A polar type system*

TREVOR JIM
AT&T Labs

Abstract

We introduce a restriction of the type system of intersection types and universal type quantification, in which quantifiers must only appear at positive occurrences in types, and intersections at negative occurrences. We show that the system is an extension of the system of rank two intersection types, and, therefore, of ML. The system is much more powerful than ML; for example, all pure normal forms are typable in the system. We show that the system has principal pairs, and that type inference for the system is decidable. These properties are retained when the system is extended by a form of polymorphic recursion.

1 Introduction

Type inference is undecidable in both the system of intersection types [1, 16, 13] and in System F [4, 14, 19]. When intersection types and universal type quantification are combined, type inference remains undecidable, but things seem improved. The combined system improves on System F because it has principal typings, which are employed by most type inference algorithms; and it improves on the system of intersection types by having simpler principal typings [2, 15, 10]. This indicates that the search for more powerful type inference algorithms should start with the combined system.

In this paper, we describe a partial type inference algorithm for the combined system, and based on this algorithm we introduce a restricted type system for which the algorithm is complete. The restricted type system can be seen as an exposition of the type inference algorithm, explaining when it will succeed and when it will fail, and facilitating comparison with other type inference algorithms.

The algorithm and its type system are based on the observation that the positive and negative occurrences in types play dual roles in type inference. Intuitively, the negative occurrences correspond to the *requirements* a term makes, and the positive occurrences correspond to the *capabilities* it provides. For example, if a term has type $\sigma \to \tau$, then it requires an argument of type σ before it can be used at type τ. A type inference algorithm will naturally try to construct typings that

*This work was done while the author was at MIT.

minimize requirements and maximize capabilities: that is, it will try to place the weakest possible types at negative occurrences and the strongest possible types at positive occurrences.

This intuition is confirmed by the principal typings of the combined system of intersection types and universal type quantification [10]. Intersections—finite conjunctions of types—appear only at negative occurrences in principal typings. Quantifiers—infinite conjunctions of types—appear only at positive occurrences. This forms the basis of both our type inference algorithm and our type system, in which we *require* that quantifiers appear at only positive occurrences and intersections appear only at negative occurrences. Since this polarity-based restriction is the basis of our type system, we call it the polar type system, **P**.

Type system **P** has principal typings and decidable type inference. It is also much more powerful than ML [11]; for example, it can type all normal forms. For comparison, the following table shows how some terms are typed in **P**, ML, and the system $\mathbf{I}_2$ of rank 2 intersection types [8].

	$\lambda f.f(\lambda x.x)$	$\lambda y.yy$	$(\lambda f.f(\lambda x.x))(\lambda y.yy)$
P	$\forall t.((\forall s.s \to s) \to t) \to t$	$\forall t,s.((t \to s) \wedge t) \to s$	$\forall t.t \to t$
ML	$\forall t,s.((t \to t) \to s) \to s$	not typable	not typable
$\mathbf{I}_2$	$((t \to t) \to s) \to s$	$((t \to s) \wedge t) \to s$	not typable

Table 1: Some terms and their types in **P**, ML, and $\mathbf{I}_2$

The first term, $\lambda f.f(\lambda x.x)$, is typable in all three type systems. However, its type in **P** has a quantifier nested under two arrows. This is a far more informative type than the ML type: it allows the argument f to assume that it will be applied to a polymorphic function. The second term, $\lambda y.yy$, self-applies a λ-bound variable, and hence is not typable in ML; but it is typable in both **P** and $\mathbf{I}_2$. The third term combines the first two, and is only typable in **P**, where it uses the fact that $\lambda y.yy$ will be applied to a polymorphic function.

Organization. Although the type system was designed after the type inference algorithm, it helps to describe it first, which we do in Section 2. Subtype satisfaction, the basis of our inference algorithm, is described in Section 3, and the algorithm itself is given in Section 4. In Section 5 we show how the system can be extended by a form of polymorphic recursion, and related work is discussed in Section 6.

Most proofs are omitted, due to lack of space. They appear in an extended version of this paper, available from the author.

2 The type system

We begin by defining the *positive types*, $\mathbf{T}^+$, the *negative types*, $\mathbf{T}^-$, and the *simple types*, $\mathbf{T}_0$:

$$\begin{array}{rcl}
\mathbf{T}_0 & = & \{t \mid t \text{ is a type variable}\} \cup \{(\sigma \to \tau) \mid \sigma, \tau \in \mathbf{T}_0\} \\
\mathbf{T}^+ & = & \mathbf{T}_0 \cup \{(\sigma \to \tau) \mid \sigma \in \mathbf{T}^-, \tau \in \mathbf{T}^+, \text{ and } \tau \text{ is not a } \forall\text{-type}\} \\
 & & \quad \cup \{(\forall t \sigma) \mid \sigma \in \mathbf{T}^+\} \\
\mathbf{T}^- & = & \mathbf{T}_0 \cup \{(\sigma \to \tau) \mid \sigma \in \mathbf{T}^+, \tau \in \mathbf{T}^-, \text{ and } \tau \text{ is not a } \wedge\text{-type}\} \\
 & & \quad \cup \{(\sigma \wedge \tau) \mid \sigma, \tau \in \mathbf{T}^-\}
\end{array}$$

