A simple type system

COMP 150 — Dataflow

February 24, 2010

Introduction and Exercises

This handout describes two exercises that will give you a little bit of practice with type checking and type inference. They are based on the same formal system, although to support type inference will require a little extra machinery to help handle unknown types.

The judgment form we're interested in is

$$\Gamma \vdash e : \tau$$

where Γ is a *type environment* mapping the name of a variable to that variable's type, e is a term, and τ is the type of that term. The judgment means that in a context where Γ gives the types of all the variables, term e has type τ.

We'll consider a very limited language of types:

$$\tau \Rightarrow \textsc{Int} \mid \textsc{Bool}$$

The language of terms is

$$\begin{array}{lll} e \Rightarrow x & & \text{variable} \\ \quad \mid \ \oplus(e_1, \ldots, e_n) & & \\ \quad \mid \ \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 & & \\ \quad \mid \ \textsc{True} \mid \textsc{False} \mid n & & \end{array}$$

Here are the rules for the typing judgment:

$$\textsc{Var}\quad \frac{x \in \operatorname{dom}\Gamma \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Apply}\quad \frac{\oplus \text{ has type } \tau_1 \times \ldots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \oplus(e_1, \ldots, e_n) : \tau} \qquad \textsc{Conditional}\quad \frac{\Gamma \vdash c : \textsc{Bool} \qquad \Gamma \vdash e_i : \tau}{\Gamma \vdash \textbf{if } c \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

$$\textsc{LitB}\quad \frac{v \in \{\textsc{True}, \textsc{False}\}}{\Gamma \vdash v : \textsc{Bool}} \qquad \textsc{LitI}\quad \Gamma \vdash n : \textsc{Int}$$

Don't forget that the equality and inequality operator may have multiple types.

There are two exercises:

1. Write a *type checker* which takes Γ and e as input and returns either a type τ such that $\Gamma \vdash e : \tau$ or else an error message:

   ```
   tycheck :: Env Ty -> Exp -> Error Ty
   ```

   (The `Error` monad is defined at the end of this handout.)

2. Write function that takes only e as an argument, and returns the *simplest* Γ and τ such that $\Gamma \vdash e : \tau$ (or else an error message). This function *infers* a *principal typing*.

   For this problem you'll need a way of representing unknown types. Normally we would use a special sort of type variable, but this exercise is simpler if we use a special "singleton type"

   $\tau \Rightarrow$ S(x)

   where $S(x)$ is pronounced "the type of x."[1]

   The rules have to change. Drop the original VAR rule and use these two rules instead:

$$\textsc{Var}' \quad \{x : \tau\} \vdash x : \tau \qquad\qquad \textsc{Weaken}\ \frac{\Gamma \vdash x : \tau \qquad y \notin \mathrm{fv}(\Gamma)}{\Gamma\{y \mapsto \tau'\} \vdash x : \tau}$$

   The rules for literals also must change to require an *empty* typing environment Γ.

   Here are some hints about how to proceed:

   - Any variable x should be given type $S(x)$ to start.
   - To use the APPLY rule, you may have to weaken environments in order to get a consistent set of environments. You may also have to substitute an actual type for $S(x)$. Examples below

Examples of principal typing

A term with variables may have multiple typings. For example, given the term x, the following judgments are all derivable:

$$\{x : \textsc{Int}\} \vdash x : \textsc{Int} \qquad \{x : \textsc{Bool}\} \vdash x : \textsc{Bool} \qquad \{x : S(x), y : \textsc{Bool}\} \vdash x : S(x) \qquad \{x : S(x)\} \vdash x : S(x)$$

The final judgment is special: by a combination of substituting for the type variable $S(x)$ and/or *weakening*, it is possible to obtain any of the other judgments. This property makes the final judgment the *principal* typing. (Not every system of formal typing rules has a principal typing—in fact, the desire to have a principal typing is what motivates the introduction of type variables.)

