

A Polar Type System

Trevor Jim
AT&T Labs

— *presented by* —
Assaf Kfoury

These are slides of a talk presented at the Workshop on Intersection Types and Related Systems (ITRS '00), with accompanying notes.

The title is “A Polar Type System,” and it describes work by Trevor Jim (AT&T Labs). [The work was done at MIT.]

The talk was given by Assaf Kfoury (Boston University), on Saturday, 15 July 2000 at the University of Geneva, Switzerland.

Summary

Partial type inference for \forall, \wedge

Complete inference for System P

- Stronger than ML
- \forall, \wedge at unbounded depth

Key idea/intuition: **POLARITY**

We describe a partial type inference algorithm for the system with universal quantification and intersection types. To understand the algorithm better, we construct a type system (System P) for which the algorithm is complete. (Helps to compare to other type systems, etc.) System P can type more terms than ML [let-desugaring] and allows quantifiers and intersections at unbounded depth in types. The key idea behind the algorithm and type system is an intuition about the role of *polarity* in type inference.

Polarity in Inference

Goal of type inference: find
most general type

$$\sigma \rightarrow \tau$$

So, *minimize* requirements σ
and, *maximize* capabilities τ

Why is polarity important in type inference? Algorithms work by divide-and-conquer: infer types of pieces, combine to get type of whole. Therefore, must get the *most general* type of each piece, or the whole approach falls apart.

Consider $\sigma \rightarrow \tau$. This is a contract, it says “supply me with a σ and I will give you a τ .” The σ in the negative position is a *requirement* that must be fulfilled before the *capability* τ can be used. To get the most general type, you should try to minimize requirements and maximize capabilities.

This is completely obvious (not new) and satisfied by just about every inference algorithm out there.

Polarity and \forall, \wedge

How to type $\omega \equiv \lambda x.xx$?

- $(\forall t.t) \rightarrow (\forall t.t)$
- $(\forall t.t \rightarrow t) \rightarrow (\forall t.t \rightarrow t)$
- $(s \wedge s \rightarrow t) \rightarrow t$
- $\forall s, t.(s \wedge s \rightarrow t) \rightarrow t$

Although the intuition is obvious, we are going to apply it in what appears to be a novel way: we will use it to *design* a type inference algorithm and type system. We take as given that we want to use \forall, \wedge . Now consider our favorite term, ω .

The first type might be used in System F: $(\forall t.t) \rightarrow (\forall t.t)$. The problem is, the requirement is too strong, very few terms have type $(\forall t.t)$. We could weaken the requirement (resulting in the second type) but the capability weakens as well. Consequence: no (good) notion of most general type for this term.

Things are better in the intersection type system, where there is a principal type. But the notion of “principal” is overly complicated.

Best of all is the combined discipline, which has principal types and they are simpler than in the intersection system. *Of course*, \forall appears at positive positions and \wedge appears at negative positions in principal types. This maximizes (\forall , infinite intersection) capabilities, minimizes (\wedge , finite intersection) requirements.

Algorithm, I

To infer type of MN :

1. Infer $M : \sigma \rightarrow \tau$
2. Infer $N : \sigma'$
3. Find solution $S \models \sigma' \leq \sigma$
4. Conclude $MN : S(\tau)$

OK, let's apply our intuition to type inference. There is really only one interesting case to type inference: application (described informally here). It matches up requirements and capabilities.

We begin by running our inference algorithm on the pieces, M and N . This gives us the most general types, $\sigma \rightarrow \tau$ and σ' . (The outputs of the algorithm are shown in boxes.) Naturally, \forall 's will appear in positive positions and \wedge 's in negative positions in these types.

Now we need to fit the pieces together: we want a solution to $\sigma' \leq \sigma$. (This is the *subtype satisfaction* problem.) As usual, we want a *best* solution, and moreover, we want it to satisfy our *polarity intuition*. In particular, we want $S(\tau)$ to have \forall 's in positive positions only, and \wedge 's in negative positions only.

We take the easy way out: we instantiate type variables to simple types only. (For both INST rule and S here!) Then, there is a best solution S , and $S(\tau)$ will obey our polarity intuition.

