Rank 2 type systems and recursive definitions

Technical Memorandum MIT/LCS/TM-531

Trevor Jim
Laboratory for Computer Science
Massachusetts Institute of Technology
August 1995

Abstract

We demonstrate an equivalence between the rank 2 fragments of the polymorphic lambda calculus (System F) and the intersection type discipline: exactly the same terms are typable in each system. An immediate consequence is that typability in the rank 2 intersection system is DEXPTIME-complete. We introduce a rank 2 system combining intersections and polymorphism, and prove that it types exactly the same terms as the other rank 2 systems. The combined system suggests a new rule for typing recursive definitions. The result is a rank 2 type system with decidable type inference that can type many examples of polymorphic recursion. Finally, we discuss some applications of the type system in data representation optimizations such as unboxing and overloading.

Keywords: Rank 2 types, intersection types, polymorphic recursion, boxing/unboxing, overloading.

1 Introduction

In the past decade, Milner's type inference algorithm for ML has become phenomenally successful. As the basis of popular programming languages like Standard ML and Haskell, Milner's algorithm is the preferred method of type inference among language implementors. And in the theoretical
community, the literature on type inference is dominated by extensions of
ML’s let-polymorphism.

In this paper we examine some alternatives to ML that have attracted
surprisingly little attention: the systems of rank 2 types introduced by
Leivant [21]. These systems are slightly more powerful than ML—strictly
more terms can be assigned types—and the increased power comes for free—
the complexity of typability is identical. But the unique feature of the rank 2
systems that justifies further study is that, in sharp contrast to other exten-
sions of ML, they abandon let-polymorphism.

We use the expression \( \lambda x.xxx \) to illustrate the limitations of let-pol-
yporphism. It is well known that this expression cannot be typed in ML:
the only way for ML to type the self-application \( xx \) is by assigning a poly-
morphic type to \( x \), and ML does not allow abstraction over variables with
polymorphic type. In ML, the only mechanism for introducing variables of
polymorphic type is the let-expression:

\[
\text{let } x = (\lambda y.y) \\
\text{in } xx
\]

This let-expression binds \( x \) to the identity function \((\lambda y.y)\), which has the
polymorphic type \( \forall t. t \rightarrow t \) in ML. By ML’s let-polymorphism, \( x \) is assigned
the type \( \forall t. t \rightarrow t \), which is sufficient to type \( xx \).

The problem with this is that we cannot typecheck the uses of \( x \) (the
application \( xx \)) separately from its definition (the function \((\lambda y.y)\)). So ML
must be extended with a module language in order to support programming
in the large, where it is impractical to require every polymorphic definition
to appear in the same source file as every use.

In contrast, \((\lambda x.xxx)\) is typable in all of the rank 2 systems we consider.
Here are two rank 2 typings:

\[
(\lambda x.xxx) : (\forall t. t \rightarrow t) \land (t \rightarrow t) \rightarrow (t \rightarrow t),
\]

The first typing says that \((\lambda x.xxx)\) is a function that, when given an argument
with type \( t \rightarrow t \) for any type \( t \), produces a result with type \( s \rightarrow s \), for any \( s \).
The identity function is an appropriate argument.

The second typing says that \((\lambda x.xxx)\) is a function that, when given an
argument having both the types \((t \rightarrow t)\) and \((t \rightarrow t) \rightarrow (t \rightarrow t)\), produces
a result of type \((t \rightarrow t)\). Once again, the identity \((\lambda y.y)\) is an appropriate
argument.
The rank 2 systems we consider are subsystems of two widely studied type systems, System F and the system of intersection types. System F, introduced independently by Girard [7] and by Reynolds [28], predates ML and can type many more terms. A recent result of Wells [34], however, shows that typability in the system is undecidable, putting type inference out of reach.

The system of intersection types, introduced independently by Coppo and Dezani [5] and by Sallé [29], can type even more terms than System F: it types all (and only) the strongly normalizing terms. The equivalence of typability and strong normalization implies that type inference, just as with System F, is unattainable.

With the goal of type inference in mind, we seek decidable restrictions of these type systems. Restrictions based on the rank of types were suggested by Leivant [21]. The rank of a type can be easily determined by examining it in tree form. A type is of rank \( k \) if no path from the root of the type to a type constructor of interest (either type intersection `\&` or type quantification `\forall`) passes to the left of \( k \) arrows. The types shown in Figure 1 are rank 2 types, because no path from root to \& or \forall passes to the left of two arrows. But the types shown in Figure 2 go beyond rank 2 (they are rank 3 types). The types given above for \((λx.xx)\) are rank 2 types.

\[
\begin{align*}
((t \& (t \to s)) \to t) \& s & \quad s \to (\forall t, t \to t) \to s \\
\end{align*}
\]

Figure 1: Examples of rank 2 types

Ranks 0 and 1 of Leivant's systems are equivalent to the simply typed

\(^{1}\text{Without the type constant } \omega.\)
lambda calculus, which can type fewer terms than ML. But starting with rank 2, the systems can type more terms than ML.

Rank 2 of System F, which we call \( \Lambda_2 \), has received the most study. McCracken [23] proposed a type inference algorithm for \( \Lambda_2 \) based on Leivant’s ideas. This algorithm is incorrect. Kfoury and Tiuryn [12] show that the complexity of typability in \( \Lambda_2 \) is identical to that of ML. Kfoury and Wells [16, 17] give a correct type inference algorithm, and show that ranks 3 and higher in System F are undecidable.

Leivant’s original paper is almost the only work on rank 2 of the intersection type discipline, which we call \( I_2 \). Leivant sketched a type inference algorithm for \( I_2 \), but the algorithm was not formalized and proved correct until recently [33]. Leivant also conjectured the undecidability of ranks 3 and higher in the intersection system; to our knowledge the details of his proof idea have never been verified.

\( I_2 \) has a significant advantage over \( \Lambda_2 \); it has principal typings. This means that for any term \( M \), if \( M \) is typable in \( I_2 \), then there is an \( I_2 \) typing judgment

\[
A \vdash M : \sigma
\]

that represents all of the possible typing judgments for \( M \). Other typings for \( M \) can be obtained from the principal typing by simple operations (substitution and subsumption).

Figure 2: Types that go beyond rank 2
Contributions of the paper

Since $I_2$ has principal typings, and $\Lambda_2$ does not, we believe $I_2$ deserves more study. The first contribution of this paper is to develop some of the basic properties of $I_2$. We establish the following equivalence: a term is typable in $I_2$ if and only if it is typable in $\Lambda_2$. An immediate corollary is that typability in $I_2$ is DEXPTIME-complete, identical to typability in $\Lambda_2$ and ML. We also consider some variants of $I_2$, and show they are all equivalent in terms of typability.

The second contribution of this paper is to introduce a new type system, $P_2$, that combines rank 2 intersection types and top-level quantification of type variables, as in ML. $P_2$ has principal typings, so it clearly improves on $\Lambda_2$. Its advantage over $I_2$ is more subtle. The addition of quantifiers makes types more expressive: the quantifiers identify generic type variables, that is, type variables which can safely be instantiated with any type. This permits a simpler definition of the type inference algorithm, and suggests a novel type inference algorithm for recursive definitions.

A recursive definition is written in the form $(\mu x . M)$, and is meant to denote a program $x$ such that

$$x = M,$$

where $M$ may contain some uses of $x$. The standard rule for typing recursive definitions looks like

$$\frac{A \cup \{x : \sigma\} \vdash M : \sigma}{A \vdash (\mu x . M) : \sigma}$$

Most type inference algorithms restrict the type $\sigma$ in this rule to be a simple type. The rule of polymorphic recursion relaxes this restriction by allowing $\sigma$ to be an ML type scheme. This gives a useful increase in typing power—it can type some natural programs that cannot be typed by the simple recursion rule. However, polymorphic recursion makes type inference undecidable [14].

We suggest another way of typing recursive definitions:

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

The rule says that as long as the type $\sigma$ of $M$ is more general than the assumption $\tau$ on $x$ needed to type $M$, we can deduce $\sigma$ as the type of the recursive definition.

---

2The equivalence between the rank 2 fragments of System F and the intersection type discipline has been shown independently by Yokouchi [35].
We extend \( P_2 \) to type recursive definitions in this way. The resulting system can type many (but not all) of the examples that seem to require polymorphic recursion. Moreover, the system has principal types and decidable type inference.

**Organization of the paper**

In §2, we introduce \( I_2 \), a syntax-directed version of \( I_2 \), and \( \Lambda_2 \), a syntax-directed version of \( \Lambda_2 \). The main result is that a term is typable in one system if and only if it is typable in the other. An immediate corollary is that typability in \( I_2 \) is DEXPTIME-complete, the same complexity as in \( \lambda I \) and \( \Lambda_2 \). In §3, we present the type inference algorithm for \( I_2 \). In §4, we discuss some other definitions of rank 2 intersection type systems, and show their equivalence with \( I_2 \). In §5, we define \( P_2 \), show that it has principal typings, and give a type inference algorithm. In §6, we discuss various ways of typing recursive definitions, and we propose an extension of \( P_2 \) that can type many examples of polymorphic recursion. We discuss applications of \( P_2 \) to compilation in §7, and we summarize our results in §8.

## 2 Rank 2 type systems

### 2.1 Preliminaries

We will be defining a number of type systems; here we develop machinery that will be useful in all of them.

We use \( x, y, \ldots \) to range over a countable set of variables, and \( t, s \) to range over a countable set, \( \mathbf{Tv} \), of type variables. The terms and types of the systems will vary, but in all cases we use \( \sigma, \tau, \ldots \) to range over types, and \( M, N, P, \ldots \) to range over terms.

The **terms of the (pure) lambda calculus** are defined by the following grammar:

\[
M ::= x \mid (M_1 M_2) \mid (\lambda x. M).
\]

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

The types of our systems will all be subsets of the types with quantification and intersection:

\[
\sigma ::= t \mid (\sigma_1 \rightarrow \sigma_2) \mid (\forall t. \sigma) \mid (\sigma_1 \wedge \sigma_2).
\]
By convention, ‘→’ associates to the right, so that, e.g., \((t \rightarrow (t \rightarrow t))\) may be written more compactly as \(t \rightarrow t \rightarrow t\), and ‘∧’ binds more tightly than ‘→’, e.g., \(\sigma \land \tau \rightarrow t\) means \((\sigma \land \tau) \rightarrow t\). The scope of a quantifier ‘∀’ extends as far to the right as possible. We write \((\forall t \sigma)\) for the type

\[
(\forall t_1(\forall t_2(\ldots(\forall t_n\sigma)\ldots)))
\]

where \(\vec{t} = t_1, t_2, \ldots, t_n\) and \(n \geq 0\).

The set of simple types, \(T_0\), is defined by the following inductive equation:

\[
T_0 = \{ t \mid t \text{ is a type variable} \} \cup \{ (\sigma \rightarrow \tau) \mid \sigma, \tau \in T_0 \}.
\]

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 \(A\), \(\text{dom}(A)\) 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 \(\text{dom}(A_1)\) and \(\text{dom}(A_2)\) are disjoint. For any set \(T\) of types, we say \(A\) is a \(T\) type environment if \(A(x) \in T\) for all \(x \in \text{dom}(A)\).

The notion of free type variable is defined as usual. We write \(\text{FTV}(\sigma)\) for the free type variables of a type \(\sigma\), and \(\text{FTV}(A)\) for the free type variables of all types appearing in \(A\). We write \(\text{Gen}(A, \tau)\) for the \(\forall\)-closure of \(\tau\) by the type variables \(\text{FTV}(\tau) - \text{FTV}(A)\).

A judgment is a relation between type environments, terms, and types, written \(A \vdash M : \sigma\). A term \(M\) is typable if \(A \vdash M : \sigma\) for some \(A\) and \(\sigma\). A pair \(<A, \sigma>\) of a type environment and a type is called simply a pair. Two pairs \(<A_1, \sigma_1>\) and \(<A_2, \sigma_2>\) are disjoint if their free type variables are disjoint. An acceptable pair of a term \(M\) in a type system is a pair \(<A, \sigma>\) such that the judgment \(A \vdash M : \sigma\) holds in the type system. We write \(\text{AP}_D(M)\) for the set of acceptable pairs of \(M\) in a type system \(D\).

A substitution is a mapping from type variables to simple types which is the identity on all but a finite number of type variables. We use \(S, R, Q, U\) to range over substitutions. The domain and range of a substitution \(S\) are defined

\[
\text{dom}(S) = \{ t \mid S t \neq t \},
\]

\[
\text{rng}(S) = \bigcup_{t \in \text{dom}(S)} \text{FTV}(St).
\]
If \( \text{dom}(S) = \{t_1, t_2, \ldots, t_n\} \) and \( St_i = \tau_i \) for all \( i \), then \( S \) can be written in the form \( \{t_1 := \tau_1, \ldots, t_n := \tau_n\} \).

The application of substitutions is extended to types, type environments, and pairs in the usual way. The composition of substitutions is denoted by juxtaposition, so that \( SRt = (SR)t = S(R(t)) \). We say \( S_1 \) and \( S_2 \) are disjoint if \( \text{dom}(S_1) \) and \( \text{dom}(S_2) \) are disjoint sets. If \( S_1 \) and \( S_2 \) are disjoint, then the substitution \( S_1 \cup S_2 \) is defined as follows:

\[
(S_1 \cup S_2)(t) = \begin{cases} 
S_1(t) & \text{if } t \in \text{dom}(S_1), \\
S_2(t) & \text{if } t \in \text{dom}(S_2), \\
\perp & \text{otherwise}.
\end{cases}
\]

Note that we have made a severe restriction on substitutions: they map type variables only to simple types, and not types in general.

\section{The rank 2 intersection type system}

There are many different formulations of intersection type systems; see van Bakel [33] for a survey. We will present a very restricted intersection type system here, the system of rank 2 intersection types. Our system is a slight generalization of van Bakel's version (see §4.1).

The terms of the intersection type system are just the terms of the lambda calculus. The sets \( T_1 \) and \( T_2 \) are defined to be the smallest sets satisfying the following equations:

\[
T_1 = T_0 \cup \{ (\sigma \land \tau) \mid \sigma, \tau \in T_1 \}, \\
T_2 = T_0 \cup \{ (\sigma \land \tau) \mid \sigma \in T_1, \tau \in T_2 \}.
\]

The set \( T_1 \) of rank 1 types consists of finite, nonempty intersections of simple types. \( T_2 \) 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 \( T_0 = T_1 \cap T_2 \), and for \( i \in \{0, 1, 2\} \), if \( \tau \in T_i \), then \( S\tau \in T_i \).

In order to simplify subsequent definitions, we adopt the following syntactic convention: we consider '\( \land \)' to be an associative, commutative, and idempotent operator, so that any \( T_1 \) type may be considered a finite, nonempty set of simple types, written in the form \( (\bigwedge_{i \in I} \sigma_i) \), where each \( \sigma_i \in T_0 \).

\textbf{Definition 1} For \( i \in \{1, 2\} \), we define the relation \( \leq_i \) as the least partial order on \( T_i \) closed under the following rules:

i) If \( \{ \tau_j \mid j \in J \} \subseteq \{ \sigma_i \mid i \in I \} \), then \( (\bigwedge_{i \in I} \sigma_i) \leq_1 (\bigwedge_{j \in J} \tau_j) \).
\[
\begin{align*}
&\text{(VAR)} \quad A, x \cup \{x : (\bigwedge_{i \in I} \tau_i)\} \vdash x : \tau_i \quad \text{(where } i_0 \in I) \\
&\text{(ABS)} \quad \underline{A, x \cup \{x : \sigma\} \vdash M : \tau} \\
&\quad \vdash (\lambda x M) : \sigma \rightarrow \tau \\
&\text{(APP)} \quad \underline{A \vdash \lambda x M : \sigma} \quad \underline{A \vdash N : \tau_i} \\
&\quad \vdash (MN) : \sigma
\end{align*}
\]

Figure 3: Typing rules of \(F_2\). Types in type environments are in \(T_1\), and derived types are in \(T_2\).

\[\text{(VAR)} \quad A, x \cup \{x : (\bigwedge_{i \in I} \tau_i)\} \vdash x : \tau_i \quad \text{(where } i_0 \in I) \\
\text{(ABS)} \quad \underline{A, x \cup \{x : \sigma\} \vdash M : \tau} \\
\quad \vdash (\lambda x M) : \sigma \rightarrow \tau \\
\text{(APP)} \quad \underline{A \vdash \lambda x M : \sigma} \quad \underline{A \vdash N : \tau_i} \\
\quad \vdash (MN) : \sigma
\]

The first rule says that \(\leq_1\) expresses the natural ordering on intersection types, and the second rule says that \(\leq_2\) obeys the usual antimonotonic ordering on function types, restricted to rank 2.

