Deriving and Solving Type Constraints

sk and dbtucker

October 25, 2000

Equational types

In the previous class, we type checked polymorphic functions by generating equations in an ad hoc manner, then using magic to solve these equations. Today we will see how to accomplish both of these tasks in a systematic way.

Remember that we were allowed to introduce unconstrained type variables into our equations. In order to use these variables in our type rules, we need to extend the type grammar:

$$\begin{array}{lcl} \text{type} & ::= & \mathsf{num} \\ & | & \mathsf{bool} \\ & | & \text{type} \rightarrow \text{type} \\ & | & \text{tvar} \end{array}$$

We also need some way to represent the equations that constrain the type variables. Every time we assign a type to expression, we'll carry along the set of equations we've generated. We call this combination an *equational type*, and we write it as:

$$\text{type} \setminus \text{set of equations}$$

For example,

$$\mathsf{num} \setminus \{\}$$

is an equational type. So is:

$$\mathsf{num} \rightarrow \mathsf{bool} \setminus \{\}$$

We can also use type variables:

$$\alpha \setminus \{\alpha = \mathsf{num}\}$$

In this equational type, α is a type variable, and the set of equations constrains α to be num.

Let's write down the type rules using equational types. The type environment Γ still maps variables to types, but now we infer an equational type for each subexpression. The basic idea is as follows. Since function arguments are no longer annotated with their types, when we come across a function, we will

say its argument has some unconstrained type (which we denote with a type variable). As we type check the body of the function, we will derive constraints on the type variables and carry them along in the equational types.

We'll start with numbers—the rule says that any number has type num, and there are no equations constraining types:

$$\Gamma \vdash \langle \text{number} \rangle : \mathsf{num} \setminus \{\}$$

Similarly, for booleans we have:

$$\Gamma \vdash \langle \text{boolean} \rangle : \mathsf{bool} \setminus \{\}$$

The rule for variables is much the same—we look up the variable's type in Γ, and there are no constraints on that type:

$$\Gamma \vdash v : \Gamma(v) \setminus \{\}$$

Next we'll look at the addition case, $(l+r)$. We know that we can infer some equational type for l and r:

$$\frac{\Gamma \vdash l : \tau_l \setminus E_l \quad \Gamma \vdash r : \tau_r \setminus E_r}{\Gamma \vdash (l+r) : ???}$$

Why didn't we just say that l must have some type $\mathsf{num} \setminus E_1$? As we'll see later, it's possible that the type of l is a type expression containing type variables, which are constrained in E_l. If we write num in place of τ_l, we are demanding that the constraints be solved and checked for τ_l being equal to num in the midst of constraint generation. We take the equivalent but more relaxed approach of constraining τ_l to be num in the consequent of the $+$ judgement:

$$\frac{\Gamma \vdash l : \tau_l \setminus E_l \quad \Gamma \vdash r : \tau_r \setminus E_r}{\Gamma \vdash (l+r) : \mathsf{num} \setminus E_l \cup E_r \cup \{\tau_l = \mathsf{num},\ \tau_r = \mathsf{num}\}}$$

Two important notes:

1. the equations generated by the subexpressions are included in the equations for the overall expression

2. τ_l is just a meta-variable—in a proof tree, it would be replaced by a type or a type variable.

The next case we'll consider is function abstraction, $(\lambda\ (v)\ e)$. Remember that the function argument v is no longer annotated with its type. Previously, we extended Γ with v and its type, and proved that the body e has some type. Now, we'll say v's type is some type variable $\boxed{\alpha}$, and then type check the body:

$$\frac{\Gamma[v : \boxed{\alpha}] \vdash e : \tau \setminus E}{\Gamma \vdash (\lambda\ (v)\ e) : ???}$$

What is the type of the function? Its argument has type $\boxed{\alpha}$, and its body has type τ_1, so it must have type $\boxed{\alpha} \to \tau_1$:

$$\frac{\Gamma[v : \boxed{\alpha}] \vdash e : \tau \setminus E}{\Gamma \vdash (\lambda\ (v)\ e) : \boxed{\alpha} \to \tau \setminus E}$$

