There Is No Magic
Type Inference for LAM

Neil Ghani and Conor McBride

November 8, 2018

1 Introduction

Our typing rules for LAM have been given as follows:

$$\frac{}{\Gamma_0, x : \sigma, \Gamma_1 \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x \to t : \sigma \to \tau} \qquad \frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash f\ s : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{\Gamma \vdash s : \mathbb{N} \quad \Gamma \vdash t : \mathbb{N}}{\Gamma \vdash (s + t) : \mathbb{N}}$$

(Writing source code in ASCII, we can't write a proper λ for abstraction, so we tend to write \ instead. When we do the theory (in LaTeX or in HTML), we have a tendency to use the symbols we really mean.)

One should rightly be suspicious of these rules. On the one hand, if types are thought of as an *input* to be *checked*, where on earth does σ come from in the rule for application? On the other hand, if types are thought of as an *output* to be *synthesized*, where on earth does the σ come from in the rule for λ? Either way, the 'right' type is seemingly conjured from thin air!

What really happens when computers give types to LAM terms?

2 Normal Forms

If we write down the grammar of β-normal terms, something interesting happens

$$\begin{array}{rcl}
\langle normal \rangle & ::= & \lambda\langle var \rangle \to \langle normal \rangle \\
& | & \langle number \rangle \\
& | & \langle neutral \rangle \\
\langle neutral \rangle & ::= & \langle var \rangle \\
& | & \langle neutral \rangle\ \langle normal \rangle \\
& | & (\langle neutral \rangle + \langle normal \rangle) \\
& | & (\langle number \rangle + \langle neutral \rangle)
\end{array}$$

We have carefully ensured that 'neutral' terms are *stuck* for some reason — always, ultimately, that there is a free variable getting in the way of computation. The 'normal' forms are things that can be value-like — numbers or λ-abstractions — but they are prevented from standing in any place that could trigger a reduction.

We *can* give a sensible algorithmic system for typing this syntax. The trick is to treat types as an input for normals but an output for neutrals. Let us write s, t, n for normals and e, f for neutrals. Our judgement forms are

$$\Gamma \vdash \tau \ni t \qquad \Gamma \vdash e \in \sigma$$

We have stolen the set theoretic $\in$ symbol and exploited its lack of symmetry! When we *check* a type, we write the type *before* the term, because it is given in advance. Here are the checking rules.

$$\frac{\Gamma, x : \sigma \vdash \tau \ni t}{\Gamma \vdash \sigma \to \tau \ni \lambda x \to t} \qquad \frac{}{\mathbb{N} \ni n} \qquad \frac{\Gamma \vdash e \in \sigma \quad \sigma = \tau}{\Gamma \vdash \tau \ni e}$$

The third rule says that if we are checking a type τ for a neutral, first synthesize the type σ of the neutral, then test if the type you get is the type you want. But note also the first rule, in which the type at which we check a λ tells us what type to give for its bound variable in the context.

Ultimately, the synthesis rules work on the basis that variables types are indeed known by the context.

$$\frac{}{\Gamma_0, x : \sigma, \Gamma_1 \vdash x \in \sigma} \qquad \frac{\Gamma \vdash f \in \sigma \to \tau \quad \Gamma \vdash \sigma \ni s}{\Gamma \vdash f\ s \in \tau}$$

$$\frac{\Gamma \vdash e \in \mathbb{N} \quad \Gamma \vdash \mathbb{N} \ni t}{\Gamma \vdash (e + t) \in \mathbb{N}} \qquad \frac{\Gamma \vdash e \in \mathbb{N}}{\Gamma \vdash (n + e) \in \mathbb{N}}$$

See that in the application rule, σ comes from the type of the function and is then used to check the argument.

So, we have no need of any magic when our terms are in normal form. If we restrict ourselves to programs that we do not want to run, we have no problem. So we still have a problem.

What we need type information for is to *justify* doing computation!

3 Type Annotations

One way to fix the problem by brute force and allow programs which compute is to extend the grammar with explicit type annotations, exactly where the information is missing.

$$\langle neutral \rangle ::= \ldots \mid \langle normal \rangle : \langle type \rangle$$

We get the rule

$$\frac{\Gamma \vdash \tau \ni t}{\Gamma \vdash t : \tau \in \tau}$$

but we have to adjust the small-step semantics to account for the fact that computations now have type annotations. Neutral terms may now compute.

$$(\lambda x \to t : \sigma \to \tau)\ s \rightsquigarrow t[s : \sigma / x] : \tau \qquad (n : \mathbb{N} + t) \rightsquigarrow (n + t : \mathbb{N}) \qquad (n + n' : \mathbb{N}) \rightsquigarrow (n + n') : \mathbb{N}$$

(In the last reduction rule, the grammar tells you that the + on the right cannot be a piece of syntax standing between two numerical values, so it indicates that we actually do the arithmetic at that point!)

