What are principal typings and what are they good for?

Technical Memorandum MIT/LCS/TM–532

Trevor Jim*

August 1995; revised November 1995

Abstract

We demonstrate the pragmatic value of the *principal typing property*, a property distinct from 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 new rule for typing recursive definitions that can type some interesting examples of polymorphic recursion.

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.

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

*545 Technology Square, Cambridge, MA 02139, trevor@theory.lcs.mit.edu. Supported by NSF grants CCR–9113196 and CCR–9417382, and ONR Contract N00014–92–J–1310.

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. This was already noted by Damas in his dissertation [6], but there have been subsequent claims that ML has the principal typing property, indicating that the distinction between principal types and principal typings is not widely appreciated. We examine ML's lack of principal typings more closely 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—we will show that they type exactly the same terms—but it possesses the additional property of principal typings. We use a variant of the intersection 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. Two rules that have been used to type recursive definitions in ML are given below.

$$(\textsc{rec-simple}) \quad \frac{A \cup \{x : \tau\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad \text{(where } \tau \text{ is a simple type)}$$

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

The rule ($\textsc{rec-simple}$) requires the body M of the recursive definition ($\mu x M$) to be typed under the assumption that x has a simple type. This restriction is relaxed in (REC-POLY), the rule of *polymorphic recursion* [26, 16], which permits M to be typed under the assumption that x has a polymorphic type.

More terms are typable under (REC-POLY) than (REC-SIMPLE), and practical examples of programs requiring polymorphic recursion are a recurring topic on the ML mailing list. But (REC-SIMPLE) is used in practice, because type inference for (REC-POLY) is undecidable [17, 9]. To understand why, 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. In the case of (REC-POLY), this "chicken and egg" problem cannot be solved.

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

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

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.

When we use this strategy to type recursive definitions in $\mathbf{P}_2$, we obtain an interesting typing rule, lying between (REC-SIMPLE) and (REC-POLY): it is able to type some, but not all, examples of polymorphic recursion.

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* [27], 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 [30] for a discussion. As a final example of the utility of principal typings, we show 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 show its connection with rank 2 of System F. 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. We describe alternatives to principal typings in §7. In §8, we describe an extension of $\mathbf{P}_2$ with principal typings. We discuss related work in §9, and we summarize our results in §10. Proofs of all theorems can be found in a separate paper [11].

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, $(\mathbf{let}\ x = M\ \mathbf{in}\ N)$ 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

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

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

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

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:

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

This says that as long as the argument of the function $(\lambda x.xx)$ has *both* the types s and $s \rightarrow t$, 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:

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

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:

$$(\mathbf{let}\ x = (\lambda y.y)\ \mathbf{in}\ xx) \quad : \quad (\forall u.u \rightarrow u).$$

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.

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

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 rank here refers to the depth of intersections below arrows, not the depth of nesting of arrows, and 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.[1] 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 $(\tau_1 \rightarrow \tau_2) \leq_2 (\sigma_1 \rightarrow \sigma_2)$.
- 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 $\sigma \leq_{\forall 2} (\forall t \tau)$.

[1] 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 $\leq_{\forall 2}$: $(\forall t.t) \not\leq_{\forall 2} (s \wedge (s \rightarrow u)) \rightarrow u$.

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 typing judgments are 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 in Figure 1.

Example 2 Recall that the typings

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

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

$$(\lambda x.xx) \quad : \quad ((s \rightarrow s) \wedge ((s \rightarrow s) \rightarrow (s \rightarrow s))) \rightarrow (s \rightarrow s).$$

And $(\forall u.u \rightarrow u) \leq_{\forall 2} (s \rightarrow s)$ and $(\forall u.u \rightarrow u) \leq_{\forall 2} ((s \rightarrow s) \rightarrow (s \rightarrow s))$, so by rules (SUB) and (APP),

