstr_tostr b =
>               b () () "e"
>                            (fn s => "z("^s^")")
>                            [(fn s => "o("^s^")"),(fn s => "o("^s^")")];`;

> observe E  == `bitstr_tostr`;

> observe NE == `bitstr_tostr`;

> std_e = \\'e. \\'n. \e:'e. \z:'n->'n. \o:'e->'n/\'n->'n. e;
std_e : E
val std_e = e

> std_z = \n:E,NE.
>            \\'e. \\'n. \e:'e. \z:'n->'n. \o:'e->'n/\'n->'n.
>                z (n ['e] ['n] e z o);
std_z : NE -> NE
val std_z = <fn>

> std_o = \n:E,NE.
>            \\'e. \\'n. \e:'e. \z:'n->'n. \o:'e->'n/\'n->'n.
>                o (n ['e] ['n] e z o);
std_o : E->NE /\ NE->NE
val std_o = <inter>
```

The successor function on bit strings in standard form can be defined by the following recursive program scheme:

```
std_succ e     = o e
std_succ (z b) = o b
std_succ (o b) = z (std_succ b)
```

Note that if the input to this scheme happens to be in standard form, then the output will be too.

To implement this scheme in terms of iterators, we need to use iteration over pairs. The first projection of the result of the iteration is the original numeral; the second projection is the successor (in standard form).

```
> type ExE   = All 'r. (E->E->'r)   -> 'r;
type ExE = All 'r. (E->E->'r) -> 'r

> type ExNE  = All 'r. (E->NE->'r)  -> 'r;
type ExNE = All 'r. (E->NE->'r) -> 'r

> type NExE  = All 'r. (NE->E->'r)  -> 'r;
type NExE = All 'r. (NE->E->'r) -> 'r

> type NExNE = All 'r. (NE->NE->'r) -> 'r;
type NExNE = All 'r. (NE->NE->'r) -> 'r

> std_pair = for 'p1 in E,NE.
>              for 'p2 in E,NE.
>                \p1:'p1. \p2:'p2.
>                   \\'r. \f:'p1->'p2->'r.
>                      f p1 p2;
std_pair : E->E->ExE /\ E->NE->ExNE /\ NE->E->NExE /\ NE->NE->NExNE
val std_pair = <inter>

> std_fst = for 'p1 in E,NE.
>         \p: (All 'r. ('p1->NS->'r)->'r).
>            p ['p1] (\p1:'p1. \p2:NS. p1);
std_fst : (All 'r. (E->NS->'r)->'r)->E /\ (All 'r. (NE->NS->'r)->'r)->NE
val std_fst = <inter>

> std_snd = for 'p2 in E,NE.
>         \p: (All 'r. (NS->'p2->'r)->'r).
>            p ['p2] (\p1:NS. \p2:'p2. p2);
std_snd : (All 'r. (NS->E->'r)->'r)->E /\ (All 'r. (NS->NE->'r)->'r)->NE
val std_snd = <inter>

> std_succ =
>    \n:E,NE.
>      std_snd
>         (n [ExNE] [NExNE]
>            (std_pair std_e (std_o std_e))
>            (\p:NExNE. std_pair (std_z (std_fst p)) (std_o (std_fst p)))
>            (\p:ExNE,NExNE. std_pair (std_o (std_fst p)) (std_z (std_snd p))));
std_succ : E->NE /\ NE->NE
val std_succ = <inter>

> zero = std_e;
zero : E
val zero = e

> one = std_succ zero;
```

```
one : NE
val one = o(e)

> two = std_succ one;
two : NE
val two = z(o(e))

> three = std_succ two;
three : NE
val three = o(o(e))

> four = std_succ three;
four : NE
val four = z(z(o(e)))
```

Addition on bit strings can be defined by the following doubly recursive scheme:

```
std_add  e     n    = n
std_add  m     e    = m
std_add (z m) (z n) = z (std_add m n)
std_add (o m) (z n) = o (std_add m n)
std_add (z m) (o n) = o (std_add m n)
std_add (o m) (o n) = z (std_add (std_add (o e) m) n)
```

To implement this scheme in terms of iterators, we need to use iteration at higher types:[15]