Some useful properties of the orderings \(\leq_1\) and \(\leq_2\) are summarized in the following lemma.

**Lemma 2**

i) If \(\sigma \in T_0\) and \(\tau \in T_1\), then \(\sigma \leq_1 \tau\) iff \(\sigma = \tau\).

ii) If \(\sigma \in T_2\) and \(\tau \in T_0\), then \(\sigma \leq_2 \tau\) iff \(\sigma = \tau\).

iii) \((\bigwedge_{i \in I} \sigma_i) \leq_1 (\bigwedge_{j \in J} \tau_j)\) iff for all \(j \in J\) there exists an \(i \in I\) such that \(\tau_j = \sigma_i\).

iv) \((\bigwedge_{j \in J} \tau_j) \rightarrow \tau \leq_2 (\bigwedge_{i \in I} \sigma_i) \rightarrow \sigma\) iff \(\tau \leq_2 \sigma\), and for all \(j \in J\) there exists an \(i \in I\) such that \(\tau_j = \sigma_i\).

v) For \(i \in \{1, 2\}\), if \(\sigma \leq_i \tau\), then \(S\sigma \leq_i S\tau\).

Judgments in our rank 2 system are defined inductively by the rules of Figure 3. We write \(F_2 \vdash A \vdash M : \sigma\) if the judgment \(A \vdash M : \sigma\) follows by these rules, with types appearing in type environments restricted to \(T_1\), and derived types restricted to \(T_2\). The superscript ‘s’ in \(F_2\) indicates that the system is syntax-directed, in contrast with a later variant (see §4).

If \(A_1\) and \(A_2\) are \(T_1\) type environments, we define \(A_1 + A_2\), a \(T_1\) type
environment, as follows: for each \( x \in \text{dom}(A_1) \cup \text{dom}(A_2) \),

\[
(A_1 + A_2)(x) = \begin{cases} 
A_1(x) & \text{if } x \notin \text{dom}(A_2), \\
A_2(x) & \text{if } x \notin \text{dom}(A_1), \\
A_1(x) \land A_2(x) & \text{otherwise.}
\end{cases}
\]

**Lemma 3 (Weakening)** If \( \Gamma_2 \vdash A \vdash \sigma \), then \( \Gamma_2 \vdash A + A' \vdash \sigma \) for any \( T_1 \) type environment \( A' \).

**Proof:** An easy induction on typing derivations. \( \square \)

**Lemma 4 (Substitutivity)** If \( \Gamma_2 \vdash A \vdash \sigma \), then \( \Gamma_2 \vdash SA \vdash \sigma \) for any substitution \( S \).

**Proof:** By induction on the structure of \( M \).

i) If \( M = x \), then \( A(x) = (\forall \xi \in I \sigma_i) \) and \( \sigma = \sigma_{i_0} \) for some \( i_0 \in I \). Then \( S A(x) = (\forall \xi \in I S \sigma_i) \), \( \Gamma_2 \vdash SA \vdash x : S \sigma_{i_0} \), and \( S \sigma = S \sigma_{i_0} \).

ii) If \( M = \lambda x N \) then \( \sigma \) must be of the form \( \tau_1 \rightarrow \tau_2 \), and \( \Gamma_2 \vdash A_x \cup \{ x : \tau_1 \} \vdash N : \tau_2 \). Then by induction, \( \Gamma_2 \vdash S(A_x \cup \{ x : \tau_1 \}) \vdash N : S \tau_2 \), so by rule (ABS), \( \Gamma_2 \vdash S A_x \vdash N : S \tau_1 \rightarrow S \tau_2 \), or \( \Gamma_2 \vdash S A_x \vdash N : S(\tau_1 \rightarrow \tau_2) \). Then by weakening, \( \Gamma_2 \vdash S A \vdash N : S(\tau_1 \rightarrow \tau_2) \).

iii) If \( M = M_1 M_2 \), then for some \( (\forall \xi \in I \tau_i) \in T_1 \) we have \( \Gamma_2 \vdash A \vdash M_1 : (\forall \xi \in I \tau_i) \rightarrow \sigma \) and \( \Gamma_2 \vdash A \vdash M_2 : \tau_i \) for all \( i \in I \). By induction we have \( \Gamma_2 \vdash SA \vdash M_1 : (\forall \xi \in I S \tau_i) \rightarrow S \sigma \) and \( \Gamma_2 \vdash SA \vdash M_2 : S \tau_i \), and by rule (APP), we have \( \Gamma_2 \vdash SA \vdash M_1 M_2 : S \sigma \), as desired. \( \square \)

### 2.3 System F

The terms of System F are exactly the terms of the lambda calculus. The types of System F are defined by the following grammar:

\[
\tau ::= t \mid (\tau_1 \rightarrow \tau_2) \mid (\forall \tau).
\]

We consider System F types to be syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers.
The types of System F can be organized into a hierarchy as follows. First, define $R(0) = T_0$. Then for $n \geq 0$, the set $R(n + 1)$ is defined to be the least set satisfying

$$R(n + 1) = R(n) \cup \{ (\sigma \rightarrow \tau) \mid \sigma \in R(n), \tau \in R(n + 1) \} \cup \{ (\forall \sigma \mid \sigma \in R(n + 1)) \}.$$ 

It will be useful to restrict types so that quantifiers do not appear to the immediate right of arrows. Therefore we define the sets

$$S = S' \cup \{ (\forall \sigma) \mid \sigma \in S \},$$

$$S' = T_0 \cup \{ (\sigma \rightarrow \tau) \mid \sigma \in S, \tau \in S' \}.$$ 

We write $S(n)$ for $S \cap R(n)$ and $S'(n)$ for $S' \cap R(n)$. Note that the $S(1)$ types are exactly the ML type schemes.

**Definition 5** Suppose $\sigma = \forall t_1 \cdots t_n. \tau \in S(1)$, and $\tau, \tau' \in T_0$. We say $\tau'$ is an instance of $\sigma$, written $\sigma \triangleright \tau'$, if and only if for some $\rho_1, \ldots, \rho_n \in T_0$, we have $\tau' = \{ t_1 := \rho_1, \ldots, t_n := \rho_n \} \tau$. We write $\sigma \triangleright (\forall s_1 \cdots s_m \tau')$ if and only if $s_1, \ldots, s_m$ are not free in $\sigma$ and $\sigma \triangleright \tau'$.

Note that the sense of ‘$\triangleright$’ is opposite to that of our other subtyping relations: both “$\sigma \leq_2 \tau$” and “$\sigma \triangleright \tau$” may be read, “$\sigma$ is more general than $\tau$.” We make an exception in the case of ‘$\triangleright$’ to be consistent with its use in ML [24].

We now define $\Lambda_2^n$, our version of the rank 2 fragment of System F. The superscript ‘$s$’ in $\Lambda_2^n$ indicates that the system is syntax-directed. See Kfoury and Tiuryn [12] for a definition of $\Lambda_2$, the non-syntax-directed version.

The judgments of the system are defined by the rules of Figure 4. We write $\Lambda_2^n \vdash A \vdash M : \tau$ if $A \vdash M : \tau$ is derivable from these rules, where types in type environments are restricted to $S(1)$, and derived types are restricted to $S'(2)$.

$\Lambda_2^n$ is closely related to the system $\Lambda_2^s$ studied by Kfoury et al. [12, 17]:

**Theorem 6**

i) If $\Lambda_2^n \vdash A \vdash M : \sigma$, then $\Lambda_2^s \vdash A \vdash M : \sigma$.

ii) If $\Lambda_2^s \vdash A \vdash M : \sigma$, then $\sigma$ is of the form $\forall t_1 \cdots t_n \sigma'$, where $\sigma' \in S'(2)$, and $\Lambda_2^n \vdash A \vdash M : \sigma'$.

This equivalence follows immediately from results of Kfoury and Wells [17]. It implies the following useful result:

**Lemma 7** If $\Lambda_2^n \vdash A \vdash M : \sigma$ and $\text{Gen}(A, \sigma) \triangleright \sigma'$, then $\Lambda_2^n \vdash A \vdash M : \sigma'$.
The typing rules of $\Lambda_2^\varphi$. Types in type environments are in $S(1)$, and derived types are in $S'(2)$.

\begin{align*}
(VAR) & \quad A_x \cup \{x : \sigma\} \vdash x : \tau & \text{(where } \sigma \triangleright \tau) \\
(ABS) & \quad A \vdash \{x : \tau_1\} \vdash M : \tau_2 \quad \Rightarrow \quad A \vdash (\lambda x M) : \tau_1 \rightarrow \tau_2 \\
(APP) & \quad A \vdash M : (\forall \tau_1 \rightarrow \tau_2), \quad A \vdash N : \tau_1 \quad \Rightarrow \quad A \vdash (MN) : \tau_2 & \text{(each } t_i \not\in \text{FTV}(A))
\end{align*}

Figure 4: Typing rules of $\Lambda_2^\varphi$. Types in type environments are in $S(1)$, and derived types are in $S'(2)$.

2.4 ML

Many different formulations of the ML type system have been studied; we choose to present a syntax-directed version here, as in Clement et al. [4] or Toft & [32].

The **types of ML** are the types $T_0$, and the **ML type schemes** are the types $S(1)$. The **terms of ML** are the terms of the lambda calculus extended with let-expressions:

$$M ::= x \mid (M_1 M_2) \mid (\lambda x M) \mid (\text{let } x = M_1 \text{ in } M_2).$$

The judgments of ML are defined inductively by the rules of Figure 5. We write $ML \triangleright A \vdash M : \tau$ if $A \vdash M : \tau$ is derivable from these rules, where types in type environments are restricted to $S(1)$, and derived types are restricted to $T_0$.

**Definition 8** An ML type $\tau$ is a **principal type for $M$ in $A$** if and only if $ML \triangleright A \vdash M : \tau$, and for all ML types $\tau'$, if $ML \triangleright A \vdash M : \tau'$, then $Gen(A, \tau) \triangleright \tau'$.

**Theorem 9 (Principal types for ML)** If $M$ is typable by $A$, then there exists a principal type for $M$ in $A$.

**Lemma 10** If $ML \triangleright A \vdash M : \tau$, and $Gen(A, \tau) \triangleright \tau'$, then $ML \triangleright A \vdash M : \tau'$.

2.5 Relationship of $\Lambda_2^\varphi$ and $I_2^\varphi$

We now show that a term is typable in $\Lambda_2^\varphi$ if and only if it is typable in $I_2^\varphi$. The left to right implication is developed first.
Figure 5: Typing rules of ML. Types in type environments are in $S(1)$, and derived types are in $T_0$.

Definitions 11

i) We define a relation $\preceq_1$ between $S(1)$ and $T_1$ as follows. Suppose $\tau \in S(1)$ and $\sigma_1, \ldots, \sigma_n \in T_0$ ($n \geq 1$). Then $\tau \preceq_1 (\Lambda_{i \in I} \sigma_i)$ if and only if $\tau \triangleright \sigma_i$ for all $i \in I$.

ii) We define the relation $\preceq_2$ between $S'(2)$ and $T_2$ inductively:

a) For any type variable $t$, $t \preceq_2 t$.

b) If $\tau \preceq_1 \tau'$ and $\sigma \preceq_2 \sigma'$, then $(\tau \rightarrow \sigma) \preceq_2 (\tau' \rightarrow \sigma')$.

Note that the relation $\preceq_2$ is monotonic in the argument of function types, in contrast to the relation $\preceq_2$. We extend the relation $\preceq_1$ to type environments as follows: $A \preceq_1 A'$ if and only if $x \in \text{dom}(A)$ and $A(x) \preceq_1 A'(x)$ whenever $x \in \text{dom}(A')$.

Theorem 12 If $A_2 \triangleright A \vdash M : \tau$, then $A_2 \triangleright A' \vdash M : \tau'$, where $A \preceq_1 A'$ and $\tau \preceq_2 \tau'$.

Proof: By induction on derivations.

i) $M = x$ and $A_2 \triangleright A \vdash x : \tau$ follows by the $A_2$ rule (VAR). Then we must have $A(x) \triangleright \tau$.

Let $A' = \{ x : \tau \}$. Clearly $A_2 \triangleright A' \vdash M : \tau$, $A \preceq_1 A'$, and $\tau \preceq_2 \tau$.

ii) $M = \lambda x N$, $\tau = \sigma \rightarrow \tau_1$, and $A_2 \triangleright A \vdash \lambda x N : \sigma \rightarrow \tau_1$ follows by the $A_2$ rule (ABS).
Then we must have
\[ \Lambda^s_n \triangleright A \cup \{x : \sigma\} \vdash N : \tau_1. \]

By induction, we have
\[ \Gamma_2 \triangleright A' \cup \{x : \sigma'\} \vdash N : \tau'_1, \]
where \( A \succeq_1 A', \sigma \succeq_1 \sigma', \) and \( \tau_1 \succeq_2 \tau'_1. \) So by the \( \Gamma_2 \) rule (ABS), we have
\[ \Gamma_2 \triangleright A' \vdash N : \sigma' \rightarrow \tau'_1, \]
where \( A \succeq_1 A' \), and \((\sigma \rightarrow \tau_1) \succeq_2 (\sigma' \rightarrow \tau'_1), \) as desired.

iii) \( M = M_1 M_2 \) and \( \Lambda^s_n \triangleright A \vdash M_1 M_2 : \tau \) follows by the \( \Lambda^s_n \) rule (APP).

Then we must have, for some \( \tau_0 \in T_0, \)
\[ \Lambda^s_n \triangleright A \vdash M_1 : (\forall \vec{t}. \tau_0) \rightarrow \tau, \]
\[ \Lambda^s_n \triangleright A \vdash M_2 : \tau_0, \]
where the type variables \( \vec{t} \) do not appear in \( \text{FTV}(A). \) Then by induction we have
\[ \Gamma_2 \triangleright A'_0 \vdash M_1 : (\bigwedge_{i \in I} \tau_i) \rightarrow \tau', \]
where \( A \succeq_1 A'_0, \tau \succeq_2 \tau', \) and \((\forall \vec{t}. \tau_0) \succeq_1 (\bigwedge_{i \in I} \tau_i). \)

Then each \( \tau_i \) is an instance of \((\forall \vec{t}. \tau_0), \) and therefore by Lemma 7, \( \Lambda^s_n \triangleright A \vdash M_2 : \tau_i \) for all \( i \in I. \)

By induction we have for all \( i \in I, \Gamma_2 \triangleright A'_i \vdash M_2 : \tau_i, \) where \( A \succeq_1 A'_i. \)
So if \( A' = A'_0 + \Sigma_{i \in I} A'_i, \) then \( A \succeq_1 A'. \) and by weakening,
\[ \Gamma_2 \triangleright A'_i \vdash M_1 : (\bigwedge_{i \in I} \tau_i) \rightarrow \tau', \]
\[ \Gamma_2 \triangleright A' \vdash M_2 : \tau_i \quad (\forall i \in I). \]

Then by the \( \Gamma_2 \) rule (APP) we have
\[ \Gamma_2 \triangleright A' \vdash M_1 M_2 : \tau', \]

as desired.

\[ \Box \]

We now show the other direction of the equivalence: any term typable in \( \Gamma_2 \) is typable in \( \Lambda^s_n. \)
Convention 13 In the remainder of this section we do not consider terms to be identical modulo α-conversion, and we will assume the following convention regarding the names of bound and free variables:

i) No variable is bound more than once.

ii) The bound and free variables are disjoint.

This convention is necessary to make the following function well-defined:

Definition 14 Let ε denote the empty sequence. The function, act, that maps terms to sequences of variables, is defined inductively by the following rules.\(^3\)

i) act(x) = ε.

ii) If act(M) = x₁, ..., xₙ then act(λxM) = y, x₁, ..., xₙ.

iii) If act(M) = y, x₁, ..., xₙ (n ≥ 0) then act(MN) = x₁, ..., xₙ.

iv) If act(M) = ε then act(MN) = ε.

Definition 15

i) γ is the rule

\[(λx(λyM))N → λy((λxM)N)\].

ii) →γ is the compatible closure of γ.

iii) A γ-redex is any term matching the left-hand side of the rule γ. We say M is a γ-normal form, or γ-nf, if no subterm of M is a γ-redex.

Note that by our convention on the distinct naming of variables, there is no capture of variables in the γ rule. We use the name “γ” in accordance with Kfoury and Wells [18]. See Barendregt [2] for a definition of “compatible.”

Lemma 16

i) →γ is strongly normalizing.

ii) →γ satisfies the diamond property.

iii) γ-nf’s are unique.