In positive types, quantifiers appear at positive positions and intersections appear at negative positions; this is reversed in negative types. For technical convenience, our definition prohibits quantifiers and intersections from appearing immediately to the right of arrows. Note that the simple types are the only types that are both positive and negative: $\tau \in \mathbf{T}_0$ if and only if $\tau \in \mathbf{T}^+$ and $\tau \in \mathbf{T}^-$. If $\vec{t} = t_1, t_2, \ldots, t_n$, $n \geq 0$, and $\sigma \in \mathbf{T}^+$ is not a $\forall$-type, we write $(\forall \vec{t} \sigma)$ for the type

$$(\forall t_1 (\forall t_2 (\ldots (\forall t_n \sigma) \ldots))).$$

It is convenient to adopt the following syntactic conventions.

Convention 1 *We consider types to be syntactically equal modulo renaming of bound type variables, reordering of adjacent quantifiers, and elimination of unnecessary quantifiers. We consider $\wedge$ to be an associative, commutative, and idempotent operator, so that negative types may be considered a finite, nonempty set of types, written in the form $\bigwedge_{i \in I} \sigma_i$, where each σ_i is not a $\wedge$-type.*

Intuition. Our type system, $\mathbf{P}$, will have judgments of the form $A \vdash M : \sigma$, where A is a type environment and M is a λ-term. According to our intuition about polarities explained in the introduction, we should require the derived type σ to be a positive type. This introduces two problems.

Problem 1: what do we do about type variables that occur both positively and negatively in a type? For example, suppose that $M : \forall t.t \to t$. We would like to be able to type (MM). In System F, this can be done by instantiating the type variable t with the type $\forall t.t \to t$, so that the first M has type $(\forall t.t \to t) \to (\forall t.t \to t)$. This type has $\forall$'s occurring both positively and negatively, violating our polarity restriction.

Our solution to this first problem is that we will only allow simple types to be instantiated for type variables. We will handle the typing of (MM) by another mechanism.

Problem 2: interaction between the standard rules (VAR) and (ABS):

$$(\text{VAR}) \qquad A \cup \{x : \sigma\} \vdash x : \sigma$$

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

Suppose σ is a nontrivial positive type. Then (VAR) derives a positive type, as desired. But then the domain of the type derived by (ABS) is positive too, so (ABS) cannot derive a positive type. A similar problem occurs if we assume σ is a nontrivial negative type. The only fix seems to be to make σ both positive and negative, that is, a simple type. We want to go beyond this. For example, we'd like to be able to type

$$\lambda f.f(\lambda x.x) : \forall t.((\forall s.s \to s) \to t) \to t.$$

Here the type of f, $(\forall s.s \to s) \to t$, is negative, which makes sense when f is abstracted, but not when used in the body.

Our solution to the second problem is to allow a variable to have a negative type *when applied*. The type of the application itself must be positive, and so variables considered alone must have positive type. This is consistent with the ML strategy of allowing the argument of a function to be polymorphic when the argument is manifest, namely, in a **let** expression.

The definition of "manifest" is a design decision. For example, in the expression

$$(\lambda x.yM_1M_2)M_3M_4,$$

it is clear that the first two arguments of y will be M_1 and M_2. But it is also clear that M_4 will be the third argument of y; we may choose to type either the application (yM_1M_2) or the application $(yM_1M_2M_4)$ specially. It turns out to be technically smoother to make the second choice, and this is the approach we take.

Typing rules. 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, and $A \backslash x$ for the type environment A with any pair for the variable x removed. We write $A_1 \cup A_2$ for the union of two type environments; by convention we assume that the variables of A_1 and A_2 are disjoint.

We write $\mathrm{FV}(M)$ for the free variables of a term M, $\mathrm{FTV}(\sigma)$ for the free type variables of a type σ, $\mathrm{FTV}(A)$ for free type variables of the types appearing in A, and $\mathrm{FTV}\langle A, \sigma\rangle$ for $\mathrm{FTV}(A) \cup \mathrm{FTV}(\sigma)$. If $\sigma \in \mathbf{T}^+$, we write $\mathrm{Gen}(A, \sigma)$ for $\forall t_1 \cdots t_n.\sigma$, where $\{t_1, \ldots, t_n\} = \mathrm{FTV}(\sigma) - \mathrm{FTV}(A)$ (we will see later that the order of the type variables $t_1 \cdots t_n$ does not matter).

