What are principal typings and what are they good for?

Technical Memorandum MIT/LCS/TM–532

Trevor Jim*
Laboratory for Computer Science
Massachusetts Institute of Technology

August 1995

Abstract

We demonstrate the pragmatic value of the *principal typing property*, a property more general than ML's principal type property, by studying a type system with principal typings. The type system is based on rank 2 intersection types and is closely related to ML. Its principal typing property provides elegant support for separate compilation, including "smartest recompilation" and incremental type inference, and for accurate type error messages. Moreover, it motivates a novel rule for typing recursive definitions that can type many examples of polymorphic recursion. Type inference remains decidable; this is surprising, since type inference for ML plus polymorphic recursion is undecidable.

Keywords: Polymorphic recursion, separate compilation, incremental type inference, error messages, intersection types.

1 Introduction

We would like to make a careful distinction between the following two properties of type systems.

Property A
Given: a term M typable in type environment A.
There exists: a type σ representing all possible types for M in A.

Property B
Given: a typable term M.
There exists: a typing $A \vdash M : \sigma$ representing all possible typings of M.

*545 Technology Square, Cambridge, MA 02139, trevor@theory.lcs.mit.edu. Supported by NSF grants CCR–9113196 and CCR–9417382.

Property A is the familiar *principal type property* of ML. By analogy, we will call Property B the *principal typing property*. The names are close enough to give us pause. In fact, some authors have used "principal typings" in reference to Property A. But "principal typings" is also the name traditionally applied to Property B, and we will not introduce a new name here.

Why do we care to make such a distinction? Property A—principal types—is certainly useful. But Property B—principal typings—is more useful still. We believe this has been overlooked because ML and its extensions completely dominate current research on type inference; and we know of no sense in which ML has principal typings. We will return to this point in §6.

In this paper, we demonstrate the usefulness of the principal typing property by studying a type system that has it. We emphasize that our results are motivated entirely by the general principal typing property, and not by the technical details of this particular case study. Any system with principal typings can benefit from our observations.

Nevertheless, we take some care in choosing our case study, so that its relevance to current practice will be immediately evident. Therefore, we seek a type system closely related to ML: it should be able to type all ML programs, it should have decidable type inference, and the complexity of type inference should be approximately the same as in ML.

The type system that satisfies all of these requirements is the system of *rank 2 intersection types*. This system is closely related to the more well-known rank 2 of System F—they type exactly the same terms—but it possesses the additional property of principal typings. We use a variant of this system, called $\mathbf{P}_2$, as our case study.

The distinction between principal types and principal typings is evident in the type inference algorithm for $\mathbf{P}_2$: it takes a single input, a term M, and produces two outputs, an A and σ such that $A \vdash M : \sigma$. The types required of the free variables of M are specified by A; but A is a byproduct of type inference, not a necessary input. Contrast this with Milner's algorithm for ML, whose let-polymorphism relies on A being an input.

We illustrate the benefits of principal typings in three areas: *recursive definitions*, *separate compilation*, and *accurate type error messages*.

Recursive definitions. The following well-known example was used by Mycroft [17] to illustrate the deficiencies of ML's handling of recursive definitions:

$$\begin{aligned} \mathit{map} &= \lambda f.\lambda l.\ \mathbf{if}\ \mathit{null}\ l\ \mathbf{then}\ \mathit{nil}\ \mathbf{else}\ f(\mathit{hd}\ l) :: \mathit{map}\ f\ (\mathit{tl}\ l) \\ \mathit{squarelist} &= \lambda l.\ \mathit{map}\ (\lambda x.\ x \times x)\ l \\ \mathit{complement} &= \lambda l.\ \mathit{map}\ (\lambda x.\ \mathit{not}\ x)\ l \end{aligned}$$

This program is not typable in ML when presented as a single, mutually recursive definition.[1] The rule of *polymorphic recursion* was introduced [17, 10] as a

[1] Note that *map* does not depend on the other functions, and it is possible for ML to type the program by considering the definition of *map* separately; but Mycroft exhibits natural

remedy:

$$\text{(REC-POLY)} \quad \frac{A_x \cup \{x : \sigma\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \text{(where } \sigma \text{ is an ML type scheme)}$$

The rule (REC-POLY) allows the body M of the recursive definition $(\mu x M)$ to be typed under the assumption that x has a polymorphic type. This is sufficient to handle the *map* example above. But now consider type inference using Milner's algorithm: in order to infer a type, σ, for the definition M, we need to know the type to use for the free variable x, that is, σ. Resolution of this "chicken and egg" problem is, in fact, impossible: type inference is undecidable [11, 5].

The principal typing property suggests a new rule for typing recursive definitions:

$$\frac{A_x \cup \{x : \tau\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \text{(where } \sigma \leq \tau\text{)}$$

In this rule, the type τ assumed for the recursive variable x need not be the same as the type σ derived for its definition M. The type τ expresses the requirements on x needed to give M the type σ; as long as σ meets these requirements ($\sigma \leq \tau$), it is safe to assume it as the type of the definition.

Now the strategy for type inference becomes clear: infer the principal typing $A \vdash M : \sigma$ for M, producing *both* σ and $\tau = A(x)$. It only remains to ensure $\sigma \leq \tau$, and this can be accomplished by *subtype satisfaction*, a procedure similar to unification. The result is a type system with decidable type inference, able to type many examples of polymorphic recursion, including the *map* example.

Separate compilation. In separate compilation, a large program is divided into smaller modules, each of which is type checked and compiled in isolation. The program as a whole is closed, but modules have free variables—a module may refer to other modules. Types play an important role in compilation; for instance, the data representations and calling conventions of a module may depend on its type. Thus the compiled machine code of a module may depend on the types of external variables that it references.

Consequently, most compilers require the user to specify the types of external variables referenced in each module. In $\mathbf{P}_2$, our ability to perform type inference on program fragments with free variables means that the user need not write these specifications: the compiler can infer them itself. More significantly, principal typings will enable us to achieve *smartest recompilation* [18], which guarantees that a module need not be recompiled unless its own definition changes. We also show that principal typings enable an elegant and efficient solution to a related problem, *incremental type inference* [1].

Error messages. Most compilers for strongly typed languages do not do a good job of pinpointing the location of type errors in programs; see Wand [21] for a discussion. As a final example of the utility of principal typings, we show

examples of functions that cannot be separated in this manner, and which cannot be typed in ML.

that principal typings help to produce error messages that accurately identify the source of type errors.

Organization of the paper. We introduce the type system $\mathbf{P}_2$ in §2, and state some of its basic properties. We describe how we type recursive definitions in §3, and we show how principal typings support separate compilation in §4. We describe how principal typings produce more accurate type error messages in §5. In §6, we address the question of whether principal typings exist for ML. In §7, we describe an extension of $\mathbf{P}_2$ with principal typings. We discuss related work in §8, and we summarize our results in §9. In an appendix, we show how our rule for typing recursive definitions could be added to ML. Proofs of all theorems can be found in a separate paper [6].

2 The type system

We now present our type system, in an expository manner. Uninteresting details have been placed in an appendix. For the most part, the system relies on familiar rules of subtyping and type assignment. However, the system is based on a notion of rank, and there are some complications due to the need to stay within rank. These complications are characteristic of all ranked systems.

Our programs are just the terms of the lambda calculus:

$$M \quad ::= \quad x \mid (M_1 M_2) \mid (\lambda x M).$$

Notice that our programs do not use ML's let-expressions. In our type system, **let** $x = M$ **in** N can be considered an abbreviation for $(\lambda x N)M$.