$$(\lambda x.xx)(\lambda y.y) \quad : \quad (s \rightarrow s).$$

Finally, by rule (GEN),

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

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

$$
\begin{array}{ll}
(\textsc{var}) & \{x : (\bigwedge_{i \in I} \tau_i)\} \vdash x : \tau_{i_0} \quad (\text{where } i_0 \in I) \\[2ex]
(\textsc{abs}) & \dfrac{A \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \rightarrow \tau} \\[2ex]
(\textsc{app}) & \dfrac{A \vdash M : (\bigwedge_{i \in I} \tau_i) \rightarrow \sigma, \quad (\forall i \in I)\ A \vdash N : \tau_i}{A \vdash (MN) : \sigma} \\[2ex]
(\textsc{gen}) & \dfrac{A \vdash M : \sigma}{A \vdash M : (\forall t \sigma)} \quad t \notin \mathrm{FTV}(A) \\[2ex]
(\textsc{sub}) & \dfrac{A \vdash M : \tau}{A \vdash M : \sigma} \quad \tau \leq_{\forall 2} \sigma \\[2ex]
(\textsc{add-hyp}) & \dfrac{A \vdash M : \sigma}{A \cup \{x : \tau\} \vdash M : \sigma}
\end{array}
$$

Figure 1: Typing rules of $\mathbf{P}_2$. Types in type environments are in $\mathbf{T}_1$, and derived types are in $\mathbf{T}_{\forall 2}$.

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, cf. [25]. 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.

2.1 Comparison with Rank 2 of System F

The system $\mathbf{P}_2$ is closely connected to Λ_2, the restriction of System F to rank 2 types. Our presentation of Λ_2 is based on that of Kfoury and Tiuryn [15].

The *types of System F* are defined by the following grammar:

$$\tau \quad ::= \quad t \mid (\tau_1 \rightarrow \tau_2) \mid (\forall t \tau).$$

We consider System F types to be syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers.

$$
\begin{array}{ll}
(\textsc{var}) & A_x \cup \{x : \sigma\} \vdash x : \sigma \\[2ex]
(\textsc{abs}) & \dfrac{A_x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \rightarrow \tau} \\[3ex]
(\textsc{app}) & \dfrac{A \vdash M : \sigma \rightarrow \tau, \quad A \vdash N : \sigma}{A \vdash (MN) : \tau} \\[3ex]
(\textsc{inst}) & \dfrac{A \vdash M : \forall t \sigma}{A \vdash M : \{t := \tau\}\sigma} \\[3ex]
(\textsc{gen}) & \dfrac{A \vdash M : \sigma}{A \vdash M : \forall t \sigma} \quad t \notin \mathrm{FTV}(A)
\end{array}
$$

Figure 2: Typing rules of Λ_2. Types in type environments are in $\mathbf{R}(1)$, and derived types are in $\mathbf{R}(2)$.

The types of System F can be organized into a hierarchy as follows. First, define $\mathbf{R}(0) = \mathbf{T}_0$. Then for $n \geq 0$, the set $\mathbf{R}(n+1)$ is defined to be the least set satisfying

$$
\begin{aligned}
\mathbf{R}(n+1) \quad = \quad & \mathbf{R}(n) \cup \{(\sigma \rightarrow \tau) \mid \sigma \in \mathbf{R}(n), \tau \in \mathbf{R}(n+1)\} \\
& \cup \{(\forall t \sigma) \mid \sigma \in \mathbf{R}(n+1)\}.
\end{aligned}
$$

The typing judgments are of the form $A \vdash M : \sigma$, where σ is an $\mathbf{R}(2)$ type, and all of the types in A are $\mathbf{R}(1)$ types. The typing rules are given in Figure 2.

Theorem 4 *A term M is typable in $\mathbf{P}_2$ iff M is typable in Λ_2 iff M is typable in the rank 2 intersection type system.*

Thus the $\mathbf{P}_2$ programs are exactly the Λ_2 programs. As we will see, however, $\mathbf{P}_2$ has the principal typing property, while no notion of principal typings is known for Λ_2 [19].

Corollary 5 *Typability in $\mathbf{P}_2$ is DEXPTIME-complete.*

The proof of Theorem 4 relies on the principal type property of ML and is given in a separate paper [11]; a similar theorem has been shown independently by Yokouchi [32]. Corollary 5 follows by the results of Kfoury and Tiuryn [15] (Λ_2 typability is polynomial time equivalent to ML typability), and Kfoury et al. [18] and Mairson [22] (ML typability is DEXPTIME-complete).

2.2 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 6

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.

2.3 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.

Definition 7 For any term M, the set $\mathrm{PP}(M)$ is defined by the following cases.

