Type Inference for ML

Tony Garnock-Jones <tonyg@ccs.neu.edu>

February 29, 2012

1 Typing without quite so much typing

Writing programs in System F is certainly possible, and the type system gives us nice benefits such as stuck-freedom. But the price we pay is in the notation: annotating programs with the types the programmer has in mind for them is tedious, difficult, brittle, and notationally heavy. Wouldn't it be nice if the system could permit us to write programs using an untyped notation, and have it somehow discover the types we have in mind? Looking at it from a Curry-Howard perspective, the program we'd write would be a proof; we'd want the system to discover the proposition the programmer had in mind.

Informally, type inference (also known as *type reconstruction*; I will be using the two terms interchangeably) is just this process of analyzing a program written using untyped notation and discovering a sensible type for it. The challenges are in producing a type specific enough to continue to guarantee stuck-freedom but general enough to be usable in all the contexts the programmer has in mind.

In these notes, I'll present two approaches to type inference for the language ML: a modular constraint-based approach, where terms are analyzed to produce constraint sets, which are subsequently solved to produce *substitutions* which give rise to *principal types* for the program; and a historically-earlier approach which interleaves the processes of constraint generation and solution. I'll be following [Krishnamurthi(2007)] and [Wand(1987)] for the development of the constraint-based approach, fleshing it out with details from chapter 22 of TAPL[Pierce(2002)] and section V of ATTAPL[Pierce(2005)]; and for algorithms W and J I'll be primarily following [Damas and Milner(1982)], occasionally referring to [Milner(1978)].

2 Defining the problem

Theorem 1. *(Hindley's Theorem.) It is decidable whether a term of the untyped lambda calculus is the image under type-erasing of a term of the simply typed lambda calculus.*

Proof. Given (constructively) in [Wand(1987)]. □

From this, it's clear that we can find *some* type for an untyped program. But is it the *best* type? What could "best" mean in this context? Let's fix a specific language and find out.

2.1 The language ML and a simple core calculus

ML the language takes many forms, most prominently the variants Standard ML (SML), OCaml, and F#. Here we'll initially be following [Damas and Milner(1982)], dealing with ML-the-calculus, a drastically simplified core language that gets to the heart of the type reconstruction problem.

Terms in ML-the-calculus are written *without* type annotations:

$$\begin{array}{rcll} e & ::= & x & \\ & | & c & \text{(constants)} \\ & | & \lambda x.e & \\ & | & e\,e & \\ & | & \textbf{let } x = e \textbf{ in } e & \end{array}$$

The straightforward CBV operational semantics is used for evaluations. While it might seem strange to include a **let** form, when it's operationally just syntactic sugar for an immediately-applied λ-term, treating it as a core expression brings a significant benefit to type reconstruction, as we will see below.

The types we will be assigning to terms are as follows:

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(type variables; we also use } \beta \text{ in places)} \\ & | & B & \text{(base types)} \\ & | & \tau \to \tau & \end{array}$$

Type *schemes* include universal quantifiers, which act as binders for type variables:

$$\sigma \quad ::= \quad \forall \alpha_1, \alpha_2, \ldots, \alpha_n.\tau \qquad (n \geq 0)$$

A type can be seen as just a type scheme without variables: $\tau = \forall \emptyset.\tau$, if you like.[1] (This is the case when $n = 0$, and is why I wrote $n \geq 0$ rather than $n > 0$ in the definition of σ.)

It's important to note here that type variables α are *monomorphic*, and may only be instantiated with types, not type schemes. We'll see why later on.

Type environments Γ mapping variables to type schemes σ and substitutions S mapping type variables to types τ are defined in the usual way. We lift substitutions pointwise over type environments, $S\Gamma = \{x : S\sigma | x : \sigma \in \Gamma\}$.

The metafunction $ftv(\tau)$ computes the set of free type variables in τ, namely, all the type variables in τ (since there are no binders available in the syntax of types). We overload the notation, providing metafunction $ftv(\Gamma)$, which computes the set of free type variables in Γ, namely

$$\begin{array}{rcl} ftv(\cdot) & = & \{\} \\ ftv(\Gamma, x : \forall \alpha_1, \ldots, \alpha_n.\tau) & = & ftv(\Gamma) \cup (ftv(\tau) \backslash \{\alpha_1, \ldots, \alpha_n\}) \end{array}$$

[1] This definition is attributed to Hindley in [Mobarakeh(2009)].

2.2 Type judgements for ML-the-calculus

$$\frac{}{\Gamma \vdash c : B}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x.e \ : \ \tau' \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau}$$

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots, \alpha_n.\tau' \quad \tau = [\beta_i / \alpha_i]\tau'}{\Gamma \vdash x : \tau} (\beta_i \text{ fresh})$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : (\forall \alpha_1, \ldots, \alpha_n.\tau') \vdash e_2 : \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \ : \ \tau} (\{\alpha_1, \ldots, \alpha_n\} = ftv(\tau') \backslash ftv(\Gamma))$$

Example. We will work through a typing for **let** $id = \lambda x.x$ **in** $id\, id$.