\(^3\)Our definition is identical to the definition of [12], but differs from [11].
Proof:

i) The proof is similar to the proof of Lemma 5.5 from Kfoury and Wells [17]:

Let \( \text{appl}(M) \) be the set of subterms of \( M \) that are applications, and let

\[
\delta(M) = \sum_{(M_1,M_2) \in \text{appl}(M)} \max(0, |\text{act}(M_1)| - 1).
\]

If \( M \rightarrow N \), then \( \delta(M) = \delta(N) + 1 \). Since for any \( M \) we have \( \delta(M) \geq 0 \), we can conclude that \( \rightarrow \) is strongly normalizing. In fact, \( \delta(M) > 0 \) iff \( M \) contains a \( \gamma \)-redex, and \( M \) normalizes in exactly \( \delta(M) \) steps.

If \(|M|\) is the size (number of subterms) of \( M \), then clearly \(|\text{appl}(M)| \leq |M| \) and \(|\text{act}(M)| \leq |M| \). Thus \( \delta(M) \leq |M|^2 \). Therefore normalization of a term \( M \) takes \( O(|M|^2) \) steps.

ii) This is a simple case analysis.

iii) This follows from (ii).

\( \square \)

Lemma 16 justifies the following definition:

**Definition 17** We write \( \gamma\text{-nf}(M) \) for the \( \gamma \)-nf of \( M \).

**Lemma 18** For \( \mathcal{D} \in \{\mathcal{I}_2, \mathcal{I}_2^e\} \), the following hold:

i) \( \mathcal{D} \vdash A \vdash (\lambda x (\lambda y M)) N : \sigma \quad \text{iff} \quad \mathcal{D} \vdash A \vdash \lambda y ((\lambda x M) N) : \sigma. \)

ii) If \( M \rightarrow N \), then \( \mathcal{D} \vdash A \vdash M : \sigma \quad \text{iff} \quad \mathcal{D} \vdash A \vdash N : \sigma. \)

iii) \( \mathcal{D} \vdash A \vdash M : \sigma \quad \text{iff} \quad \mathcal{D} \vdash A \vdash \gamma\text{-nf}(M) : \sigma. \)

**Proof:**

i) Simple case analysis.

ii) Use (i) and induction on the definition of compatible.

iii) Use (ii) and induction on the length of rewriting.

\( \square \)
Lemma 19 If $\text{act}(M) = x_1, \ldots, x_n$ and $E_n \vdash A \vdash M : \sigma$, then $\sigma$ is of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, where $\tau \in T_0$.

Proof: By induction on the structure of $M$.

i) If $M = x$, then $n = 0$ by the definition of $\text{act}$, and $\sigma \in T_0$ by rule (VAR).

ii) If $M = \lambda x_1 N$, then $E_n \vdash A \vdash M : \sigma$ follows by rule (ABS), and therefore $\sigma$ is of the form $\sigma_1 \rightarrow \sigma'$, where $\sigma_1 \in T_1$.

Also we must have $\text{act}(N) = x_2, \ldots, x_n$ ($n \geq 1$) and $E_n \vdash A \cup \{x_1 : \sigma_1\} \vdash N : \sigma'$. By induction $\sigma'$ must be of the form $\sigma_2 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, where $\sigma_2, \ldots, \sigma_n \in T_1$ and $\tau \in T_0$.

iii) If $M = M_1 M_2$, then $E_n \vdash A \vdash M : \sigma$ follows by rule (APP), and therefore we have $E_n \vdash M_1 : \sigma' \rightarrow \sigma$, where $\sigma' \in T_1$.

We consider two cases. If $\text{act}(M_1) = \epsilon$, then $\text{act}(M) = \epsilon$, so we only need prove $\sigma \in T_0$. And by induction, we have $(\sigma' \rightarrow \sigma) \in T_0$, so $\sigma \in T_0$.

\[\square\]

Note 20 A similar lemma holds for $\Lambda_3^\alpha$, c.f. Kfoury et al. [12], Lemma 15.

Lemma 21 Suppose $M$ is a $\gamma$-nf. Then

$\text{act}(M) \neq \epsilon$ iff $M = \lambda y N$ for some $y, N$.

Proof: By induction on the structure of $M$. The cases $M = x$ and $M = \lambda y N$ are trivial, so assume $M = M_1 M_2$. We must show $\text{act}(M) = \epsilon$.

By way of contradiction, assume that $\text{act}(M) = x_1, \ldots, x_n$ ($n \geq 1$). By the definition of $\text{act}$, we must have $\text{act}(M_1) = y, x_1, \ldots, x_n$ for some $y$. Then $\text{act}(M_1) \neq \epsilon$, so by induction we have $M_1 = \lambda y M'_1$, and $\text{act}(M'_1) = x_1, \ldots, x_n$. Since $n \geq 1$, $\text{act}(M'_1) \neq \epsilon$, and by induction $M'_1 = \lambda x_1 M''_1$. But then $M$ is a $\gamma$-redex, contradiction. \[\square\]

Definition 22 We define a mapping, $\text{ml}$, from terms to ML terms:

i) $\text{ml}(x) = x$.

ii) $\text{ml}(\lambda x M) = (\lambda x \text{ml}(M))$. 

17
iii) $\mathtt{ml}(M_1 M_2) = \begin{cases} \mathtt{(let}\ x = \mathtt{ml}(M_2)\ \mathtt{in}\ \mathtt{ml}(N)) & \text{if } M_1 = \lambda x N, \\ \mathtt{ml}(M_1) \mathtt{ml}(M_2) & \text{otherwise.} \end{cases}$

**Definition 23**

i) A *generalization* of a set $T$ of simple types is a type $\sigma \in S(1)$ such that $\sigma \triangleright \tau$ for every $\tau \in T$. A generalization $\sigma$ of $T$ is the *least common generalization* of $T$ if $\sigma' \triangleright \sigma$ for any other generalization $\sigma'$ of $T$.

ii) If $(\wedge_{i \in I} \tau_i) \in T_1$, we define $\mathtt{lcg}(\wedge_{i \in I} \tau_i)$ to be the least common generalization of $\{\tau_i | i \in I\}$. If $\sigma_1, \ldots, \sigma_n \in T_1$ and $\tau \in T_0$, then

$$\mathtt{lcg}(\sigma_1 \to \cdots \to \sigma_n \to \tau) = \mathtt{lcg}(\sigma_1) \to \cdots \to \mathtt{lcg}(\sigma_n) \to \tau.$$ 

The function $\mathtt{lcg}$ is extended to type environments in the usual way.

The use of “least” in the name “least common generalization” is consistent with the relation ‘$\triangleright$’. Recall that the sense of ‘$\triangleright$’ is opposite to that of our other subtyping relations, so that “least” for ‘$\triangleright$’ means “greatest” for the other relations.

The concept of least common generalizations was developed by Plotkin [26] and Reynolds [27]. They showed that any finite nonempty set of simple types has a least common generalization, and they gave an algorithm to compute it.

**Lemma 24** If $M$ is a $\gamma$-nf and $\sigma \in T_0$, then

i) $P_2 \triangleright A \vdash M : \sigma$ implies $ML \triangleright \mathtt{lcg}(A) \vdash \mathtt{ml}(M) : \sigma$; and

ii) $A \triangleright A \vdash M : \sigma$ if and only if $ML \triangleright A \vdash \mathtt{ml}(M) : \sigma$.

**Proof:**

i) By induction on the structure of $M$.

a) The case $M = x$ is trivial.

b) If $M = \lambda y N$, then $P_2 \triangleright A \vdash M : \sigma$ follows by the $P_2$ rule (ABS), so $\sigma$ must be of the form $\tau \to \sigma'$ where $\tau, \sigma' \in T_0$, and $P_2 \triangleright A \cup \{y : \tau\} \vdash N : \sigma'$. Note that $N$ is a $\gamma$-nf, so we can apply the induction hypothesis to get

$$ML \triangleright \mathtt{lcg}(A \cup \{y : \tau\}) \vdash \mathtt{ml}(N) : \sigma'.$$

Now $\tau \in T_0$, so $\mathtt{lcg}(A \cup \{y : \tau\}) = \mathtt{lcg}(A) \cup \{y : \tau\}$. Therefore $ML \triangleright \mathtt{lcg}(A) \cup \{y : \tau\} \vdash \mathtt{ml}(N) : \sigma'$, so by the ML rule (ABS), $ML \triangleright \mathtt{lcg}(A) \vdash \mathtt{ml}(\lambda y N) : \tau \to \sigma'$, as desired.
c) If $M = (\lambda y M_1)M_2$, then our judgment must follow by $\textbf{I}_2$ rules (ABS) and (APP). Thus we have

$$\textbf{I}_2 \triangleright A \cup \{y : (\lambda_{i \in I} \sigma_i)\} \vdash M_1 : \sigma,$$

$$\forall i \in I \quad \textbf{I}_2 \triangleright A \vdash M_2 : \sigma_i.$$

Let $\forall \tilde{\tau} = \text{lcg}(\lambda_{i \in I} \sigma_i)$, where $\tau \in T_0$, and no $t_i$ appears in $A$. By induction, we have

$$\text{ML} \triangleright \text{lcg}(A) \cup \{y : \forall \tilde{\tau}\} \vdash \text{ml}(M_1) : \sigma,$$

$$\forall i \in I \quad \text{ML} \triangleright \text{lcg}(A) \vdash \text{ml}(M_2) : \sigma_i.$$

By the principal type property of ML, we have

$$\text{ML} \triangleright \text{lcg}(A) \vdash \text{ml}(M_2) : \tau.$$

Then since $\text{ml}(M) = (\text{let } y = \text{ml}(M_2) \text{ in } \text{ml}(M_1))$, we have

$$\text{ML} \triangleright \text{lcg}(A) \vdash \text{ml}(M) : \sigma$$

by the ML rule (LET).

d) If $M = M_1M_2$, where $M_1$ is not an abstraction, then by the $\textbf{I}_2$ rule (APP), we have for some $\sigma'$,

$$\textbf{I}_2 \triangleright A \vdash M_1 : \sigma' \rightarrow \sigma,$$

$$\textbf{I}_2 \triangleright A \vdash M_2 : \sigma'.$$

$M_1$ is a $\gamma$-nf and is not an abstraction, so by Lemma 21, we have $\text{act}(M_1) = \epsilon$. Then by Lemma 19, $\sigma' \rightarrow \sigma \in T_0$, and therefore $\sigma' \in T_0$. $M_2$ is also a $\gamma$-nf, so we may apply the induction hypothesis to both judgments above, to get

$$\text{ML} \triangleright \text{lcg}(A) \vdash \text{ml}(M_1) : \sigma' \rightarrow \sigma,$$

$$\text{ML} \triangleright \text{lcg}(A) \vdash \text{ml}(M_2) : \sigma'.$$

Then by the ML rule (APP), we have

$$\text{ML} \triangleright \text{lcg}(A) \vdash \text{ml}(M_1M_2) : \sigma,$$

as desired.

ii) Similar, but easier.
Note 25 The converse of Lemma 24(i) does not hold. For instance, if $\sigma = \tau_3$ and $A = \{x : t_1 \land t_2\}$, then $\text{lcg}(A) = \{x : \forall t.t\}$, $\text{ml}(xx) = xx$, and $\text{ML} \vdash \{x : \forall t.t\} \vdash xx : \tau_3$, but the judgment $\{x : t_1 \land t_2\} \vdash xx : \tau_3$ cannot be derived in $\Pi_2^\#$. 

Theorem 26 If $\Pi_2 \vdash A \vdash M : \sigma$, then $\Lambda_2^\# \vdash \text{lcg}(A) \vdash M : \text{lcg}(\sigma)$.

Proof: Suppose $\text{act}(M) = x_1, \ldots, x_n$. Then by Lemma 19, $\sigma$ is of the form $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$, where $\tau \in T_0$, and by Lemma 21, the $\gamma$-nf of $M$ is of the form $\lambda x_1 \cdots \lambda x_n N$, where $N$ is a $\gamma$-nf. By Lemma 18(iii),

$$\Pi_2 \vdash A \vdash \lambda x_1 \cdots \lambda x_n N : \sigma.$$

This judgment must follow by $n$ uses of the $\Pi_2$ rule (ABS), so we have

$$\Pi_2 \vdash A \cup \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash N : \tau.$$

Then by Lemma 24, we have

$$\Lambda_2^\# \vdash \text{lcg}(A \cup \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}) \vdash N : \tau.$$

By $n$ uses of the $\Lambda_2^\#$ rule (ABS), we have

$$\Lambda_2^\# \vdash \text{lcg}(A) \vdash \lambda x_1 \cdots \lambda x_n N : \text{lcg}(\sigma),$$

and by Lemma 18(iii), we have

$$\Lambda_2^\# \vdash \text{lcg}(A) \vdash M : \text{lcg}(\sigma).$$

\[\square\]

Theorem 27 If $M$ is a term of the pure lambda calculus, then $M$ is typable in $\Pi_2$ if and only if $M$ is typable in $\Lambda_2^\#$.

Therefore, typability in $\Pi_2$ is $\text{DEXPTIME}$-complete.

Proof: The equivalence of $\Pi_2$ and $\Lambda_2^\#$ typability follows from Theorems 12 and 26.

Kfoury and Tiuryn [12] show that $\Lambda_2^\#$ typability is polynomial time equivalent to $\text{ML}$ typability. $\text{ML}$ typability was shown to be $\text{DEXPTIME}$-complete independently by Kfoury et al. [15] and by Mairson [22].

This equivalence has been shown independently by Yokouchi [35].
3 Type inference for $I_2$

We present the type inference algorithm for $I_2$ and a proof that it infers principal pairs. The algorithm is not new: it was described briefly in Leivant’s original paper [21], and was defined rigorously by van Bakel in his dissertation [33]. We include it here because the algorithm provides a way to compare a variety of type systems based on rank 2 intersection types.

The algorithm takes as input a term $M$, and produces a pair $\langle A, \sigma \rangle$ such that $I_2 \vdash A \vdash M : \sigma$. Moreover, the pair $\langle A, \sigma \rangle$ is principal in the sense that any other acceptable pair of $M$ can be obtained from $\langle A, \sigma \rangle$ by some well-understood operations.

**Definition 28**

i) We write $A \leq_1 A'$ if $x \in \text{dom}(A)$ and $A(x) \leq_1 A'(x)$ for all $x \in \text{dom}(A')$.

ii) The ordering $\leq$ on $(T_1 \text{ type environment, } T_2 \text{ type})$ pairs is defined as follows:

$$\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle \text{ if and only if } A' \leq_1 A \text{ and } \sigma \leq_2 \sigma'.$$

iii) A pair $\langle A, \sigma \rangle$ is a principal pair for $M$ if $\langle A, \sigma \rangle \in \text{AP}_{I_2}(M)$, and for any other pair $\langle A', \sigma' \rangle \in \text{AP}_{I_2}(M)$, there is a substitution $S$ such that $S\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle$.

Note that $\leq_1$ and $\leq$ are transitive, and $A + A' \leq_1 A$ for all $T_1 \text{ type environments } A, A'$.

3.1 Subtype satisfaction

In this section we give a decision procedure for one of our subtyping relations, and show how to solve a more general problem, subtype satisfaction, that we use in our type inference algorithm.

Up until now, we have relied on some syntactic conventions to simplify our presentation, namely, that ‘$\land$’ is an associative, commutative, and idempotent operator. Part of the problem we are addressing here is how to decide whether two types are equivalent under these assumptions. Therefore, in this section, we do not rely on the syntactic conventions in any way.

Subtype satisfaction is a generalization of the well-known problem of unification, and the techniques we use here are based on those used to solve unification. For more details, consult a survey on unification [19, 20, 30, 10,
6, 31, 1]. One difference between unification and our satisfaction problems is that we work with types that go beyond simple types, but our substitutions involve only simple types. This is not the typical case with unification, and it makes our problem easier to solve.

If \( S_1, S_2 \) are substitutions and \( V \) is a set of type variables, we say \( S_1 \) and \( S_2 \) are equivalent on \( V \), written \( S_1 =_V S_2 \), if \( S_1 t = S_2 t \) for every \( t \in V \). We say \( S_1 \) is more general than \( S_2 \) on \( V \), written \( S_1 \preceq_V S_2 \), if there is a substitution \( S_3 \) such that \( S_2 =_V S_3 S_1 \). The relation \( \preceq_V \) is a partial order modulo \( =_V \). We omit \( V \) when \( V = TV \). A substitution \( S \) is idempotent if \( S = SS \), or, equivalently, if \( \text{dom}(S) \cap \text{rng}(S) = \emptyset \).

We define the relation \( \preceq_{2,1} \) between \( T_2 \) and \( T_1 \) as the least relation closed under the rule:

- If \( \sigma \preceq_{2,1} \tau \); for all \( i \in I \), then \( \sigma \preceq_{2,1} (\bigwedge_{i \in I} \tau_i) \).