```
> std_add = for 'm in E,NE.
>             \m:'m.
>                m [E->E/\NE->NE] [E->NE/\NE->NE]
>                  (\n:E,NE. n)
>                  (for 'n in E,NE.
>                   \addm':E->NE/\NE->NE. \n:'n.
>                     n [NE] [NE]
>                        (std_z (addm' std_e))
>                        (\addm'n':NE. std_z addm'n')
>                        (\addm'n':NE. std_o addm'n'))
>                  (for 'n in E,NE.
>                   \addm':E->E/\NE->NE,E->NE/\NE->NE. \n:'n.
>                     n [NE] [NE]
>                        (std_o (addm' std_e))
>                        (\addm'n':E,NE. std_o addm'n')
>                        (\addm'n':E,NE. std_z (std_succ addm'n'))));
std_add : E->E->E /\ E->NE->NE /\ NE->E->NE /\ NE->NE->NE
```

[15] The code generation and execution phases for this example are omitted; with the present naive implementation of the code generator they consume too much memory.

4.6 Finitary Quantification and Typechecking Efficiency

This section demonstrates how the finitary polymorphism of the `for` construct may be used to improve the efficiency of typechecking over simpler languages, like Forsythe, allowing intersection types but omitting `for`.

To make the example read more like Forsythe, we'll work in terms of two "built-in" primitive types, integer and real:

```
> observe int == `fn i:int => makestring i`;

> observe real == `fn r:real => makestring r`;

> zero : int == `0`;
zero : int

> succ : int->int == `fn x => x+1`;
succ : int -> int

> plustwo = \x:int. succ (succ x);
plustwo : int -> int
val plustwo = <fn>

> two = plustwo zero;
two : int
val two = 2
```

To introduce a primitive inclusion between the types `int` and `real` we need to tell the back-end how to generate code for transforming from int to real:

```
> prim int <= real == `real`;
```

Now we can define a successor function that works for both integers and reals:

```
> irsucc : int->int /\ real->real
>      == `[R.c (fn (x:int) => x+1), R.c (fn (x:real) => x+1.0)]`;
irsucc : int->int /\ real->real

> irplustwo = \x:int,real. irsucc irsucc x;
irplustwo : NS
val irplustwo = <nonsense>

> irtwo = irplustwo zero;
irtwo : NS
val irtwo = <nonsense>
```

Similarly, the `plus` function operates on both integers and reals:

```
> plus : int->int->int /\ real->real->real
>      == `[R.c (fn (x:int) => fn (y:int) => x+y),
>           R.c (fn (x:real) => fn (y:real) => x+y)]`;
```

```
plus : int->int->int /\ real->real->real

> double = \x:int,real. plus x x;
double : int->int /\ real->real
val double = <inter>
```

We can realize a substantial gain in typechecking efficiency for functions like "plus" by using the `for` construct explicitly instead of annotating each bound variable with both `int` and `real`. Here, the body of the polynomial function "poly" is typechecked only twice, instead of 16 times as would be necessary in Forsythe:

```
> poly = for 'a in int,real.
>           \w:'a. \x:'a. \y:'a. \z:'a.
>               plus (double x) (plus (plus w y) z);
poly : int->int->int->int->int /\ real->real->real->real->real
val poly = <inter>
```

4.7 Default Parameters

A completely different example of the practical utility of intersection types comes from their ability to express procedures with default parameters. For example, we can give the type `string -> int -> (string/\char->string)` to a built-in function that takes a string `s` and an integer i and returns *both* the string `s` padded with enough blanks to make its length i *and* a function that given a character `c` returns `s` padded with enough `c`'s to make its length i. The result of applying `pad` to `s` and i can either be used directly as a string (by applying a `prnt` function to it, for example) or further applied to a character `c`.

To implement this scheme, we define the following primitives:

```
> install `fun dupl c 0 = ""
>             | dupl c n = c ^ (dupl c (n-1));
>           fun primpadstr s i c = s ^ (dupl c (max(0,i-(size s))));`;

> pad : string -> int -> (string/\char->string)
>     == `[R.c (fn s => fn i => primpadstr s i " "),
>               R.c (fn s => fn i => fn c => primpadstr s i c)]`;
pad : string->int->string /\ string->int->char->string