These rules are far from being algorithmic! In particular, that τ' is freely chosen in the λ rule gives us an infinite number of possible proof trees even for such a simple term as $\lambda x.x$.

To start to pin things down toward a single "best" type, we'll need to define which of these infinite trees we want.

2.3 Principal types

In [Wand(1987)], Wand proves Hindley's theorem by giving an algorithm that when given a term in an untyped calculus produces not only an equivalent typed term but also a *principal type* for that term. A principal type is exactly the "best" type for the term: the type that under *all* substitutions remains a valid type for the term.

Definition 2. (Principal type.) A type τ is called a *principal type* for a (closed) term e iff

1. $\cdot \vdash e : \tau$ and
2. $\forall \tau'. \cdot \vdash e : \tau' \implies (\exists S. \tau' = S\tau)$

That is to say, e can be given type τ and for all other types τ' that can be given to e, there exists some substitution that converts τ into τ'. The type τ is in this sense the *least specific* type for e.

It should be clear that principal types for a term form an equivalence class with respect to alpha conversion of type variables, so let's start to operate modulo alpha-equivalence and say that *"the"* principal type for a term is a member of this equivalence class, if any such members exist.

3 Constraint-based Type Inference

As we saw in the example above, the problem with the free choice of types in the declarative type judgements for ML-the-calculus is that we are forced to "guess" a type assignment that's "likely" to be correct. The judgements themselves don't offer any guidance to us. It's as if, when it comes time to assign a type to a variable binding, we need information only available at the use-site of the variable,

which of course we haven't examined yet because we are so strictly following the syntax of the term involved.

Constraint-based type inference algorithms split the process of reconstruction into two parts: in a first, syntax-directed phase, they introduce a large number of type variables and build up a set of constraints over those variables that expresses the minimum necessary side-conditions on instantiations of the variables. Second, they process the constraint set, looking for a substitution that respects every constraint.

By separating the process of analysing the term's syntax from the process of computing the types for each part, we *postpone* the decision of exactly which type a variable binding should have until *all* the relevant information is available, in the second phase.

A second, and very important, benefit of taking a constraint-based approach is that the algorithm then becomes *modular* in the exact constraint system used: by enriching the language of constraints (and the corresponding constraint-solving algorithm) we can infer types for richer type systems. As type systems become increasingly sophisticated, it becomes increasingly more difficult to extend pure syntax-directed approaches to type reconstruction to match.

[Wand(1987)] presents just such a modular constraint-based algorithm for type inference, and then gives an example use of the system for reconstructing types for the simply-typed lambda calculus. In this section, we'll start from Wand's paper before moving on to a closely-related system that can express full let-polymorphism as seen in ML-the-calculus.

3.1 Algorithm Skeleton

The algorithm as presented (almost verbatim) in [Wand(1987)] is as follows:

Input. A term e_0.

Initialization. Set $E = \emptyset$ and $G = \{(\Gamma_0, e_0, \alpha_0)\}$, where α_0 is a fresh type variable and Γ_0 maps the free variables of e_0 to other distinct fresh type variables.

Loop Step. If $G = \emptyset$, halt and return E. Otherwise, choose a subgoal (Γ, e, τ) from G, remove it from G, and add to E and G new verification conditions and subgoals as specified in an *action table* for the language.

Each triple (Γ, e, τ) is a *hypothesis* about the input term that we hope is true. Once the loop ends, E will contain a set of constraints that, if solvable, when solved will produce a substitution S such that $S\Gamma \vdash e : S\tau$. In particular, that S leads us to $S\Gamma_0 \vdash e_0 : S\alpha_0$. If E turns out not to be solvable, we know that the term cannot be well-typed.

Recursive presentation. Wand presents the algorithm skeleton in an iterative style, maintaining a worklist of hypotheses to process and an accumulator for constraints so far generated. It can also be

presented as a recursive function depending on the action table:[2][3]

$$\begin{array}{rcl} \mathit{constraints} & : & \Gamma \times e \times \tau \to \{C\} \\ \mathit{action} & : & \Gamma \times e \times \tau \to \{C\} \times \{\Gamma \times e \times \tau\} \end{array}$$

$$\begin{array}{rcl} \mathit{constraints}(\Gamma, e, \tau) & = & E \cup \left(\bigcup_{(\Gamma', e', \tau') \in G} \mathit{constraints}(\Gamma', e', \tau') \right) \\ & & \text{where } (E, G) = \mathit{action}(\Gamma, e, \tau) \end{array}$$

3.2 Constraints and Action Table for the STLC

The action table Wand gives in his paper covers the simply-typed lambda-calculus, but no more. In our setting, we require a little more because ML-the-calculus includes let-polymorphism. Before getting to that, it's worth looking at the constraints and action table involved in inferring types for the STLC.

Constraints. Constraints are of the form $\tau = \tau'$. It's important to note that "=" isn't to be read as equality but as a constraint on unification: the constraint itself reads as something like "in order for the system to have a solution, τ must be unifiable with τ'."