A \( \preceq_{2,1} \)-satisfaction problem is a pair \( \exists \bar{s}. P \), where \( P \) is a finite set whose every element is either: 1) an equality between simple types; or 2) an inequality between a \( T_2 \) type and a \( T_1 \) type. When \( \bar{s} \) is empty \( \exists \bar{s} \) may be omitted. We use \( \pi \) to range over \( \preceq_{2,1} \)-satisfaction problems.

A substitution \( S \) is a solution to \( \exists \bar{s}. P \) if there is a substitution \( S' \) such that \( S(t) = S'(t) \) for all \( t \notin \bar{s} \), \( S' \sigma \preceq_{2,1} S' \tau \) for all inequalities \( (\sigma \preceq \tau) \in P \), and \( S' \sigma = S' \tau \) for all equalities \( (\sigma = \tau) \in P \). The (possibly empty) set of solutions to a problem \( \pi \) is written \( \text{Solutions}(\pi) \). Two problems \( \pi_1 \) and \( \pi_2 \) are equivalent if \( \text{Solutions}(\pi_1) = \text{Solutions}(\pi_2) \).

**Definition 29**

i) A substitution \( U \) is a most general solution to \( \pi \) if it satisfies the following conditions.

a) \( U \in \text{Solutions}(\pi) \).

b) If \( S \in \text{Solutions}(\pi) \) then \( U \preceq_{\text{FTV}(\pi)} S \).

c) \( U \) is idempotent.

d) \( \text{dom}(U) \subseteq \text{FTV}(\pi) \).

ii) We write \( \text{MGS}(\pi) \) for the (possibly empty) set of most general solutions to a \( \preceq_{2,1} \)-satisfaction problem \( \pi \).

We require the last two conditions on most general solutions for technical convenience only. We could relax the definition by eliminating those conditions; but any \( \pi \) has a solution under the relaxed definition if and only if it has a solution under our definition.
Sometimes it is useful to ensure that a most general solution does not interfere with a set of “protected” variables. For any set \( W \) of type variables, we say \( U \) is a most general solution to \( \pi \) away from \( W \) if \( U \in \text{MGS}(\pi) \) and \( W \cap \text{rng}(U) = \emptyset \), and we write \( \text{MGS}(\pi)[W] \) for the (possibly empty) set of most general solutions to \( \pi \) away from \( W \).

**Lemma 30** If \( U \in \text{MGS}(\pi)[W] \) and \( S \in \text{Solutions}(\pi) \), then \( U \leq_{\text{W} \cup \text{FTV}(\pi)} S \).

**Proof:** Since \( U \leq_{\text{FTV}(\pi)} S \), there is some \( R \) such that \( RU =_{\text{FTV}(\pi)} S \). Define

\[
R'(t) = \begin{cases} 
R(t) & \text{if } t \in \text{rng}(U), \\
S(t) & \text{otherwise}.
\end{cases}
\]

If \( t \in \text{FTV}(\pi) \), then \( R'(U(t)) = R(U(t)) = S(t) \). And if \( t \in W - \text{FTV}(\pi) \), then \( t \notin (\text{dom}(\pi) \cup \text{rng}(\pi)) \), so \( R'(U(t)) = R'(t) = S(t) \).

A unification problem is a subtype satisfaction problem involving only equalities. Algorithms for solving unification problems are well known; in particular, we have the following theorem.

**Theorem 31** Let \( \pi \) be a unification problem and \( W \) be a finite set of type variables.

i) \( \text{Solutions}(\pi) = \emptyset \) iff \( \text{MGS}(\pi) = \emptyset \) iff \( \text{MGS}(\pi)[W] = \emptyset \).

ii) There is an algorithm that decides whether \( \pi \) has a solution, and, if so, returns an element of \( \text{MGS}(\pi)[W] \).

**Proof:** See for example Snyder [31], Lemma 3.3.11.

**Theorem 32** Every \( \leq_{2,1} \)-satisfaction problem is equivalent to a unification problem, and moreover, there is an algorithm that transforms every \( \leq_{2,1} \)-satisfaction problem into an equivalent unification problem.

**Corollary 33** Let \( \pi \) be a \( \leq_{2,1} \)-satisfaction problem and \( W \) be a finite set of type variables.

i) \( \text{Solutions}(\pi) = \emptyset \) iff \( \text{MGS}(\pi) = \emptyset \) iff \( \text{MGS}(\pi)[W] = \emptyset \).

ii) There is an algorithm that decides whether \( \pi \) has a solution, and, if so, returns an element of \( \text{MGS}(\pi)[W] \).
\[(\sigma_1 \rightarrow \sigma_2) \leq \tau \quad \Rightarrow \quad \exists t_1, t_2, \{t_1 \leq \sigma_1, \sigma_2 \leq t_2, t = t_1 \rightarrow t_2\}
\text{if } t_1, t_2 \text{ are fresh}\]
\[(\sigma_1 \rightarrow \sigma_2) \leq (\tau_1 \rightarrow \tau_2) \quad \Rightarrow \quad \{\tau_1 \leq \sigma_1, \sigma_2 \leq \tau_2\}\]
\[\sigma \leq (\tau_1 \land \tau_2) \quad \Rightarrow \quad \{\sigma \leq \tau_1, \sigma \leq \tau_2\}\]
\[t \leq \tau \quad \Rightarrow \quad \{t = \tau\}\]
\text{if } \tau \text{ is a simple type}\]

Figure 6: Transformational rules for \(\leq_{2,1}\)-satisfaction problems

We will prove Theorem 32 by giving an algorithm that transforms any \(\leq_{2,1}\)-satisfaction problem into an equivalent unification problem. Corollary 33 follows by combining the transformation with any unification algorithm.

Our transformation is defined by rules of the form
\[\sigma \leq \tau \quad \Rightarrow \quad \exists \overrightarrow{t}. P.\]

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 \(\overrightarrow{t}\) of the right-hand side (but they are not the only source of variables in \(\overrightarrow{t}\)).

The rules are used to define a rewrite relation on problems:
\[
\sigma \leq \tau \quad \Rightarrow \quad \exists \overrightarrow{t}. P \quad \overrightarrow{\exists. P'} \uplus \{\sigma \leq \tau\} \quad \Rightarrow \quad \exists \overrightarrow{t}. P' \uplus P
\]

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

The rules for transforming a \(\leq_{2,1}\)-satisfaction problem into a unification problem are given in Figure 6.

Proof of Theorem 32: We show that the rules of Figure 6 constitute an algorithm for converting any \(\leq_{2,1}\)-satisfaction problem into an equivalent unification problem.

First, note that every rule transforms a \(\leq_{2,1}\)-satisfaction problem into another \(\leq_{2,1}\)-satisfaction problem (equalities are between simple types, inequalities are between \(T_2\) and \(T_1\) types).
Second, note that each rule preserves the set of solutions, so that each application of a rule transforms a problem into an equivalent problem.

Third, note that repeated application of these rules must halt: every rule reduces the number of type constructors (‘→’ or ‘∧’) in inequalities or reduces the number of inequalities.

Finally, note that a normal form contains no inequalities, and is therefore a unification problem.

\[ \text{Theorem 34} \quad \text{The subtyping relation } \preceq_{2,1} \text{ is decidable.} \]

\[ \text{Proof:} \quad \text{To see whether } \sigma \preceq_{2,1} \tau, \text{ compute } U \in \text{MGS}(\{\sigma \leq \tau\}) \text{ and check to see whether } U \text{ is the identity substitution.} \]

Decision procedures for the other subtyping relations can be obtained in a similar way.

Because we so often want to ensure that \( U \in \text{MGS}(\pi) \) is chosen “away” from a set of type variables, we adopt the following convention.

\[ \text{Convention 35} \quad \text{Whenever } U \in \text{MGS}(\pi) \text{ occurs in any mathematical context, we assume that } U \text{ is chosen so that it does not interfere with “current” type variables, that is, } U \in \text{MGS}(\pi)[W] \text{ where } W \cup \text{FTV}(\pi) \text{ is the set of type variables present in the context.} \]

### 3.2 Type inference

**Definition 36** For any term \( M \), we define the set \( \text{PP}_E(M) \) of pairs by induction:

1. If \( M = x \), then for any type variable \( t \), \( \langle \{x : t\}, t \rangle \in \text{PP}_E(x) \).
2. If \( M = \lambda x N \), and \( \langle A, \sigma \rangle \in \text{PP}_E(N) \), then:
   a. If \( x \notin \text{dom}(A) \), and \( t \) is a type variable not appearing in \( \langle A, \sigma \rangle \), then \( \langle A, t \rightarrow \sigma \rangle \in \text{PP}_E(\lambda x N) \).
   b. If \( x \in \text{dom}(A) \), then \( \langle A_x, A(x) \rightarrow \sigma \rangle \in \text{PP}_E(\lambda x N) \).
3. If \( M = M_1 M_2 \), then:
   a. If \( \langle A_1, t \rangle \in \text{PP}_E(M_1) \) and \( \langle A_2, \sigma_2 \rangle \in \text{PP}_E(M_2) \) are disjoint, and \( U \in \text{MGS}(\{ t = t_1 \rightarrow t_2, \sigma_2 \leq t_1 \}) \) where \( t_1, t_2 \) are fresh, then
      \[ U \langle A_1 + A_2, t_2 \rangle \in \text{PP}_E(M_1 M_2) \].
b) If \( \langle A_i, (\bigwedge_{i \in I} \sigma_i) \rightarrow \sigma_1 \rangle \in \text{PP}_I(M_1) \), and \( \langle A_i, \tau_i \rangle \in \text{PP}_I(M_2) \) for all \( i \in I \), where all pairs are chosen disjoint, and \( U \in \text{MGS}(\{\tau_i \leq \sigma_1 \mid i \in I\}) \), then
\[
U \langle A_1 + \sum_{i \in I} A_i, \sigma_1 \rangle \in \text{PP}_I(M_1 M_2).
\]

The following lemma establishes that the elements of \( \text{PP}_I(M) \) are just trivial variants of each other. Therefore, the requirement of disjointness used in the definition of \( \text{PP}_I \) is easily satisfied, and Definition 36 can be adapted to a type inference algorithm.

Lemma 37

i) If \( \langle A, \sigma \rangle \in \text{PP}_I(M) \), then \( x \in \text{dom}(A) \) if and only if \( x \) is free in \( M \).

ii) Suppose \( \langle A_1, \sigma_1 \rangle \in \text{PP}_I(M) \). Then \( \langle A_2, \sigma_2 \rangle \in \text{PP}_I(M) \) if and only if there is a bijection \( R \) of type variables such that \( R\langle A_1, \sigma_1 \rangle = \langle A_2, \sigma_2 \rangle \).

Proof: An easy induction on Definition 36.

\[ \square \]

Theorem 38 There is an algorithm that decides, for any \( M \), whether the set \( \text{PP}_I(M) \) is empty; and furthermore, if \( \text{PP}_I(M) \) is not empty, it produces a member of \( \text{PP}_I(M) \).

Proof: Just follow the rules of Definition 36, generating "fresh" type variables as necessary, and use the algorithm of Corollary 33 to compute \( \text{MGS} \).

\[ \square \]

Example 39 We show how the algorithm finds the type of \( (\lambda x.xxx) \).

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

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

iii) To calculate \( \text{PP}_I(xx) \), 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 \( \langle \{x : t_3 \land (t_3 \rightarrow t_4)\}, t_4 \rangle \in \text{PP}_I(xx) \).

iv) Finally, \( \text{PP}_I(\lambda x.xxx) \) produces \( \langle \emptyset, \forall t_3, t_4, t_3 \land (t_3 \rightarrow t_4) \rightarrow t_4 \rangle \).

We now establish the soundness of \( \text{PP}_I \).
Theorem 40 If \( \langle A, \sigma \rangle \in \text{PP}_{E}(M) \), then \( \langle A, \sigma \rangle \in \text{AP}_{I}(M) \).

Proof: By induction on the definition of \( \text{PP}_{E}(M) \).

i) If \( M = x \), then \( \langle A, \sigma \rangle = \langle \{x : t\}, t \rangle \), and we have \( \langle A, \sigma \rangle \in \text{AP}_{I}(x) \) by rule \( \text{(VAR)} \).

ii) If \( M = \lambda xN \), then by Lemma 37(i) we have the following two cases:

a) \( x \) is not free in \( N \), and \( \sigma = t \rightarrow \sigma' \), where \( \langle A, \sigma' \rangle \in \text{PP}_{I}(N) \).

By induction and weakening, \( \langle A \cup \{x : t\}, \sigma' \rangle \in \text{AP}_{I}(N) \) (note that \( A \cup \{x : t\} \) is well-formed by Lemma 37(i)).

So by rule \( \text{(ABS)} \), \( \langle A, t \rightarrow \sigma' \rangle = \langle A, \sigma \rangle \in \text{AP}_{I}(\lambda xN) \).

b) \( x \) is free in \( N \) and \( \langle A, \sigma \rangle = \langle A', \lambda'(x) \rightarrow \sigma' \rangle \), where \( \langle A', \sigma' \rangle \in \text{PP}_{I}(N) \).

By induction \( \langle A', \sigma' \rangle \in \text{AP}_{I}(N) \), so \( \langle A, \sigma \rangle \in \text{AP}_{I}(\lambda xN) \) by rule \( \text{(ABS)} \).

iii) If \( M = M_1 M_2 \), then one of the following cases holds:

a) \( \langle A, \sigma \rangle = U(A_1 + A_2, t_1) \), where \( \langle A_1, t_1 \rangle \in \text{PP}_{I}(M_1) \), \( \langle A_2, \sigma_2 \rangle \in \text{PP}_{I}(M_2) \), and \( U \in \mathbf{MGS}\{t = t_1 \rightarrow t_2, \sigma_2 \leq \sigma'_1\} \).

Then by induction, weakening, and substitution,

\[
U\langle A_1 + A_2, t \rangle \in \text{AP}_{I}(M_1), \\
U\langle A_2 + A_2, \sigma_2 \rangle \in \text{AP}_{I}(M_2).
\]

Since \( U \sigma_2 \leq U t_1 \), by Lemma 2(ii) we have \( U \sigma_2 = U t_1 \). And \( U t = (U t_1) \rightarrow (U t_2) \), so by rule \( \text{(APP)} \) we have \( U \langle A_1 + A_2, t_1 \rangle \in \text{AP}_{I}(M) \).

b) \( \langle A, \sigma \rangle = U(A_1 + \Sigma_{i \in I} A_i, \sigma_1) \), where \( \langle A_i, \tau_i \rangle \in \text{PP}_{E}(M_2) \) for all \( i \in I \), \( \langle A_1, (\Lambda_{i \in I} \sigma_i) \rightarrow \sigma_1 \rangle \in \text{PP}_{E}(M_1) \), and \( U \in \mathbf{MGS}(\{\tau_i \leq \sigma_i \mid i \in I\}) \).

Then by induction, weakening, and substitution,

\[
U\langle A_1 + \Sigma_{i \in I} A_i, (\Lambda_{i \in I} \sigma_i) \rightarrow \sigma_1 \rangle \in \text{AP}_{E}(M_1), \\
U\langle A_1 + \Sigma_{i \in I} A_i, \tau_i \rangle \in \text{AP}_{E}(M_2) \quad (\forall i \in I).
\]

By Lemma 2(ii) and the fact that \( U \tau_i \leq U \sigma_i \), we have \( U \tau_i = U \sigma_i \). Then by rule \( \text{(APP)} \) we have \( U \langle A_1 + \Sigma_{i \in I} A_i, \sigma_1 \rangle \in \text{AP}_{E}(M) \).

\( \square \)
Theorem 41 (Principal pairs for $\Pi_2$) If $\langle A, \sigma \rangle \in \text{AP}_{\Pi_2}(M)$, then there is a pair $\langle A', \sigma' \rangle \in \text{PP}_{\Pi_2}(M)$ and a substitution $S$ such that $S(A', \sigma') \leq \langle A, \sigma \rangle$.

Proof: By cases on the structure of $M$.

i) If $M = x$, then $\langle A, \sigma \rangle \in \text{AP}_{\Pi_2}(M)$ by rule (VAR), and therefore, $A(x) = (\bigwedge_{i \in I} \sigma_i)$ and $\sigma = \sigma_{i_0} \in T_0$ for some $i_0 \in I$.