> prnt : string -> unit == `fn s => prt ("\"" ^ s ^ "\"\n")`;
prnt : string -> unit

> blank : char == `" "`;
blank : char

> dot : char == `"."`;
dot : char

> mesg : string == `"hello"`;
mesg : string
```

```
> ten : int == `10`;
ten : int
```

Now we can use pad and prnt as described above:

```
> prnt (pad mesg ten);
it : unit
"hello     "
val it = <prim>

> prnt (pad mesg ten dot);
it : unit
"hello....."
val it = <prim>
```

In fact, this notion can be supported as a general language feature by supplying one built-in constant that is used to build functions with default parameters

```
> default : All 'a. All 'b. ('a -> 'b) -> 'a -> ('b/\'a->'b)
>         == `[R.c (fn () => fn () => fn f => fn v => f v),
>                  R.c (fn () => fn () => fn f => fn v => f)]`;
default :
     (All 'a. (All 'b. ('a->'b)->'a->'b))
  /\ (All 'a. (All 'b. ('a->'b)->'a->'a->'b))

> myprimpad : string -> int -> char -> string
>           == `primpadstr`;
myprimpad : string -> int -> char -> string

> mypad = \s:string. \l:int. default [char] [string] (myprimpad s l) blank;
mypad : string->int->string /\ string->int->char->string
val mypad = <inter>
```

5 Alternatives

This section discusses some choices in the formulation of the type system given in Section 3.

5.1 Primitive Inclusions

The type system of Forsythe is parameterized by a set of *primitive types* with a primitive preorder structure that is inherited by the preorder on the set of all type expressions:

$$\frac{\rho_1 \ \leq_{\text{prim}} \ \rho_2}{\rho_1 \ \leq \ \rho_2} \qquad \text{(Sub-Prim)}$$

The types and subtyping rules of $\lambda(\forall, \wedge, \vee)$ can be extended straightforwardly to include this notion of primitive types. In fact, the prototype compiler described in Section 4 implements this extension already.

5.2 Bounded Quantification

A primary design choice for a calculus combining polymorphism with intersection types is between ordinary and bounded quantification. It would seem at first that bounded quantification would be more appropriate, since it explicitly takes the subtype structure into account. But in a calculus with intersection types, bounds on quantifiers do not actually add much power, since every well-typed expression in a calculus with bounded quantification and intersections (call it $\lambda(\forall_{\leq}, \wedge, \vee)$) can be translated into a well-typed expression in $\lambda(\forall, \wedge, \vee)$ by rewriting bounded quantifiers as ordinary ones and moving the bounds into the body of the quantified expression:[16]

$$\begin{array}{lcl} \Lambda\alpha \leq \sigma.\ \mathtt{e} & \longrightarrow & \Lambda\alpha.\ [\sigma\wedge\alpha/\alpha]\mathtt{e} \\ \forall\alpha \leq \sigma.\ \tau & \longrightarrow & \forall\alpha.\ [\sigma\wedge\alpha/\alpha]\tau. \end{array}$$

Nevertheless, there is at least one compelling reason for studying a system with bounded quantification in preference to one with ordinary quantification only: the treatment of primitive types.

Experience leads us to expect that when we take the step from a first-order calculus to one with impredicative quantification, the primitive types that were the base case of the first-order system will no longer be needed, since their role can be played equally well by type variables. But this is not the case here: if our quantifiers do not mention bounds, then there is no way to introduce subtyping assumptions among the variables representing primitive types, and we lose the effect of the SUB-PRIM rule.

5.3 The UNION-E Rule

The union elimination rule used in earlier sections of this report includes an explicit syntactic marker, `case`, to guide the typechecker in applying the rule. The rule can also be formulated with no syntactic marker

$$\frac{\Gamma \vdash \mathtt{e\ in}\ \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x}:\tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash [\mathtt{e}/\mathtt{x}]\mathtt{e}' \in \tau} \qquad \text{(UNION-E')}$$

leaving it up to the typechecker to determine when it ought to be applied during the derivation of a minimal type for an expression.

Of course, this rule presents severe difficulties for a typechecking procedure for $\lambda(\wedge, \vee, \rho)$. Because the syntactic forms of $\mathtt{e}$ and $\mathtt{e}'$ are unconstrained in the conclusion, UNION-E can potentially be applied to *any* term $\mathtt{e}'$, choosing any subset of the occurrences of any subterm $\mathtt{e}$. Worse yet, both versions share the difficulty that every term $\mathtt{e}$ has an infinite number of union types and any $\mathtt{e}$ with a type strictly less than `NS` has an infinite number of *inequivalent* union types.

Nevertheless, the more "implicit" form of the rule seems somewhat more elegant; it may preferable for more theoretical investigations of union types.

5.4 Encoding Union Types

In a type system with intersection types and polymorphism, union types can be *encoded* using the following translation:

$$\sigma\vee\tau \quad \stackrel{\text{def}}{=} \quad \forall\alpha.(\sigma\to\alpha) \wedge (\tau\to\alpha) \to \alpha.$$

This allows us to achieve the effect of expressions like

[16] John Mitchell showed me this translation.