- If $M = x$, then $\langle\{x : t\}, t\rangle \in \mathrm{PP}(x)$ for any type variable t.
- 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:
  - If $x \notin \mathbf{dom}(A)$, and t is a fresh type variable, then $\langle A, \forall t\vec{s}(t \rightarrow \sigma)\rangle \in \mathrm{PP}(\lambda x N)$.
  - If $x \in \mathbf{dom}(A)$, then $\langle A_x, \mathrm{Gen}(A_x, A(x) \rightarrow \sigma)\rangle \in \mathrm{PP}(\lambda x N)$.
- If $M = M_1 M_2$, and $\langle A_1, \forall\vec{s}\sigma_1\rangle \in \mathrm{PP}(M_1)$, then:
  - If $\sigma_1 = t$ (a type variable), t_1 and t_2 are fresh type variables, the type variables of $\langle A_2, \sigma_2\rangle \in \mathrm{PP}(M_2)$ are fresh, $U \in \mathbf{MGS}(\{\sigma_2 \leq t_1, t = t_1 \rightarrow t_2\})$, and $A = U(A_1 + A_2)$, then

$$\langle A, \mathrm{Gen}(A, Ut_2)\rangle \in \mathrm{PP}(M).$$

– If $\sigma_1 = (\bigwedge_{i \in I} \tau_i) \to \tau$, $(\forall i \in I)$ the type variables of $\langle A_i, \sigma_i \rangle \in \mathrm{PP}(M_2)$ are fresh, $U \in \mathbf{MGS}(\{\sigma_i \leq \tau_i \mid i \in I\})$, and $A = U(A_1 + \sum_{i \in I} A_i)$, then

$$\langle A, \mathrm{Gen}(A, U\tau) \rangle \in \mathrm{PP}(M).$$

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.

Example 8 We show how the algorithm finds a principal typing for $(\lambda x.xx)$.

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

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

such as $\{t_2 := t_3, t_1 := t_3 \to t_4\}$. Then

$$\langle \{x : t_3 \wedge (t_3 \to t_4)\}, t_4 \rangle \in \mathrm{PP}(xx).$$

iv) Finally, $\mathrm{PP}(\lambda x.xx)$ produces

$$\langle \emptyset, \forall t_3, t_4.(t_3 \wedge (t_3 \to t_4)) \to t_4 \rangle.$$

Theorem 9 (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 $(\mu x M)$ represents the program x such that $x = M$, where M may contain occurrences of x.

As we remarked in the introduction, the principal typing property suggests that we type recursive definitions by a rule of the following form.

$$(\textsc{rec}) \quad \frac{A \cup \{x : \tau\} \vdash M : \sigma}{A \vdash (\mu x M) : \sigma} \quad \sigma \leq_{\forall 2,1} \tau$$

The rule (REC) can type strictly more terms than the rule (REC-SIMPLE) of ML. For example, the following term is typable in $\mathbf{P}_2 + (\textsc{rec})$, but not in $\mathbf{P}_2 + (\textsc{rec-simple})$:

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

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

However, (REC) cannot type as many terms as (REC-POLY). For example, the term $(\mu x.xx)$ has type $(\forall t.t)$ in ML + (REC-POLY), but it is not typable with our rules.

It is interesting to compare (REC) with a rule, (FIX'), that Mycroft [26] suggested in the context of ML:

$$(\text{FIX'}) \quad \frac{A \vdash \lambda x_1 \cdots x_n.M' : \tau_1 \to \cdots \to \tau_n \to \tau}{A \vdash (\mu x M) : \tau}$$

Here M is a term with n occurrences of x, M' is M with each occurrence of x renamed to a fresh variable x_i, $\tau_1, \ldots, \tau_n, \tau$ are simple types, and $\mathrm{Gen}(A, \tau) \leq \tau_i$ for all $i \leq n$.

The idea behind Mycroft's rule is that each of the finite occurrences of x in M may have a different simple type (so long as M can be shown to satisfy those types). The same idea explains the typing power of (REC). Note, however, that *this idea was not the motivation for* (REC). Instead, (REC) arose as an instance of a general rule motivated by the principal typing property. Other interesting typing rules may arise as instances of the general rule, in type systems other than $\mathbf{P}_2$.