For any $t$, $\langle \{ x : t \}, t \rangle \in \text{PP}_{\Pi_2}(M)$. Then $\{ t := \sigma \}$ is a well-formed substitution and

$$\{ t := \sigma \}\langle \{ x : t \}, t \rangle = \langle \{ x : \sigma \}, \sigma \rangle \leq \langle A, \sigma \rangle.$$

ii) If $M = \lambda x. N$, then by the definition of $\Pi_2$, $\sigma$ must be of the form $\sigma_1 \rightarrow \sigma_2$, and $\langle A_x \cup \{ x : \sigma_1 \}, \sigma_2 \rangle \in \text{AP}_{\Pi_2}(N)$. By induction, there is a substitution $S$ and pair $\langle A', \sigma'_2 \rangle \in \text{PP}_{\Pi_2}(N)$ such that

$$S(A', \sigma'_2) \leq \langle A_x \cup \{ x : \sigma_1 \}, \sigma_2 \rangle. \quad (1)$$

We consider two cases.

a) If $x \notin \text{dom}(A')$, then for any fresh type variable $t$, $\langle A', t \rightarrow \sigma'_2 \rangle \in \text{PP}_{\Pi_2}(\lambda x. N)$.

Note that $\sigma_1$ is of the form $\bigwedge_{i \in I} \sigma_i$, and therefore, we can pick $\sigma'_1 \in T_0$ such that $\sigma_1 \leq 1 \sigma'_1$ (choose any $\sigma_i$). Then let $S' = \{ t := \sigma'_1 \} \cup S$. By (1) and the definition of $\leq$,

$$S'(A', t \rightarrow \sigma'_2) = \langle S A', \sigma'_1 \rightarrow S \sigma'_2 \rangle \leq \langle A_x, \sigma_1 \rightarrow \sigma_2 \rangle.$$

Since $A \leq 1 A_x$, we have $S'(A', t \rightarrow \sigma'_2) \leq \langle A, \sigma_1 \rightarrow \sigma_2 \rangle$, as desired.

b) If $x \in \text{dom}(A')$, then $\langle A'_x, A'(x) \rightarrow \sigma'_2 \rangle \in \text{PP}_{\Pi_2}(\lambda x. N)$. Then by (1) and the definition of $\leq$,

$$S(A'_x, A'(x) \rightarrow \sigma'_2) \leq \langle A_x, \sigma_1 \rightarrow \sigma_2 \rangle,$$

and since $A \leq 1 A_x$, we have $S(A'_x, A'(x) \rightarrow \sigma'_2) \leq \langle A, \sigma_1 \rightarrow \sigma_2 \rangle$, as desired.

iii) If $M = M_1 M_2$, then by the definition of $\Pi_2$, $\langle A, (\bigwedge_{i \in I} \sigma_i) \rightarrow \sigma \rangle \in \text{AP}_{\Pi_2}(M_1)$ and $\langle A, \sigma_i \rangle \in \text{AP}_{\Pi_2}(M_2)$ for all $i \in I$.

By induction, $\text{PP}_{\Pi_2}(M_1)$ is nonempty, and by Lemma 37(ii), it is sufficient to consider the following cases on the structure of pairs in $\text{PP}_{\Pi_2}(M_1)$.
a) \( \langle A_1, t \rangle \in \PP_{E}(M_1) \). By induction, there is a substitution \( S_1 \) such that
\[
S_1 \langle A_1, t \rangle \leq \langle A, (\bigwedge_{i \in I} \sigma_i) \rightarrow \sigma \rangle.
\]
By the definition of \( \leq_v \), \( S_1 t = \sigma_i \rightarrow \sigma' \) for some \( i \in I \) and \( \sigma' \in T_0 \).
Then by induction and Lemma 37(ii), there is a disjoint pair \( \langle A_2, \tau \rangle \in \PP_{E}(M_2) \) and substitution \( S_2 \) such that
\[
S_2 \langle A_2, \tau \rangle \leq \langle A, \sigma_i \rangle.
\]
Let \( \sigma = \{ t = t_1 \rightarrow t_2, \tau \leq t_1 \} \), where \( t_1, t_2 \) are fresh. Then \( S = S_1 \cup S_2 \cup \{ t_1 := \sigma_i, t_2 := \sigma' \} \) is a solution to \( \sigma \).
Pick \( U \in \text{MGS}(\sigma) \). Then \( U \langle A_1 + A_2, t_2 \rangle \in \PP_{E}(M_1 M_2) \).
By Convention 35, there exists an \( R \) such that \( RU \langle A_1 + A_2, t_2 \rangle = S \langle A_1 + A_2, t_2 \rangle \).
And
\[
S \langle A_1 + A_2, t_2 \rangle = \langle S_1 A_1 + S_2 A_2, \sigma' \rangle \leq \langle A, \sigma \rangle,
\]
as desired.

b) \( \langle A_1, (\bigwedge_{j \in J} \sigma'_j) \rightarrow \sigma' \rangle \in \PP_{E}(M_1) \).
By induction there is a substitution \( S_1 \) such that
\[
S_1 \langle A_1, (\bigwedge_{j \in J} \sigma'_j) \rightarrow \sigma' \rangle \leq \langle A, (\bigwedge_{i \in I} \sigma_i) \rightarrow \sigma \rangle.
\]
By the definition of \( \leq_v \), \( \{ S_1 \sigma'_j \mid j \in J \} \subseteq \{ \sigma_i \mid i \in I \} \), so without loss of generality we assume \( J \subseteq I \) and \( S_1 \sigma'_j = \sigma_j \) for all \( j \in J \).
By induction and Lemma 37(ii), for all \( j \in J \) there are disjoint pairs \( \langle A_j, \rho_j \rangle \in \PP_{E}(M_2) \) and substitutions \( S_j \) such that
\[
S_j \langle A_j, \rho_j \rangle \leq \langle A, \sigma_j \rangle.
\]
Let \( \sigma = \{ \rho_j \leq \sigma'_j \mid j \in J \} \). Then \( S = S_1 \cup (\bigcup_{j \in J} S_j) \) is a solution to \( \sigma \): \( S \rho_j = S_j \rho_j \leq \sigma_j = S_1 \sigma'_j = S \sigma'_j \).
Pick \( U \in \text{MGS}(\sigma) \). Then
\[
U \langle A_1 + \sum_{j \in J} A_j, \sigma' \rangle \in \PP_{E}(M_1 M_2).
\]
By Convention 35, there exists an \( R \) such that
\[
RU \langle A_1 + \sum_{j \in J} A_j, \sigma' \rangle = S \langle A_1 + \sum_{j \in J} A_j, \sigma' \rangle.
\]
And
\[ S(A_1 + \Sigma_{j \in J} A_j, \sigma') = (S_1 A_1 + \Sigma_{j \in J} S_j A_j, S_1 \sigma') \leq (A, \sigma), \]
as desired.

\[ \square \]

4 Other systems of rank 2 intersection types

4.1 A restriction of \( \text{I}_2 \)

Van Bakel [33] defined a rank 2 intersection type system that is a slight restriction of our system \( \text{I}_2 \). A version of his rules is presented below.

\[(\text{VAR}) \quad \{x : \tau\} \vdash x : \tau \quad \text{(where } \tau \in T_0)\]

\[(\text{ABS}) \quad \frac{A_x \cup \{x : \tau_1\} \vdash M : \tau_2}{A_x \vdash (\lambda x.M) : \tau_1 \rightarrow \tau_2} \]

\[(\text{APP}) \quad \frac{A \vdash M : (\bigwedge_{i \in I} \tau_i) \rightarrow \tau, \quad (\forall i \in I) A_i \vdash N : \tau_i}{A + \Sigma_{i \in I} A_i \vdash (MN) : \tau} \]

We write \( \text{I}_2^b \vdash A \vdash M : \sigma \) if the judgment \( A \vdash M : \sigma \) follows by these rules, under the following restrictions: environment types are in \( T_1 \); derived types are in \( T_2 \); and in every judgment \( A \vdash M : \tau \), the type environment \( A \) contains only assumptions actually used in the derivation of \( A \vdash M : \tau \). For example, the rule (VAR) has been intentionally restricted to rule out a judgment such as

\[ \{x : \sigma_1 \wedge \sigma_2\} \vdash x : \sigma_1, \]
in which the type \( \sigma_2 \) assumed for \( x \) is not used. Similarly, \( \{x : \sigma_1, y : \sigma_2\} \vdash x : \sigma_1 \) is not derivable because the assumption \( y : \sigma_2 \) is not used. The exact relation between \( \text{I}_2^b \) and \( \text{I}_2 \) is summed up in the following lemma.

Lemma 42 (Comparison of \( \text{I}_2^b \) and \( \text{I}_2 \))

i) If \( \text{I}_2^b \vdash A \vdash M : \sigma \), then \( \text{I}_2 \vdash A \vdash M : \sigma \). The converse does not hold.

ii) A term \( M \) is typable in \( \text{I}_2^b \) if and only if it is typable in \( \text{I}_2 \).

Proof:

i) Just note that the \( \text{I}_2^b \) rule (VAR) is a special case of the \( \text{I}_2 \) rule (VAR), that the \( \text{I}_2^b \) rule (ABS) is identical to the \( \text{I}_2 \) rule (ABS), and that the \( \text{I}_2^b \) rule (APP) follows from the \( \text{I}_2 \) rule (ABS) and weakening.
The examples above show that the converse does not hold.

ii) This follows because the definition of principal pair in van Bakel’s system is identical to our own.

4.2 An extension of \( I_2 \)

A natural extension of \( I_2 \) is obtained by adding the following rule to the rules of \( I_2 \):

\[
(\text{SUB}) \quad \frac{A \vdash M : \tau}{A \vdash M : \sigma} \quad \text{(where } \tau \leq \sigma)\]

We write \( I_2 \vdash A \vdash M : \sigma \) if the judgment \( A \vdash M : \sigma \) follows by the rules of \( I_2 \) plus \( (\text{SUB}) \), with types appearing in type environments restricted to \( T_1 \), and derived types restricted to \( T_2 \).

Clearly, every judgment of \( I_2 \) is a judgment of \( I_2 \). The converse does not hold; for example, the judgment

\[
\{x : \sigma \rightarrow \tau\} \vdash x : (\sigma \land \sigma') \rightarrow \tau
\]

is derivable in \( I_2 \) for any \( \sigma \neq \sigma' \in T_0 \), but is not derivable in \( I_2 \).

\( I_2 \) has principal pairs, and indeed, they are identical to the principal pairs of \( I_2 \) (the proof is a simple extension of the proof of Theorem 4.1). An immediate consequence is that the terms typable in \( I_2 \) are exactly the same as the terms typable in \( I_2 \).

In summary:

**Lemma 43 (Comparison of \( I_2 \) and \( I_2 \))**

i) If \( I_2 \vdash A \vdash M : \sigma \), then \( I_2 \vdash A \vdash M : \sigma \). The converse does not hold.

ii) A term \( M \) is typable in \( I_2 \) if and only if it is typable in \( I_2 \).

Although it does not type any more terms than \( I_2 \), \( I_2 \) has other advantages over \( I_2 \).

**Example 44** The acceptable pairs of \( I_2 \) are not closed under the operation \( \leq \):

\[
I_2 \vdash \{x : s \rightarrow t\} \vdash x : s \rightarrow t,
\]

and

\[
\langle \{x : s \rightarrow t\}, s \rightarrow t \rangle \leq \langle \{x : s \rightarrow t\}, (s \land t) \rightarrow t \rangle,
\]

31
but the judgment

\[ \{ x : s \to t \} \vdash x : (s \land t) \to t \]

is not derivable in \( I_2 \).

On the other hand, \( I_2 \) is closed under \( \leq \):

**Lemma 45 (Weakening for \( I_2 \))** If \( I_2 \vdash A \vdash M : \sigma \) and \( \langle A, \sigma \rangle \leq \langle A', \sigma' \rangle \), then \( I_2 \vdash A' \vdash M : \sigma' \).

For this reason, we prefer \( I_2 \) to either \( I_{2b} \) or \( I_3 \). However, it was still useful to develop \( I_3 \). In particular, the example above shows that Lemma 19 does not hold for \( I_2 \); it was convenient to have Lemma 19 for the proof of the equivalence of typability with \( \Lambda_2 \).

# 5 Combining intersections and quantification

## 5.1 The system \( P_2 \)

We now describe a type system that combines aspects of rank 2 intersection types and rank 2 polymorphic types. The system is called \( P_2 \), as it is the rank 2 subset of a type system \( P \) (described elsewhere).

The types of the system are the rank 2 intersection types extended with top-level quantifiers:

\[ T_{v_2} = T_2 \cup \{ (\forall t \sigma) \mid \sigma \in T_{v_2} \} \]

In order to simplify the definition of subtyping, we consider \( T_{v_2} \) types syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers. When a \( T_{v_2} \) type is written in the form \( \forall \bar{s} \sigma \), we assume \( \sigma \in T_2 \).

**Definition 46**

i) The relation \( \leq_{v_2} \) is the least partial order on \( T_{v_2} \) closed under the following rules:

a) If \( \sigma \leq_2 \tau \), then \( \sigma \leq_{v_2} \tau \).

b) If \( \tau \in T_0 \), then \( (\forall t \sigma) \leq_{v_2} \{ t := \tau \} \sigma \).

c) If \( \sigma \leq_{v_2} \tau \) and \( t \) is not free in \( \sigma \), then \( \sigma \leq_{v_2} (\forall t \tau) \).

ii) The relation \( \leq_{v_2,1} \) between \( T_{v_2} \) and \( T_1 \) is the least relation closed under the rule:
a) If $\sigma \le_{\forall \tau} \tau_i$ for all $i \in I$, then $\sigma \le_{\forall 2} (\land_{i \in I} \tau_i)$.

The rules for $\le_{\forall 2}$ express the intuition that a type is a subtype of its instances. They are equivalent to the following rule, similar to ML’s notion of *generic instance*:

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

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., $(\forall t.I)$ is not a least type in the ordering $\le_{\forall 2}$: $(\forall t.I) \not\le_{\forall 2} (s_1 \land (s_1 \to s_2)) \to s_2$.

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

$$(\forall t \tau) \le_{\forall 2,1} (\bigwedge_{i \in I} \sigma_i),$$

the variable $t$ may be instantiated differently for each $\sigma_i$.

Some basic properties of $\le_{\forall 2}$ and $\le_{\forall 2,1}$ are summarized in the following lemma.

**Lemma 47**

i) If $\sigma, \tau \in T_0$, then $\sigma \le_{\forall 2} \tau$ iff $\sigma \le_{\forall 2,1} \tau$ iff $\sigma = \tau$.

ii) If $\sigma, \tau \in T_2$, then $\sigma \le_{\forall 2} \tau$ iff $\sigma \le_{\forall 2} \tau$.

iii) If $\sigma \le_{\forall 2} \tau$, then $(\forall t \sigma) \le_{\forall 2} (\forall t \tau)$.

iv) If $\sigma \in T_2$ and $\tau \in T_0$, then $\forall \vec{t} \sigma \le_{\forall 2} \tau$ iff for some substitution $S$ with $\text{dom}(S) \subseteq \vec{t}$, we have $S \sigma = \tau$.

v) For any substitution $S$ and types $\sigma, \tau \in T_{\forall 2}$, if $S \sigma \le_{\forall 2} \tau$, then $S(\forall t \sigma) \le_{\forall 2} \tau$.

vi) For any substitution $S$, types $\sigma, \tau \in T_{\forall 2}$, and type environment $A$, if $S \sigma \le_{\forall 2} \tau$, then $S(\text{Gen}(A, \sigma)) \le_{\forall 2} \tau$.

vii) If $\sigma_1 \le_{\forall 2} \sigma_2 \le_{\forall 2,1} \sigma_3 \le_{\forall 2,1} \sigma_4$, then $\sigma_1 \le_{\forall 2,1} \sigma_4$.

The typing rules of the system are presented in Figure 7. We write $P_2 \triangleright A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ follows by these rules, with types appearing in type environments restricted to $T_1$, and derived types
restricted to $T_{\forall 2}$. Note that the familiar rules $(\text{inst})$ and $(\text{gen})$ are special cases of the rule $(\text{sub})$:

\[
\frac{A \vdash M : \forall t \sigma}{A \vdash M : \{t := \tau\} \sigma} \quad \text{(where $\tau \in T_0$)}
\]

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

The ordering $\leq$ of Definition 28 is extended to pairs with $T_{\forall 2}$ types as follows:

\[
\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle \text{ if and only if } A' \leq_1 A \text{ and } \sigma \leq_{\forall 2} \sigma'.
\]

**Lemma 48 (Weakening for $P_2$)** If $P_2 \vdash A \vdash M : \sigma$ and $\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle$, then $P_2 \vdash A' \vdash M : \sigma'$.

**Lemma 49 (Substitutivity for $P_2$)** If $P_2 \vdash A \vdash M : \sigma$, then $P_2 \vdash SA \vdash M : S\sigma$ for any substitution $S$.

### 5.2 Extending subtype satisfaction

In order to perform type inference for $P_2$, we will need to solve problems that generalize the $\leq_{21}$-satisfaction problems of §3.1.