We will be defining several classes of types, each of which is a restriction of the types with quantification and intersection:

$$\sigma \quad ::= \quad t \mid (\sigma_1 \rightarrow \sigma_2) \mid (\forall t \sigma) \mid (\sigma_1 \wedge \sigma_2).$$

For those unfamiliar with intersection types, we present a brief example. A term of type $(\sigma \wedge \tau)$ is thought of as having *both* the type σ and the type τ. For example, the identity function has both type $(t \rightarrow t)$ and $(s \rightarrow s) \rightarrow (s \rightarrow s)$, so

$$(\lambda y.y) : (t \rightarrow t) \wedge ((s \rightarrow s) \rightarrow (s \rightarrow s)).$$

By this intuition, a quantified type stands for the *infinite* intersection of its instances:

$$(\lambda y.y) : (\forall u.u \rightarrow u).$$

The types $(t \rightarrow t)$ and $(s \rightarrow s) \rightarrow (s \rightarrow s)$ are instances of $(\forall u.u \rightarrow u)$, so in some sense this typing is "more general" than the first.

Our ranked system will allow only a limited use of intersections: they may only appear to the left of a single arrow. For example, we will be able to derive the following type in our system:

$$(\lambda x.xx) : \forall s,t.(s \wedge (s \rightarrow t)) \rightarrow t.$$

This says that as long as the argument of the function $(\lambda x.xx)$ has *both* the types s and $s \to t$, for some s and t, the result will be of type t. Note that this term is not typable in ML. An appropriate argument for this function is the identity function:

$$(\lambda x.xx)(\lambda y.y) : (\forall u.u \to u).$$

Again, we will be able to derive this type in our system. This example is typable in ML, *provided* it is translated into a let-expression:

$$\textbf{let } x = (\lambda y.y) \textbf{ in } xx : (\forall u.u \to u).$$

We now give the details of our ranked system, called $\mathbf{P}_2$. The sets $\mathbf{T}_0$, $\mathbf{T}_1$, $\mathbf{T}_2$, and $\mathbf{T}_{\forall 2}$ of types are defined inductively by the equations below.

$$\begin{aligned}
\mathbf{T}_0 &= \{\, t \mid t \text{ is a type variable} \,\} \cup \{(\sigma \to \tau) \mid \sigma, \tau \in \mathbf{T}_0\}, \\
\mathbf{T}_1 &= \mathbf{T}_0 \cup \{(\sigma \wedge \tau) \mid \sigma, \tau \in \mathbf{T}_1\}, \\
\mathbf{T}_2 &= \mathbf{T}_0 \cup \{(\sigma \to \tau) \mid \sigma \in \mathbf{T}_1, \tau \in \mathbf{T}_2\}, \\
\mathbf{T}_{\forall 2} &= \mathbf{T}_2 \cup \{(\forall t\sigma) \mid \sigma \in \mathbf{T}_{\forall 2}\}.
\end{aligned}$$

The set $\mathbf{T}_0$ is the set of simple types, and $\mathbf{T}_1$ is the set of finite, nonempty intersections of simple types. $\mathbf{T}_2$ is the set of rank 2 intersection types: these are types possibly containing intersections, but only to the left of a single arrow. Note that $\mathbf{T}_0 = \mathbf{T}_1 \cap \mathbf{T}_2$. Finally, $\mathbf{T}_{\forall 2}$ adds top-level quantification of type variables to $\mathbf{T}_2$.

Just as we have several classes of types, we have several subtyping relations.[2] Their definition is simplified by observing the following conventions: we consider types to be syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers; and we consider '$\wedge$' to be an associative, commutative, and idempotent operator, so that any $\mathbf{T}_1$ type may be considered a finite, nonempty set of simple types, written in the form $(\bigwedge_{i \in I} \sigma_i)$, where each $\sigma_i \in \mathbf{T}_0$.

Definition 1 For $i \in \{1, 2, \forall 2\}$, we define the relation $\leq_i$ as the least partial order on $\mathbf{T}_i$ closed under the following rules:

- If $\{\tau_j \mid j \in J\} \subseteq \{\sigma_i \mid i \in I\}$, then $(\bigwedge_{i \in I} \sigma_i) \leq_1 (\bigwedge_{j \in J} \tau_j)$.
- If $\sigma_1 \leq_1 \tau_1$ and $\tau_2 \leq_2 \sigma_2$, then $(\tau_1 \to \tau_2) \leq_2 (\sigma_1 \to \sigma_2)$.
- If $\sigma \leq_2 \tau$, then $\sigma \leq_{\forall 2} \tau$.
- If $\tau \in \mathbf{T}_0$, then $(\forall t\sigma) \leq_{\forall 2} \{t := \tau\}\sigma$.
- If $\sigma \leq_{\forall 2} \tau$ and t is not free in σ, then $\sigma \leq_{\forall 2} (\forall t\tau)$.

[2] These could be combined into a single subtyping relation, but it is technically convenient to keep them separate.

The first rule says that $\leq_1$ expresses the natural ordering on intersection types. The second rule says that $\leq_2$ obeys the usual antimonotonic ordering on function types, restricted to rank 2. The rules for $\leq_{\forall 2}$ express the intuition that a type is a subtype of its instances (we write $\{t := \tau\}\sigma$ for the substitution of τ for t in σ). They are equivalent to the following rule, similar to ML's notion of *generic instance*:

- If $\{\vec{s} := \vec{\rho}\}\sigma \leq_2 \tau$, where $\vec{\rho}$ is a vector of simple types, and the type variables $\vec{t}$ are not free in $(\forall \vec{s}\sigma)$, then $\forall \vec{s}\sigma \leq_{\forall 2} \forall \vec{t}\tau$.

Note that we only allow instantiation of simple types. This ensures that instantiation does not take us beyond rank 2. It also has less desirable implications, e.g., $(\forall t.t)$ is not a least type in the ordering $\leq_{\forall 2}$: $(\forall t.t) \not\leq_{\forall 2} (s_1 \wedge (s_1 \to s_2)) \to s_2$.

A fourth subtyping relation will play an important role in the type system. The relation $\leq_{\forall 2,1}$ between $\mathbf{T}_{\forall 2}$ and $\mathbf{T}_1$ is the smallest relation satisfying the rule:

- If $\sigma \leq_{\forall 2} \tau_i$ for all $i \in I$, then $\sigma \leq_{\forall 2,1} (\bigwedge_{i\in I} \tau_i)$.

The relation $\leq_{\forall 2,1}$ is not a partial order; it is not even reflexive. This is because it relates types "across rank." Note that in a comparison

$$(\forall t\sigma) \leq_{\forall 2,1} (\bigwedge_{i\in I} \tau_i),$$

the type variable t may be instantiated differently for each τ_i.

The type system derives judgments of the form $A \vdash M : \sigma$, where σ is a $\mathbf{T}_{\forall 2}$ type, and all of the types in A are $\mathbf{T}_1$ types. The typing rules are given below.

$$\text{(VAR)} \qquad A \cup \{x : \sigma\} \vdash x : \tau \qquad \text{(where } \sigma \leq_1 \tau \in \mathbf{T}_0\text{)}$$

$$\text{(ABS)} \qquad \frac{A_x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \to \tau}$$

$$\text{(APP)} \qquad \frac{A \vdash M : \tau_1 \to \tau_2 \quad A \vdash N : \sigma}{A \vdash (MN) : \tau_2} \quad \text{(where } \sigma \leq_{\forall 2,1} \tau_1\text{)}$$

$$\text{(GEN)} \qquad \frac{A \vdash M : \sigma}{A \vdash M : (\forall t\sigma)} \qquad \text{(where } t \text{ is not free in } A\text{)}$$

$$\text{(SUB)} \qquad \frac{A \vdash M : \tau}{A \vdash M : \sigma} \qquad \text{(where } \tau \leq_{\forall 2} \sigma\text{)}$$

Example 2 Recall that the typings

$$\begin{aligned} (\lambda x.xx) &: \forall s,t.(s \wedge (s \to t)) \to t, \\ (\lambda y.y) &: (\forall u.u \to u), \end{aligned}$$

hold in our system. Then by rule (SUB),

$$(\lambda x.xx) : (s \to s) \wedge ((s \to s) \to (s \to s)) \to (s \to s).$$

And $(\forall u.u \to u) \leq_{\forall 2,1} (s \to s) \wedge ((s \to s) \to (s \to s))$, so by rule (APP),

$$(\lambda x.xx)(\lambda y.y) : (s \to s).$$

Finally, by rule (GEN),

$$(\lambda x.xx)(\lambda y.y) : \forall s.s \to s.$$

We now give the definition of principal typings appropriate to our system.