A *substitution* is a finite set $\{t_1 := \tau_1, \ldots, t_n := \tau_n\}$ of (type variable, simple type) pairs, where the type variables $t_1, \ldots, t_n$ are distinct. We use S, R, Q, U to

(ABS)
$$\frac{A \backslash x \cup \{x : \sigma\} \vdash M : \tau}{A \vdash (\lambda x M) : \sigma \to \tau}$$
$$(\text{where } \sigma \in \mathbf{T}_0 \text{ if } x \notin \mathrm{FV}(M))$$

(VAR-APP)
$$\frac{A \vdash M_1 : \tau_1, \cdots, A \vdash M_n : \tau_n}{A \vdash (x M_1 \cdots M_n : \tau)}$$
$$(\text{where } n \geq 0,\ A(x) \leq^- (\tau_1 \to \cdots \to \tau_n \to \tau), \text{ and } \tau \in \mathbf{T}_0)$$

(ABS-APP)
$$\frac{\begin{array}{l} A \vdash (\lambda x M M_1 \cdots M_n) : (\bigwedge_{i \in I} \sigma_i) \to \tau \\ (\forall i \in I) \quad A \vdash N : \sigma'_i, \quad \sigma'_i \leq^{\pm} \sigma_i \end{array}}{A \vdash (\lambda x M) N M_1 \cdots M_n : \tau}$$
$$(\text{where } n \geq 0 \text{ and } x \notin \mathrm{FV}(M_1 \cdots M_n))$$

(GEN-SUB)
$$\frac{A \vdash M : \sigma}{A \vdash M : \sigma'} \quad (\text{where } \mathrm{Gen}(A, \sigma) \leq^+ \sigma')$$

Table 2: Typing rules of **P**

range over substitutions. If S is a substitution, we write St for the type paired with t in S, and $\mathrm{dom}(S)$ for the set $\{t \mid \exists \tau.(t := \tau) \in S\}$. A substitution may be extended to a total function by defining $St = t$ if $t \notin \mathrm{dom}(S)$. Then the application of substitutions is extended to types and type environments in the usual way (avoiding the capture of free type variables). We write SR for the composition of substitutions S and R.

The system **P** derives judgments of the form $A \vdash M : \tau$, where the types assigned by A are all negative and the derived type, τ, is always positive. The typing rules are given in Table 2.

The system uses three different subtyping relations, in the rules (VAR-APP), (ABS-APP), and (GEN-SUB). In the rule (VAR-APP) we are comparing two negative types with the relation $\leq^-$; in rule (ABS-APP) we are comparing a positive type to a negative type with the relation $\leq^{\pm}$; and in rule (GEN-SUB) we are comparing a positive type to a positive type, using the relation $\leq^+$. The subtyping relations are defined in Table 3.

Note that the instantiation of quantified type variables used in the definitions of $\leq^+$ and $\leq^{\pm}$ is restricted: we may only instantiate a simple type for the type variable. This ensures that we remain within the positive/negative type universe. However, we also lose some power; for example, the type $(\forall t.t)$ is *not* a least type in the orderings $\leq^+$ and $\leq^{\pm}$.

We write $\sigma \sim^- \tau$ if and only if $\sigma \leq^- \tau$ and $\tau \leq^- \sigma$, and we write $\sigma \sim^+ \tau$ if and only if $\sigma \leq^+ \tau$ and $\tau \leq^+ \sigma$. The relations $\leq^-$ and $\leq^+$ are preorders, so it

$$t \leq^+ t$$

$$\frac{\tau_1 \leq^- \sigma_1, \quad \sigma_2 \leq^+ \tau_2}{(\sigma_1 \to \sigma_2) \leq^+ (\tau_1 \to \tau_2)} \qquad \sigma \leq^+ (\forall t \sigma),\ t \notin \mathrm{FTV}(\sigma)$$

$$(\forall t \sigma) \leq^+ \{t := \tau\}\sigma, \quad \tau \in \mathbf{T}_0$$

$$\frac{\sigma \leq^+ \tau}{(\forall t \sigma) \leq^+ (\forall t \tau)}$$

$$\frac{\sigma_1 \leq^+ \sigma_2, \quad \sigma_2 \leq^+ \sigma_3}{\sigma_1 \leq^+ \sigma_3}$$

$$t \leq^- t \qquad\qquad t \leq^\pm t$$

$$\frac{\tau_1 \leq^+ \sigma_1, \quad \sigma_2 \leq^- \tau_2}{(\sigma_1 \to \sigma_2) \leq^- (\tau_1 \to \tau_2)} \qquad\qquad \frac{\tau_1 \leq^\pm \sigma_1, \quad \sigma_2 \leq^\pm \tau_2}{(\sigma_1 \to \sigma_2) \leq^\pm (\tau_1 \to \tau_2)}$$

$$\frac{\sigma \leq^- \tau_1, \quad \sigma \leq^- \tau_2}{\sigma \leq^- (\tau_1 \wedge \tau_2)} \qquad\qquad \frac{\sigma \leq^\pm \tau_1, \quad \sigma \leq^\pm \tau_2}{\sigma \leq^\pm (\tau_1 \wedge \tau_2)}$$