A $\leq_{\forall 21}$-satisfaction problem $\pi$ is a pair $\exists \delta P$, where $P$ is a finite set whose every element is either: 1) an equality between simple types; or 2)
an inequality between a $T_{v_2}$ type and a $T_1$ type. A substitution $S$ is a solution to $P$ if there is a substitution $S'$ such that $S(t) = S'(t)$ for all $t \notin S$, $S'\sigma \leq_{v_2,1} S'\tau$ for all inequalities $(\sigma \leq \tau) \in P$, and $S'\sigma = S'\tau$ for all equalities $(\sigma = \tau) \in P$.

Note that any $\leq_{2,1}$-satisfaction problem is a $\leq_{v_{2,1}}$-satisfaction problem with the same set of solutions. Therefore we abuse notation and write $\text{Solutions}(\pi)$, $\text{MGS}(\pi)$, and $\text{MGS}(\pi)[W]$ for the solutions, most general solutions, and most general solutions away from $W$ of a $\leq_{v_{2,1}}$-satisfaction problem $\pi$.

Similarly, $\leq_{v_{2,1}}$-satisfaction problems can be solved by extending the transformational algorithm of Figure 6 by the following rule:

$$(\forall t \sigma \leq \tau \Rightarrow \exists t \{ \sigma \leq \tau \}$$

if $\tau$ is not an $\land$-type, and $t$ is not free in $\tau$

**Theorem 50** Every $\leq_{v_{2,1}}$-satisfaction problem is equivalent to a unification problem, and moreover, there is an algorithm that transforms every $\leq_{v_{2,1}}$-satisfaction problem into an equivalent unification problem.

**Proof:** We show that the rules of Figure 6, augmented by the rule above, constitute an algorithm for converting any $\leq_{v_{2,1}}$-satisfaction problem into an equivalent unification problem (equalities are between simple types, inequalities are between $T_{v_2}$ and $T_1$ types).

First, note that every rule transforms a $\leq_{v_{2,1}}$-satisfaction problem into another $\leq_{v_{2,1}}$-satisfaction problem.

Second, note that each rule preserves the set of solutions, so that each application of a rule transforms a problem into an equivalent problem.

Third, note that repeated application of these rules must halt: every rule reduces the number of type constructors (‘$\rightarrow$’, ‘$\land$’, or ‘$\forall$’) in inequalities or reduces the number of inequalities.

Finally, note that a normal form contains no inequalities, and is therefore a unification problem.

**Corollary 51** Let $\pi$ be a $\leq_{v_{2,1}}$-satisfaction problem and $W$ be a finite set of type variables.

i) $\text{Solutions}(\pi) = \emptyset$ iff $\text{MGS}(\pi) = \emptyset$ iff $\text{MGS}(\pi)[W] = \emptyset$.

ii) There is an algorithm that decides whether $\pi$ has a solution, and, if so, returns an element of $\text{MGS}(\pi)[W]$.

**Theorem 52** The subtyping relation $\leq_{v_{2,1}}$ is decidable.
Proof: To see whether $\sigma \leq_{\forall, 1} \tau$, compute $U \in \text{MGS}(\{\sigma \leq \tau\})$ and check to see whether $U$ is the identity substitution. \hfill \Box

5.3 Type inference for $P_2$

Definition 53 For any term $M$, we define the set $\text{PP}_{P_2}(M)$ by induction on $M$.

i) If $M = x$, then for any type variable $t$, $\langle \{x : t\}, t \rangle \in \text{PP}_{P_2}(x)$.

ii) If $M = \lambda x.N$, and $\langle A, \forall \overline{s} \sigma \rangle \in \text{PP}_{P_2}(N)$, where the type variables $\overline{s}$ are distinct from all other type variables, then:
   a) If $x \notin \text{dom}(A)$, and $t$ is a type variable not appearing in $\langle A, \forall \overline{s} \sigma \rangle$, then $\langle A, \forall \overline{s} (t \rightarrow \sigma) \rangle \in \text{PP}_{P_2}(\lambda x N)$.
   b) If $x \in \text{dom}(A)$, then $\langle A \angle A x A \rightarrow \sigma \rangle \in \text{PP}_{P_2}(\lambda x N)$.

iii) If $M = M_1 M_2$, the pairs $\langle A_1, \forall \overline{s} \sigma_1 \rangle \in \text{PP}_{P_2}(M_1)$ and $\langle A_2, \sigma_2 \rangle \in \text{PP}_{P_2}(M_2)$ are disjoint, and the type variables $\overline{s}$ are distinct from all other type variables, then:
   a) If $\sigma_1$ is a type variable $t$, $t_1$ and $t_2$ are fresh type variables, $U \in \text{MGS}(\{\sigma_2 \leq t_1, t = t_1 \rightarrow t_2\})$, and $A = U(A_1 + A_2)$, then
      \[ \langle A, \text{Gen}(A, U t_2) \rangle \in \text{PP}_{P_2}(M) \]
   b) If $\sigma_1 = \tau_1 \rightarrow \tau_2$, $U \in \text{MGS}(\{\sigma_2 \leq \tau_1\})$, and $A = U(A_1 + A_2)$, then
      \[ \langle A, \text{Gen}(A, U \tau_2) \rangle \in \text{PP}_{P_2}(M) \]

Just as with $P_1$, the elements of $\text{PP}_{P_2}(M)$ are trivial variants of each other, so Definition 53 can easily be adapted to a type inference algorithm.

Lemma 54

i) If $\langle A, \sigma \rangle \in \text{PP}_{P_2}(M)$, then $x \in \text{dom}(A)$ if and only if $x$ is free in $M$.

ii) Suppose $\langle A_1, \sigma_1 \rangle \in \text{PP}_{P_2}(M)$. Then $\langle A_2, \sigma_2 \rangle \in \text{PP}_{P_2}(M)$ if and only if there is a bijection $R$ of type variables such that $R(\langle A_1, \sigma_1 \rangle) = \langle A_2, \sigma_2 \rangle$.

Proof: An easy induction on Definition 53. \hfill \Box

Theorem 55 There is an algorithm that decides, for any $M$, whether the set $\text{PP}_{P_2}(M)$ is empty; and furthermore, if $\text{PP}_{P_2}(M)$ is not empty, it produces a member of $\text{PP}_{P_2}(M)$. 36
Proof: Just follow the rules of Definition 53, generating “fresh” type variables as necessary, and use the algorithm of Corollary 51 to compute MGS.

We now establish the soundness of $\text{PP}_{2}$.

**Theorem 56** If $\langle A, \sigma \rangle \in \text{PP}_{2}(M)$, then $\langle A, \sigma \rangle \in \text{AP}_{2}(M)$.

**Proof:** By induction on the definition of $\text{PP}_{2}(M)$.

i) If $M = x$, then $\langle A, \sigma \rangle = \langle \{ x : t \}, t \rangle$ for some type variable $t$.

Then we have $\langle A, \sigma \rangle \in \text{AP}_{2}(x)$ by rule (VAR).

ii) If $M = \lambda x N$, then by Lemma 54(i) we have the following two cases:

   a) $x$ is not free in $N$, $\langle A, \forall \bar{s}\sigma' \rangle \in \text{PP}_{2}(N)$ for some $\sigma'$, and $\sigma = \forall t \bar{s}(t \rightarrow \sigma')$ for some fresh type variable $t$.

      By induction, $\langle A, \forall \bar{s}\sigma' \rangle \in \text{AP}_{2}(N)$, and by weakening,

      $$\langle A \cup \{ x : t \}, \sigma' \rangle \in \text{AP}_{2}(N)$$

      (note that $A \cup \{ x : t \}$ is well-formed by Lemma 54(i)).

      So by rule (ABS), $\langle A, t \rightarrow \sigma' \rangle \in \text{AP}_{2}(\lambda x N)$, and by then by rule (SUB),

      $$\langle A, \forall t \bar{s}(t \rightarrow \sigma') \rangle = \langle A, \sigma \rangle \in \text{AP}_{2}(\lambda x N).$$

   b) $x$ is free in $N$ and $\langle A, \sigma \rangle = \langle A', \text{Gen}(A'_{x}, A'(x) \rightarrow \sigma') \rangle$, where $\langle A', \forall \bar{s}\sigma' \rangle \in \text{PP}_{2}(N)$.

      By induction and rule (SUB), $\langle A', \sigma' \rangle \in \text{AP}_{2}(N)$, so by rule (ABS),

      $$\langle A', A'(x) \rightarrow \sigma' \rangle \in \text{AP}_{2}(\lambda x N).$$

      Then by rule (SUB),

      $$\langle A'_{x}, \text{Gen}(A'_{x}, A'(x) \rightarrow \sigma') \rangle = \langle A, \sigma \rangle \in \text{AP}_{2}(\lambda x N).$$

iii) If $M = M_{1}M_{2}$, then we have disjoint pairs $\langle A_{1}, \forall \bar{s}\sigma_{1} \rangle \in \text{PP}_{2}(M_{1})$ and $\langle A_{2}, \sigma_{2} \rangle \in \text{PP}_{2}(M_{2})$. By induction, $\langle A_{1}, \forall \bar{s}\sigma_{1} \rangle \in \text{AP}_{2}(M_{1})$ and $\langle A_{2}, \sigma_{2} \rangle \in \text{AP}_{2}(M_{2})$. By rule (SUB), we have $\langle A_{1}, \sigma_{1} \rangle \in \text{AP}_{2}(M_{1})$.

   We now consider two cases.

   a) If $\sigma_{1}$ is a type variable $t$, then we must have $A = U(A_{1} + A_{2})$ and $\sigma = \text{Gen}(A, Ut_{2})$, where $U \in \text{MGS}(\{ \sigma_{2} \leq t_{1}, t_{1} \rightarrow t_{2} \})$ for fresh type variables $t_{1}$ and $t_{2}$.
By substitutivity we have
\[ U\langle A_1, \sigma_1 \rangle = \langle U A_1, (Ut_1) \rightarrow (Ut_2) \rangle \in \text{AP}_2(M_1) \]
and
\[ U\langle A_2, \sigma_2 \rangle = \langle U A_2, U \sigma_2 \rangle \in \text{AP}_2(M_2). \]

By weakening,
\[ \langle U A_1 + U A_2, (Ut_1) \rightarrow (Ut_2) \rangle \in \text{AP}_2(M_1) \]
and
\[ \langle U A_1 + U A_2, U \sigma_2 \rangle \in \text{AP}_2(M_2). \]

Then since \( U \sigma_2 \leq \forall \sigma_1 \, Ut_1 \), by rule (APP) we have
\[ \langle U A_1 + U A_2, U t_2 \rangle = \langle A, Ut_2 \rangle \in \text{AP}_2(M_1 M_2). \]

Then by rule (SUB),
\[ \langle A, \text{Gen}(A, Ut_2) \rangle = \langle A, \sigma \rangle \in \text{AP}_2(M_1 M_2). \]

b) The case \( \sigma_1 = \tau_1 \rightarrow \tau_2 \) is almost identical to the last.

\[ \square \]

**Theorem 57 (Principal pairs for P_2)** If \( \langle A, \sigma \rangle \in \text{AP}_2(M) \), then there is a pair \( \langle A', \sigma' \rangle \in \text{PP}_2(M) \) and a substitution \( S \) such that \( S\langle A', \sigma' \rangle \leq \langle A, \sigma \rangle \).

**Proof**: By induction on the definition of \( \text{AP}_2(M) \).

i) If \( \langle A, \sigma \rangle \in \text{AP}_2(M) \) by rule (VAR), then \( M = x \) for some variable \( x \), \( A(x) = (\land_{i \in I} \sigma_i) \), and \( \sigma = \sigma_{i_0} \in T_0 \) for some \( i_0 \in I \).

By the definition of \( \text{PP}_2 \), \( \langle \{ x : t \}, t \rangle \in \text{PP}_2(M) \), where \( t \) is a fresh type variable.

Then \( \{ t := \sigma \} \) is a well-formed substitution and
\[ \{ t := \sigma \}\{ x : t \}, t \rangle = \langle \{ x : \sigma \}, \sigma \rangle \leq \langle A, \sigma \rangle \. \]

ii) If \( \langle A, \sigma \rangle \in \text{AP}_2(M) \) by rule (ABS), then \( M = \lambda x N \), \( \sigma \) is of the form \( \sigma_1 \rightarrow \sigma_2 \), and \( \langle A_x \cup \{ x : \sigma_1 \}, \sigma_2 \rangle \in \text{AP}_2(N) \).

By induction, there is a substitution \( S' \) and pair \( \langle A', \forall \sigma' \rangle \in \text{PP}_2(N) \) such that
\[ S'\langle A', \forall \sigma' \rangle \leq \langle A_x \cup \{ x : \sigma_1 \}, \sigma_2 \rangle. \]

We consider two cases.
a) If \( x \not\in \text{dom}(A') \), then for any fresh type variable \( t \),
\[
\langle A', \forall t \bar{s}(t \rightarrow \sigma'_2) \rangle \in \text{PP}_2(\lambda x N).
\]

It remains to show that there is a substitution \( S \) such that
\[
S\langle A', \forall t \bar{s}(t \rightarrow \sigma'_2) \rangle \leq \langle A, \sigma \rangle.
\]

Just let \( S = S' \). By (2), we have \( A \leq_1 A_x \leq_1 S'A' \), so we only need show
\[
S'(\forall t \bar{s}(t \rightarrow \sigma'_2)) \leq \forall_2 \sigma_1 \rightarrow \sigma_2.
\]

We can assume \( t, \bar{s} \) are fresh, so that
\[
S'(\forall t \bar{s}(t \rightarrow \sigma'_2)) = \forall t \bar{s}(t \rightarrow S'\sigma'_2).
\]

And by (2),
\[
\{ t := \sigma_1 \} \forall \bar{s}(t \rightarrow \sigma'_2) = \forall \bar{s}(\sigma_1 \rightarrow \sigma'_2) \leq \forall_2 \sigma_1 \rightarrow \sigma_2,
\]

so by the definition of \( \leq \), \( S'(\forall t \bar{s}(t \rightarrow \sigma'_2)) \leq \forall_2 \sigma_1 \rightarrow \sigma_2 \) as desired.

b) If \( x \in \text{dom}(A') \), then \( \langle A'_x, \text{Gen}(A'_x, A'(x) \rightarrow \sigma'_2) \rangle \in \text{PP}_2(\lambda x N) \).

Then by (2) and the definition of \( \leq \),
\[
S'(\langle A'_x, \text{Gen}(A'_x, A'(x) \rightarrow \sigma'_2) \rangle) \leq \langle A_x, \sigma_1 \rightarrow \sigma_2 \rangle,
\]

and since \( A \leq_1 A_x \), we have \( S'(\langle A'_x, \text{Gen}(A'_x, A'(x) \rightarrow \sigma'_2) \rangle) \leq \langle A, \sigma_1 \rightarrow \sigma_2 \rangle \), as desired.

iii) If \( \langle A, \sigma \rangle \in \text{AP}_2(M) \) by rule \( \text{(APP)} \), then \( M = M_1 M_2, \langle A, \tau_1 \rightarrow \sigma \rangle \in \text{AP}_2(M_1), \langle A, \tau \rangle \in \text{AP}_2(M_2) \), and \( \tau \leq \forall_2, \tau_1 \).

By induction, we have substitutions \( S_1 \) and \( S_2 \), and disjoint pairs \( \langle A_1, \forall \bar{s} \rangle \in \text{PP}_2(M_1) \) and \( \langle A_2, \tau' \rangle \in \text{PP}_2(M_2) \), such that
\[
S_1\langle A_1, \forall \bar{s} \rangle \leq \langle A, \tau_1 \rightarrow \sigma \rangle, \quad (3)
\]
\[
S_2\langle A_2, \tau' \rangle \leq \langle A, \tau \rangle. \quad (4)
\]

We may assume without loss of generality that \( \text{dom}(S_1), \text{dom}(S_2) \), and \( \bar{s} \) are disjoint.

We now consider two subcases.
a) \( \rho \) is a type variable \( t \).

Let \( t_1, t_2 \) be fresh type variables, and let \( \tau = \{ \tau' \leq t_1, t = t_1 \to t_2 \} \).

By (3), we have

\[
S_1(\forall \bar{z}, t) \leq_{\nu_2} \tau_1 \to \sigma,
\]

and therefore, \((S_1 \cup S'_1)t \leq_{\nu_2} \tau_1 \to \sigma \) for some \( S'_1 \) with domain \( \bar{z} \).