Definition 3

i) A typing $B \vdash M : \tau$ is an *instance* of a typing $A \vdash M : \sigma$ if there is a substitution S such that $S\sigma \leq_{\forall 2} \tau$ and $B(x) \leq_1 S(A(x))$ for all $x \in \mathbf{dom}(A)$.

ii) A *principal typing* for a term M is a typing $A \vdash M : \sigma$ of which any other typing of M is an instance.

This definition is standard, c.f. [16]. Note in particular that the notion of instance is monotonic in the derived type, but antimonotonic in the type environment. The intuition is, a principal typing EXPECTS LESS of its free variables, and PROVIDES MORE than any other typing judgment.

The close connection between all of the rank 2 systems is expressed by the following theorem.

Theorem 4 *A term M is typable in $\mathbf{P}_2$ iff M is typable in rank 2 of System F iff M is typable in rank 2 of the intersection type discipline. Therefore, typability in $\mathbf{P}_2$ is DEXPTIME-complete.*

The equivalence between rank 2 of System F and the rank 2 intersection discipline has been shown independently by Yokouchi [23].

2.1 Subtype satisfaction

In order to perform type inference, we must solve *subtype satisfaction problems*, which generalize unification. Solving subtype satisfaction also gives a decision procedure for subtyping. We will focus on the relation $\leq_{\forall 2,1}$, as it is the most important for type inference; all of the other relations can be handled in a similar manner.

A $\leq_{\forall 2,1}$-*satisfaction problem* π is a pair $\exists \vec{s}.P$, where P is a set whose every element is either: 1) an equality between simple types; or 2) an inequality between a $\mathbf{T}_{\forall 2}$ type and a $\mathbf{T}_1$ type. A substitution S is a *solution* to $\exists \vec{s}.P$ if there is a substitution S' such that $S(t) = S'(t)$ for all $t \notin \vec{s}$, $S'\sigma \leq_{\forall 2,1} S'\tau$ for all inequalities $(\sigma \leq \tau) \in P$, and $S'\sigma = S'\tau$ for all equalities $(\sigma = \tau) \in P$. We write $\mathbf{MGS}(\pi)$ for the set of most general solutions to a $\leq_{\forall 2,1}$-satisfaction problem π (as with unification, most general solutions are not unique).

Theorem 5

i) *The relation $\leq_{\forall 2,1}$ is decidable.*

ii) *If a $\leq_{\forall 2,1}$-satisfaction problem π is solvable, then there is a most general solution for π. Moreover, there is an algorithm that decides, for any π, whether π is solvable, and, if so, returns a most general solution.*

Algorithms for deciding $\leq_{\forall 2,1}$ subtyping and solving $\leq_{\forall 2,1}$-satisfaction problems are given in Appendix B.

2.2 Type inference

The type inference algorithm is presented in the style favored by the intersection type community: for any M, we define a set, $\mathrm{PP}(M)$, called the *principal pairs of* M. Every element of $\mathrm{PP}(M)$ is a pair $\langle A, \sigma \rangle$ such that $A \vdash M : \sigma$ is a principal typing of M.

The following technical property is used to show that $\mathrm{PP}(M)$ indeed specifies a type inference algorithm: the set $\mathrm{PP}(M)$ is an equivalence class of pairs under permutations, i.e., $\langle A_1, \sigma_1 \rangle, \langle A_2, \sigma_2 \rangle \in \mathrm{PP}(M)$ iff $\langle A_1, \sigma_1 \rangle = S\langle A_2, \sigma_2 \rangle$ for some bijection S of type variables. Therefore, in choosing $\langle A, \sigma \rangle \in \mathrm{PP}(M)$ it is always possible to guarantee that the type variables of $\langle A, \sigma \rangle$ are "fresh".

To perform type inference, simply follow the definition of $\mathrm{PP}(M)$, choosing "fresh" type variables and using the **MGS** algorithm as necessary.

Definition 6 For any term M, the set $\mathrm{PP}(M)$ is defined by the following rules.

- If $M = x$, then for any type variable t, $\langle \{x : t\}, t \rangle \in \mathrm{PP}(x)$.
- If $M = \lambda x N$, and $\langle A, \forall \vec{s} \sigma \rangle \in \mathrm{PP}(N)$, where the type variables $\vec{s}$ are distinct from all other type variables, then:
  - i) If $x \notin \mathbf{dom}(A)$, and t is a type variable not appearing in $\langle A, \forall \vec{s} \sigma \rangle$, then $\langle A, \forall t \vec{s} (t \to \sigma) \rangle \in \mathrm{PP}(\lambda x N)$.
  - ii) If $x \in \mathbf{dom}(A)$, then $\langle A_x, \mathrm{Gen}(A_x, A(x) \to \sigma) \rangle \in \mathrm{PP}(\lambda x N)$.
- If $M = M_1 M_2$, the type variables of $\langle A_1, \forall \vec{s} \sigma_1 \rangle \in \mathrm{PP}(M_1)$ and $\langle A_2, \sigma_2 \rangle \in \mathrm{PP}(M_2)$ are disjoint, and the type variables $\vec{s}$ are distinct from all other type variables:
  - i) If σ_1 is a type variable t, t_1 and t_2 are fresh type variables, $U \in \mathbf{MGS}(\{\sigma_2 \leq t_1, t = t_1 \to t_2\})$, and $A = U(A_1 + A_2)$, then
  $$\langle A, \mathrm{Gen}(A, U t_2) \rangle \in \mathrm{PP}(M).$$
  - ii) If $\sigma_1 = \tau_1 \to \tau_2$, $U \in \mathbf{MGS}(\{\sigma_2 \leq \tau_1\})$, and $A = U(A_1 + A_2)$, then $\langle A, \mathrm{Gen}(A, U\tau_2) \rangle \in \mathrm{PP}(M)$.

Example 7 We show how the algorithm finds the type of $(\lambda x.xx)$.

i) $\mathrm{PP}(x)$ produces a pair $\langle \{x : t_1\}, t_1 \rangle$.

ii) $\mathrm{PP}(x)$ (again) produces a pair $\langle \{x : t_2\}, t_2 \rangle$.

iii) To calculate $\mathrm{PP}(xx)$, we find a most general solution to

$$\{t_2 \leq t_3, t_1 = t_3 \rightarrow t_4\},$$

such as $\{t_2 := t_3, t_1 := t_3 \rightarrow t_4\}$. Then $\langle \{x : t_3 \wedge (t_3 \rightarrow t_4)\}, t_4 \rangle \in \mathrm{PP}(xx)$.

iv) Finally, $\mathrm{PP}(\lambda x.xx)$ produces $\langle \emptyset, \forall t_3, t_4.t_3 \wedge (t_3 \rightarrow t_4) \rightarrow t_4 \rangle$.

Theorem 8 (Principal typings) *If M is typable in $\mathbf{P}_2$, then there is a pair $\langle A, \sigma \rangle \in \mathrm{PP}(M)$ such that $A \vdash M : \sigma$ is a principal typing for M.*

3 Recursive definitions

We now add recursive definitions to our language. A term of the form $(\mu x M)$ is meant to represent the program x such that $x = M$, where M may contain occurrences of x. The following rule is a straightforward way to type such definitions, and is the rule adopted by ML:

$$\text{(REC-SIMPLE)} \quad \frac{A_x \cup \{x : \sigma\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \text{(where } \sigma \text{ is a simple type)}$$

As remarked in the introduction, this *simple recursion* is not able to type the *map* example, and other examples of interest. And the rule (REC-POLY) is not appropriate for our system, because we do not allow quantified types in our type environments.