Solving Constraints (Robinson's Unification Algorithm). The constraints produced by the algorithm's main loop as it halts are fed to the following routine, which produces a substitution as its output:

$$\begin{array}{rcl} \mathit{unify} & : & \{C\} \to S \end{array}$$

$$\begin{array}{rcl} \mathit{unify}(\emptyset) & = & [\,] \\ \mathit{unify}(\{\tau = \tau\} \cup C) & = & \mathit{unify}(C) \qquad \text{(structurally equal)} \\ \mathit{unify}(\{\alpha = \tau\} \cup C) & = & \mathit{unify}([\alpha \mapsto \tau]C) \circ [\alpha \mapsto \tau] \text{ when } \alpha \notin \mathit{ftv}(\tau) \\ \mathit{unify}(\{\tau = \alpha\} \cup C) & = & \mathit{unify}([\alpha \mapsto \tau]C) \circ [\alpha \mapsto \tau] \text{ when } \alpha \notin \mathit{ftv}(\tau) \\ \mathit{unify}(\{\tau_1 \to \tau_1' = \tau_2 \to \tau_2'\} \cup C) & = & \mathit{unify}(\{\tau_1 = \tau_2, \tau_1' = \tau_2'\} \cup C) \\ \mathit{unify}(\{\tau_1 = \tau_2\} \cup C) & = & \bot \qquad \text{(failure in all other cases)} \end{array}$$

Note that $\circ$ is substitution composition. The constraint that $\alpha \notin \mathit{ftv}(\tau)$ in the variable-binding cases is called the *occurs check*, and is used to avoid inferring recursive types. See section 5.5 for a discussion of what happens if you omit the occurs check.

Action Table. We represent the action table as an action function, taking a triple from the main algorithm and returning a pair of a set of new constraints to add to our E and a set of new hypothesis

[2] It's worth pointing out that Wand has a particular constraint language in mind: sets of equations between types. For the richer constraint language explored below for ML-the-calculus, we'd want to use $\wedge$ instead of $\cup$ to accumulate our constraints. If we interpret Wand's sets-of-equations as *conjunctions* of equations, then $\wedge$ makes sense here for that constraint language too.

[3] My apologies for the ad-hoc notation for expressing the types of these functions, here and below. Where I've written $\{X\}$ I mean "a set of Xs".

triples to add to our G.[4]

$$
\begin{array}{rcl}
action & : & \Gamma \times e \times \tau \rightarrow \{C\} \times \{\Gamma \times e \times \tau\} \\
\hline
action(\Gamma, x, \tau) & = & (\{\tau = \Gamma(x)\}, \emptyset) \\
action(\Gamma, e_1\, e_2, \tau) & = & (\emptyset, \{(\Gamma, e_1, \alpha \rightarrow \tau), (\Gamma, e_2, \alpha)\}) \\
 & & \text{where } \alpha \text{ fresh} \\
action(\Gamma, \lambda x.e, \tau) & = & (\{\tau = \alpha_1 \rightarrow \alpha_2\}, \{((\Gamma, x : \alpha_1), e, \alpha_2)\}) \\
 & & \text{where } \alpha_1, \alpha_2 \text{ fresh}
\end{array}
$$

Example. Let's run Wand's algorithm on $\lambda x.\lambda y.\lambda z.(xz)(yz)$.

3.3 Constraint-based inference for ML-the-calculus

One approach to extending Wand's algorithm to take into account ML's let-polymorphism is to follow the form of the type judgements given above directly. If we do so, however, we run into a difficulty when it comes to processing **let** expressions:

$$
\begin{array}{rcl}
 & \vdots & \\
action(\Gamma, x, \tau) & = & (\{\tau = [\beta_i/\alpha_i]\tau'\}, \emptyset) \\
 & & \text{where } \Gamma(x) = \forall\alpha_1, \ldots, \alpha_n.\tau' \text{ and } \beta_i \text{ fresh} \\
 & \vdots & \\
action(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2, \tau) & = & (\emptyset, \{(\Gamma, e_1, \beta), ((\Gamma, x : (\forall\alpha_1, \ldots, \alpha_n.\beta)), e_2, \tau)\}) \\
 & & \text{where } \beta \text{ fresh} \\
 & & \text{and } \{\alpha_1, \ldots, \alpha_n\} = ftv(\tau') \backslash ftv(\Gamma) \text{ ????}
\end{array}
$$

In order to form an appropriate type scheme $\forall\alpha_1, \ldots, \alpha_n.\beta$ to associate to the **let**-bound variable x in the augmented Γ in the subgoal we would want to generate, we would need to know enough about the structure of β to be able to compute $ftv(\beta)$. But in Wand's algorithm, we strictly separate constraint generation from constraint solution, so all we know about β is that it is a fresh type variable that will eventually be instantiated with some monomorphic type!

What's required is an alternative constraint language that lets us not only specify the required equalities between types but also the places where the constraints are polymorphic. Pottier and Rémy in section V of [Pierce(2005)] use HM(X), originally published in [Sulzmann et al.(1997)Sulzmann, Odersky, and Wehr], for this purpose.[5] The key ideas are to permit *existential quantification* of type variables in constraints and, crucially, to extend universal quantifications in the type language with constraints that need to be instantiated at the same time the variables quantified over are instantiated. The extension to universal quantifiers lets us replicate constraints into each use-site of a polymorphic variable, and the addition of existential quantification lets us appropriately freshen type variables after replicating a set of constraints.