Algorithm, II

To infer the type of $y \text{ id}$:

1. $\text{id} : \forall u. u \rightarrow u$
2. $y : (\forall u. u \rightarrow u) \rightarrow v \vdash y \text{ id} : v$

If we only consider solutions using simple types, how do we get quantifiers and intersections at arbitrary depth, as we claimed?

We use a kind of “local type inference” (see Pierce and Turner for comparison). To infer the type of MN , we look more closely and type $x M_1 \cdots M_n$ instead.

Here we first infer the type of the argument id , then record an assumption about the head of the application, y : it takes an argument of the argument type to a fresh type variable v . (No subtype satisfaction is needed if the head of the application is a variable.)

Note, the assumption is a negative type and the conclusion is a positive type. Thus when we abstract over y we’ll get a positive type...

Algorithm, III

To infer type of $(\lambda y.y \text{ id})\omega$:

1. $(\lambda y.y \text{ id})$:

$$\forall v.((\forall u.u \rightarrow u) \rightarrow v) \rightarrow v$$

2. ω : $\forall s, t.(s \wedge s \rightarrow t) \rightarrow t$

3. $S \models \forall s, t.(s \wedge s \rightarrow t) \rightarrow t$
 $\leq (\forall u.u \rightarrow u) \rightarrow v$

We still need subtype satisfaction if the head of the application is a lambda abstraction. Here we use the example from the previous slide: we abstract over the variable y . By the usual (ABS) and (GEN) rules we get the type shown here.

The argument ω has the principal type shown (we won't go through the details).

Now to infer the type of the application we have to solve the subtype satisfaction problem shown. (This is just an instance of the slide Algorithm, I.)

Subtype satisfaction

Convert $\sigma \leq \tau$ to an equivalent unification problem. Key rules:

- $\sigma \leq (\tau_1 \wedge \tau_2)$
 $\Rightarrow \sigma \leq \tau_1, \sigma \leq \tau_2$
- $(\forall t \sigma) \leq \tau \Rightarrow \exists t. (\sigma \leq \tau)$

We solve subtype satisfaction by transforming the problem to an equivalent unification problem (it has the same set of solutions). The key rules involve intersections and quantifiers.

To solve an inequality with an intersection on the right, solve two simpler inequalities. (A source of exponential blowup.)

To solve an inequality with a quantifier on the left, show that there is a **SIMPLE** type that can be instantiated for the variable to solve the simpler inequality. (Recall instantiation is restricted to simple types!)

Examples

$(\lambda y.y \text{ id})\omega : \forall t.t \rightarrow t$

$((\lambda x.\lambda y.yx) \text{ id})\omega$ not typable.

The algorithm succeeds on the first term. Notice that the normal form of the term is the identity, so it is not surprising that principal type is the type of the polymorphic identity function.

The second term, on the other hand, does not type. It is a β -expansion of the first term. It does not type because the variable y has “lost” the information that it will be applied to the polymorphic identity. This information was crucial in allowing ω to be the actual argument for the formal y .

System P

$$\frac{A \vdash M_i : \tau_i}{A \vdash xM_1 \cdots M_n : \tau}$$

where

$$A(x) \leq^- \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$$

and τ is a simple type.

System P is a type system designed after the type inference algorithm just introduced. The algorithm is complete for System P, hence System P helps explain the algorithm and relate it to other systems.

The key rule lets us type applications of variables specially, and it is given here. It would be a derived rule in the combined type system, but must be handled specially here due to polarities.

Note that every derived type (the τ_i) is positive. Therefore the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ is negative. That's why we don't have a judgment $A \vdash x : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$. Instead, we only allow x to assume this type when it is *syntactically evident* that it is applied to arguments of the right type.

Polymorphic Recursion

$$\frac{A, x : \sigma \vdash M : \sigma}{A \vdash \mu x.M : \sigma}$$

$$\frac{A, x : \sigma \vdash M : \tau}{A \vdash \mu x.M : \tau} \quad \tau \leq \sigma$$

We discuss polymorphic recursion in passing.

The first rule is the usual style of polymorphic recursion. It interacts badly with polymorphism: in ML inference becomes undecidable. Notice that it violates the polarity discipline: the type σ appears both negatively (in the type environment) and positively.