Instead, we propose the following rules for typing recursive definitions:

$$\text{(REC)} \quad \frac{A_x \cup \{x : \tau\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \text{(where } \sigma \leq_{\forall 2,1} \tau)$$

$$\text{(REC-VAC)} \quad \frac{A_x \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \text{(where } x \text{ is not free in } M)$$

The rule (REC-VAC) is necessary to type terms like

$$(\mu w(\lambda x.xx)) : \forall s, t.(s \wedge (s \rightarrow t)) \rightarrow t.$$

In order to use the rule (REC) in this case, we would need a type $\tau \in \mathbf{T}_1$ such that $\forall s, t.(s \wedge (s \rightarrow t)) \rightarrow t \leq_{\forall 2,1} \tau$. There is no such type, because s and $s \rightarrow t$ cannot be unified.

The rules (REC) and (REC-VAC) can type strictly more terms than (REC-SIMPLE), but not as many terms as (REC-POLY).

Example 9

i) The following term is typable in $\mathbf{P}_2$ + (REC) + (REC-VAC), but not in $\mathbf{P}_2$ + (REC-SIMPLE):

$$(\mu x.(\lambda yz.z)(xx)) : \forall t.t \rightarrow t.$$

The self-application xx cannot be typed if x is assigned just a simple type.

ii) The term $(\mu x.xx)$ is typable in ML+(REC-POLY), but not in $\mathbf{P}_2$+(REC)+(REC-VAC). In ML + (REC-POLY) it has type $(\forall t.t)$.

Now we add mutually recursive definitions to the language:

$$(\textbf{letrec } x_1 = M_1, \ldots, x_n = M_n \textbf{ in } M),$$

where all of the x_i are distinct. The corresponding typing rule is:

$$(\text{LETREC}) \quad \frac{\begin{array}{rl} & A_{x_1 \cdots x_n} \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash M : \sigma \\ (\forall i \leq n) & A_{x_1 \cdots x_n} \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash (\mu x_i M_i) : \sigma_i \end{array}}{\begin{array}{r} A \vdash (\textbf{letrec } x_1 = M_1, \ldots, x_n = M_n \textbf{ in } M) : \sigma \\ (\text{where } \forall i \leq n, \sigma_i \leq_{\forall 2,1} \tau_i) \end{array}}$$

In the hypothesis of this rule, we are careful to type each definition M_i as a recursive but not *mutually* recursive definition. Thus at first, each M_i needs to satisfy only the constraints on x_i implied by the occurrences of x_i in M_i itself; constraints implied by occurrences in M or other M_j are satisfied second. In between, the type of M_i can be generalized.

This is the technical trick that lets us type examples like *map*. Even when we are presented with *map*, *squarelist*, and *complement* in a mutually recursive definition, we will type each of them first without mutual recursion.

Example 10 Let $M_{\rm m}$, $M_{\rm s}$, and $M_{\rm c}$ abbreviate the definitions of *map*, *squarelist*, and *complement*, and let

$$\begin{array}{rcl} A & = & \{\mathit{map} : ((\text{int} \to \text{int}) \to \text{int list} \to \text{int list}) \\ & & \quad \wedge ((\text{bool} \to \text{bool}) \to \text{bool list} \to \text{bool list})\}. \end{array}$$

Since

$$\begin{array}{l} A \vdash (\mu \mathit{map}.M_{\rm m}) : \forall s,t.(s \to t) \to s \text{ list} \to t \text{ list}, \\ A \vdash (\mu \mathit{squarelist}.M_{\rm s}) : \text{int list} \to \text{int list}, \\ A \vdash (\mu \mathit{complement}.M_{\rm c}) : \text{bool list} \to \text{bool list}, \end{array}$$

and $(\forall s,t.(s \to t) \to s \text{ list} \to t \text{ list}) \leq_{\forall 2,1} A(\mathit{map})$, by rule (LETREC) the term

$$\begin{array}{ll} (\textbf{letrec} & \mathit{map} = M_{\rm m}, \\ & \mathit{squarelist} = M_{\rm s}, \\ & \mathit{complement} = M_{\rm c} \\ \textbf{in} & 0) \end{array}$$

is typable.

Adding the following clauses to Definition 6 gives a type inference algorithm.

- If $M = (\mu x N)$ and $\langle A, \sigma \rangle \in \text{PP}(N)$, then:
  - i) If $x \notin \textbf{dom}(A)$, then $\langle A, \sigma \rangle \in \text{PP}(M)$.
  - ii) If $x \in \textbf{dom}(A)$ and $U \in \textbf{MGS}(\{\sigma \leq_{\forall 2,1} A(x)\})$, then $\langle UA_x, \text{Gen}(UA_x, U\sigma) \rangle \in \text{PP}(M)$.

- If $M = (\mathbf{letrec}\ x_1 = M_1, \ldots, x_n = M_n\ \mathbf{in}\ M_0)$,

  and $\langle A_i, \sigma_i \rangle \in \mathrm{PP}(\mu x_i M_i)$ for $1 \leq i \leq n$,
  $\langle A_0, \sigma_0 \rangle \in \mathrm{PP}(M_0)$,
  $A' = A_0 + \Sigma_{1 \leq i \leq n} A_i$,
  $U \in \mathbf{MGS}(\{\sigma_i \leq A'(x_i) \mid 1 \leq i \leq n,\ x_i \in \mathbf{dom}(A')\})$,
  and $A'' = A'_{x_1, \ldots, x_n}$,

  then $\langle UA'', \mathrm{Gen}(UA'', U\sigma_0) \rangle \in \mathrm{PP}(M)$.

4 Separate compilation

Any separate compilation system manages a collection of small program fragments that together make up a single large program. Two questions must be answered by such a system. First, does the program as a whole type check? And second, how do we generate code for each program fragment, and how can we combine these code fragments into an executable program?

We consider each of these questions in turn.

4.1 Incremental type inference

The problem of *incremental type inference* [1] can be described as follows. A user develops a program in an incremental fashion, by entering a sequence of definitions to a read-eval-print loop:

$$x_1 = M_1, x_2 = M_2, x_3 = M_3, \ldots$$

After each definition is entered, the compiler performs type inference to ensure the type-correctness of the partial program. Definitions may be re-defined as the programmer detects and corrects bugs, and they may be mutually recursive. Most relevant, a "bottom-up" style of program development is made possible by allowing definitions to refer to other definitions which have not yet been entered. This is exactly the strength of principal typings: type inference can be performed without knowing the types of free variables.

Incremental type inference is simply the type checking task of separate compilation, on an extremely fine scale: not just every module, but every definition is typed and compiled separately. The type checking task can be solved elegantly and efficiently using principal typings.

Consider a partial program $x_1 = M_1, \ldots, x_n = M_n$. If a variable is defined twice, the latter definition takes precedence; let $y_1 = N_1, \ldots, y_m = N_m$ be the sequence with duplicates discarded. To check that the program is well-typed, we can perform type inference on the expression

$$(\mathbf{letrec}\ y_1 = N_1, \ldots, y_m = N_m\ \mathbf{in}\ 0).$$

By our algorithm, type inference requires, first, computing the principal pair $\langle A_i, \sigma_i \rangle$ of each $(\mu y_i N_i)$, and second, reconciling the type of each definition with

its uses. That is, if $A = \Sigma A_i$, we want to satisfy the problem

$$\{\sigma_i \leq A(y_i) \mid 1 \leq i \leq m, y_i \in \mathbf{dom}(A)\}.$$

We have already shown how to accomplish both tasks.

Now suppose the user enters a brand new definition, $y_{m+1} = N_{m+1}$. Again, we must compute $\langle A_i, \sigma_i \rangle$ for each $(\mu y_i N_i)$. But if $i \neq m+1$, then *by the principal typing property, $\langle A_i, \sigma_i \rangle$ is unchanged.* The principal pair for each definition need only be computed once, as it is entered by the user; it does not need to be recomputed at each new definition or re-definition.