The modified universal quantification is written here $\forall\alpha\{C\}.\tau$, where α is bound in the τ but is to be substituted out at instantiation time in the constraint C,[6] so the definition of type schemes σ now becomes

[4] In Wand's paper, he uses Γ_e on the right-hand-sides of various action rules to indicate Γ restricted to the domain of the free variables in e. Such a restriction is not needed for the algorithm, but makes the proofs in the paper work.

[5] HM(X) is also readily extended to type systems involving more complex constraints like subtyping. See [Sulzmann et al.(1997)Sulzmann, Odersky, and Wehr] and ATTAPL section V for details.

[6] ATTAPL uses a slightly different notation.

$$\sigma \quad ::= \quad \forall \alpha_1, \alpha_2, \ldots, \alpha_n \{C\}.\tau \qquad (n \geq 0)$$

where constraints C are defined by

$$C \quad ::= \quad \tau = \tau' \mid \exists \alpha_1, \ldots, \alpha_n.C \mid C \wedge C$$

Generating constraints. Given our richer constraint and type languages, we can properly specify the process of constraint generation. Wand's loop skeleton and action table are replaced by a straightforward structural recursion.[7]

$[[\cdot \vdash \cdot : \cdot]]$	:	$\Gamma \times e \times \tau \to C$
$[[\Gamma \vdash c : \tau]]$	=	$\tau = B$
$[[\Gamma \vdash x : \tau]]$	=	$\tau = \Gamma(x)$
$[[\Gamma \vdash \lambda x.e : \tau]]$	=	$\exists \alpha_1 \alpha_2.([[\Gamma, x : \alpha_1 \vdash e : \alpha_2]] \wedge \tau = \alpha_1 \to \alpha_2)$
$[[\Gamma \vdash e_1\, e_2 : \tau]]$	=	$\exists \alpha.([[\Gamma \vdash e_1 : \alpha \to \tau]] \wedge [[\Gamma \vdash e_2 : \alpha]])$
$[[\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau]]$	=	$[[\Gamma, x : (\forall \alpha \{[[\Gamma \vdash e_1 : \alpha]]\}.\alpha) \vdash e_2 : \tau]]$ where all the various αs above are fresh

Example. Let's examine the constraints generated from **let** $id = \lambda x.x$ **in** $id\, id$. We're after the constraint C resulting from

$$C = [[\cdot \vdash \textbf{let } id = \lambda x.x \textbf{ in } id\, id : \alpha]]$$

Following the steps of the algorithm,

$$\begin{array}{rcl}
C & = & [[id : (\forall \beta \{[[\cdot \vdash \lambda x.x : \beta]]\}.\beta) \vdash id\, id : \alpha]] \\
\text{but } [[\cdot \vdash \lambda x.x : \beta]] & = & \exists \alpha_1 \alpha_2.([[x : \alpha_1 \vdash x : \alpha_2]] \wedge \beta = \alpha_1 \to \alpha_2) \\
\text{and } [[x : \alpha_1 \vdash x : \alpha_2]] & = & \alpha_2 = \alpha_1 \\
\text{so let } \tau_{id} & = & \forall \beta \{\exists \alpha_1 \alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\
\text{and then } C & = & [[id : \tau_{id} \vdash id\, id : \alpha]] \\
& = & \exists \alpha_3.([[id : \tau_{id} \vdash id : \alpha_3 \to \alpha]] \wedge [[id : \tau_{id} \vdash id : \alpha_3]]) \\
\text{but } [[id : \tau_{id} \vdash id : \alpha_3 \to \alpha]] & = & (\alpha_3 \to \alpha) = \tau_{id} \\
\text{and } [[id : \tau_{id} \vdash id : \alpha_3]] & = & \alpha_3 = \tau_{id} \\
\text{so then } C & = & \exists \alpha_3.((\alpha_3 \to \alpha) = \tau_{id} \wedge \alpha_3 = \tau_{id})
\end{array}$$

which, when written out in full, is

$$\begin{array}{lll}
\exists \alpha_3 & .(& (\alpha_3 \to \alpha) = \forall \beta \{\exists \alpha_1 \alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\
& \wedge & \alpha_3 = \forall \beta \{\exists \alpha_1 \alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\
&) &
\end{array}$$

[7] [Pierce(2005)] presents this differently because implementing the naive algorithm given here is inefficient. ((But I'm not sure why—is it something to do with excessive copying and substitution? Actually, looking at the worked example, it might be to do with the need for simplification before pushing a constraint under a $\forall$.))