The only reduction rule for normal terms is

$$t : \tau \rightsquigarrow t$$

which says 'a normal form which not being *used* does not need a type annotation'.

Everywhere computation happens, we can see the type at which it is happening. In particular, the β rule does computation at a function type $\sigma \to \tau$ which may trigger further computation, but only at smaller types, σ and τ. Fundamentally, that is why all well typed programs in this language terminate.

4 Definitions

In ordinary programming, we don't just write one great big expression: we make definitions, naming components we intend to use later, and we use them just by invoking their names.

Another way to avoid the need for magic is to give types when we make definitions. Instead of putting annotations directly into neutrals, let us consider

$$\langle neutral \rangle ::= \ldots \mid \texttt{let}\ x = \langle normal \rangle : \langle type \rangle\ \texttt{in}\ \langle neutral \rangle$$

with the typing rule

$$\frac{\Gamma \vdash \sigma \ni s \quad \Gamma, x : \sigma \vdash e \in \tau}{\Gamma \vdash \texttt{let}\ x = s : \sigma\ \texttt{in}\ e \in \tau}$$

Again, we supply just the information that is needed. Programs do not contain β-redexes, but they do contain definitions and defined symbols. To *run* a program, turn it into an ordinary LAM term by substituting out all the definitions. It will still be well typed and not go wrong.

That is to say, giving types for definitions, which is good documentation anyway, should always be enough.

5 Oh, All Right, There Is Magic, But It's Just Algebra

As it happens, for a language as simple as LAM, you can get away with writing no types at all. The method we use is straightfoward:

1. Extend the grammar of types with *variables* which stand for types we don't know yet. (Let's write $?\langle number \rangle$ for such a variable.)
2. Ensure that we can always invent fresh variables to stand for new things we don't know. (As we're numbering them, we can always remember the lowest number we haven't used yet.)
3. When we don't know a type, we invent a variable to stand for it.
4. When we make demands about types, that tells us how to solve for those variables.

Specifically,

- When inferring a type for $\lambda x \to t$, invent a fresh variable for x's type, put that in the context, and hope to figure it out later.
- Every time you have something whose type is a variable $?n$, but you want a $\mathbb{N}$ (i.e., in the addition rule), congratulations! You now know that $?n = \mathbb{N}$ and can make that substitution.
- Every time you have something whose type is a variable $?n$, but you want a function type (i.e., demanding a function in the application rule), congratulations! You can make up two new type variables $?s$ and $?t$ for the unknown source and target types, then solve $?n = ?s \to ?t$.
- Every time that you want two types to be *equal* (i.e., when demanding that a function's argument has the function's source type), solve the equation between the types, and report an error if you cannot.

To solve

- $\mathbb{N} = \mathbb{N}$
  You win!
- $\sigma \to \tau = \sigma' \to \tau'$
  Solve $\sigma = \sigma'$ and $\tau = \tau'$.
- $\mathbb{N} = \sigma' \to \tau'$ or vice versa
  You lose!
- $?n = ?n$
  You win! But you don't learn what $?n$ is.

- $?n = \tau$ or vice versa, if variable $?n$ occurs strictly inside type τ
  You lose! (We cannot come up with any type which is a subformula of itself.)
- $?n = \tau$ or vice versa, if $?n$ does not occur in τ
  Substitute τ for $?n$

The above algorithm is called *first-order unification*. It was published by Alan Robinson in 1965. The insight to use it to figure out the missing parts of types was due to Robin Milner, in the mid-1970s. It has the property that it makes only those solutions that are forced by the constraints we face. As a result, if any type variables remain unsolved, it does not matter what type they stand for!

Let us have an example. Infer a type for $\lambda f \to \lambda x \to f\ (f\ x)$. We're trying to derive

$$\frac{\dots}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : \dots}$$

Make up a type for f,

$$\frac{\dots}{\dfrac{\dots}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : \dots}}$$

Now make up a type for x.

$$\frac{\dots}{\dfrac{\dots}{\dfrac{\dots}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : \dots}}}$$

Now, the application rule demands a *function* type for the function. The box marks the thing that is not yet what we need.

$$\frac{\dfrac{\dfrac{}{f : ?0, x : ?1 \vdash f : \boxed{?0}} \quad \dfrac{\dots}{f : ?0, x : ?1 \vdash f\ x : \dots}}{f : ?0, x : ?1 \vdash f\ (f\ x) : \dots}}{\dfrac{f : ?0 \vdash \lambda x \to f\ (f\ x) : \dots}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : \dots}}$$

So we make up a function type $?2 \to ?3$ and use it to solve $?0$.