We must also calculate a solution to the new satisfaction problem. However, the new problem is almost identical to the previous problem, adding only a few more constraints. We may be able to incorporate large parts of the old solution into the new solution. Our algorithm for subtype satisfaction, described in Appendix B, solves problems by transforming them into equivalent, simpler problems until a solution is reached. Such an algorithm is ideally suited to incorporating parts of the old solution. The transformations that applied to the old problem will, for the most part, be identical to the transformations applicable to the new problem.

4.2 Smartest recompilation

Once we have solved the type checking task of separate compilation, we face the task of code generation. Types determine data representations, calling conventions, and other implementation details. Thus we regard compilers as functions from typing judgments to machine code. For example, the compilation of a module M that imports a module x can be written

$$\text{Compile}(\{x : \sigma\} \vdash M : \tau) \quad = \quad \langle \text{machine code for } M \rangle.$$

There are two difficulties with this strategy. First, the compiler requires as input a typing judgment, or, at least, the types of external variables. The typical solution is to require the user to supply the types. A better solution is available in $\mathbf{P}_2$, where the compiler itself can infer a judgment $\{x : \sigma\} \vdash M : \tau$ for a term M with free variable x.

The second difficulty arises when we need to link all of the code fragments together into a single program. In particular, consider *re*compilation, in which a user changes a single module x and the system attempts to recompile as small a portion of the entire program as possible. Certainly the definition of x must be recompiled. Moreover, an *unchanged* module M that imports x may have to be recompiled: if the type of x changes, then the *typing judgment* of M, and thus its compiled output, changes.

This is where principal typings help. Suppose that we have compiled a module M by compiling its principal typing, $A \vdash M : \tau$. At link time, we discover that in order to be consistent with the rest of the program, we should instead have compiled M by a different typing, $B \vdash M : \sigma$. The principal typing property tells us that the second judgment is an *instance* of the first: in $\mathbf{P}_2$,

it can be obtained by substitution and subsumption from the principal typing. More formally,

$$\langle B, \sigma \rangle \quad = \quad \mathcal{C}\langle A, \tau \rangle,$$

where $\mathcal{C}$ is an operator that applies substitution and subsumption to the pair $\langle A, \tau \rangle$.

Stating the problem in this way lets us study the operator $\mathcal{C}$ in isolation. The operations of substitution and subsumption specified by $\mathcal{C}$ can be implemented via *coercions*. These coercions can be "wrapped" around the code generated for the typing $A \vdash M : \sigma$ at link time, making it behave like code generated for $B \vdash M : \tau$. That is,

$$\text{Compile}(B \vdash M : \sigma) \quad \cong \quad \text{Link}(\mathcal{C}, \text{Compile}(A \vdash M : \tau)),$$

where Link produces machine code that implements the coercions specified by $\mathcal{C}$.

Using this strategy, *a module need not be recompiled unless its definition changes*. This property was dubbed *smartest recompilation* by Shao and Appel [18]. They achieved smartest recompilation for ML by relating ML to a restriction of $\mathbf{P}_2$ with principal typings.

Shao and Appel identified the following problem with smartest recompilation. If a module references many free variables, e.g., functions from the standard library, then the type environment of the principal typing becomes large. This can be alleviated in the following way. Let B be a type environment specifying the $\mathbf{T}_{\forall 2}$ types of our library functions. We modify our type system to use two type environments, so that typings are of the form

$$A, B \vdash M : \sigma.$$

We modify our old rules to ignore this new type environment, and add a rule that allows us to use it:

$$(\textsc{var-new}) \quad A, B \cup \{x : \sigma\} \vdash x : \sigma$$

This system does not have principal typings, but it does have a useful "weak" form of principal typing property: given a term M typable in type environment B, there exists a typing $A, B \vdash M : \sigma$ representing all possible typings for M in B. We say that M has a principal typing *with respect to* the type environment B, and that we have smartest compilation *with respect to* B. Since B only specifies types for identifiers that are relatively stable, we gain most of the benefits of full smartest recompilation.

As an aside, we remark that this immediately suggests an extension to the type system: restore let-expressions to the language and add the rule

$$(\textsc{let}) \quad \frac{A, B \vdash M : \sigma, \quad A, B_x \cup \{x : \sigma\} \vdash N : \tau}{A, B \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N : \tau}$$

We call this a "rank 2.5" system, since it lies between ranks 2 and 3. For instance, it can type a term that is untypable in rank 2:

$$\mathbf{let}\ g = (\lambda x.xx)\ \mathbf{in}\ g(\lambda y.y) : \forall t.t \to t.$$

We will not pursue this further, because we already know how to extend $\mathbf{P}_2$ to a more general system, called $\mathbf{P}$, that does not rely on let-polymorphism. The description of $\mathbf{P}$ will appear in a future paper.

We do not claim that we have solved the smartest recompilation problem for Standard ML. Standard ML has a rich module system, with type components in modules, and generative, user-definable, recursive datatypes. Our simple language does not support such features (nor does the work of Shao and Appel [18]). However, we have identified principal typings, or some equivalent, as the key ingredient of such a system.

5 Error messages

Up until now, we have concentrated on one benefit of principal typings: a term can be given a type without regard to the definitions of its free variables.

The flip side of this benefit is that a definition can be typed independently of its uses. We now show how this allows us to produce accurate error messages when our type inference algorithm is faced with a program containing type errors.

Consider a definition, $(\lambda x M)N$, in which some uses of the variable x cause type errors: they require types that N cannot satisfy. To perform type inference, we calculate the principal typings of both the operator and the operand, say

$$A \vdash (\lambda x M) : \sigma \to \tau,$$
$$A' \vdash N : \sigma'.$$

By the principal typing property, we can calculate these principal typings in any order. To complete type inference, we simply check whether we can satisfy $\sigma' \leq_{\forall 2,1} \sigma$. At this point we will discover all of the type errors related to x: the type σ' will not be able to satisfy some of the constraints expressed by σ. If we take care to label each constraint with the use of x that produced it, we can output the offending uses, all in one batch.

Contrast this with the situation in ML. Assuming the definition is polymorphic, we must perform type inference on a let-expression $\mathbf{let}\ x = N\ \mathbf{in}\ M$. Without principal typings, we are forced to first calculate the principal type, σ, of N. We then process M, instantiating σ at each use of x. Errors are reported as they are encountered, at each use. But note, the errors of one definition can be interspersed with errors for other definitions, or with run-on errors. And the type σ may have been specialized for that particular (erroneous) use, leaving the programmer to understand a type only remotely related to the type σ of the definition.

6 Does ML have principal typings?

We have deliberately stated the principal typing property in a broad way, so that it can be applied to many different type systems.[3] In particular, we have not precisely defined what it means to *represent* all possible typings, because this will vary from one type system to another.

This imprecision makes it impossible for us to *prove* that a given type system lacks the principal typing property. Nevertheless, we do not know of a sensible formulation of principal typings for ML, and in particular, ML does not have principal typings in the sense of our Definition 3. For example, consider the following ML typings of the term xx.

$$\{x : \forall t.t\} \vdash xx : \forall t.t,$$
$$\{x : \forall t.t \to t\} \vdash xx : \forall t.t \to t.$$

Our intuition is that a principal typing EXPECTS LESS of its free variables and PROVIDES MORE than any other typing. We certainly cannot hope to derive a more general type for the term xx than $(\forall t.t)$, so the first judgment provides more than the second. However, the first judgment also makes a strong requirement on x: the type environment indicates that it too must have type $(\forall t.t)$. Thus the second judgment expects less than the first, and neither typing is more general than the other. Moreover, there is no typing more general than both the typings above. The obvious candidate,

$$\{x : \forall t.t \to t\} \vdash xx : \forall t.t,$$

is not derivable.