Solving constraints. Clearly, something more powerful than Robinson's algorithm is needed here, since we are required to unify types like $\alpha_3 \to \alpha$ with types like $\forall\beta\{\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta$. A constraint solver appropriate for the job is presented in ATTAPL section V. It's a large, complex system, though, so I shall simply walk through an informal *reduction* of the constraints from the example above. Existentials lead to freshening; "unification" involving a universal quantification on one side of an equal-sign leads to instantiation of the quantified-over variables, and instantiation and release of the carried constraints into the wider context.[8]

$$\begin{array}{rcl} \text{Let } C_1 & = & (\alpha_3 \to \alpha) = \forall\beta\{\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\ \text{and } C_2 & = & \alpha_3 = \forall\beta\{\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\ \text{then } C & = & \exists\alpha_3.(C_1 \wedge C_2). \end{array}$$

First, let's simply imagine we "freshen" α_3 by leaving it alone, since it doesn't appear anywhere outside the outermost existential. Then $C = C_1 \wedge C_3$. Rewriting C_1 first of all,

$$\begin{array}{rcl} C_1 = & & \alpha_3 \to \alpha = \forall\beta\{\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\ & \Longrightarrow & (\alpha_3 \to \alpha = \alpha_3 \to \alpha) \wedge (\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \alpha_3 \to \alpha = \alpha_1 \to \alpha_2)) \\ & \Longrightarrow & (\alpha_3 \to \alpha = \alpha_3 \to \alpha) \wedge (\alpha_5 = \alpha_4) \wedge (\alpha_3 \to \alpha = \alpha_4 \to \alpha_5) \\ & \Longrightarrow & (\alpha_5 = \alpha_4) \wedge (\alpha_3 \to \alpha = \alpha_4 \to \alpha_5) \\ & \Longrightarrow & (\alpha_3 \to \alpha = \alpha_4 \to \alpha_4) \\ & \Longrightarrow & (\alpha_3 = \alpha_4) \wedge (\alpha = \alpha_4) \\ & \Longrightarrow & \alpha = \alpha_3 \end{array}$$

Then, rewriting C_2,

$$\begin{array}{rcl} C_2 = & & \alpha_3 = \forall\beta\{\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \beta = \alpha_1 \to \alpha_2)\}.\beta \\ & \Longrightarrow & (\alpha_3 = \alpha_3) \wedge (\exists\alpha_1\alpha_2.(\alpha_2 = \alpha_1 \wedge \alpha_3 = \alpha_1 \to \alpha_2)) \\ & \Longrightarrow & (\alpha_3 = \alpha_3) \wedge (\alpha_7 = \alpha_6) \wedge (\alpha_3 = \alpha_6 \to \alpha_7) \\ & \Longrightarrow & (\alpha_7 = \alpha_6) \wedge (\alpha_3 = \alpha_6 \to \alpha_7) \\ & \Longrightarrow & (\alpha_3 = \alpha_6 \to \alpha_6) \end{array}$$

Now, since $C = C_1 \wedge C_2$ we can place them together again:

$$\begin{array}{rcl} C_1 \wedge C_2 = & & (\alpha = \alpha_3) \wedge (\alpha_3 = \alpha_6 \to \alpha_6) \\ & \Longrightarrow & \alpha = \alpha_6 \to \alpha_6 \end{array}$$

and we are done, having learned that

$$\cdot \vdash \textbf{let } id = \lambda x.x \textbf{ in } id\, id : \alpha_6 \to \alpha_6$$

[8] I'm totally winging it on what happens with universal quantifiers here.

4 Damas-Milner: Algorithms W and J

We abandoned the straightforward syntax-directed approach to type reconstruction above because of the difficulty we got into trying to guess the right type to use at each variable binding in a term. But we can use the idea of principal types to let us *interleave* constraint generation and constraint solution while performing a simple structural recursion on terms. [Pierce(2002)], page 330, has this to say:

> "The idea of principal types can be used to build a type reconstruction algorithm that works more incrementally than the one we have developed here. Instead of generating all the constraints first and then trying to solve them, we can interleave generation and solving, so that the type reconstruction algorithm actually returns a principal type at each step. The fact that the types are always principal ensures that the algorithm never needs to re-analyze a subterm: it makes only the minimum commitments needed to achieve typability at each step. One major advantage of such an algorithm is that it can pinpoint errors in the user's program much more precisely."

This is actually the way type inference was first defined, in [Milner(1978)] but also in the other early presentations of the idea. It's interesting to note the tradeoff here: we get a simple, structurally-recursive definition of type reconstruction, but the cost is a lack of modularity in our reconstruction algorithm. Extending such an algorithm to richer type systems is much more difficult than in algorithms which separate constraint generation from constraint solution.

[Milner(1978)] gives *Algorithm W* for type-reconstruction for ML. It relies on Robinson's unification algorithm, and interleaves recursive calls to W with calls to $unify$ in places. As we will see, W lets us write down rules that look much closer to the declarative type judgements we initially wrote down.

The notation used in [Milner(1978)] is quite different from modern notation, and so can be difficult to follow. A clearer (and shorter) presentation of the same algorithm can be found in [Damas and Milner(1982)], and that's going to be the paper we follow in this section. It's also the paper that proved completeness for W,[9] and so gave the algorithm its modern name: the Damas-Milner Type Reconstruction Algorithm.[10]

4.1 Algorithm W

W is a function from a type environment ("assumptions" in the paper) and a term to a pair of a substitution S and a type τ.[11]

[9] Milner had already covered soundness in 1978, and had a strong suspicion that the algorithm was complete.

[10] Some will name it Hindley-Milner; others Hindley-Damas-Milner; take your pick.