A version compatible with the polarity discipline is given below. It allows a negative assumption (in the environment) to be discharged by a positive conclusion via the subtyping relation.

The importance of this is, it gives more support for my intuition:
1) Polymorphic recursion violates polarity intuition and is undecidable
2) My version that obeys polarity intuition is decidable and (slightly?) more powerful than simple recursion.

AND it indicates a general technique for other type systems/static analyses to pursue.

“Turning the Crank”

If a type system satisfies

- inference is decidable
- M typable \Rightarrow SN(M)
- $\text{Type}(\text{NF}(M)) \leq \text{Type}(M)$

then build a stronger system
with decidable inference

Finally, we give a general recipe for strengthening type systems. It is not particular to system P; see for example MacQueen and Tofte. However we will see it is similar to the idea in System P of using surrounding clues to infer stronger types (as in the variable application rule). [It uses reduction to get MORE clues.]

Note, we may wish to use the idea with other forms of reduction. For example, use β I-reduction if erasure is bothersome. To handle recursion, don't do unfolding.

The Idea

Design the cranked system so

$$\text{Type}(M) = \text{Type}(\text{NF}(M))$$

Naive algorithm:

1. Reduce to NF
2. Use non-cranked inference

Sounds good, but, reduction to NF might not terminate.

Key Case: Application

To infer type of MN :

1. Infer $M : \sigma \rightarrow \tau$
2. Infer $N : \sigma'$
3. Check $\sigma' \leq \sigma$ (so $\text{SN}(MN)$)
4. Infer $\text{NF}(MN) : \tau'$

So we have to go carefully.

IF we infer types of function and argument separately, and typability guarantees SN, THEN we can β -reduce each and get a better type for each. (So lines 1 and 2 involve β -reduction.)

Then if their types match up, they are SN when applied. Reduce to NF and infer the type of the result.

(This is just an informal explanation. The details will differ for each type system. I usually use judgments of the form $A \vdash M \Rightarrow M' : \tau$ where the term M' on the right of the \Rightarrow is the normal form of M . Thus inference/checking and reduction happen in an intertwined fashion.)

Cranking Simple Types

$(\lambda x.(\lambda y.x)(xz))z$ not typable

BUT: $(\lambda x.(\lambda y.x)(xz))$ typable

its NF is $(\lambda x.x)$

$(\lambda x.x)z$ typable

SO: $(\lambda x.(\lambda y.x)(xz))z$ typable

by turning the crank

In the original term, one step of reduction produces a self-application of z , so the term is not simply-typable.

If the function is reduced first, however, the self-application goes away and the term is simply-typable.

Hence the term is typable in the cranked system.

Cranking System P

$((\lambda x. \lambda y. yx) \text{ id})\omega$ not typable

BUT: $(\lambda x. \lambda y. yx) \text{ id}$ typable

its NF is $(\lambda y. y \text{ id})$

$(\lambda y. y \text{ id})\omega$ typable

so $((\lambda x. \lambda y. yx) \text{ id})\omega$ typable by
turning the crank in P

Things are more interesting in System P. In this example there is no erasure. Instead, reduction makes the application of y to the polymorphic identity manifest.

Open Problems, I

Better algorithms for subtype satisfaction (e.g., remove limitation to simple types)

Applications to other static analyses

If/when we find a way to solve subtype satisfaction that produces more general solutions (removing the restriction to simple types) and stays within our polarity discipline, we can just plug it in to get a more powerful type system/inference algorithm.

Type systems are closely related to other static analyses (e.g., control flow analyses) and our techniques may be applicable to them.

Open Problems, II

Is P type sound?

What is a *denotational* model of “turning the crank” ?

What is the complexity of checking/inference?

Since P lives inside of a type sound system, type soundness of P is not a big concern. (But I conjecture P is type sound.)

Obviously the model of the simply typed lambda calculus is not going to work for the cranked system. We don't even know if the notion is sensible without a denotational model.

The complexity of checking or inference for “turning the crank” is clearly beyond elementary recursive. For System P alone, one can try to study a ranked hierarchy. Rank 2 is the same as ML, I conjecture that each step up in the hierarchy costs an additional exponential. (Check out the subtype satisfaction algorithm to see the source of the exponential, and get an upper bound.)