```
> \f:a->c/\b->c. \x:a\/b. f x;
it : (a->c/\b->c)->a->c /\ (a->c/\b->c)->b->c
```

by writing:

```
> \f:a->c/\b->c. \x:(All 'x. (a->'x/\b->'x)->'x). x [c] f;
it : (a->c/\b->c) -> (All 'x. (a->'x/\b->'x)->'x) -> c
```

This encoding is not entirely satisfactory, however; for example, it does not validate the rule

$$\sigma \leq \sigma \vee \tau.$$

6 Future Work

The preliminary work presented here suggests a number of paths for future research. In fact, several novel type systems can be formed by combinating subsets of the various features described here:

	Meets only	Meets and joins
No quantification	$\lambda(\wedge, \rho)$ [Forsythe]	$\lambda(\wedge, \vee, \rho)$
Ordinary quantification	$\lambda(\forall, \wedge, \rho)$	$\lambda(\forall, \wedge, \vee, \rho)$ [this report]
Bounded quantification	$\lambda(\forall_{\leq}, \wedge)$ [33]	$\lambda(\forall_{\leq}, \wedge, \vee)$

Among the five novel calculi in the table, $\lambda(\forall, \wedge)$ appears the most tractable. The tractability of $\lambda(\forall_{\leq}, \wedge)$ is still open; a prototype implementation of this calculus[17] would be an excellent next step toward understanding the role of bounded quantification and its interaction with intersection types. The simplest union calculus, $\lambda(\wedge, \vee, \rho)$, provides an important case for studying the semantics of union types, but from a practical perspective $\lambda(\forall, \wedge, \vee, \rho)$ is not very much harder to implement and appears to be much more expressive. The bottom corner, $\lambda(\forall_{\leq}, \wedge, \vee)$, though it undoubtedly holds many fascinating surprises, should probably be saved for last.

Besides detailed investigation of the proof theory and semantics of the more tractable calculi above, there are several issues to be pursued:

Higher-order Polymorphism Some of the examples in Section 4 — especially those involving pairs — could have been expressed more elegantly using higher-order polymorphism. The extension from second-order to higher-order polymorphism, at least for quantifiers without bounds, appears to be unproblematic.

Records The Forsythe language includes an elegant treatment of "extensible records" based on intersection types. Unfortunately, the approach used to add records to the core type system of Forsythe cannot be generalized to calculi with polymorphism: the Forsythe record extension operator must be able to examine the type of the record being extended and remove all instances of the field at which it is being extended. When the type of the record being extended can involve type variables, this is no longer possible.

[17] Now underway.

A treatment of extensible records based on a notion of "constrained quantification" [21, 22] seems promising, but interactions between record types and intersection types may present problems.

Pragmatics of Compilation In addition to the usual problems of establishing decidability for the type systems discussed here, there are some serious efficiency issues that must be addressed before any claims can be made that they form a foundation for practical programming languages. In particular, the compilation scheme used in the prototype compiler described here, where intersection types are interpreted as cartesian products at runtime, leads to an enormous explosion of object code if implemented naively, since a λ-abstraction that is given an intersection type with fifteen conjuncts will be compiled in fifteen separate versions. The union elimination rule causes similar problems.

