Implicit Polymorphism

sk and dbtucker

2002-11-15

1 The Problem

Consider the function

(**lambda** (*x*) *x*)

The type inference engine would infer that this function has type

$$\alpha \rightarrow \alpha$$

(or some other type, modulo renaming the type variable).

Now consider a program of the form (the **let** construct is similar to the `with` in our interpreted language):

```
(let ([id (lambda (x) x)])
  (+ (id 5)
     (id 6)))
```

First we need a type judgment for **let**. Here is a reasonable one: it is exactly what one gets by using the existing rules for functions and applications, since we have consistently defined `with` as an application of an immediate function:

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma[x \leftarrow \tau'] \vdash b : \tau}{\Gamma \vdash (\mathbf{let}([x\, v])\, b) : \tau}$$

Given this judgment, the type variable α in the type inferred for *id* would unify with the type of 5 (namely, number) at one application and with the type of 6 (also number) at the other application. Since these are consistent, the unification algorithm would conclude that *id* is being used as if it had the type

$$\mathsf{number} \rightarrow \mathsf{number}$$

in this program.

Now suppose we use *id* in a context where we apply it to values of different types. The following program is certainly legal in Scheme:

```
(let ([id (lambda (x) x)])
  (if (id true)
      (id 5)    ;; then
      (id 6)))  ;; else
```

This should be legal even in typed Scheme, because we're returning the same type of value in both branches of the conditional. But what happens when we supply this program to our type system? It infers that *id* has type $\alpha \rightarrow \alpha$. But it then unifies α with the type of each of the arguments. Two of these are the same (number) but the first application is to a value of type boolean. This forces the type inference algorithm to try and unify number with boolean. Since these are distinct base types, type inference fails with a type error!

We definitely do not want this program to be declared erroneous. The problem is not with the program itself, but with the algorithm we employ to infer types. That suggests that we should try to improve the algorithm.

2 A Solution

What's the underlying problem in the type algorithm? We infer that *id* has type $\alpha \to \alpha$; we are then stuck with that type for every use of *id*. It must therefore be *either* number $\to$ number or boolean $\to$ boolean—but it can't be both. That is, we cannot use it in a truly polymorphic manner!

This analysis makes clear that the problem has something to do with type variables and their unification. We arrive at a contradiction because α must unify with both number and boolean. But what if α didn't need to do that? What if we didn't use the same type variable every time? Then perhaps we could avoid the problem entirely.

One way to get fresh type variables for each application of *id* is to *literally substitute* the uses of *id* with their value. That is, instead of type checking the program above, suppose we were to check the following program:

```
(let ([id (lambda (x) x)])
  (if ((lambda (x) x) true)
      ((lambda (x) x) 5)
      ((lambda (x) x) 6)))
```

We don't want to have to *write* this program, of course, but that's okay: a simple pre-processor can substitute every **let**-bound identifier in the body before type-checking. If we did that, we get a different result from type-checking:

```
(let ([id (lambda (x) x)])
  (if (1 (lambda (x) x) true)
      (2 (lambda (x) x) 5)
      (3 (lambda (x) x) 6)))
```

Each use of *id* results in a different type; for instance, the *id* procedure at 1 might have type $\alpha \to \alpha$, 2 might have type $\beta \to \beta$ and 3 might have $\gamma \to \gamma$. Then α could unify with type boolean, β with type number and γ with type number. Because these are distinct type variables, they need not unify with one another. Each application would succeed, and the entire program would successfully pass the type checker. This in fact corresponds more accurately with what happens during execution, because on the first invocation the identifier *x* in *id* holds a value of boolean type, and on the subsequent invocation (in the first branch of the conditional, in this case) it holds a number. The separate type variables accurately reflect this behavior.

3 A Better Solution

The solution we've presented above has two problems:

1. It can lead to considerable code size explosion. For instance, consider this program:

   ```
   (let ([x
          (let ([y
                 (let ([z 3])
                   (+ z z))])
            (+ y y))])
     (+ x x))
   ```

   Expand it in full. In general, how big can a program grow upon expansion?

2. Since **let** does not permit recursion, consider **letrec** or **local**, the Scheme analog of `rec`. What happens if we substitute code in a recursive definition?

In short, the code substitution solution is not workable, but it does contain the germ of a good idea. We see that what it does is generate *fresh type variables* at every use: this is the essence of the solution. So perhaps we can preserve the essence while dispensing with that particular implementation.