$$\frac{\sigma_1 \leq^- \tau}{(\sigma_1 \wedge \sigma_2) \leq^- \tau} \qquad\qquad \frac{\{t := \rho\}\sigma \leq^\pm \tau}{(\forall t \sigma) \leq^\pm \tau} \quad \rho \in \mathbf{T}_0$$

$$\frac{\sigma_1 \leq^- \sigma_2, \quad \sigma_2 \leq^- \sigma_3}{\sigma_1 \leq^- \sigma_3} \qquad\qquad \frac{\sigma_1 \leq^+ \sigma_2, \quad \sigma_2 \leq^\pm \tau_1, \quad \tau_1 \leq^- \tau_2}{\sigma_1 \leq^\pm \tau_2}$$

Table 3: The subtyping relations $\leq^+$, $\leq^-$, and $\leq^\pm$

is sensible to consider $\mathbf{T}^+$ types syntactically equal modulo $\sim^+$, and $\mathbf{T}^-$ types syntactically equal modulo $\sim^-$, and we will adopt this convention. Then $\leq^+$ and $\leq^-$ become partial orders.

As an example, we present the crucial step in the derivation of the judgment

$$\vdash (\lambda f.f(\lambda x.x))(\lambda y.yy) : \forall t.t \to t.$$

Let $t^2 = t \to t$. Then

$$(\text{ABS-APP}) \quad \frac{\begin{array}{l}\vdash (\lambda f.f(\lambda x.x)) : ((\forall s.s \to s) \to t^2) \to t^2 \\ \vdash (\lambda y.yy) : ((t^2 \to t^2) \wedge t^2) \to t^2\end{array}}{\vdash (\lambda f.f(\lambda x.x))(\lambda y.yy) : t^2}$$

because

$$\begin{aligned}&(\forall s.s \to s) \leq^{\pm} (t^2 \to t^2) \text{ and } (\forall s.s \to s) \leq^{\pm} t^2\\ &\quad \Rightarrow (\forall s.s \to s) \leq^{\pm} (t^2 \to t^2) \wedge t^2\\ &\quad \Rightarrow ((t^2 \to t^2) \wedge t^2) \to t^2 \leq^{\pm} (\forall s.s \to s) \to t^2.\end{aligned}$$

Discussion. The rules (VAR-APP), (ABS-APP), and (GEN-SUB) subsume some familiar typing rules. The following version of the rule (VAR) is a special case of the rule (VAR-APP):

$$(\text{VAR'}) \quad A \cup \{x : \sigma\} \vdash x : \tau, \quad (\text{where } \sigma \leq^{-} \tau \text{ and } \tau \in \mathbf{T}_0)$$

And the rules (INST) and (GEN) below are special cases of the rule (GEN-SUB):

$$(\text{INST}) \quad \frac{A \vdash M : \forall t \sigma}{A \vdash M : \{t := \tau\}\sigma} \quad (\text{where } \tau \in \mathbf{T}_0)$$

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

Note that in the rule (INST), the type instantiated for the type variable is restricted to be a simple type; this guarantees that the result of the substitution is a positive type.

Finally, the following version of the application rule (APP) is implied by our rules (by an induction on typing derivations):

$$(\text{APP'}) \quad \frac{A \vdash M_1 : \sigma_1 \to \sigma_2, \quad A \vdash M_2 : \tau}{A \vdash M_1 M_2 : \sigma_2} \quad (\text{where } \tau \leq^{\pm} \sigma_1)$$

The rules (VAR'), (APP'), (ABS), (GEN), and (INST) clearly generalize the typing rules of the system $\mathbf{P}_2$, introduced in [6]. In turn, $\mathbf{P}_2$ generalizes $\mathbf{I}_2$, the rank 2 intersection type discipline; and this generalizes the type system of ML, giving the following result.

Theorem 1 *If M is typable in ML, then its **let**-desugaring is typable in* $\mathbf{P}$.

The type system is not as powerful as the full discipline with $(\forall,\wedge)$ types, in which all (and only) the strongly normalizing terms are typable [10]. For example, the term $y(x\,3)(x\,\mathbf{true})$ is typable in our system (when extended with constants), but its β expansion $(\lambda z(y(z\,3)(z\,\mathbf{true}))x$ is not typable. Both terms are typable in the full discipline.

3 Subtype satisfaction

We show that the subtyping relation $\leq^{\pm}$ is decidable and give a subtyping algorithm.

Up until now, we have relied on some syntactic conventions to simplify our presentation (Convention 1). 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 (see for example [18]). One difference between unification and our satisfaction problems is that we work with types that go beyond simple types, but we will only consider solutions (substitutions) involving simple types. This is not the typical case with unification, and it makes our problem easier to solve.