$$\frac{\dfrac{\dfrac{}{f : ?2 \to ?3, x : ?1 \vdash f : ?2 \to ?3} \quad \dfrac{\dots}{f : ?2 \to ?3, x : ?1 \vdash f\ x : \dots}}{f : ?2 \to ?3, x : ?1 \vdash f\ (f\ x) : \dots}}{\dfrac{f : ?2 \to ?3 \vdash \lambda x \to f\ (f\ x) : \dots}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : \dots}}$$

In fact, we can fill in a lot of missing types, now.

$$\frac{\dfrac{\dfrac{}{f : ?2 \to ?3, x : ?1 \vdash f : ?2 \to ?3} \quad \dfrac{\dots}{f : ?2 \to ?3, x : ?1 \vdash f\ x : ?2}}{f : ?2 \to ?3, x : ?1 \vdash f\ (f\ x) : ?3}}{\dfrac{f : ?2 \to ?3 \vdash \lambda x \to f\ (f\ x) : ?1 \to ?3}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : (?2 \to ?3) \to ?1 \to ?3}}$$

We know that we want the remaining application to give us a ?2, but what have we got?

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : ?2 \to ?3, x : ?1 \vdash f : ?2 \to \boxed{?3}} \quad \dfrac{}{f : ?2 \to ?3, x : ?1 \vdash x : \boxed{?1}}}{f : ?2 \to ?3, x : ?1 \vdash f\ x : ?2}}{f : ?2 \to ?3, x : ?1 \vdash f\ (f\ x) : ?3}}{f : ?2 \to ?3 \vdash \lambda x \to f\ (f\ x) : ?1 \to ?3}}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : (?2 \to ?3) \to ?1 \to ?3}$$

For the $f\ x$ to be well typed, we need ?1 =?2 and for it to fit where we are putting it, we need ?3 =?2, so we can solve ?1, then ?3:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : ?2 \to ?2, x : ?2 \vdash f : ?2 \to ?2} \quad \dfrac{}{f : ?2 \to ?2, x : ?2 \vdash x : ?2}}{f : ?2 \to ?2, x : ?2 \vdash f\ x : ?2}}{f : ?2 \to ?2, x : ?2 \vdash f\ (f\ x) : ?2}}{f : ?2 \to ?2 \vdash \lambda x \to f\ (f\ x) : ?2 \to ?2}}{\vdash \lambda f \to \lambda x \to f\ (f\ x) : (?2 \to ?2) \to ?2 \to ?2}$$

We have no solution for ?2, but no constraints on it, either. It is whatever you want it to be. That is, the type of the x does not matter, but the f has to be a function that maps that type to itself, so that you can do it twice.

Let's have another example, this time of an error: $\lambda x \to x\ x$. We're trying to derive

$$\dfrac{\dfrac{\ldots}{\vdash \lambda x \to x\ x : \ldots}}{}$$

so we make up a variable for the type of x.

$$\dfrac{\dfrac{\dfrac{\ldots}{x : ?0 \vdash x\ x : \ldots}}{\vdash \lambda x \to x\ x : \ldots}}{}$$

When we deploy the application rule and the variable rule, we learn that ?0 must be a function type, so we make one up and substitute it. We now know that the application gives back ?2

$$\dfrac{\dfrac{\dfrac{}{x : ?0 \vdash \boxed{?0}} \quad \ldots}{x : ?0 \vdash x\ x : \ldots}}{\vdash \lambda x \to x\ x : \ldots} \qquad \dfrac{\dfrac{\dfrac{}{x : ?1 \to ?2 \vdash x : ?1 \to ?2} \quad \ldots}{x : ?1 \to ?2 \vdash x\ x : \ldots}}{\vdash \lambda x \to x\ x : \ldots} \qquad \dfrac{\dfrac{\dfrac{}{x : ?1 \to ?2 \vdash x : ?1 \to ?2} \quad \ldots}{x : ?1 \to ?2 \vdash x\ x : ?2}}{\vdash \lambda x \to x\ x : (?1 \to ?2) \to ?2}$$

The trouble arrives when we check the argument. The variable rule gives us its type.

$$\dfrac{\dfrac{\dfrac{}{x : ?1 \to ?2 \vdash x : ?1 \to ?2} \quad \dfrac{}{x : ?1 \to ?2 \vdash x : \boxed{?1 \to ?2}}}{x : ?1 \to ?2 \vdash x\ x : ?2}}{\vdash \lambda x \to x\ x : (?1 \to ?2) \to ?2}$$

For a valid use of the application rule, we must solve

$$?1 \to ?2 \ = ?1$$

That is, we need a function type equal to its own source type. That cannot happen, so this term is rejected.

6 What Have We Learned?

1. You do not need to do anything clever to figure out types as long as you are willing to annotate redexes or write types for definitions.
2. You can figure out missing types by introducing variables to stand for unknown types, then solving the constraints that the rules impose on those variables.
3. In our LAM language, the constraints can easily be solved by first-order unification.
4. If in some fancy language, the constraints are just TOO HARD to solve by an algorithm, see 1.