Mycroft's rule is actually more powerful than (REC). The side condition, $\mathrm{Gen}(A, \tau) \leq \tau_i$, permits τ to be generalized by any type variable not appearing in A, *including type variables appearing in the* τ_i. This is not allowed by (REC). The term $(\mu x.xx)$ is one place where this makes a difference: it is typable with (FIX') but not (REC). For a more practical example, consider the following ML code. It comes from the ML mailing list, and has arisen in practice.

```
datatype 'a T = EMPTY
              | NODE of 'a * ('a T) T

fun collect EMPTY = nil
  | collect (NODE(n,t)) =
      n :: flatmap collect (collect t)
```

Here `'a T` is a polymorphic tree type, and `flatmap` is the mapping function of type `('a -> 'b list) -> 'a list -> 'b list`. The function `collect`, which collects all the labels of an `'a T` and returns them in an `'a list`, is typable with (REC-POLY) and with (FIX'), but not with (REC). Of course, we could generalize our rule along the lines of (FIX'):

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

The system would retain principal typings and decidable type inference, but for simplicity, we stay with (REC).

$$\text{(LETREC-SIMPLE)}\quad \frac{(\forall j \in I) \quad A \cup \{x_i : \tau_i \mid i \in I\} \vdash M_j : \tau_j \quad (\tau_j \in \mathbf{T}_0) \qquad A \cup \{x_i : \mathrm{Gen}(A, \tau_i) \mid i \in I\} \vdash M : \sigma}{A \vdash (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ M) : \sigma}$$

$$\text{(LETREC-VAR)}\quad \frac{\forall j \in I \quad A \cup \{x_i : \tau_i \mid i \in I\} \vdash M_j : \sigma_j \quad (\sigma_j \leq_{\forall 2,1} \tau_j)}{A \vdash (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_{i_0}) : \sigma_{i_0}} \quad i_0 \in I$$

$$\text{(LETREC)}\quad \frac{A \cup \{x_i : \tau_i \mid i \in I\} \vdash N : \sigma \qquad \forall j \in I \quad A \vdash_\wedge (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_j) : \tau_j}{A \vdash (\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ N) : \sigma} \quad N \notin \{x_i \mid i \in I\}$$

Figure 3: Rules for typing mutually recursive definitions. The rule (LETREC-SIMPLE) is used by ML, while $\mathbf{P}_2$ uses (LETREC-VAR) and (LETREC).

3.1 Mutual recursion

In order to support the applications of principal typings in the next section, we add mutually recursive definitions to the language. Such definitions are written

$$(\mathbf{letrec}\ x_1 = M_1, \ldots, x_n = M_n\ \mathbf{in}\ N)$$

or

$$(\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ N),$$

where all of the x_i are distinct.

The typing rules for **letrec** are given in Figure 3. ML uses the rule (LETREC-SIMPLE) to type mutual recursion. In (LETREC-SIMPLE), the recursive definitions must be typed under the assumption that the recursive variables have simple type. In typing the body of the **letrec**, however, the types of the recursive variables can be generalized, so that they can be used polymorphically.

We cannot use (LETREC-SIMPLE) with $\mathbf{P}_2$, because $\mathbf{P}_2$ does not permit quantified types to appear in type environments. And it is not easily adapted to $\mathbf{P}_2$. In ML, the polymorphic type $\mathrm{Gen}(A, \tau_i)$ of x_i is easily obtained from the simple type τ_i used in typing the recursive definitions. The equivalent of $\mathrm{Gen}(A, \tau_i)$ in $\mathbf{P}_2$ is some intersection $(\bigwedge_{j \in J} \tau_j)$, where each τ_j is an instance of $\mathrm{Gen}(A, \tau_i)$. It is not immediately clear how to get directly from τ_i to $(\bigwedge_{j \in J} \tau_j)$.

Instead, our rules for $\mathbf{P}_2$ are based on the following observation: the typings of any term $(\mathbf{letrec}\ B\ \mathbf{in}\ N)$ can be expressed in terms of the typings for terms $(\mathbf{letrec}\ B\ \mathbf{in}\ x)$, where x is a variable defined by B. Formally, for any $B = \{x_1 = M_1, x_2 = M_2, \ldots, x_n = M_n\}$ and $M = (\mathbf{letrec}\ B\ \mathbf{in}\ N)$, we define $\langle\!\langle M \rangle\!\rangle$ to be the term

$$\begin{array}{rcl} \langle\!\langle M \rangle\!\rangle & = & (\mathbf{let}\ x_1 = (\mathbf{letrec}\ B\ \mathbf{in}\ x_1) \\ & & \qquad \vdots \\ & & \qquad x_n = (\mathbf{letrec}\ B\ \mathbf{in}\ x_n) \\ & & \ \ \mathbf{in}\ \ N). \end{array}$$

The following lemma is easily proved.