We say substitutions S_1 and S_2 are *disjoint* if $\mathrm{dom}(S_1)$ and $\mathrm{dom}(S_2)$ are disjoint sets, and we write $S_1 \cup S_2$ for the union of two disjoint substitutions. 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 $S_1 \leq_V S_2$, if there is a substitution S_3 such that $S_2 =_V S_3 S_1$. The relation $\leq_V$ is a partial order modulo $=_V$. We omit V when it is the set of all type variables. A substitution S is *idempotent* if $S = SS$, or, equivalently, if $\mathrm{dom}(S) \cap \mathrm{rng}(S) = \varnothing$.

A $\leq^{\pm}$*-satisfaction problem* is a pair $\exists \vec{s}.P$, where P is a set whose every element is either: 1) an equality between simple types; or 2) an inequality between a $\mathbf{T}^{+}$ type and a $\mathbf{T}^{-}$ type. When $\vec{s}$ is empty $\exists \vec{s}$ may be omitted. We use π to range over $\leq^{\pm}$-satisfaction problems.

A $\mathbf{T}_0$ substitution S is a *solution* to $\exists \vec{s}.P$ if there is a substitution S' such that $S(t) = S'(t)$ for all $t \notin \vec{s}$, $S'\sigma \leq^{\pm} S'\tau$ for all inequalities $(\sigma \leq \tau) \in P$, and $S'\sigma = S'\tau$ for all equalities $(\sigma = \tau) \in P$. The (possibly empty) set of solutions to a problem π is written $\mathbf{Solutions}(\pi)$. Two problems π_1 and π_2 are *equivalent* if $\mathbf{Solutions}(\pi_1) = \mathbf{Solutions}(\pi_2)$.

Definition 1 *A substitution U is a* most general solution to π *if it satisfies the following conditions.*

- $U \in \mathbf{Solutions}(\pi)$.
- *If* $S \in \mathbf{Solutions}(\pi)$ *then* $U \leq_{\mathrm{FTV}(\pi)} S$.
- U *is idempotent.*
- $\mathrm{dom}(U) \subseteq \mathrm{FTV}(\pi)$.

We write $\mathbf{MGS}(\pi)$ *for the (possibly empty) set of most general solutions to a* $\leq^{\pm}$*-satisfaction problem* π.

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 π 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* π *away from* W if $U \in \mathbf{MGS}(\pi)$ and $W \cap \mathrm{rng}(U) = \varnothing$, and we write $\mathbf{MGS}(\pi)[W]$ for the (possibly empty) set of most general solutions to π away from W.

Lemma 1 *If* $U \in \mathbf{MGS}(\pi)[W]$ *and* $S \in \mathbf{Solutions}(\pi)$*, then* $U \leq_{W \cup \mathrm{FTV}(\pi)} S$.

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

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

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

Lemma 2 *Let* π *be a unification problem and* W *be a finite set of type variables.*

- $\mathbf{Solutions}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi)[W] = \varnothing$.
- *There is an algorithm that decides whether* π *has a solution, and, if so, returns an element of* $\mathbf{MGS}(\pi)[W]$.

Proof: See for example Snyder [18], Lemma 3.3.11. □

Theorem 2 *Every* $\leq^{\pm}$*-satisfaction problem is equivalent to a unification problem, and moreover, there is an algorithm that transforms every* $\leq^{\pm}$*-satisfaction problem into an equivalent unification problem.*

Corollary 1 *Let* π *be a* $\leq^{\pm}$*-satisfaction problem and* W *be a finite set of type variables.*

- $\mathbf{Solutions}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi) = \varnothing$ *iff* $\mathbf{MGS}(\pi)[W] = \varnothing$.
- *There is an algorithm that decides whether* π *has a solution, and, if so, returns an element of* $\mathbf{MGS}(\pi)[W]$.

$t_1 \leq t_2$	$\Rightarrow$	$\{t_1 = t_2\}$
$t \leq (\tau_1 \to \tau_2)$	$\Rightarrow$	$\exists t_1, t_2.\{\tau_1 \leq t_1, t_2 \leq \tau_2, t = t_1 \to t_2\}$ if t_1, t_2 are fresh
$(\sigma_1 \to \sigma_2) \leq t$	$\Rightarrow$	$\exists t_1, t_2.\{t_1 \leq \sigma_1, \sigma_2 \leq t_2, t = t_1 \to t_2\}$ if t_1, t_2 are fresh
$(\sigma_1 \to \sigma_2) \leq (\tau_1 \to \tau_2)$	$\Rightarrow$	$\{\tau_1 \leq \sigma_1, \sigma_2 \leq \tau_2\}$
$\sigma \leq (\tau_1 \wedge \tau_2)$	$\Rightarrow$	$\{\sigma \leq \tau_1, \sigma \leq \tau_2\}$
$(\forall t \sigma) \leq \tau$	$\Rightarrow$	$\exists t\{\sigma \leq \tau\}$ if τ is not a $\wedge$-type, and t is not free in τ

Table 4: Transformational rules for $\leq^{\pm}$-satisfaction problems

We will prove Theorem 2 by giving an algorithm that transforms any $\leq^{\pm}$-satisfaction problem into an equivalent unification problem. Corollary 1 follows by combining the transformation with any unification algorithm.