[11] Be warned: there are some typographic errors in the specification of W in [Damas and Milner(1982)]. They're fairly apparent, but you can always go to [Milner(1978)] to cross-check. Here I've presented the corrected version of the rules (I hope).

$$
\begin{array}{rcl}
W & : & \Gamma \times e \to S \times \tau \\
\hline
W(\Gamma, x) & = & ([\,], [\beta_i/\alpha_i]\tau') \\
 & & \text{where } \Gamma(x) = \forall\alpha_1, \ldots, \alpha_n.\tau' \\
 & & \text{and } \beta_i \text{ are fresh} \\
W(\Gamma, e_1\, e_2) & = & (V \circ S_2 \circ S_1, V\beta) \\
 & & \text{where } (S_1, \tau_1) = W(\Gamma, e_1) \\
 & & \text{and } (S_2, \tau_2) = W(S_1\Gamma, e_2) \\
 & & \text{and } V = unify(\{S_2\tau_1 = \tau_2 \to \beta\}) \\
 & & \text{and } \beta \text{ is fresh} \\
W(\Gamma, \lambda x.e) & = & (S, S\beta \to \tau) \\
 & & \text{where } (S, \tau) = W((\Gamma, x : \beta), e) \\
 & & \text{and } \beta \text{ is fresh} \\
W(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) & = & (S_2 \circ S_1, \tau_2) \\
 & & \text{where } (S_1, \tau_1) = W(\Gamma, e_1) \\
 & & \text{and } (S_2, \tau_2) = W((S_1\Gamma, x : (\forall\alpha_1, \ldots, \alpha_n.\tau_1)), e_2) \\
 & & \text{and } \{\alpha_1, \ldots, \alpha_n\} = ftv(\tau_1) \backslash ftv(S_1\Gamma)
\end{array}
$$

4.2 Algorithm J

W is side-effect free, and takes great care to apply and compose substitutions in the right order. Because this repeated application of substitutions to terms can seriously harm the asymptotic efficiency of the reconstruction algorithm, Milner presented a more efficient (but equivalent) imperative variation on W called *Algorithm J* in [Milner(1978)]. The imperative updates to the substitution that Milner uses aren't essential to the algorithm, so here I've threaded the substitution through in the usual functional style.

$$
\begin{array}{rcl}
J & : & S \times \Gamma \times e \to S \times \tau \\
\hline
J(S,\Gamma,x) & = & (S, [\beta_i/\alpha_i]\tau') \\
 & & \text{where } \Gamma(x) = \forall\alpha_1,\ldots,\alpha_n.\tau' \\
 & & \text{and } \beta_i \text{ are fresh} \\
J(S,\Gamma,e_1\,e_2) & = & (V,\beta) \\
 & & \text{where } (S_1,\tau_1) = J(S,\Gamma,e_1) \\
 & & \text{and } (S_2,\tau_2) = J(S_1,\Gamma,e_2) \\
 & & \text{and } V = unify'(\tau_1,\tau_2 \to \beta, S_2) \\
 & & \text{and } \beta \text{ is fresh} \\
J(S,\Gamma,\lambda x.e) & = & (S_1,\beta \to \tau) \\
 & & \text{where } (S_1,\tau) = J(S,(\Gamma,x:\beta),e) \\
 & & \text{and } \beta \text{ is fresh} \\
J(S,\Gamma,\textbf{let } x = e_1 \textbf{ in } e_2) & = & (S_2,\tau_2) \\
 & & \text{where } (S_1,\tau_1) = J(S,\Gamma,e_1) \\
 & & \text{and } (S_2,\tau_2) = J(S_1,(\Gamma,x:(\forall\alpha_1,\ldots,\alpha_n.\tau_1)),e_2) \\
 & & \text{and } \{\alpha_1,\ldots,\alpha_n\} = ftv(S_2\tau_1)\backslash ftv(S_2\Gamma)
\end{array}
$$

The supporting function $unify'(\tau,\tau',S)$ extends the substitution S it is given with bindings resulting from the unification of τ with τ' in the context of S.

Notice that substitutions are only applied where absolutely necessary, namely, in order to determine the set of free type variables in the type of the bound variable in a **let**.

Example. Let's evaluate $J([\,], \cdot, \textbf{let } id = \lambda x.x \textbf{ in } id\, id)$. Call the result R.