Programing applications The examples presented here serve to demonstrate that the combination of intersection types, union types, and polymorphism gives rise to intriguing and entertaining programming examples. The question of its importance in practice remains open, pending the development of larger examples that illustrate concrete benefits at acceptable cost.

7 Acknowledgements

I am grateful for productive discussions with Franco Barbanera, Luca Cardelli, Manfred Droste, Andrzej Filinski, Tim Freeman, Robert Harper, Nevin Heintze, Susumu Hayashi, Frank Pfenning, Didier Rémy, and John Reynolds.

A Summary of Rules

A.1 Subtyping

$$\tau \le \tau \qquad \text{(Sub-Refl)}$$

$$\frac{\sigma \le \theta \qquad \theta \le \tau}{\sigma \le \tau} \qquad \text{(Sub-Trans)}$$

$$\frac{\tau_1 \le \sigma_1 \qquad \sigma_2 \le \tau_2}{\sigma_1 \to \sigma_2 \;\le\; \tau_1 \to \tau_2} \qquad \text{(Sub-Arrow)}$$

$$\frac{\sigma \le \tau}{\forall\alpha.\sigma \;\le\; \forall\alpha.\tau} \qquad \text{(Sub-All)}$$

$$\frac{\text{for all } i,\ \sigma \le \tau_i}{\sigma \;\le\; \bigwedge[\tau_1..\tau_n]} \qquad \text{(Sub-Inter-G)}$$

$$\bigwedge[\tau_1..\tau_n] \;\le\; \tau_i \qquad \text{(Sub-Inter-LB)}$$

$$\frac{\text{for all } i,\ \sigma_i \le \tau}{\bigvee[\sigma_1..\sigma_n] \;\le\; \tau} \qquad \text{(Sub-Union-L)}$$

$$\tau_i \;\le\; \bigvee[\tau_1..\tau_n] \qquad \text{(Sub-Union-UB)}$$

$$\bigwedge[\sigma\to\tau_1 \,..\, \sigma\to\tau_n] \;\le\; \sigma\to\bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-AI)}$$

$$\bigwedge[\sigma_1\to\tau \,..\, \sigma_n\to\tau] \;\le\; \bigvee[\sigma_1..\sigma_n]\to\tau \qquad \text{(Sub-Dist-AU)}$$

$$\bigwedge[\forall\alpha.\tau_1 \,..\, \forall\alpha.\tau_n] \;\le\; \forall\alpha.\bigwedge[\tau_1..\tau_n] \qquad \text{(Sub-Dist-QI)}$$

$$\frac{\begin{array}{c}\mathtt{U} \;\equiv\; [\bigvee[\tau_{1,1} \,..\, \tau_{1,m_1}] \,..\, \bigvee[\tau_{n,1} \,..\, \tau_{n,m_n}]] \\ \mathtt{I} \;\equiv\; [\bigwedge[\tau_{1,j_1} \,..\, \tau_{n,j_n}] \mid 1 \le j_i \le m_i]\end{array}}{\bigvee\mathtt{I} \;\le\; \bigwedge\mathtt{U}} \qquad \text{(Sub-Dist-IU)}$$

$$\frac{\begin{array}{c}\mathtt{I} \;\equiv\; [\bigwedge[\tau_{1,1} \,..\, \tau_{1,m_1}] \,..\, \bigwedge[\tau_{n,1} \,..\, \tau_{n,m_n}]] \\ \mathtt{U} \;\equiv\; [\bigvee[\tau_{1,j_1} \,..\, \tau_{n,j_n}] \mid 1 \le j_i \le m_i]\end{array}}{\bigwedge\mathtt{U} \;\le\; \bigvee\mathtt{I}} \qquad \text{(Sub-Dist-UI)}$$

A.2 Typing

$$\frac{\Gamma \;\vdash\; \mathtt{e} \in \sigma \qquad \sigma \le \tau}{\Gamma \;\vdash\; \mathtt{e} \in \tau} \qquad \text{(Subsumption)}$$

$$\Gamma_1, \mathtt{x} : \tau, \Gamma_2 \;\vdash\; \mathtt{x} \in \tau \qquad \text{(Var)}$$

$$\frac{\Gamma, \mathtt{x} : \sigma \;\vdash\; \mathtt{e} \in \tau}{\Gamma \;\vdash\; \lambda\mathtt{x}{:}\sigma.\,\mathtt{e} \in \sigma\to\tau} \qquad \text{(Arrow-I)}$$

$$\frac{\Gamma \;\vdash\; \mathtt{f} \in \sigma\to\tau \qquad \Gamma \;\vdash\; \mathtt{e} \in \sigma}{\Gamma \;\vdash\; \mathtt{f}\ \mathtt{e} \;\in\; \tau} \qquad \text{(Arrow-E)}$$

$$\frac{\Gamma \;\vdash\; \mathtt{e} \in \tau \qquad \alpha \notin \mathrm{FTV}(\Gamma)}{\Gamma \;\vdash\; \Lambda\alpha.\,\mathtt{e} \in \forall\alpha.\tau} \qquad \text{(All-I)}$$

$$\frac{\Gamma \;\vdash\; \mathtt{e} \in \forall\alpha.\tau}{\Gamma \;\vdash\; \mathtt{e}[\sigma] \in [\sigma/\alpha]\tau} \qquad \text{(All-E)}$$

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

$$\frac{\Gamma \vdash \mathtt{e\ in} \bigvee[\tau_1..\tau_n] \qquad \text{for all } i,\ \Gamma, \mathtt{x} : \tau_i \vdash \mathtt{e}' \in \tau}{\Gamma \vdash \mathtt{case\ e\ of\ x} \Rightarrow \mathtt{e}' \in \tau} \quad \text{(Union-E)}$$

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

References

[1] Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types (preliminary version). Draft manuscript (to appear in TACS '91 proceedings), 1991.