Why doesn't ML's principal type property imply the existence of principal typings? You might think that the principal typing of a term could be obtained from the principal type of the λ-closure of the term. But ML has only a restricted abstraction rule:

$$(\text{ABS}) \quad \frac{A_x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \to \tau} \quad \text{(where } \sigma \text{ is a simple type)}$$

In ML, we cannot abstract over variables of polymorphic type; the only way of introducing polymorphic variables is through let-expressions. In languages with a "true" abstraction rule, the principal type property and the principal typing property may coincide. This is the case in $\mathbf{P}_2$. It is also the case in rank 2 of System F, which *lacks* both principal types and principal typings [12].

If we want to work in a language lacking the principal typing property, we may still achieve some of its benefits by finding a "representation" for all possible typings. That is, we may relax the principal typing condition that the representatives themselves be typings.

Pushed to an extreme, this is nonsense—after all, M itself is a representation of all typings of M! But there is a middle ground. An example is the smartest

[3] In fact, we could have stated it more broadly still: we assumed typing judgments were of the form $A \vdash M : \sigma$, but this is not always the case.

recompilation system of Shao and Appel [18]. Their type system is a restriction of $\mathbf{P}_2$. Its typings are not ML typings, but for any ML term M, there is a typing in their system that encodes all of the ML typings for M. Similarly, typings in either $\mathbf{P}_2$ or rank 2 of the intersection system can be regarded as the principal "typings" of rank 2 of System F.

7 An extension

The system $\mathbf{P}_2$ is the rank 2 fragment of a type system, $\mathbf{P}$, that can type many more terms. The description of $\mathbf{P}$ is beyond the scope of this paper. However, we will present a few examples of its typing power.

If we define terms M and N by

$$\begin{aligned} M &= (\lambda g.g(\lambda f.f(\lambda x.x))), \\ N &= (\lambda w.w(\lambda y.yy)), \end{aligned}$$

then the following typings hold in $\mathbf{P}$:

$$\begin{aligned} M &: \forall t.((\forall s.((\forall u.u \to u) \to s) \to s) \to t) \to t, \\ N &: \forall u.((\forall st.(s \wedge (s \to t)) \to t) \to u) \to u, \\ MN &: \forall t.t \to t. \end{aligned}$$

Only M is typable in ML or $\mathbf{P}_2$, and only at less informative types. Note that in the type of M, the inner quantifier, $\forall u$, is under the left of four arrows, well beyond rank 2.

The system $\mathbf{P}$ has the principal typing property, decidable type inference, and a rule in the style of (REC) for typing recursive definitions. The crucial technical advance is a way of solving subtype satisfaction problems for types with quantifiers and intersections at arbitrary depth.

8 Related work

Principal typings are not a new concept. A number of existing type systems have principal typings, including the simply typed lambda calculus [22], the system of recursive types [4], the system of simple subtypes [16], and the system of intersection types [3]. Our contribution is to highlight the practical uses of the principal typing property, and to distinguish it from the principal type property. A number of authors have published offhand claims that ML possesses the principal typing property, which suggests that this distinction is not widely appreciated.

The system of rank 2 intersection types is also not new, but as with the principal typing property, it has attracted little attention. It was first suggested by Leivant in 1983 [14], but he did not give a formal definition of the type inference algorithm or proof of correctness. In an oft-referenced 1984 paper [15], McCracken gave a type inference algorithm for rank 2 of System F, inspired by

Leivant's ideas. This algorithm is incorrect. A correct algorithm for rank 2 of System F was finally given by Kfoury and Wells [12] in 1993. Their algorithm is completely unrelated to Leivant's algorithm. The earliest formal definition and proof of Leivant's algorithm was published in 1993, by van Bakel [20].

Our addition of top-level quantification is a significant technical improvement to the rank 2 intersection system, allowing a smoother development. In particular, the simplicity of our rule for typing recursive definitions is due to the power of quantifiers and the subtyping relation $\leq_{\forall 2,1}$. It is possible to formulate an equivalent rule for typing recursive definitions without top-level quantification, but the machinery is cumbersome and simply duplicates the functionality of the quantifiers.

We have shown that rank 2 of System F is closely related to our type system. However, rank 2 of System F does not have principal types or principal typings [12]. Launchbury and Peyton Jones [13] describe an interesting constant with a rank 2 System F type. Rank 2 System F types are not part of our type system, and we do not know how to handle their constant without resort to a special typing rule. This is the same solution employed by Launchbury and Peyton Jones.

The system of Aiken and Wimmers [2] uses ML's let-polymorphism, and, therefore, we believe it does not have principal typings. The subsystem without let-polymorphism, though, is still of interest, and may have principal typings (but this is not clear). The constraint-based systems of Jones [7], Kaes [9], and Smith [19] are also based on ML.

Constraint satisfaction, including subtype satisfaction, is an important component of each of these systems. Our method for solving constraints involving quantifiers ($\leq_{\forall 2,1}$-satisfaction) is a significant advance over these systems. Along with intersections, this is the central mechanism by which let-polymorphism is avoided and principal typings are achieved. In our work on the system $\mathbf{P}$, we will show how to solve some subtype satisfaction problems for types with quantifiers and intersections at arbitrary depth, giving type inference for a system with a much richer class of types.

9 Conclusion

We have shown that the principal typing property has practical applications, including smartest recompilation, incremental type inference, and accurate type error messages. Inspired by the principal typing property, we proposed a novel rule for typing recursive definitions. The type inference algorithm of our system $\mathbf{P}_2$ is easily extended to infer principal typings for recursive definitions under the new rule, resulting in a type system with decidable type inference that can type many examples of polymorphic recursion.

A number of languages, including ML, seem to lack the principal typing property. In such languages, we may achieve some of the benefits of principal typings by finding a way to represent all possible typings for terms.

Although our primary goal was to draw attention to the principal typing

property, a secondary contribution is to draw attention to the system of rank 2 intersection types, which also seems to have been overlooked. Our particular version of this system, $\mathbf{P}_2$, makes an important technical contribution by showing how to solve subtype satisfaction problems for types containing quantifiers. Our types only have quantifiers at top level, but the method is easily extended to types with quantifiers at arbitrary depth, as we will show in a forthcoming paper.

Acknowledgments. This paper has benefited from the comments of Assaf Kfoury, Albert Meyer, Jens Palsberg, and Mona Singh.

References

[1] Shail Aditya and Rishiyur Nikhil. Incremental polymorphism. In *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 379–405. Springer-Verlag, 1991.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, June 1993.

[3] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symbolic Logic*, 48(4):931–940, December 1983.

[4] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.

[5] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[6] Trevor Jim. Rank 2 type systems and recursive definitions. Technical Memorandum MIT/LCS/TM–531, Massachusetts Institute of Technology, August 1995.

[7] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, November 1994.

[8] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, chapter 8, pages 257–321. MIT Press, 1991.

[9] Stefan Kaes. Typing in the presence of overloading, subtyping, and recursive types. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 193–204, 1992.

[10] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. A proper extension of ML with an effective type-assignment. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 58–69, 1988.

[11] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[12] A.J. Kfoury and J.B. Wells. A direct algorithm for type inference in the rank 2 fragment of the second-order lambda-calculus. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 196–207, 1994.

[13] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 24–35, 1994.

[14] Daniel Leivant. Polymorphic type inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–98, 1983.

[15] Nancy McCracken. The typechecking of programs with implicit type structure. In G. Kahn, D.B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 301–315, June 1984.

[16] John Mitchell. Type inference with simple subtypes. *J. Functional Programming*, 1(3):245–285, July 1991.

[17] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the International Symposium on Programming, Toulouse*, volume 167 of *Lecture Notes in Computer Science*, pages 217–239. Springer-Verlag, 1984.

[18] Zhong Shao and Andrew Appel. Smartest recompilation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450, 1993.

[19] Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[20] Steffen van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Mathematisch Centrum, Amsterdam, February 1993.

[21] Mitchell Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, 1986.

[22] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

[23] Hirofumi Yokouchi. Embedding a second-order type system into an intersection type system. *Information and Computation*, 117(2):206–220, March 1995.