$$\begin{array}{rcl}
R & = & J([\,], \cdot, \textbf{let } id = \lambda x.x \textbf{ in } id\, id) \\
 & & \text{(first, examine subterm } \lambda x.x) \\
J([\,], \cdot, \lambda x.x) & = & ([\,], \beta \to \beta) \\
\text{because } J([\,], x : \beta, x) & = & ([\,], \beta) \\
 & & \text{(now compute } \{\alpha_1, \ldots, \alpha_n\}) \\
ftv([\,]\, \beta \to \beta) & = & \{\beta\} \\
ftv([\,]\, \cdot) & = & \emptyset \\
\therefore \alpha_1 & = & \beta \quad \text{(so our } id \text{has type } \forall\beta.\beta \to \beta) \\
 & & \text{(now examine subterm } id\, id) \\
 & & \text{(and immediately examine the first use of } id) \\
J([\,], id : (\forall\beta.\beta \to \beta), id) & = & ([\,], \beta_1 \to \beta_1) \\
 & & \text{(which flows into examining the second use of } id) \\
J([\,], id : (\forall\beta.\beta \to \beta), id) & = & ([\,], \beta_2 \to \beta_2) \\
 & & \text{(generate a fresh } \beta\text{; call it } \beta_3) \\
unify'(\beta_1 \to \beta_1, (\beta_2 \to \beta_2) \to \beta_3, [\,]) & = & [\beta_3 \mapsto \beta_1, \beta_1 \mapsto (\beta_2 \to \beta_2)] \\
J([\,], id : (\forall\beta.\beta \to \beta), id\, id) & = & ([\beta_3 \mapsto \beta_1, \beta_1 \mapsto (\beta_2 \to \beta_2)], \beta_3) \\
 & & \text{(it remains to assemble the result of the } \textbf{let}) \\
\therefore R & = & ([\beta_3 \mapsto \beta_1, \beta_1 \mapsto (\beta_2 \to \beta_2)], \beta_3)
\end{array}$$

When we apply the substitution $[\beta_3 \mapsto \beta_1, \beta_1 \mapsto (\beta_2 \to \beta_2)]$ to β_3, we arrive at $\beta_2 \to \beta_2$, which is a sensible principal type for our term.

5 Problems and Solutions

The type inference algorithms we've examined here are at a sweet spot for simplicity combined with reasonable expressiveness. If we want to extend the systems to include richer type-system features, though, we do have to be a little careful. The line of undecidability is, in some cases, not far away; in other cases, it's easy to become unsound.

5.1 Side-effects can make polymorphic typing unsound

Let's imagine we have mutable reference cells in our language, of type $ref\ \alpha$. Our initial type environment will then include the bindings

$$\begin{array}{rcl}
box & \mapsto & \forall\alpha.\alpha \to ref\ \alpha \\
unbox & \mapsto & \forall\alpha.ref\ \alpha \to \alpha \\
setbox & \mapsto & \forall\alpha.ref\ \alpha \to \alpha \to \alpha
\end{array}$$

With the rules we've given so far, and with imperative-update semantics for $ref\ \alpha$, it's possible to write programs that go wrong.

Example. The program

$$\textbf{let } f = box\ \lambda x.x \textbf{ in } setbox\ f\ (\lambda y.(y + 5)); (unbox\ f)\ \textbf{true}$$

has type $bool$, but if you run it, it will get stuck trying to reduce the term $\textbf{true} + 5$.

The problem is that $box\,\lambda x.x$ has type $ref\,(\alpha \to \alpha)$, which the let-binding rules will generalize to $\forall\alpha.ref\,(\alpha \to \alpha)$. When the bound variable f is used later, it is instantiated with a different value for alpha at each usage: first, when the box is set, it treats f as if $\cdot \vdash f : ref\,(int \to int)$, but later, when the box is opened, it treats f as if $\cdot \vdash ref\,(bool \to bool)$, even though we just placed a function expecting an int into it.

As it happens, the original production ML system was in use for years before this bug became painful enough to notice and fix. The first attempt at fixing the type system was a complex-sounding tracking of *ref*s specifically (excluding other side-effects) in the types,[12] but a much simpler alternative, the *value restriction*, was debated on and off for years, until Andrew Wright's 1995 survey of a large body of SML code showed that almost no-one was using the extra expressiveness afforded by the complex type extension, and that almost no code would need to change if the value restriction were adopted as the official solution to the problem that mutable reference cells introduce.

The value restriction is a simple *syntactic* determination of whether a particular let-bound variable is to be given a polymorphic type or not:

- If the value on the right-hand-side of the binding is syntactically a value, e.g. a λ-term, constructor applied to values, or constant, then it is given polymorphic type.
- If it is *not* syntactically a value, including for example if it is a function call, then it is given monomorphic type.

This is enough to make our example program above fail type reconstruction: since we are calling *ref* on the right-hand-side of the binding for f, the type reconstruction algorithm assigns f the monomorphic type $ref\,(\alpha \to \alpha)$, and when f is used in the body of the let at two different types, the inconsistency is detected, and the algorithm complains that f cannot have both $ref\,(int \to int)$ and $ref\,(bool \to bool)$ as types.

Adopting the value restriction is not a completely painless fix, however. It affects pure functional programs as well as those that use mutable state. For example, it is no longer possible to use *revlists* from

$$\textbf{let}\, revlists = map\, rev\, \textbf{in}\ \ldots$$

at more than one type at once, even though to do so would be perfectly safe. In such cases, one is forced to eta-expand the right-hand-side to get a polymorphic type for *revlists*:

$$\textbf{let}\, revlists = \lambda x.\, map\, rev\, x\, \textbf{in}\ \ldots$$

5.2 Type reconstruction for impredicative polymorphism is undecidable

We've been careful so far to make sure that let-bound variables are the only ones given polymorphic types, making a big deal out of the fact that type variables are only permitted to be instantiated with types and never with type schemes. The reason for this is that type reconstruction for impredicative polymorphism is undecidable, and so we must avoid higher-order types.