By the definition of \( \leq_{\nu_2} \), we must have

\[
(S_1 \cup S'_1)t = \tau_1' \to \sigma',
\]

\[
\tau_1 \leq_{\nu_2} \tau_1',
\]

\[
\sigma' \leq_{\nu_2} \sigma,
\]

for some \( \tau_1', \sigma' \in T_0 \). And by (4),

\[
S_2 \tau' \leq_{\nu_2} \tau \leq_{\nu_2} \tau_1.
\]

Therefore, \( S = (S_1 \cup S'_1 \cup S_2 \cup \{ t_1 := \tau_1', t_2 := \sigma' \}) \) is a solution to \( \pi \).

Pick \( U \in \text{MGS}(\pi) \), and let \( A' = U(A_1 + A_2) \). Then

\[
\langle A', \text{Gen}(A', Ut_2) \rangle \in \text{PP}_{p_2}(M).
\]

By Convention 35, there exists an \( R \) such that \( RA' = RU(A_1 + A_2) = S(A_1 + A_2) \) and \( RU t_2 = St_2 \).

Since \( RA' = S(A_1 + A_2) = S_1 A_1 + S_2 A_2 \), we have \( A \leq_{\nu_2} RA' \).

And \( RU t_2 = St_2 = \sigma' \leq_{\nu_2} \sigma \), so by Lemma 47(vi), we have \( R(\text{Gen}(A', Ut_2)) \leq_{\nu_2} \sigma \).

Therefore,

\[
R(\langle A', \text{Gen}(A', Ut_2) \rangle) \leq \langle A, \sigma \rangle,
\]

as desired.

b) \( \rho = \rho_1 \to \rho_2 \).

By (3), we have

\[
S_1(\forall \bar{z}, \rho_1 \to \rho_2) \leq_{\nu_2} \tau_1 \to \sigma,
\]

and therefore,

\[
(S_1 \cup S'_1)\rho_2 \leq_{\nu_2} \sigma,
\]

\[
\tau_1 \leq_{\nu_2} (S_1 \cup S'_1)\rho_1,
\]

40
for some $S'_1$ with domain $\bar{x}$. And by (4),
\[
S_2 \tau' \leq_{\forall 2} \tau \leq_{\forall 2,1} \tau_1.
\]
Therefore, $S = (S_1 \cup S'_1 \cup S_2)$ is a solution to the problem $\pi = \{\tau' \leq \rho_1\}$.

Pick $U \in \text{MGS}(\pi)$, and let $A' = U(A_1 + A_2)$. Then
\[
\langle A', \text{Gen}(A', U \rho_2) \rangle \in \text{PP}_2(M).
\]

By Convention 35, there exists an $R$ such that $RA' = RU(A_1 + A_2) = S(A_1 + A_2)$ and $RU \rho_2 = S \rho_2$.

By (3) and (4),
\[
RA' = S(A_1 + A_2) = S_1 A_1 + S_2 A_2 \leq A.
\]

And since $RU \rho_2 = S \rho_2 = (S_1 \cup S'_1) \rho_2 \leq_{\forall 2} \sigma$, by Lemma 47(vi) we have $R(\text{Gen}(A', U \rho_2)) \leq_{\forall 2} \sigma$.

Therefore,
\[
R(\langle A', \text{Gen}(A', U \rho_2) \rangle) \leq \langle A, \sigma \rangle,
\]
as desired.

iv) If $\langle A, \sigma \rangle \in \text{AP}_2(M)$ by rule (sub), then for some $\sigma'$, we have a shorter derivation of $\langle A, \sigma' \rangle \in \text{AP}_2(M)$, and $\text{Gen}(A, \sigma') \leq_{\forall 2} \sigma$.

By induction there is a pair $\langle A', \sigma'' \rangle \in \text{PP}_2(M)$ and a substitution $S$ such that $S(A', \sigma'') \leq \langle A, \sigma' \rangle$.

We now show that if $t \not\in \text{FTV}(A)$, then $t \not\in \text{FTV}(S \sigma'')$. Since $S \sigma'' \leq_{\forall 2} \sigma'$, this implies $S \sigma'' \leq_{\forall 2} \text{Gen}(A, \sigma')$, and therefore by transitivity, $S \sigma'' \leq_{\forall 2} \sigma$.

Assume by way of contradiction that $t \not\in \text{FTV}(A)$ and $t \in \text{FTV}(S \sigma'')$. Since $A \leq_{1} SA'$, $\text{FTV}(SA') \subseteq \text{FTV}(A)$. Therefore, $t \not\in \text{FTV}(A) \Rightarrow t \not\in \text{FTV}(SA')$.

Since $t \not\in \text{FTV}(SA')$ and $t \in \text{FTV}(S \sigma'')$, there must be some $u \in \text{FTV}(\sigma'') \setminus \text{FTV}(A')$ such that $t \in \text{FTV}(Su)$. However, it is easily checked that $\langle A', \sigma'' \rangle \in \text{PP}_2(M) \Rightarrow \text{FTV}(\sigma'') \setminus \text{FTV}(A') = \emptyset$, so we have reached a contradiction.

\[\Box\]

The next result shows the strong connection between the systems $I_2$ and $P_2$: a term is typable in one system if and only if it is typable in the other.
Theorem 58  For any $M$, $P_2 \Downarrow A \vdash M : \forall \vec{t} \sigma$ for some $\vec{t}$ if and only if $I_2 \Downarrow A \vdash M : \sigma$.

Proof: Each direction can be proved by induction on derivations.

- $I_2 \Rightarrow P_2$: The rules $(\text{VAR})$, $(\text{ABS})$, and $(\text{SUB})$ are trivial, so assume $M = M_1 M_2$ and $I_2 \Downarrow A \vdash M_1 M_2 : \sigma$ follows by rule $(\text{APP})$. Then we must have
  
  $I_2 \Downarrow A \vdash M_1 : (\bigwedge_{i \in I} \tau_i) \rightarrow \sigma$

  and
  
  $(\forall i \in I) I_2 \Downarrow A \vdash M_2 : \tau_i$.

  By induction we have
  
  $P_2 \Downarrow A \vdash M_1 : (\bigwedge_{i \in I} \tau_i) \rightarrow \sigma$

  and
  
  $(\forall i \in I) P_2 \Downarrow A \vdash M_2 : \tau_i$.

  By the principal typing property of $P_2$, there is a pair $\langle A', \sigma' \rangle$ and substitution $S$ such that
  
  $P_2 \Downarrow SA' \vdash M_2 : S\sigma'$,

  $A \leq_1 SA'$, and $S\sigma' \leq_{\forall 2} \tau_i$ for all $i \in I$. By weakening,

  $P_2 \Downarrow A \vdash M_2 : S\sigma'$.

  Then by the $P_2$ rule $(\text{APP})$,

  $I_2 \Downarrow A \vdash M_1 M_2 : \sigma$.

- $P_2 \Rightarrow I_2$: The rules $(\text{VAR})$ and $(\text{ABS})$ are trivial.

  If $P_2 \Downarrow A \vdash M : \forall \vec{t} \sigma$ follows by rule $(\text{SUB})$, then we must have a shorter derivation of

  $P_2 \Downarrow A \vdash M : \forall \vec{s} \tau$,

  and $\text{Gen}(A, \forall \vec{s} \tau) = (\forall \vec{n} \tau) \leq_{\forall 2} (\forall \vec{t} \sigma)$. We must show

  $I_2 \Downarrow A \vdash M : \sigma$.

  By induction,

  $I_2 \Downarrow A \vdash M : \tau$.  

42
Furthermore, by the definition of \( \leq_{\nu 2} \), for some sequence \( \vec{\rho} \) of simple types, we have \( \{ \vec{\nu} := \vec{\rho} \} \tau \leq_{2} \sigma \). We may assume that the type variables \( \vec{\nu} \) do not appear in \( A \). Then by substitutivity,

\[
I_2 \triangleright A \vdash M : \{ \vec{\nu} := \vec{\rho} \} \tau,
\]

and by the \( I_2 \) rule (SUB), we have \( I_2 \triangleright A \vdash M : \sigma \), as desired.

Otherwise, \( M = M_1 M_2 \) and \( P_2 \triangleright A \vdash M_1 M_2 : \sigma \) follows by rule (APP). Then

\[
P_2 \triangleright A \vdash M_1 : \sigma' \rightarrow \sigma
\]

and

\[
P_2 \triangleright A \vdash M_2 : \forall \vec{s} \tau,
\]

where \( \forall \vec{s} \tau \leq_{\nu 2,1} \sigma' \). By induction we have

\[
I_2 \triangleright A \vdash M_2 : \tau.
\]

If \( \sigma' = \Lambda_{i \in I} \sigma_i \), then by the definition of \( \leq_{\nu 2,1} \) and by substitutivity,

\[
I_2 \triangleright A \vdash M_2 : \sigma_i
\]

for all \( i \in I \). Then by the \( I_2 \) rule (APP),

\[
P_2 \triangleright A \vdash M_1 M_2 : \sigma.
\]

\( \Box \)

## 6 Recursive definitions

We now consider ways of typing recursive definitions. We extend the grammar of our language to include terms of the form \((\mu x M)\). Such a term is meant to represent the program \( x \) such that \( x = M \), where \( M \) may contain occurrences of \( x \).

In ML, recursive definitions are typed by the following rule:

\[
\begin{array}{c}
(\text{REC-SIMPLE}) \\
A \cup \{ x : \tau \} \vdash M : \tau \\
A \vdash (\mu x M) : \tau \\
\end{array}
\quad (\text{where } \tau \in T_0)
\]

43
6.1 Recursive definitions in $\Lambda_2$

In $\Lambda_2$ and ML, the rule (rec-simple) seems overly restrictive. Both systems allow ML type schemes to appear in type environments and as derived types, suggesting the rule of polymorphic recursion:

\[
(\text{rec-poly}) \quad \frac{A \cup \{x : \tau\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad (\text{where } \tau \in S(1))
\]

Example 59 When extended by (rec-poly), both ML and $\Lambda_2$ can type the following terms:

\[
(\mu w, (\lambda xyz.z)(w \, 3)(w \, \text{true})) : \forall t. t \to t,
\]

\[
(\mu x.xx) : \forall t.t.
\]

Neither is typable with the rule (rec-simple). Other examples are given by Mycroft [25] and Kfoury et al. [13, 15], who introduced (rec-poly) independently.

Unfortunately, type inference for $\Lambda_2$ or ML extended by (rec-poly) is undecidable [14, 9], so (rec-simple) is used in practice.

6.2 Recursive definitions in $I_2$

The rule (rec-simple) is one way of typing recursive definitions in intersection type systems. However, as with ML and $\Lambda_2$, it seems overly restrictive. The rule (rec-poly) involves $S(1)$ types, so it is not appropriate for the intersection type systems. Instead, we might consider a rule like the following:

\[
(\text{rec-int}) \quad \frac{A \cup \{x : \tau\} \vdash M : \tau}{A \vdash (\mu x M) : \tau} \quad (\text{where } \tau \in T_1)
\]

Note that the full power of the rule is achieved only by allowing $T_1$ derived types, so the rule is not compatible with the rank 2 intersection type systems that we have defined so far. However, the rule can be adapted to our systems as follows:

\[
(\text{rec-int}) \quad \frac{\forall i \in I \ A \cup \{x : (\Lambda j \in I \tau_j)\} \vdash M : \tau_i}{A \vdash (\mu x M) : \tau_{i_0}} \quad (\text{where } i_0 \in I)
\]

44
The system $I_2 + (\text{rec-simple})$ can type the following terms:

$$(\mu w. (\lambda x y z). (w 3)(w \text{true})) : \tau \rightarrow \tau,$$

$$(\mu w. (\lambda x y). (w w)) : \tau \rightarrow \tau,$$

where $\tau$ is any simple type. Neither term is typable in $I_2 + (\text{rec-simple})$.

The close connection between $I_2$ and $I_2 + (\text{rec-int})$ casts some doubt on the decidability of the system $I_2 + (\text{rec-int})$. However, $I_2 + (\text{rec-int})$ cannot type all of the terms that can be typed by $\Lambda_2 + (\text{rec-polym})$. For example, the term $(\mu x. x x)$ cannot be typed in $I_2 + (\text{rec-int})$. The decidability of $I_2 + (\text{rec-int})$ is an open question.

### 6.3 Recursive definitions in $P_2$

The system $P_2$ could be extended to type recursive definitions with either the rule $(\text{rec-simple})$ or the rule $(\text{rec-int})$ (the rule $(\text{rec-polym})$ is not appropriate since it requires $S(1)$ types to appear in type environments). Surprisingly, however, we can do better: we now propose two rules, $(\text{rec})$ and $(\text{rec-vac})$, for typing recursive definitions in $P_2$. The rules will allow use to type more terms than $(\text{rec-simple})$, and we retain principal typings and decidable type inference. We will also give a typing rule for mutually recursive definitions, of the form

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

where the variables $x_i$ are distinct.

The typing rules are defined below.

**Rule (rec)**

$$A_x \cup \{x : \tau\} \vdash M : \sigma \quad \text{(where } \sigma \leq \forall \tau_1 \tau)$$

$$A \vdash (\mu x. M) : \sigma$$

**Rule (rec-vac)**

$$A_x \vdash M : \sigma$$

$$A \vdash (\mu x. M) : \sigma$$

**Rule (letrec)**

$$A_{x_1, \ldots, x_n} \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash M : \sigma$$

$$A \vdash (\text{letrec } x_1 = M_1, \ldots, x_n = M_n \text{ in } M) : \sigma$$

**(where } \forall \leq n, \sigma_i \leq \forall \tau_1 \tau)$$

We write $P_2^R \vdash A \vdash M : \sigma$ if the judgment $A \vdash M : \sigma$ follows by the rules of $P_2$ and the rules $(\text{rec})$, $(\text{rec-vac})$, and $(\text{letrec})$, with types appearing in type environments restricted to $T_1$, and derived types restricted to $T_{\forall 2}$. 
The rule (\textsc{rec-vac}) is necessary to type terms like

\[(\mu w(\lambda x.xx)) : \forall s, t. s \land (s \rightarrow t) \rightarrow t.\]

In order to use the rule (\textsc{rec}) in this case, we would need a type \(\tau \in T_1\) such that \(\forall s, t. (s \land (s \rightarrow t)) \rightarrow t \leq \forall \nu_{1,1} \tau\). There is no such type, because \(s\) and \(s \rightarrow t\) cannot be unified.

Note in the hypothesis of the rule (\textsc{letrec}), we are careful to type each definition \(M_i\) as a recursive but not \textit{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.

\textbf{Example 60}

1) The following terms are typable in \(P_R^2\), but not in \(P_2 + (\text{\textsc{rec-simple}})\):

\[(\mu w.(\lambda x y z. z)(w 3)(w \text{\textsc{true}})) : \forall t. t \rightarrow t.\]

\[(\mu w.(\lambda x y y)(w w)) : \forall t. t \rightarrow t.\]

2) The term \((\mu x.x x)\) is not typable in \(P_R^2\). It has type \((\forall t. t)\) in ML + (\textsc{\textsc{rec-poly}}) and \(\Lambda_2 + (\text{\textsc{rec-poly}})\).

\textbf{Definition 61} The set \(PP_{P_R^2}(M)\) of principal pairs for a term \(M\) is defined just as \(PP_{P_2}\), with the addition of the following clauses:

4) If \(M = (\mu x.N)\) and \(\langle A, \sigma \rangle \in PP_{P_2^R}(N)\), then:

a) If \(x \notin \text{\textsc{dom}}(A)\), then \(\langle A, \sigma \rangle \in PP_{P_2^R}(M)\).

b) If \(x \in \text{\textsc{dom}}(A)\) and \(U \in \text{\textsc{MGS}}(\sigma \leq A(x))\),

then \(\langle U A_x, \text{\textsc{Gen}}(U A_x, U \sigma) \rangle \in PP_{P_2^R}(M)\).

v) If \(M = (\text{\textsc{letrec}} x_1 = M_1, \ldots, x_n = M_n\ in M_0)\),

and \(\langle A_i, \sigma_i \rangle \in PP_{P_2^R}(\mu x_i.M_i)\) for \(1 \leq i \leq n\),

\(\langle A_0, \sigma_0 \rangle \in PP_{P_2^R}(M_0)\),

\(A' = A_0 + \Sigma_{1 \leq i \leq n} A_i\),

\(U \in \text{\textsc{MGS}}(\{\sigma_i \leq A'(x_i) \mid 1 \leq i \leq n, x_i \in \text{\textsc{dom}}(A')\})\),

and \(A'' = A''_{x_1, \ldots, x_n}\),

then \(\langle U A'', \text{\textsc{Gen}}(U A'', U \sigma_0) \rangle \in PP(M)\).
Theorem 62 If \( \langle A, \sigma \rangle \in \text{PP}_{P_2}(M) \), then \( \langle A, \sigma \rangle \in \text{AP}_{P_2}(M) \).