A Technical details of the type system

We use $x, y, \ldots$ to range over a countable set of (term) variables, and $M, N, \ldots$ to range over terms. The terms of the language are just the terms of the lambda calculus:

$$M \quad ::= \quad x \mid (M_1 M_2) \mid (\lambda x M).$$

Terms are considered syntactically equal modulo renaming of bound variables, and we adopt the usual conventions that allow us to omit parentheses: application associates to the left, and the scope of an abstraction λx extends to the right as far as possible. We write $\lambda x_1 \cdots x_n.M$ for $(\lambda x_1(\cdots(\lambda x_n M)\cdots))$.

We use $s, t, u \ldots$ to range over a countable set, $\mathbf{Tv}$, of type variables, and $\sigma, \tau, \ldots$ to range over types. We define several classes of types, each of which is a restriction of the types with quantification and intersection:

$$\sigma \quad ::= \quad t \mid (\sigma_1 \to \sigma_2) \mid (\forall t \sigma) \mid (\sigma_1 \wedge \sigma_2).$$

The constructor '$\wedge$' binds more tightly than '$\to$', e.g., $\sigma \wedge \tau \to t$ means $(\sigma \wedge \tau) \to t$, and the scope of a quantifier '$\forall$' extends as far to the right as possible. If $\vec{t} = t_1, t_2, \ldots, t_n$, $n \geq 0$, and $\sigma \in \mathbf{T}_2$, we write $(\forall \vec{t} \sigma)$ for the type $(\forall t_1(\forall t_2(\ldots(\forall t_n \sigma)\ldots)))$.

A type environment is a finite set $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of (variable, type) pairs, where the variables $x_1, \ldots, x_n$ are distinct, and $\sigma_1, \ldots, \sigma_n \in \mathbf{T}_1$. We use A to range over type environments. We write $A(x)$ for the type paired with x in A, $\mathbf{dom}(A)$ for the set $\{x \mid \exists \tau.(x : \tau) \in A\}$, and A_x for the type environment A with any pair for the variable x removed. We write $A_1 \cup A_2$ for the union of two type environments; by convention we assume that the domains of A_1 and A_2 are disjoint. We define $A_1 + A_2$ as follows: for each $x \in \mathbf{dom}(A_1) \cup \mathbf{dom}(A_2)$,

$$(A_1 + A_2)(x) = \begin{cases} A_1(x) & \text{if } x \notin \mathbf{dom}(A_2), \\ A_2(x) & \text{if } x \notin \mathbf{dom}(A_1), \\ A_1(x) \wedge A_2(x) & \text{otherwise.} \end{cases}$$

We write $\mathrm{Gen}(A, \sigma)$ for the $\forall$-closure of variables free in σ but not A.

B A subtype satisfaction algorithm

A *unification problem* is a satisfaction problem involving only equalities. Unification algorithms, such as Robinson's algorithm, can determine, for any uni-

fication problem, whether a solution exists, and, if so, produce a most general solution. Two problems are *equivalent* if they have the same solutions.

We will show how to transform a $\leq_{\forall 2,1}$-satisfaction problem into an equivalent unification problem. The transformation is defined by rules of the form

$$\sigma \leq \tau \quad \Rightarrow \quad \exists \vec{s}.P.$$

The rules may need to introduce fresh type variables, that is, type variables that do not appear on the left-hand side. These variables will appear in the variables $\vec{s}$ of the right-hand side (but they are not the only source of variables in $\vec{s}$).

The rules are used to define a rewrite relation on problems:

$$\frac{\sigma \leq \tau \quad \Rightarrow \quad \exists \vec{t}.P}{\exists \vec{s}.P' \uplus \{\sigma \leq \tau\} \quad \Rightarrow \quad \exists \vec{s} \uplus \vec{t}.P' \cup P}$$

The operator '$\uplus$' is disjoint union; on the right of the consequent, it means that the variables $\vec{t}$ must be fresh (this can always be achieved by renaming).

The rules for transforming a $\leq_{\forall 2,1}$-unification problem into a unification problem are given below.

$$\begin{array}{lcl}
(\sigma_1 \rightarrow \sigma_2) \leq t & \Rightarrow & \exists t_1, t_2.\{t_1 \leq \sigma_1, \sigma_2 \leq t_2, t = t_1 \rightarrow t_2\} \\
 & & \text{if } t_1, t_2 \text{ are fresh} \\
(\sigma_1 \rightarrow \sigma_2) \leq (\tau_1 \rightarrow \tau_2) & \Rightarrow & \{\tau_1 \leq \sigma_1, \sigma_2 \leq \tau_2\} \\
\sigma \leq (\tau_1 \wedge \tau_2) & \Rightarrow & \{\sigma \leq \tau_1, \sigma \leq \tau_2\} \\
t \leq \tau & \Rightarrow & \{t = \tau\} \\
 & & \text{if } \tau \text{ is a simple type} \\
(\forall t \sigma) \leq \tau & \Rightarrow & \exists t\{\sigma \leq \tau\} \\
 & & \text{if } \tau \text{ is not a } \wedge\text{-type, and } t \text{ is not free in } \tau
\end{array}$$

To see that these rules constitute an algorithm for producing an equivalent unification problem, observe that the rules preserve solutions, that the system is terminating, and that normal forms contain no inequalities, and thus are unification problems.

A unification algorithm in a transformational style, taken from [8], is given

below.[4]

$$
\begin{array}{lcl}
P \uplus \{\sigma = \sigma\} & \Rightarrow & P \\
P \uplus \{\sigma_1 \to \sigma_2 = \tau_1 \to \tau_2\} & \Rightarrow & P \cup \{\sigma_1 = \tau_1, \sigma_2 = \tau_2\} \\
P \uplus \{t_1 = t_2\} & \Rightarrow & \{t_1 := t_2\}P \cup \{t_1 = t_2\} \\
 & & \text{if } t_1, t_2 \in \mathrm{FTV}(P) \text{ and } t_1 \neq t_2 \\
P \uplus \{t = \sigma\} & \Rightarrow & F \\
 & & \text{if } t \in \mathrm{FTV}(\sigma) \text{ and } \sigma \notin \mathbf{T_V} \\
P \uplus \{t = \sigma\} & \Rightarrow & \{t := \sigma\}P \cup \{t = \sigma\} \\
 & & \text{if } t \notin \mathrm{FTV}(\sigma),\ \sigma \notin \mathbf{T_V}, \text{ and } t \in \mathrm{FTV}(P)
\end{array}
$$

The normal forms of the rewrite system are in *solved form*, a set of equations that corresponds immediately to a most general substitution. Note the special problem F, used to denote failure of unification.

The combination of these two transformation systems is an algorithm for finding most general solutions to $\leq_{\forall 2,1}$-satisfaction problems. As a special case, we obtain a decision procedure for subtyping: to see whether $\sigma \leq_{\forall 2,1} \tau$, compute a member of $\mathbf{MGS}(\{\sigma \leq \tau\})$ and check whether it is the identity (empty) substitution.

C Typing recursive definitions in ML

In this appendix, we show how to integrate our type inference strategy for recursive declarations into ML. The result is not as elegant as our rank 2 system, but it demonstrates that an existing ML implementation could easily be modified to take advantage of our rules.

To simplify our presentation we make the following assumptions, all of which can be relaxed without technical difficulty. We assume that no variable is bound more than once, and free and bound variables are distinct; and that variables are divided into two classes, *recursive* variables, which can be bound only by **letrec** and μ, and *ordinary* variables, which can be bound by λ and **let**. We use w to range over recursive variables and x to range over ordinary variables.

We introduce a new type system with judgments of the form

$$A, B \vdash M : \sigma,$$

where A maps recursive variables to $\mathbf{T}_1$ types, B maps ordinary variables to ML type schemes, and σ is an ML type scheme. The environments A and B are kept separate for purposes of presentation; they might well be merged in an implementation.