Our transformation is defined by rules of the form

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

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

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

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

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

The rules for transforming a $\leq^{\pm}$-unification problem into a unification problem are given in Table 4.

Proof of Theorem 2: We show that the rules of Table 4 constitute an algorithm for converting any $\leq^{\pm}$-satisfaction problem into an equivalent unification problem.

First, note that every rule transforms a $\leq^{\pm}$-satisfaction problem into another $\leq^{\pm}$-satisfaction problem (equalities are between simple types, inequalities are between $\mathbf{T}^{+}$ and $\mathbf{T}^{-}$ 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 every inequality matches the left-hand side of exactly one rule.

Fourth, note that repeated application of these rules must halt. This is shown by defining a metric on problems for which the rules are strictly decreasing. The size of a problem will be defined as the sum of the size of its inequalities. The size of an inequality will be defined using induction. By our last observation, it is sufficient to consider the cases given by the left-hand sides of the rules of Table 4. For the base case we set $|t_1 \leq t_2| = 1$. Every other case is simply defined so that the size of the left-hand side of a rule is greater than the sum of the inequalities appearing on the corresponding right-hand side. For example, define $|(\sigma_1 \to \sigma_2) \leq t| = |t_1 \leq \sigma_1| + |\sigma_2 \leq t_2| + 1$, and $|\sigma \leq (\tau_1 \wedge \tau_2)| = |\sigma \leq \tau_1| + |\sigma \leq \tau_2| + 1$. Types on right-hand sides either appear as syntactic subtypes on left-hand sides, or are type variables, so this is an inductive definition; and by definition it gives a decreasing metric on problems.

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

Lemma 3 (Subtyping is decidable) *The relation $\leq^{\pm}$ is decidable.*

Proof: To see whether $\sigma \leq^{\pm} \tau$, compute $U \in \mathbf{MGS}(\{\sigma \leq \tau\})$ and check to see whether U is the identity substitution. $\square$

We have focused on $\leq^{\pm}$ and **MGS** because they play a crucial role in type inference, which we describe in the next section. Decision procedures for the other subtyping relations, $\leq^{+}$ and $\leq^{-}$, can be obtained in a similar way. One complication is that the subtype satisfaction problems for the other subtyping relations are not *unitary*; that is, they do not always have a *single* most general solution. Instead, they may have a finite number of most general solutions. Thus they belong to the (well-understood) class of *finitary* problems.

Lemma 4 *For any $\sigma \in \mathbf{T}^{+}$ and $\tau \in \mathbf{T}^{+}$, if the set $\mathcal{S} = \{S \mid S(\sigma) \leq^{+} S(\tau)\}$ is nonempty, then there is a finite subset $\{S_i \mid i \in I\}$ of $\mathcal{S}$ such that for all $S \in \mathcal{S}$, $S_i \leq S$ for some $i \in I$.*

For any $\sigma \in \mathbf{T}^{-}$ and $\tau \in \mathbf{T}^{-}$, if the set $\mathcal{S} = \{S \mid S(\sigma) \leq^{-} S(\tau)\}$ is nonempty, then there is a finite subset $\{S_i \mid i \in I\}$ of $\mathcal{S}$ such that for all $S \in \mathcal{S}$, $S_i \leq S$ for some $i \in I$.

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

Convention 2 (Non-interference) *Whenever $U \in \mathbf{MGS}(\pi)$ occurs in any mathematical context, we assume that U is chosen so that it does not interfere with "current" type variables, that is, $U \in \mathbf{MGS}(\pi)[W]$ where $W \cup \mathrm{FTV}(\pi)$ is the set of type variables present in the context.*

$\mathrm{PP}(M) =$
If $M = xM_1 \cdots M_n$
then let $\langle A_1, \sigma_1 \rangle = \mathrm{PP}(M_1), \ldots, \langle A_n, \sigma_n \rangle = \mathrm{PP}(M_n)$
  t be a fresh type variable
 in $\langle A_1 + \cdots + A_n + \{x : \sigma_1 \to \cdots \to \sigma_n \to t\}, t \rangle$
else if $M = \lambda x N$
then let $\langle A, (\forall \vec{s} \sigma) \rangle = \mathrm{PP}(N)$
 in if $x \notin \mathrm{dom}(A)$
  then let t be fresh
   in $\langle A, (\forall t \vec{s} (t \to \sigma)) \rangle$
  else $\langle A \backslash x, \mathrm{Gen}(A \backslash x, A(x) \to \sigma) \rangle$
else if $M = (\lambda x M_0) N M_1 \cdots M_n$
then let $\langle A_1, \forall \vec{t} ((\bigwedge_{i \in I} \tau_i) \to \sigma) \rangle = \mathrm{PP}(\lambda x M_0 M_1 \cdots M_n)$
  $\langle A_i, \tau_i' \rangle = \mathrm{PP}(N)$ for all $i \in I$
  $U \in \mathbf{MGS}(\{\tau_i' \leq \tau_i \mid i \in I\})$
  $A = U(A_1 + \sum_{i \in I} A_i)$
 in $\langle A, \mathrm{Gen}(A, U\sigma) \rangle$.