Here is a more complex derivation. To make it fit the space, I have left out the side conditions on the types of $>$ and $+$:

$$\textsc{Conditional}\ \dfrac{\dfrac{\dfrac{\textsc{Weaken}\ \dfrac{\textsc{Var}'\ \dfrac{}{\{x : \textsc{Int}\} \vdash x : \textsc{Int}}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash x : \textsc{Int}} \quad \vdots \quad \dfrac{\textsc{Var}'\ \dfrac{}{\{y : \textsc{Int}\} \vdash y : \textsc{Int}}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash y : \textsc{Int}}\ \textsc{Weaken}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash x > y : \textsc{Bool}}\ \textsc{Apply} \quad \vdots \quad \textsc{Weaken}\ \dfrac{\textsc{Var}'\ \dfrac{}{\{x : \textsc{Int}\} \vdash x : \textsc{Int}}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash x : \textsc{Int}}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash \mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y + 1 : \textsc{Int}} \quad \dfrac{\textsc{Weaken}\ \dfrac{\{y : \textsc{Int}\} \vdash y : \textsc{Int}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash y : \textsc{Int}} \quad \textsc{Weaken}\ \dfrac{\textsc{Weaken}\ \dfrac{\textsc{LitI}'\ \{\} \vdash 1 : \textsc{Int}}{\{x : \textsc{Int}\} \vdash 1 : \textsc{Int}}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash 1 : \textsc{Int}}}{\{x : \textsc{Int}, y : \textsc{Int}\} \vdash y + 1 : \textsc{Int}}\ \textsc{Apply} \quad \vdots}{}$$

[1] The S stands for *singleton*, but this is a slight abuse of terminology and notation.

Inference algorithm

How can you reconstruct a principal typing? For example, how does the inference algorithm "know" to assign type INT to x? There are three key ideas:

- Use principal types throughout.
- In the APPLY rule, use weakening to make type environments equal.
- When two types (or two type environments have to be equal), make them equal by substituting for type variables. If this is not possible (i.e., if you try to make INT and BOOL equal, halt with a type error.

Here's an example of how the derivation above might be constructed *algorithmically*, working from the bottom up.

1. Given term x, the principal typing says $\{x : S(x)\} \vdash x : S(x)$.
2. Similarly, $\{y : S(y)\} \vdash y : S(y)$.
3. What is the principal typing of $x > y$? We need the APPLY rule, so our first step is to take the principal typings of the subterms x and y and make the environments equal through a combination of weakening and substitution. This step yields environment $\Gamma_> = \{x : S(x), y : S(y)\}$.

   The type of $>$ is $\text{INT} \times \text{INT} \rightarrow \text{BOOL}$. By examining the type part of the subterms, we arrive at the constraints $S(x) = \text{INT}$ and $S(y) = \text{INT}$. These constraints can be solved by substituing INT for the type variable $S(x)$ and INT for the type variable $S(y)$, yielding the following principal typing:

   $$\{x : \text{INT}, y : \text{INT}\} \vdash x > y : \text{BOOL}.$$

4. Similar procedures on the other two subterms if the if-then-else yield two more principal typings:

   $$\{x : S(x)\} \vdash x : S(x) \qquad\qquad \{y : \text{INT}\} \vdash y + 1 : \text{INT}$$

   To compute the principal typing of the if-then-else, we must make all three type environments equal. In this case, we require both weakening *and* substitution, and the final type environment is $\{x : \text{INT}, y : \text{INT}\}$. The principal typing of the entire term is

   $$\{x : \text{INT}, y : \text{INT}\} \vdash \textbf{if } x > y \textbf{ then } x \textbf{ else } y + 1 : \text{INT}.$$

(Coding hints on next page)

Coding hints

Here are some declarations that might be helpful:

```
type Env a = [(Variable, a)] -- association list; there are predefined
                             -- lookup functions, and you can sort to
                             -- unify environments

type TySubst = Ty -> Ty    -- built-in 'id' is the identity substitution
tySubst :: Variable -> Ty -> TySubst

unify :: Ty -> Ty -> Error TySubst
  -- if theta = unify tau tau', then  theta tau == theta tau'

unifyEnv :: Env Ty -> Env Ty -> Error (Env Ty, TySubst)
  -- if (gamma'', theta) = unifyEnv gamma gamma' then
  --   1. theta gamma  can be obtained from gamma'' by weakening
  --   2. theta gamma' can be obtained from gamma'' by weakening

ptyping :: Exp -> Error (Env Ty, Ty)
```

where `Error` is the error monad:

```
data Error a = Error String
             | OK a

instance Monad Error where
  return = OK
  fail   = Error
  Error msg >>= _ = Error msg
  OK a      >>= k = k a
```

If the monad confuses you, you can just have your code crash when it encounters a term with no principal typing.