Lemma 10 *In* ML + (LETREC-SIMPLE)*,* $A \vdash M : \sigma$ *iff* $A \vdash \langle\!\langle M \rangle\!\rangle : \sigma$*.*

This is the intuition behind the rules (LETREC-VAR) and (LETREC) of Figure 3. (LETREC-VAR) is a straightforward generalization of the rule (REC) for terms of the form (**letrec** B **in** x), where x is a variable defined in B. Lemma 10 suggests that we type other **letrec** expressions by a rule of the form

$$\frac{A \vdash \langle\!\langle \textbf{letrec } B \textbf{ in } N \rangle\!\rangle : \sigma}{A \vdash (\textbf{letrec } B \textbf{ in } N) : \sigma} \quad (N \text{ is not defined by } B)$$

Our rule (LETREC) is obtained simply by desugaring the let-expression formed by $\langle\!\langle \cdot \rangle\!\rangle$ into abstractions and applications, and considering how the resulting term would be typed by (ABS) and (APP). We make the rule more compact by using the notation $A \vdash_\wedge M : (\bigwedge_{i \in I} \tau_i)$ to abbreviate $(\forall i \in I)\ A \vdash M : \tau_i$.

We write Λ_2^{R} for the system Λ_2 + (REC-SIMPLE) + (LETREC-SIMPLE), and $\mathbf{P}_2^{\mathrm{R}}$ for the system $\mathbf{P}_2$ + (REC) + (LETREC-VAR) + (LETREC).

Theorem 11 *If* M *is typable in* Λ_2^{R}*, then* M *is typable in* $\mathbf{P}_2^{\mathrm{R}}$*.*

Definition 12 The type inference algorithm of Definition 7 can be extended to $\mathbf{P}_2^{\mathrm{R}}$ by adding the following cases.

- If $M = (\mu x N)$ and $\langle A, \sigma \rangle \in \mathrm{PP}(N)$, then:
  - If $x \notin \mathbf{dom}(A)$, and $U \in \mathbf{MGS}(\{\sigma \leq t\})$ where t is a fresh type variable,
    then $\langle UA, \mathrm{Gen}(UA, U\sigma) \rangle \in \mathrm{PP}(M)$.
  - If $x \in \mathbf{dom}(A)$ and $U \in \mathbf{MGS}(\{\sigma \leq A(x)\})$,
    then $\langle UA_x, \mathrm{Gen}(UA_x, U\sigma) \rangle \in \mathrm{PP}(M)$.
- If $M = (\textbf{letrec } \{x_i = M_i \mid i \in I\} \textbf{ in } x_{i_0})$, where $i_0 \in I$,

  and $\langle A_i, \sigma_i \rangle \in \mathrm{PP}(M_i)$ for $i \in I$,
  $A' = \sum_{i \in I} A_i$,
  $A'' = A' \cup \{x_i : t_i \mid x_i \notin \mathbf{dom}(A'), t_i \text{ fresh}\}$,
  $U \in \mathbf{MGS}(\{\sigma_i \leq A''(x_i) \mid i \in I\})$,
  and $A = UA''_{\{x_i \mid i \in I\}}$,

  then $\langle A, \mathrm{Gen}(A, U\sigma_{i_0}) \rangle \in \mathrm{PP}(M)$.
- If $M = (\textbf{letrec } \{x_i = M_i \mid i \in I\} \textbf{ in } N)$, where $N \notin \{x_i \mid i \in I\}$, and $\langle A, \sigma \rangle \in \mathrm{PP}(\langle\!\langle M \rangle\!\rangle)$,

  then $\langle A, \sigma \rangle \in \mathrm{PP}(M)$.

Theorem 13 (Principal typings) *If* M *is typable in* $\mathbf{P}_2^{\mathrm{R}}$*, 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 *in* $\mathbf{P}_2^{\mathrm{R}}$*.*

An important limitation of our rules for mutual recursion is illustrated by the following well-known example of Mycroft [26]:

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

This program is not typable under our rules (or ML's rules) when presented as a single, mutually recursive definition. The function *map* is used polymorphically by the other functions, and our rules do not allow sufficient polymorphism for the program to type. Note that *map* does not depend on the other functions; if *map* is placed in a separate recursive definition, the program can be typed by our rules.

Thus to type an unordered set of definitions, it is necessary to examine the call graph of the program to determine an order in which to type the definitions. This complication must be addressed by the applications of the next section.[2]

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.