Table 5: The type inference algorithm for **P**

4 Type inference and principal pairs

We now show that the type system has a notion of principal type, and we show that type inference is decidable. First we state some necessary definitions.

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

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

We write $A' \leq^- A$ iff $x \in \mathrm{dom}(A) \Rightarrow x \in \mathrm{dom}(A')$ and $A'(x) \leq^- A(x)$. We write $\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle$ iff $A' \leq^- A$ and $\sigma \leq^+ \sigma'$.

A pair $\langle A, \sigma \rangle$ of a type environment and a type is called simply a *pair*. Two pairs $\langle A_1, \sigma_1 \rangle$ and $\langle A_2, \sigma_2 \rangle$ are *disjoint* if their free type variables are disjoint. An *acceptable pair of a term M* is a pair $\langle A, \sigma \rangle$ such that $A \vdash M : \sigma$ is derivable. We write $\mathrm{AP}(M)$ for the set of acceptable pairs of M. The application of substitutions is extended to pairs in the following way: $S\langle A, \sigma \rangle = \langle SA, S\sigma \rangle$. A pair $\langle A, \sigma \rangle$ is a *principal pair for M* if $\langle A, \sigma \rangle \in \mathrm{AP}(M)$, and for any other pair $\langle A', \sigma' \rangle \in \mathrm{AP}(M)$, there is a substitution S such that $S\langle A, \sigma \rangle \leq \langle A', \sigma' \rangle$.

Table 5 defines the algorithm PP that takes as input a term M, and either produces a (type environment,type) pair as output, or fails.

Theorem 3 (Soundness) *If $\langle A, \sigma\rangle \in \mathrm{PP}(M)$, then $\langle A, \sigma\rangle \in \mathrm{AP}(M)$.*

Theorem 4 (Principal pairs) *If $\langle A, \sigma\rangle \in \mathrm{AP}(M)$, then there is a pair $\langle A', \sigma'\rangle \in \mathrm{PP}(M)$ and a substitution S such that $S\langle A', \sigma'\rangle \leq \langle A, \sigma\rangle$.*

Proofs have been omitted due to lack of space, but they appear in an extended version of the paper, available from the author.

Note, when the algorithm PP is used on a normal form M, only the first two cases, $M = xM_1 \cdots M_n$ and $M = \lambda xN$, are ever needed. These cases can never fail: the only way that $\mathrm{PP}(M)$ can fail is when no solution is found by **MGS** in the case $M = (\lambda xM_0)NM_1 \cdots M_n$. This gives us the following result.

Theorem 5 *All normal forms are typable in* **P**.

5 Recursive definitions

The rule of *polymorphic recursion* was proposed as an extension to ML:

$$(\text{REC-ML}) \quad \frac{A \backslash x \cup \{x : \sigma\} \vdash M : \sigma}{A \vdash (\mu xM) : \sigma}$$

In this rule, σ is any ML type scheme (quantified simple type). The typability problem for ML extended with this rule is undecidable [7, 5]. A similar rule can be added to the rank 2 intersection type system—the only difference is that σ ranges over finite intersections of simple types. It is not known whether typability in the resulting system is decidable.

Note that in both cases, there is a violation of the polarity discipline we are advocating: in the extension of ML, the rule uses a positive type in the type environment; and in the extension of $\mathbf{I}_2$, there is a negative derived type. This motivates adding the following rule to our type system:

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

For example, we can derive $\mu w.(\lambda x(\lambda y(\lambda zz)))(w\,3)(w\,\mathbf{true}) : \forall t(t \to t)$ using the rule (REC). In order to handle the vacuous case of a recursive definition (μwM) where w is not free in M, we also have the rule

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

The rule (REC-VAC) is necessary to type, e.g., $\mu w(\lambda x.xx) : \forall s,t.s \wedge (s \to t) \to t$. In order to use the rule (REC) in this case, we would intuitively need a type $\tau_1 \to \tau_2 \in \mathbf{T}^-$ such that $\forall s,t.(s \wedge (s \to t)) \to t \leq^{\pm} \tau_1 \to \tau_2$. There is no such type, because there is no type $\tau_2 \in \mathbf{T}^-$ that is an upper bound for every simple type.