Now we'll write the rule for function application. Here's a start

$$\frac{\Gamma \vdash f : \tau_f \setminus E_f \quad \Gamma \vdash a : \tau_a \setminus E_a}{\Gamma \vdash (f\ (a)) : ???}$$

The function has type τ_1, but we don't know its return type (remember, τ_1 could be a type variable). So we'll introduce a new type variable $\boxed{\alpha}$ and add the constraint that it is the return type of the function:

$$\frac{\Gamma \vdash f : \tau_f \setminus E_f \quad \Gamma \vdash a : \tau_a \setminus E_a}{\Gamma \vdash (f\ (a)) : \boxed{\alpha} \setminus E_f \cup E_a \cup \{\tau_f = \tau_a \to \boxed{\alpha}\}}$$

The type rule for **if** should now be straightforward:

$$\frac{\Gamma \vdash c : \tau_c \setminus E_c \quad \Gamma \vdash y : \tau_y \setminus E_y \quad \Gamma \vdash n : \tau_n \setminus E_n}{\Gamma \vdash (\textbf{if}\ \ c\ \ y\ \ n) : \boxed{\alpha} \setminus E_c \cup E_y \cup E_n \cup \{\tau_c = \mathsf{bool},\ \boxed{\alpha} = \tau_y,\ \boxed{\alpha} = \tau_n\}}$$

Let's try out these rules on a small example. What is the type of the following expression?

$$(\lambda\ (x)\ (x\ \ 2))$$

Since x is applied to 2, it must be an arrow type $\mathsf{num} \to \alpha$, thus the overall function has type $(\mathsf{num} \to \alpha) \to \alpha$. Using our rules, we get this type derivation:

$$\frac{\dfrac{\{x : \boxed{\alpha}\} \vdash x : \boxed{\alpha} \setminus \{\} \quad \{x : \boxed{\alpha}\} \vdash 2 : \mathsf{num} \setminus \{\}}{\{x : \boxed{\alpha}\} \vdash (x\ \ 2) : \boxed{\beta} \setminus \{\boxed{\alpha} = \mathsf{num} \to \boxed{\beta}\}}}{\vdash (\lambda\ (x)\ (x\ \ 2)) : \boxed{\gamma} \setminus \{\boxed{\gamma} = \boxed{\alpha} \to \boxed{\beta},\ \boxed{\alpha} = \mathsf{num} \to \boxed{\beta}\}}$$

If we conjure some magic to solve the equations, we see the type of the expression is $(\mathsf{num} \to \boxed{\beta}) \to \boxed{\beta}$, which is what we expected.

Solving the equations

What we have so far is not a type checker. For one, it doesn't reject any programs—it only generates constraints on types. Now we'll give an algorithm for solving these constraints.

Step 1: Generate constraints

This is what we did above.

Step 2: Close constraints

To compute the closure of our constraint set E, we need three properties to hold:

1. Symmetry. If $a = b \in E$, then $b = a \in E$.
2. Transitivity. If $a = b \in E$ and $b = c \in E$, then $a = c \in E$.
3. Compatibility. The idea here is to "break up" constructed types into their components. For example, if $x = a \to b \in E$ and $x = c \to d \in E$, then add $a = c$ and $b = d$ to E.

Step 3: Detect conflicts

Next we see if there are any conflicts in our constraint set. In our current set of types, the possible conflicts are:

$$\mathsf{num} = \mathsf{bool}$$

$$\mathsf{num} = a \to b$$

$$\mathsf{bool} = a \to b$$

If any of these equations appear in our set, the system of equations is not consistent. This means the program has a type error, so we reject it.

Although we know there's an error, we don't know its source. As an exercise, can you determine where the error came from using some slight modification of the constraints?

Step 4: Printing types

Now we have some type $\boxed{\alpha} \setminus E$. What type do we report to the programmer? We'll study this question in the next class.