The reason that let-bound variables are safe places to introduce polymorphism, while λ-bound variables are not, is that we have more information to work with in the **let** case: the expression to be bound to the variable—call it e_1—is always syntactically present in a **let**-expression, but it's rare to see analogous program terms of the form $((\lambda x.e_2)\, e_1)$. The rules inferring types for let-bound variables can exploit their immediate knowledge of the type of e_1 in deciding whether or not it's safe to give it a polymorphic type. The rules for lambda, on the other hand, limited as they are to analyzing the λ-expression on its own without knowledge of the type of the argument it will eventually be applied to, must be conservative when inferring a type for the formal parameter to the function.

[12] This is described in a little more detail on p336 of [Pierce(2002)] as "weak type variables".

There's a large body of research attempting to extend type reconstruction to full System F without crossing the line into undecidability. Often this takes the form of requiring the occasional type annotation in program terms, to help the type inferencer decide which way to go. See section 23.6 (p354) of [Pierce(2002)] for more information on this topic. It's interesting that, like so many other problems in computer science, *solving* the problem of type reconstruction for System F is undecidable, but *checking* a particular solution is decidable.

5.3 Supporting recursion

Adding recursion to ML-the-calculus in the form of a construct like **fix** $x\ e$ is quite straightforward. An interesting design question arises when we consider polymorphic recursive functions, however: what type should be assigned to x in e? Should it be polymorphic, or monomorphic? The easy and sound choice is to make x monomorphic in e, thus forbidding polymorphic recursion. Certain attempts to support polymorphic recursion have been shown to make type reconstruction undecidable: there's a little more information and some pointers toward the end of the notes in section 22.8 (p338) of [Pierce(2002)].

5.4 Don't close off too many type variables

Note the side condition on the let rule in the type judgements above. If it is instead $\{\alpha_1, \ldots, \alpha_n\} = ftv(\tau')$ (note no restriction against $ftv(\Gamma)$), things become unsound.

Example. The program

$$\lambda y.\mathbf{let}\, f = \lambda x.y\, \mathbf{in}\, \mathbf{if}\, (f\, \mathbf{true})\, \mathbf{then}\, (f\, \mathbf{true}) + 5\, \mathbf{else}\, 6$$

will have type $bool \to int$ inferred for it, but when it is run, will get stuck evaluating $y + 5$, where y has type $bool$. The problem is that the modified let rule causes f to be polymorphic in the type-variable associated with y, even though at the time the body of the outermost λ is running, it's going to have been instantiated at some fixed type.

5.5 Recursive types

Recursive types (such as $\mu\alpha.list(\alpha)$) do not actually pose a problem for Hindley-Milner-style type reconstruction algorithms! All that is required is to remove the occurs check from the unification algorithm or constraint solver. This leads to regular (infinite) types, which can be a user-interface problem, and causes problems when users make mistakes in their programs: the error messages that are produced are often pointing at locations far away from the source of the error. For more information, see the section on recursive types in section V of [Pierce(2005)].

6 History

Type inference has a long history. [Wand(1987)] and [Pierce(2002)] give a good summary of it, with references, and Pottier and Rémy provide extensive and detailed background in section V of [Pierce(2005)]. Roger Hindley wrote an account of his understanding of the history of the idea in an email to the TYPES list in 1988 [Hindley(1988)]. He writes that:

- Curry used type reconstruction informally in the 50s, perhaps even earlier, and wrote it up formally in 1967 (though it was only published in 1969). Curry's work independently develops a unification algorithm.

- Tarski is rumoured to have used a principal-type-scheme or unification algorithm in the 1920s.
- His own 1967 presentation of the idea depends on Robinson's 1965 unification algorithm.
- J. H. Morris independently developed the idea, including a reinvention of unification, in hist 1968 thesis.
- Milner's 1978 presentation of Algorithm W depends on Robinson's unification algorithm too.
- Carew Meredith was using a Hindley-like algorithm in the 1950s.

Many people independently arrived at the central ideas of type inference over the course of several decades. Hindley's 1988 email concludes with the following quip: "There must be a moral to this story of continual re-discovery; perhaps someone along the line should have learned to read. Or someone else learn to write."

References

[Damas and Milner(1982)] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–212, 1982.

[Hindley(1988)] Roger Hindley. Email to the Types mailing-list, April 1988. URL `http://www.cis.upenn.edu/ bcpierce/types/archives/1988/msg00042.html`.

[Krishnamurthi(2007)] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 2007. URL `http://www.cs.brown.edu/ sk/Publications/Books/ProgLangs/2007-04-26/`.

[Milner(1978)] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mobarakeh(2009)] S. Saeidi Mobarakeh. Type inference algorithms, February 2009. URL `http://www.win.tue.nl/ hzantema/semssm.pdf`.

[Pierce(2002)] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 978-0262162098.

[Pierce(2005)] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005. ISBN 978-0262162289.

[Sulzmann et al.(1997)Sulzmann, Odersky, and Wehr] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type Inference with Constrained Types. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, Paris, France, January 1997.

[Wand(1987)] Mitchell Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae*, 10:115–122, 1987.