Incremental type inference is thus the type checking task of separate compilation on an extremely fine scale: not just every module, but every definition is typed and compiled separately.

[2] A generalization of our rules along the lines of Mycroft's (FIX') could handle the *map* example, but not all such examples.

Consider a partial program $x_1 = M_1, \ldots, x_n = M_n$, where duplicate definitions have been discarded. To check that the program is well-typed, it is sufficient to perform type inference on the expression

$$(\mathbf{letrec}\ B_1\ \mathbf{in}\ \cdots(\mathbf{letrec}\ B_m\ \mathbf{in}\ 0)\cdots)$$

derived from the call graph of the program: each B_i is a strongly connected component (SCC) of mutually recursive bindings, and the B_i are topologically sorted.

This can be accomplished by any type inference algorithm that works on terms with free variables. But this is not enough to solve the incremental problem efficiently: when the user enters the next definition, $x_{n+1} = M_{n+1}$, we must do better than just running the type inference algorithm on the new expression

$$(\mathbf{letrec}\ B_1'\ \mathbf{in}\ \cdots(\mathbf{letrec}\ B_{m'}'\ \mathbf{in}\ 0)\cdots).$$

A close inspection of the $\mathbf{P}_2^{\mathrm{R}}$ type inference algorithm will show that principal typings are the key to efficient incremental type inference.

If $P = (\mathbf{letrec}\ B_1\ \mathbf{in}\ \cdots(\mathbf{letrec}\ B_m\ \mathbf{in}\ 0)\cdots)$ is our partial program, and the variables defined by each B_i are denoted $x_{i,1}, \ldots, x_{i,n_i}$, then by the $\mathbf{P}_2^{\mathrm{R}}$ equivalent of Lemma 10, type inference for P is equivalent to type inference for the expression

$$\begin{array}{rcl} \langle\!\langle P \rangle\!\rangle & = & (\mathbf{let}\ x_{1,1} = (\mathbf{letrec}\ B_1\ \mathbf{in}\ x_{1,1}) \\ & & \qquad \vdots \\ & & \qquad x_{1,n_1} = (\mathbf{letrec}\ B_1\ \mathbf{in}\ x_{1,n_1}) \\ & & \qquad \vdots \\ & & \qquad x_{m,1} = (\mathbf{letrec}\ B_1\ \mathbf{in}\ x_{m,1}) \\ & & \qquad \vdots \\ & & \qquad x_{m,n_m} = (\mathbf{letrec}\ B_m\ \mathbf{in}\ x_{m,n_m}) \\ & & \ \mathbf{in}\ \ 0), \end{array}$$

where **let**'s are desugared into applications of abstractions.

We now show that type inference for such an expression is equivalent to solving a subtype satisfaction problem constructed from the principal pair of each $(\mathbf{letrec}\ B_i\ \mathbf{in}\ x_{i,j})$.

Definition 14 For any term M, we define the set $L^*(M)$ inductively as follows.

- If $M = (\lambda x M_1) M_2$,

  and $\langle A_1, \sigma, \pi \rangle \in L^*(M_1)$

  $\quad\langle A_2, \sigma_2 \rangle \in \mathrm{PP}(M_2)$

  $\quad(\bigwedge_{i \in I} \tau_i) = \begin{cases} t & \text{if } x \notin \mathbf{dom}(A_1) \text{ and } t \text{ is fresh,} \\ A_1(x) & \text{otherwise.} \end{cases}$

  $\quad(\forall i \in I)\ S_i$ renames $\mathrm{FTV}(A_2, \sigma_2)$ to fresh type variables

  then $\langle A_1 + (\sum_{i \in I} S_i A_2), \sigma, \pi \cup \{S_i \sigma_2 \leq \tau_i \mid i \in I\} \rangle \in L^*(M)$.

- Otherwise, $\langle A, \sigma, \emptyset\rangle \in L^*(M)$ iff $\langle A, \sigma\rangle \in \mathrm{PP}(M)$.