Type inference for the system extended by the rules (REC) and (REC-VAC) is decidable, using the following extension of $\mathrm{PP}(M)$:

- If $M = (\mu x N)$, and $\langle A, \sigma \rangle \in \mathrm{PP}(N)$, we consider two cases.
  - If $x \notin \mathrm{dom}(A)$, then $\langle A, \sigma \rangle \in \mathrm{PP}(\mu x N)$.
  - If $x \in \mathrm{dom}(A)$, $A(x) = \tau$, and $U \in \mathbf{MGS}(\{\sigma \leq^{\pm} \tau\})$, then

$$\langle UA \backslash x, \mathrm{Gen}(UA \backslash x, U\sigma) \rangle \in \mathrm{PP}(\mu x N).$$

The rules (REC) and (REC-VAC) were first introduced in a rank 2 restriction of **P** [6].

6 Related work and future directions

Local Type Inference. Pierce and Turner [12] describe a strategy for type inference that seeks to reconstruct missing arguments of type applications in a term by examining adjacent nodes of the syntax tree of the term. In particular, if x has polymorphic type, then the Pierce-Turner algorithm will reconstruct the type argument for a term application xM. This corresponds to our use of "manifest" arguments in the rule (VAR-APP) and the matching case of our inference algorithm. However, the Pierce-Turner system does not employ intersection types, an important distinction. Inference algorithms for systems that rely exclusively on type quantification for polymorphism usually work in a top-down fashion. Consider, for example, how Hindley-Milner type inference handles **let** expressions:

$$\mathbf{let}\ x = M\ \mathbf{in}\ N$$

In the Hindley-Milner system, x can be used polymorphically in N, but the type inference algorithm must construct the type scheme of M before examining N. The Pierce-Turner algorithm works the same way for polymorphic x's. One consequence is that their algorithm cannot infer a type for a term $\lambda x M$ where x is used polymorphically in M.

Simple intersection types. Coppo and Giannini defined a restriction of the combined system of intersection types and type quantification [3]. Their system has a notion principal type and decidable type inference, and, like **P**, can type all pure normal forms. However, their type system cannot type all terms typable in rank 2 of the intersection type discipline; so, some **P**-typable terms are not typable in their system.

"Turning the crank." We can considerably strengthen **P** by incorporating term evaluation into the typing rules and inference algorithm. The idea is to extend judgments to derive not only the type of a term, but also its normal form: judgments will have the form $A \vdash M \Rightarrow M' : \tau$, where M' is the normal form of M, and τ is its type. This will help us type more terms because the type inference algorithm for **P** can deduce a more general type for the normal form of some terms than the unnormalized terms. The following informal rule captures the essential idea:

$$\frac{A \vdash \lambda x M \Rightarrow M' : \sigma \to \tau \quad A \vdash N \Rightarrow N' : \sigma \quad A \vdash \mathrm{NF}(M'N') : \tau'}{A \vdash (\lambda x M)N \Rightarrow \mathrm{NF}(M'N') : \tau'}$$

Intuitively, once we have determined that the application $(\lambda x M)N$ is typable, we can "turn the crank" of evaluation to normal form, and apply type inference to the result to get a more general typing. Type checking and inference remain decidable because, as in many type systems, typability implies strong normalization.

The author first saw this trick at work in MacQueen and Tofte's type system for higher order modules [9].

References

[1] M. Coppo and M. Dezani. A new type-assignment for lambda terms. *Archiv für Math. Logik*, 19:139–156, 1978.

[2] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Principal type schemes and λ-calculus semantics. In Seldin and Hindley [17], pages 535–560.

[3] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In J.-C. Raoult, editor, *17th Colloquium on Trees in Algebra and Programming*, volume 581 of *Lecture Notes in Computer Science*, pages 102–123. Springer-Verlag, 1992.

[4] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, 2nd *Scandinavian Logic Symp.*, pages 63–92. North-Holland Publishing Co., 1971.

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

[6] Trevor Jim. What are principal typings and what are they good for? In *Conference Record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

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

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

[9] David MacQueen and Mads Tofte. A semantics for higher-order functors. In *5^{th} European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, April 1994.

[10] I. Margaria and M. Zacchi. Principal typing in a $\forall\wedge$-discipline. *Journal of Logic and Computation*, 5(3):367–382, June 1995.

[11] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[12] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL '98: 25^{rd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, 1998.

[13] Garrel Pottinger. A type assignment for the strongly normalizable λ-terms. In Seldin and Hindley [17], pages 561–577.

[14] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium: Proceedings, Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[15] S. Ronchi Della Rocca and B. Venneri. Principal type schemes for an extended type theory. *Theoretical Computer Science*, 28(1,2):151–169, January 1984.

[16] Patrick Sallé. Une extension de la theorie des types en λ-calcul. In G. Ausiello and C. Böhm, editors, *Automata, Languages and Programming: Fifth Colloquium*, volume 62 of *Lecture Notes in Computer Science*, pages 398–410. Springer-Verlag, July 1978.

[17] J.P. Seldin and J.R. Hindley, editors. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[18] Wayne Snyder. *A Proof Theory for General Unification*, volume 11 of *Progress in Computer Science and Applied Logic*. Birkhäuser, 1991.

[19] J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98, 1999.