[4] This particular unification algorithm is inefficient, because the size of the output may be exponential in the size of the input. It is possible to specify efficient unification algorithms in this style, but in order to simplify the presentation we use this more straightforward algorithm.

(VAR-REC)	$A \cup \{w : \sigma\}, B \vdash w : \tau$	$(\sigma \leq_1 \tau \in \mathbf{T}_0)$
(REC)	$\dfrac{A \cup \{w : \tau\}, B \vdash M : \sigma}{A, B \vdash (\mu w M) : \sigma}$	$(\sigma \succ \tau)$
(LETREC)	$\dfrac{\begin{array}{c} A \cup \{w_1 : \tau_1, \ldots, w_n : \tau_n\}, B \vdash M : \sigma \\ (\forall i \leq n) \quad A \cup \{w_1 : \tau_1, \ldots, w_n : \tau_n\}, B \vdash (\mu w_i M_i) : \sigma_i \end{array}}{\begin{array}{r} A, B \vdash (\mathbf{letrec}\ w_1 = M_1, \ldots, w_n = M_n\ \mathbf{in}\ M) : \sigma \\ (\forall i \leq n, \sigma_i \succ \tau_i) \end{array}}$	
(VAR)	$A, B \cup \{x : \sigma\} \vdash x : \sigma$	
(ABS)	$\dfrac{A, B \cup \{x : \tau_1\} \vdash M : \tau_2}{A, B \vdash (\lambda x M) : \tau_1 \rightarrow \tau_2}$	$(\tau_1, \tau_2 \in \mathbf{T}_0)$
(APP)	$\dfrac{A, B \vdash M : \tau_1 \rightarrow \tau_2 \quad A, B \vdash N : \tau_1}{A, B \vdash (MN) : \tau_2}$	
(GEN)	$\dfrac{A, B \vdash M : \sigma}{A, B \vdash M : (\forall t \sigma)}$	(t is not free in A or B)
(INST)	$\dfrac{A, B \vdash M : (\forall t \sigma)}{A, B \vdash M : \{t := \tau\}\sigma}$	$(\tau \in \mathbf{T}_0)$

Figure 1: A new way of typing recursive definitions in ML

The typing rules are given in Figure 1. The rules (REC), (VAR-REC), and (LETREC) are new; all of the other rules have been taken from ML, and modified to handle the type environment A. Note that this system does not have the rule (REC-VAC); since we do not use rank 2 types, it is not necessary.

The rules (REC) and (LETREC) use a subtyping relation, $\succ$, that is the restriction of $\leq_{\forall 2,1}$ so that only ML type schemes can appear in the left position. It can be defined by the following rules:

- $\forall \vec{t}\sigma \succ \{\vec{t} := \vec{\tau}\}\sigma$, where σ and every τ_i is a simple type.
- If $\sigma \succ \tau_i$ for all $i \in I$, then $\sigma \succ (\bigwedge_{i \in I} \tau_i)$.

Because the relation does not involve proper rank 2 types, the corresponding subtype satisfaction problem can be solved more easily than the general case. The problems are of the form $\exists \vec{s}.P$, where P is a set of equalities between simple types and inequalities ($\sigma \succ \tau$) between ML type schemes and $\mathbf{T}_1$ types. The following rules are sufficient to transform such problems into equivalent unification problems.

$$\begin{array}{lcl} \sigma \succ \tau & \Rightarrow & \{\sigma = \tau\} \\ & & \text{if } \sigma, \tau \in \mathbf{T}_0 \\ \\ \sigma \succ (\tau_1 \wedge \tau_2) & \Rightarrow & \{\sigma \succ \tau_1, \sigma \succ \tau_2\} \\ \\ (\forall t \sigma) \succ \tau & \Rightarrow & \exists t \{\sigma \succ \tau\} \\ & & \text{if } \tau \text{ is not a } \wedge\text{-type, and } t \text{ is not free in } \tau \end{array}$$

To find $U \in \mathbf{MGS}(\pi)$, use the rules to transform π into π' with only equalities, and then use unification on π' to find U.

Now we consider how to perform type inference. The type inference algorithm W^* of Shao and Appel [18] can easily be extended to this system. But most compilers are based on Milner's algorithm W, so we use W as a starting point.

The modified algorithm, called W', is given in Figure 2, and it behaves as follows. If $W'(B, M) = \langle A, S, \tau \rangle$, the principal typing of M with respect to B is

$$A, SB \vdash M : \mathrm{Gen}(A, SB, \tau).$$

Note that we have extended the operator Gen so that $\mathrm{Gen}(A, B, \sigma)$ is the $\forall$-closure of σ by type variables that do not appear free in A or B. To simplify our presentation, we have only considered the case of **letrec** expressions that define exactly two variables. The general case introduces no technical difficulties.

$W'(B, M) =$ case M of

$w \quad \Rightarrow$ let t be a fresh type variable
in $\langle \{w : t\}, \{\}, t \rangle$

$(\mu w M_1) \Rightarrow$ let $\langle A, S, \tau \rangle = W'(M_1)$
in if $w \notin \mathbf{dom}(A)$ then $\langle A, S, \tau \rangle$
else let $\sigma = \mathrm{Gen}(A, SB, \tau)$
$U \in \mathbf{MGS}(\{\sigma \succ A(w)\})$
$\forall \vec{t}\tau' = U\sigma$
$\vec{s}$ be fresh type variables
in $\langle UA_w, US, \{\vec{t} := \vec{s}\}\tau' \rangle$

(**letrec** $w_1 = M_1, w_2 = M_2$ **in** M_0)
$\Rightarrow$ let $\langle A_0, S_0, \tau_0 \rangle = W'(B, M_0)$
$\langle A_1, S_1, \tau_1 \rangle = W'(S_0 B, (\mu w_1 M_1))$
$\langle A_2, S_2, \tau_2 \rangle = W'(S_1 S_0 B, (\mu w_2 M_2))$
$A = S_2 S_1 A_0 + S_2 A_1 + A_2$
$S = S_2 S_1 S_0$
$\sigma_1 = \mathrm{Gen}(A, SB, S_2 \tau_1)$
$\sigma_2 = \mathrm{Gen}(A, SB, \tau_2)$
$U \in \mathbf{MGS}(\{\sigma_i \succ A(w_i) \mid i \in \{1, 2\}, w_i \in \mathbf{dom}(A)\})$
in $\langle UA_{w_1, w_2}, US, US_2 S_1 \tau_0 \rangle$

$x \quad \Rightarrow$ if $x \notin \mathbf{dom}(B)$ then fail
else let $\forall \vec{t}\tau = B(x)$
$\vec{s}$ be fresh type variables
in $\langle \{\}, \{\}, \{\vec{t} := \vec{s}\}\tau \rangle$

$\lambda x M_1 \quad \Rightarrow$ let t be a fresh type variable
$\langle A, S, \tau \rangle = W'(B \cup \{x : t\}, M_1)$
in $\langle A, S, St \to \tau \rangle$

$M_1 M_2 \quad \Rightarrow$ let $\langle A_1, S_1, \tau_1 \rangle = W'(B, M_1)$
$\langle A_2, S_2, \tau_2 \rangle = W'(S_1 B, M_2)$
t be a new type variable
$S_3 \in \mathbf{MGS}(\{S_2 \tau_1 = \tau_2 \to t\})$
in $\langle S_3 S_2 A_1 + S_3 A_2, S_3 S_2 S_1, S_3 t \rangle$

let $x = M_1$ **in** M_2
$\Rightarrow$ let $\langle A_1, S_1, \tau_1 \rangle = W'(B, M_1)$
$\langle A_2, S_2, \tau_2 \rangle = W'(S_1 B \cup \{x : \mathrm{Gen}(A_1, S_1 B, \tau_1)\}, M_2)$
in $\langle S_2 A_1 + A_2, S_2 S_1, \tau_2 \rangle$

Figure 2: Extending ML's type inference algorithm for recursive definitions