Lemma 15 *$\langle A, \sigma\rangle \in \mathrm{PP}(M)$ iff for some A', σ', π and U, $\langle A', \sigma', \pi\rangle \in L^*(M)$, $U \in \mathbf{MGS}(\pi)$, and $\langle A, \sigma\rangle = \langle UA', \mathrm{Gen}(UA', U\sigma')\rangle$.*

Therefore, we can perform type inference for P by calculating $\langle A, \sigma, \pi\rangle \in L^*(\langle\!\langle P\rangle\!\rangle)$, and finding a solution to π. And $L^*(\langle\!\langle P\rangle\!\rangle)$ is calculated from the principal pair of each ($\mathbf{letrec}\ B_i\ \mathbf{in}\ x_{i,j}$).

Now consider the incremental case. When the user enters the next definition, $x_{n+1} = M_{n+1}$, we must perform type inference on a new partial program, P'. Just as before, this means calculating a new π' from $L^*(\langle\!\langle P'\rangle\!\rangle)$. And again, this requires the principal pair of each ($\mathbf{letrec}\ B'_{i'}\ \mathbf{in}\ x_{i',j'}$).

We now argue that $L^*(\langle\!\langle P'\rangle\!\rangle)$ can be constructed incrementally. First note that the new definition may not change the SCC's of the existing call graph: the SCCs change only when the new definition is mutually recursive with a previous definition. So most often, ($\mathbf{letrec}\ B'_{i'}\ \mathbf{in}\ x_{i',j'}$) will equal some previous ($\mathbf{letrec}\ B_i\ \mathbf{in}\ x_{i,j}$); and then, *by the principal typing property, the principal pair of* ($\mathbf{letrec}\ B_i\ \mathbf{in}\ x_{i,j}$) *is unchanged.*

We even benefit if the SCC's change. If $B = \{x_i = M_i \mid i \in I\}$ is a new SCC, we must calculate the principal pair of ($\mathbf{letrec}\ \{x_i = M_i \mid i \in I\}\ \mathbf{in}\ x_j$) for all $j \in I$. This involves computing $\mathrm{PP}(M_i)$ for each M_i; but if $i \neq n+1$, then by the principal typing property, $\mathrm{PP}(M_i)$ is unchanged.

Thus 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. This is not the case in the system of Aditya and Nikhil, where a new definition may cause the entire program to be reprocessed (see [8], p. 104).

We must also calculate a solution to the new satisfaction problem. However, the new problem may be almost identical to the previous problem. In particular, if the new definition does not change the SCCs of the call graph, the new satisfaction problem will be a superset of the old problem. 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.

Finally, we remark that the SCCs and topological sort may be computed incrementally by off-the-shelf algorithms [10, 23].

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 [27]. 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:

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

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 [27]). 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

$$\begin{aligned}&A \vdash (\lambda x M) : (\textstyle\bigwedge_{i \in I} \sigma_i) \to \tau,\\ &A' \vdash N : \sigma'.\end{aligned}$$

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

$$\pi \quad = \quad \{S_i\sigma' \leq \sigma_i \mid i \in I\},$$

where each S_i renames $\mathrm{FTV}(A', \sigma')$ to fresh type variables. At this point we will discover all of the type errors related to x: for some i, the type $S_i\sigma'$ will not be able to satisfy the constraints expressed by σ_i. 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,

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

[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.

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:

$$\frac{A \cup \{x : \tau_1\} \vdash M : \tau_2}{A \vdash (\lambda x M) : \tau_1 \to \tau_2} \quad \tau_1, \tau_2 \in \mathbf{T}_0$$

In ML, we cannot abstract over variables of polymorphic type; the only way of introducing polymorphic variables is through let-expressions.

7 Living without principal typings

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. For example, the "representation" may be a typing *in another type system.*

This idea was the basis of the smartest recompilation system of Shao and Appel [27]. They defined a type system with the following property: for any ML typable term M, there is a judgement in the Shao-Appel system that encodes all of the ML typings for M, in an appropriate sense. They did not prove a principal typing property for their system, but it is essentially identical to a system of Damas [6]. Damas proved a principal typing theorem for his system, and showed that it types exactly the same terms as ML.

The systems $\mathbf{P}_2$ and Λ_2 are a second example of this phenomenon. We have shown that $\mathbf{P}_2$ has principal typings and types exactly the same terms as Λ_2. However, Λ_2 does not have principal typings in the sense of Definition 3. The counterexample xx that we used for ML also works for Λ_2. Unlike ML, Λ_2 has a "true" abstraction rule; this is not a contradiction, because in addition to lacking principal typings, Λ_2 lacks principal types [19].