[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] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] 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.

[6] Luca Cardelli. Typeful programming. Research Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1989.

[7] 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.

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

[9] 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.

[10] Felice Cardone and Mario Coppo. Two extensions of curry's type inference system. In Odifreddi, editor, *Logic For Computer Science*. Academic Press, 1990.

[11] Robert Cartwright and Mike Fagan. Soft typing. Submitted to PLDI '91, November 1990.

[12] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[13] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of basic functionality theory for λ-calculus. *Notre-Dame Journal of Formal Logic*, 21:685–693, 1980.

[14] 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.

[15] 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.

[16] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. *Mathematical Structures in Computer Science*, 1991. To appear.

[17] Mike Fagan. *Soft Typing: An Approach to Type Checking for Dynamically Typed Languages.* PhD thesis, Rice University, December 1990.

[18] Tim Freeman and Frank Pfenning. Refinement types for ML. Submitted to PLDI '91, November 1990.

[19] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[20] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[21] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, School of Computer Science, Carnegie Mellon University, Feburary 1990.

[22] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando FL*, pages 131–142. ACM, January 1991. Extended version available as Carnegie Mellon Technical Report CMU-CS-90-157.

[23] Susumu Hayashi. Lecture on Union Types at the Logical Frameworks meeting, Antibes, May 1990.

[24] Susumu Hayashi. Personal communication, December 1990.

[25] Susumu Hayashi and Yukihide Takayama. Extended projection method with Kreisel-Troelstra realizability. Submitted to Information and Computation, 1990.

[26] Bart Jacobs, Ines Margaria, and Maddalena Zacchi. Expansion and conversion models in the lambda calculus from filters with polymorphic types. Manuscript, March 1989.

[27] Daniel Leivant. Discrete polymorphism (summary). In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 288–297, 1990.

[28] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[29] Spiro Michaylov and Frank Pfenning. Compiling the polymorphic λ-calculus. Ergo Report 89–088, School of Computer Science, Carnegie Mellon University, November 1989.

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

[31] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic lambda-calculus. *Theoretical Computer Science*, 1990. To appear. A preliminary version appeared in *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359, Springer-Verlag LNCS 352, March 1989.

[32] 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.

[33] Benjamin Pierce. Bounded quantification and intersection types. Thesis proposal (unpublished), September 1989.

[34] 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.

[35] 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.

[36] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[37] John Reynolds. Three approaches to type structure. In *Mathematical Foundations of Software Development*. Springer-Verlag, 1985. Lecture Notes in Computer Science No. 185.

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

[39] S. Ronchi della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28:151–169, 1984.