**Proof:** By induction on the definition of \( \text{PP}_{P_2}(M) \). For the rules of \( P_2 \), see the proof of Theorem 56. We only need to consider the following cases.

iv) If \( M = (\mu x.N) \), we consider two cases.

a) If \( x \) is not free in \( N \), then \( \langle A, \sigma \rangle \in \text{PP}_{P_2}(N) \). By induction, \( \langle A, \sigma \rangle \in \text{AP}_{P_2}(N) \), and by rule (rec-vac), \( \langle A, \sigma \rangle \in \text{AP}_{P_2}(\mu x.N) \).

b) If \( x \) is free in \( N \), then for some \( \langle A', \sigma' \rangle \in \text{PP}_{P_2}(N) \) and \( U \in \text{MGS}(\sigma' \leq A'(x)) \), we have

\[
\langle A, \sigma \rangle = \langle U A'_x, \text{Gen}(U A'_x, U \sigma') \rangle.
\]

By induction, \( \langle A', \sigma' \rangle \in \text{AP}_{P_2}(N) \). Then \( \langle U A', U \sigma' \rangle \in \text{AP}_{P_2}(N) \) by substitutivity. Since \( U \sigma' \leq_{\nu} U A'(x) \), by rule (rec) we have \( \langle U A'_x, U \sigma' \rangle \in \text{AP}_{P_2}(\mu x.N) \). Finally by rule (sub),

\[
\langle U A'_x, \text{Gen}(U A'_x, U \sigma') \rangle \in \text{AP}_{P_2}(\mu x.N).
\]

v) If \( M = (\text{letrec } x_1 = M_1, \ldots, x_n = M_n \text{ in } M_0) \),

then \( \langle A, \sigma \rangle = \langle U A', \text{Gen}(U A'', U \sigma_0) \rangle \), where

\[
\langle A_i, \sigma_i \rangle \in \text{PP}_{P_2}(\mu x_i M_i) \text{ for } 1 \leq i \leq n,
\]

\[
\langle A_0, \sigma_0 \rangle \in \text{PP}_{P_2}(M_0),
\]

\[
A' = A_0 + \sum_{1 \leq i \leq n} A_i,
\]

\[
U \in \text{MGS}(\{ \sigma_i \leq A'(x_i) \mid 1 \leq i \leq n, x_i \in \text{dom}(A') \}),
\]

and \( A'' = A'_{x_1, \ldots, x_n} \).

By induction, \( \langle A_i, \sigma_i \rangle \in \text{AP}_{P_2}(\mu x_i M_i) \text{ for } 1 \leq i \leq n \), and \( \langle A_0, \sigma_0 \rangle \in \text{AP}_{P_2}(M_0) \).

By weakening and substitutivity, \( \langle U A', U \sigma_i \rangle \in \text{AP}_{P_2}(\mu x_i M_i) \text{ for } 1 \leq i \leq n \), and \( \langle U A', U \sigma_0 \rangle \in \text{AP}_{P_2}(M) \).

Then by rule (letrec), \( \langle U A'', U \sigma_0 \rangle \in \text{AP}_{P_2}(M) \), and by (sub),

\[
\langle A, \sigma \rangle = \langle U A'', \text{Gen}(U A'', U \sigma_0) \rangle \in \text{AP}_{P_2}(M).
\]

\[ \square \]

**Theorem 63 (Principal pairs for \( P_2^R \))** If \( \langle A, \sigma \rangle \in \text{AP}_{P_2^R}(M) \), then there is a pair \( \langle A', \sigma' \rangle \in \text{PP}_{P_2^R}(M) \) and a substitution \( S \) such that \( S(A', \sigma') \leq \langle A, \sigma \rangle \).

47
Proof: By induction on the definition of $\text{AP}_{\mathbb{P}_2}(M)$. For the rules of $\mathbb{P}_2$, see the proof of Theorem 57. We only need to consider the following cases.

v) If $\langle A, \sigma \rangle \in \text{AP}_{\mathbb{P}_2}(M)$ by rule (rec-vac), then $M = (\mu x N)$, $x$ is not free in $N$, and $\langle A_x, \sigma \rangle \in \text{AP}_{\mathbb{P}_2}(N)$.

By induction, we have a pair $\langle A', \sigma' \rangle \in \text{PP}_{\mathbb{P}_2}(N)$ and a substitution $S$ such that $S \langle A', \sigma' \rangle \leq \langle A_x, \sigma \rangle$.

By Lemma 54(i), $x \notin \text{dom}(A')$, so $\langle A', \sigma' \rangle \in \text{PP}_{\mathbb{P}_2}(\mu x N)$ as desired.

vi) If $\langle A, \sigma \rangle \in \text{AP}_{\mathbb{P}_2}(M)$ by rule (rec), then $M = (\mu x N)$, and for some $\tau \in T_1$, we have $\langle A_x \cup \{x : \tau\}, \sigma \rangle \in \text{AP}_{\mathbb{P}_2}(N)$ and $\sigma \leq \nu_2,1 \tau$.

By induction, we have a pair $\langle A', \sigma' \rangle \in \text{PP}_{\mathbb{P}_2}(N)$ and a substitution $S$ such that

$$S \langle A', \sigma' \rangle \leq \langle A_x \cup \{x : \tau\}, \sigma \rangle. \quad (5)$$

We consider two cases.

a) If $x \notin \text{dom}(A')$, then $\langle A', \sigma' \rangle \in \text{PP}_{\mathbb{P}_2}(\mu x N)$, and by (5), $A \leq_1 A_x \leq_1 S A'$ and $S \sigma' \leq_2 \sigma$, as desired.

b) If $x \in \text{dom}(A')$, and $\tau' = A'(x)$, then by (5), $S \sigma' \leq_2 \sigma \leq_2,1 \tau \leq_1 S \tau'$, so $S$ is a solution to $\pi = \{\sigma' \leq \tau'\}$.

Then pick $U \in \text{MGS}(\pi)$, so that

$$\langle U A'_x, \text{Gen}(U A'_x, U \sigma') \rangle \in \text{PP}_{\mathbb{P}_2}(\mu x N).$$

By Convention 35, there exists an $R$ such that $RU A'_x = S A'_x$ and $RU \sigma' = S \sigma'$.

By (5), $A \leq_1 S A'_x$ and by (5) and Lemma 47(vi), $R(\text{Gen}(U A', U \sigma')) \leq_2 \sigma$. Therefore

$$R(\langle U A'_x, \text{Gen}(U A'_x, U \sigma') \rangle) \leq \langle A, \sigma \rangle,$$

as desired.

vii) If $\langle A, \sigma \rangle \in \text{AP}_{\mathbb{P}_2}(M)$ by rule (letrec), then for some $\vec{x}, N, \vec{\tau}$ of length $n$, $M = (\text{letrec } \vec{x} = N \text{ in } N_0)$, $\langle A_{\vec{x}} \cup \{\vec{x} : \vec{\tau}\}, \sigma \rangle \in \text{AP}_{\mathbb{P}_2}(N_0)$, and $\langle A_{\vec{x}} \cup \{\vec{x} : \vec{\tau}\}, \sigma_i \rangle \in \text{AP}_{\mathbb{P}_2}(\mu x_i N_i)$ for all $i \leq n$.

By induction, we have pairs $\langle A_0, \sigma'_0 \rangle \in \text{PP}_{\mathbb{P}_2}(N_0)$, and $\langle A_i, \sigma'_i \rangle \in \text{PP}_{\mathbb{P}_2}(\mu x_i N_i)$, and substitutions $S_0, S_1, \ldots, S_n$ such that

$$S_0 \langle A_0, \sigma'_0 \rangle \leq \langle A_{\vec{x}} \cup \{\vec{x} : \vec{\tau}\}, \sigma \rangle, \quad (6)$$

$$S_i \langle A_i, \sigma'_i \rangle \leq \langle A_{\vec{x}} \cup \{\vec{x} : \vec{\tau}\}, \sigma_i \rangle \quad (7)$$

48
for all $i \leq n$.

Let $A' = A_0 + \Sigma_{1 \leq i \leq n} A_i$, $S = S_0 \cup S_1 \cup \cdots \cup S_n$, and $\pi = \{ \sigma'_i \leq A'(x_i) \mid 1 \leq i \leq n, x_i \in \text{dom}(A') \}$. Then by (7), if $x_i \in \text{dom}(A')$, then

$S_i \sigma'_i \leq_{\forall 2} \sigma_i \leq_{\forall 2,1} \tau_i \leq_{1} A'(x_i)$.

Therefore $S$ is a solution to $\pi$.

Pick $U \in \text{MGS}(\pi)$ and let $A'' = A'_{x_1, \ldots, x_n}$. Then

$\langle U A'', \text{Gen}(U A'', U \sigma'_0) \rangle \in \text{PP}_P(M)$.

By Convention 35, there exists an $R$ such that $RU A'' = SA''$ and $RU \sigma'_0 = S\sigma'_0$.

Then by (6) and (7), $A \leq_{1} SA''$, and $S\sigma'_0 \leq_{\forall 2} \sigma$, as desired.

\[\square\]

**Theorem 64**

i) If $I_2 + (\text{rec-int}) \triangleright A \vdash M : \sigma$, then $P^R_2 \triangleright A \vdash M : \sigma$.

ii) If $P_2 + (\text{rec-int}) \triangleright A \vdash M : \sigma$, then $P^R_2 \triangleright A \vdash M : \sigma$.

**Proof:**

i) By induction on derivations. The cases for all the rules except $(\text{rec-int})$ are just as for Theorem 58, so assume that $I_2 + (\text{rec-int}) \triangleright A \vdash \mu x M : \sigma$ holds by rule $(\text{rec-int})$. We must have

$(\forall i \in I) \quad I_2 \triangleright A_x \cup \{ x : \land_{j \in I} \tau_j \} \vdash M : \tau_i$,

and $\sigma = \tau_{i_0}$ for some $i_0 \in I$. By induction,

$(\forall i \in I) \quad P^R_2 \triangleright A_x \cup \{ x : \land_{j \in I} \tau_j \} \vdash M : \tau_i$.

By the principal pair property of $P^R_2$, there is a pair $\langle A', \sigma' \rangle$ and substitution $S$ such that

$P^R_2 \triangleright SA' \vdash M : S\sigma'$,

$A_x \cup \{ x : \land_{j \in I} \tau_j \} \leq_{1} S A'$, and $S\sigma' \leq_{\forall 2} \tau_i$ for all $i \in I$. By weakening,

$P^R_2 \triangleright A_x \cup \{ x : \land_{j \in I} \tau_j \} \vdash M : S\sigma'$.
Then by rule \( \text{rec} \),

\[
P_2^R \vdash A \vdash (\mu x \, M) : S_\sigma',
\]

and by rule \( \text{sub} \),

\[
P_2^R \vdash A \vdash (\mu x \, M) : \tau_\alpha.
\]

ii) Identical to the last case.

\( \Box \)

7 Compiling with rank 2 intersection types

We briefly discuss some applications of rank 2 intersections in compilation.

Polymorphism allows a function \( F \) of type \( \forall t. t \to t \) to be applied to arguments of any type. Unfortunately, it also requires that the data representation of its arguments be reduced to a lowest common denominator: the machine code for \( F \) cannot handle both a 32-bit integer in a general purpose register and a 64-bit floating point number in a float register. In practice, arguments are “boxed,” or represented as a pointer to the actual data value stored in main memory. Boxing and unboxing coercions slow program execution.

These overheads can be reduced when more is known about the uses of the polymorphic function. For example, consider the program

\[
M = (\lambda f. (f \, 3, f \, \text{true})) \, F.
\]

A naive implementation would insert instructions to box the arguments 3 and \( \text{true} \) before passing them to \( F \). A more clever implementation would recognize that the only arguments of \( F \) are integers and booleans, both of which can be represented in a single 32-bit register; so \( F \) could be compiled to expect an unboxed value as its argument.

This can easily be achieved in \( P_2 \). To compile \( M \), we first calculate the principal typings of the operator and operand:

\[
(\lambda f. (f \, 3, f \, \text{true})) : \forall s, u. (\text{int} \to s) \land (\text{bool} \to u) \to s \times u,
\]

\[
F : \forall t. t \to t.
\]

The type of the operator indicates that \( F \) will only be applied to integers and booleans, and the compiler can take advantage of this in generating the machine code for \( F \). Note that this improves on Bjørner’s minimal typing derivations [3], which would require the arguments to be boxed.
\( \text{P}_2 \) also supports other data representation strategies. For example, in compiling the program \((\lambda f.(f\,3,f\,2.4))F\), we will calculate the principal typing

\[
(\lambda f.(f\,3,f\,2.4)) : \forall s,u.(\text{int} \to s) \land (\text{float} \to u) \to s \times u.
\]

If floating point numbers are 64-bit values, we can’t just compile \( F \) to expect its argument in a 32-bit register, as before. Boxing is one solution. But another solution is possible: specialization [8]. We can generate two versions of \( F \), one expecting an unboxed integer in a 32-bit register, and one expecting an unboxed float in a 64-bit register. We are essentially overloading the variable \( f \), so the application \((f\,3)\) invokes the integer-expecting \( F \), and \((f\,2.4)\) invokes the float-expecting \( F \).

8 Conclusion

We discussed a variety of rank 2 type systems: \( \Lambda_2 \), the rank 2 fragment of System F; \( \text{I}_2 \), \( \text{I}^b_2 \), and \( \text{I}^v_2 \), all variants of the rank 2 intersection type discipline; and \( \text{P}_2 \), which adds ML-style, top-level quantification of type variables to \( \text{I}_2 \). We showed that all of the systems are equivalent in terms of typability—a term is typable in one system if and only if it is typable in another. An immediate corollary is that typability in all of these systems is DEXPTIME-complete. We have also determined that the sequence \( \text{I}^b_2, \text{I}^v_2, \text{I}_2, \text{P}_2 \) is in order of increasing “expressiveness.” For example, a judgment of \( \text{I}^b_2 \) is a judgment of \( \text{P}_2 \), but not vice versa.

We proposed a new rule for typing recursive definitions that can type many examples of polymorphic recursion. The extension of \( \text{P}_2 \) by this rule results in a system with principal typings and decidable type inference.

Finally, we discussed some applications of intersections in compilation. The finite polymorphism of intersections expresses data representation constraints more accurately than polymorphism by quantification. The accurate expression of these constraints leads to data representations that require fewer boxing and unboxing coercions at runtime.

Acknowledgments. Albert Meyer, Jens Palsberg, and Mona Singh made many useful suggestions. I would particularly like to thank Assaf Kfoury for his careful reading of a draft.
References


54


Index

Types
- $T_v$, 6
- $T_b$, 7
- $T_1$, 8
- $T_2$, 8
- $T_{\lor 2}$, 32
- types of System F, 10
- $R(n)$, 11
- $S(n)$, 11
- $S'(n)$, 11

Type systems
- $\Lambda_{\text{3}}$, 11
- $\Lambda_{\text{3}}^\ast$, 11
- $I_{\text{3}}$, 31
- $I_{\text{3}}^\ast$, 9
- $I_{\text{3}}^\beta$, 30
- $\text{ML}$, 12
- $P_{\text{2}}$, 33
- $P_{\text{R}}$, 45

Relations and operators
- $\sigma \leq_1 \tau$, 8
- $A \leq A'$, 21
- $\sigma \leq_2 \tau$, 8
- $\sigma \leq_2 \tau$, 22
- $\sigma \leq_{\lor 2} \tau$, 32
- $\sigma \leq_{\lor 2,1} \tau$, 32
- $\sigma \succ \tau$, 11
- $\sigma \preceq_1 \tau$, 13
- $\sigma \preceq_2 \tau$, 13
- $\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle$, 21, 34
- $S \leq S'$, 22
- $S \leq_{\lor} S'$, 22
- $S = S'$, 22
- $S =_{\lor} S'$, 22

General concepts
- $A + A'$, 9
- $A \cup A'$, 7
- $S \cup S'$, 8
- $M \rightarrow_\gamma M'$, 15
- $\gamma\text{-nf}(M)$, 16
- $\text{act}(M)$, 15
- $\text{AP}(M)$, 7
- $\text{appl}(M)$, 16
- $\text{dom}(A)$, 7
- $\text{dom}(S)$, 7
- $\text{FTV}$, 7
- $\text{Gen}(A, \sigma)$, 7
- $\text{lcg}(\sigma)$, 18
- $\text{MGS}(\pi)$, 22, 35
- $\text{MGS}(\pi)[W]$, 23, 35
- $\text{ml}(M)$, 17
- $\text{PP}_{\text{E}}(M)$, 25
- $\text{PP}_{\beta}(M)$, 36
- $\text{PP}_{\beta}(M)$, 46
- $\text{rng}(S)$, 7
- $\text{Solutions}(\pi)$, 22, 35

56