And for a third example, Palsberg and Scott[4] have shown that the recursive type system of Amadio and Cardelli [3] types exactly the same terms as a type system based on constraints [7]. Palsberg has shown that the Amadio-Cardelli system does not have principal typings, and Jim has shown that the constraint-based system does have principal typings.

8 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.

[4] Personal communication, September 1995.

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 unlimited depth.

9 Related work

Principal typings are not a new concept. A number of existing type systems have principal typings, including the simply typed lambda calculus [31], the system of recursive types [5], the system of simple subtypes [25], and the system of intersection types [4]. 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, despite the early remarks of Damas [6] to the contrary.

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 [21], but he did not give a formal definition of the type inference algorithm or proof of correctness. In an oft-referenced 1984 paper [24], 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 [19] 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 [29].

Our addition of top-level quantification is a useful technical improvement to the rank 2 intersection system. 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 [19]. Launchbury and Peyton Jones [20] 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 [12], Kaes [14], and Smith [28] 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 **P**, we will show how to solve some subtype satisfaction problems for types with quantifiers and intersections at unlimited depth, giving type inference for a system with a much richer class of types.

10 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 new 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 some interesting 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 a term. In particular, a term's principal typing in one type system may serve as a representative of all of its typings in another type system. This technique serves for Λ_2, whose typings can be represented by principal typings in $\mathbf{P}_2$.

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 unlimited depth, as we will show in a forthcoming paper.

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

referees. We thank Fritz Henglein for pointing out Mycroft's rule (FIX') and the work of Damas.

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] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[4] 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.

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

[6] Luis Manuel Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[7] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and it application to OOP. In *Proc. Mathematical Foundations of Programming Semantics*, 1995. To appear.

[8] Shail Aditya Gupta. An incremental type inference system for the programming language Id. Master's thesis, Massachusetts Institute of Technology, November 1990. Available as MIT/LCS Technical Report TR–488.

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

[10] Monika Rauch Henzinger and Valerie King. Fully dynamic biconnectivity and transitive closure. To appear in Proc. 36^{th} Annual Symp. on Foundations of Computer Science, 1995.

[11] Trevor Jim. Rank 2 type systems and recursive definitions. Technical Memorandum MIT/LCS/TM–531, M.I.T. Lab. for Computer Science, August 1995.

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

[13] 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.

[14] 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.

[15] A.J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257, June 1992.

[16] 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.

[17] 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.

[18] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41(2), March 1994.

[19] 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.

[20] 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.

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

[22] Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 382–401, 1990.

[23] Marchetti-Spaccamela, Nanni, and Rohnert. On-line graph algorithms for incremental compilation. In *Graph-Theoretic Concepts in Computer Science, International Workshop WG*, 1993.

[24] 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.

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

[26] 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.

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

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

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

[30] 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.

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

[32] 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 '$\rightarrow$', e.g., $\sigma \wedge \tau \rightarrow t$ means $(\sigma \wedge \tau) \rightarrow t$, 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. 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 unification 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 in Figure 4. 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.

$$
\begin{array}{lll}
(\sigma_1 \rightarrow \sigma_2) \leq t & \Rightarrow & \exists t_1, t_2 . \{t_1 \leq \sigma_1, \sigma_2 \leq t_2, t = t_1 \rightarrow t_2\} \\
 & & \text{if } t_1, t_2 \text{ are fresh} \\[1em]
(\sigma_1 \rightarrow \sigma_2) \leq (\tau_1 \rightarrow \tau_2) & & \\
 & \Rightarrow & \{\tau_1 \leq \sigma_1, \sigma_2 \leq \tau_2\} \\[1em]
\sigma \leq (\tau_1 \wedge \tau_2) & \Rightarrow & \{\sigma \leq \tau_1, \sigma \leq \tau_2\} \\[1em]
t \leq \tau & \Rightarrow & \{t = \tau\} \\
 & & \text{if } \tau \text{ is a simple type} \\[1em]
(\forall t \sigma) \leq \tau & \Rightarrow & \exists t \{\sigma \leq \tau\} \\
 & & \text{if } \tau \text{ is not a } \wedge\text{-type, and } t \text{ is not} \\
 & & \text{free in } \tau
\end{array}
$$

Figure 4: Transformational rules for $\leq_{\forall 2,1}$-satisfaction problems

A unification algorithm in a transformational style,[5] taken from [13], is given in Figure 5. 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.

[5] 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.

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

Figure 5: A set of transformational rules for solving unification of simple types.