Indeed, we can build further on the intuition we have developed. A closure has only one name for an identifier, but a closure can be used multiple times, even recursively, without confusion. This is because, in effect, each application consistently renames the bound identifier(s) so they cannot be confused across instances. Working backwards, since

we want fresh identifiers that cannot be confused across instances, we want to create a kind of *type closure* that we instantiate at every use of a polymorphic function.

We will therefore use a modified rule for typing **let**:

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma[x \leftarrow \mathsf{CLOSE}(\tau')] \vdash b : \tau}{\Gamma \vdash (\mathbf{let}([x\,v])\,b) : \tau}$$

That is, we bind x to a "closed" type when we check the body. The idea is, whenever we encounter this special type in the body, we instantiate its type variables with fresh type variables:

$$\frac{\Gamma \vdash e : \mathsf{CLOSE}(\tau')}{\Gamma \vdash e : \tau}$$

where τ is the same as τ', except all type variables have been renamed consistently to unused type variables.

Returning to the identity procedure, the type inference algorithm binds *id* to the type $\mathsf{CLOSE}(\alpha \to \alpha)$. At each use of *id*, the type checker renames the type variables, generating types such as $\alpha_1 \to \alpha_1$, $\alpha_2 \to \alpha_2$, and so on. As we have seen before, these types permit the body to successfully type check. Therefore, we have successfully captured the intuition behind code-copying without the difficulties associated with it.

Puzzle (How) Does the use of type closures instead of code substitution affect the handling of recursion?

4 A Significant Subtlety

Alas something is still rotten in the state of inferring polymorphic types. When we rename all type variables in a CLOSE type, we may rename variables that were not bound in the **let** expression: for instance,

```
(lambda (y)
  (let ([f (lambda (x) y)])
    (if (f true)
        (+ (f true) 5)
        6)))
```

Our algorithm would infer the type $\mathsf{CLOSE}(\alpha \to \beta)$ (or the equivalent under renaming) for f. (Note that f is not the identity function; the type $\mathsf{CLOSE}(\alpha \to \alpha)$ would not be valid, because the type of *y* may be different from that of *x*.

At the first application, in the test of the conditional, we generate fresh type names, α_1 and β_1. The type α_1 unifies with boolean, and β_1 unifies with boolean (since it's used in a conditional context). At the second application, the algorithm generates two fresh names, α_2 and β_2. α_2 will unify with boolean (since that is the type of the argument to f), while β_2 unifies with number, due to the use in an addition expression. In other words, the program successfully passes the type checker.

But this program should fail! Simply looking at it, it's obvious that f can return *either* a boolean or a numeric value, but not both. Indeed, if we apply the entire expression to true, there will be a type error at the addition; if we apply it to 42, the type error will occur at the conditional. Sure enough, in our type system prior to today, it would have failed with an error while unifying the constraints on the return types of f. So how did it slip through?

The program successfully passed the type checker because of our use of type closures. We did not, however, correctly apply our intuition about closures. When we apply a closure, we only get new identifiers for those bound by the closures—not those in its lexical scope. The variables in the closure's lexical scope are shared between all applications of the closure. So should it be in the case of type closures. We should only generate fresh type variables for the types introduced by the **let**.

Concretely, we must modify our rule for **let** and our type closure scheme to track the identifiers that must be renamed:

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma[x \leftarrow \mathsf{CLOSE}(\tau', \Gamma)] \vdash b : \tau}{\Gamma \vdash (\mathbf{let}([x\,v])\,b) : \tau}$$

That is, a type closure tracks the environment of closure creation. Correspondingly,

$$\frac{\Gamma \vdash e : \mathsf{CLOSE}(\tau', \Gamma')}{\Gamma \vdash e : \tau}$$

where τ is the same as τ', except all type variables *not in* Γ' have been renamed consistently to fresh type variables.

Applying these rules to the example above, we rename the α's but not β, so the first use of f gets type $\alpha_1 \rightarrow \beta$ and the second use $\alpha_2 \rightarrow \beta$. This forces $\beta = \mathsf{number} = \mathsf{boolean}$, which results in a type error during unification.

5 Why Let and not Lambda?

The kind of polymorphism we have seen above is called *let-based polymorphism*, in honor of the ML programming language, which introduced this concept. Note that **let** in ML is recursive (so it corresponds to Scheme's **letrec** or **local**, and the `rec` we have studied in this class). In particular, ML treats `let` as a primitive construct, rather than expanding it into an immediate function application as Scheme does (and as we did with `with` in our interpreters).

The natural question is to wonder why we would have a rule that makes **let**-bound identifiers polymorphic, but not admit the same polymorphic power for **lambda**-bound identifiers. The reason goes back to our initial approach to polymorphism, which was to substitute the body for the identifier. When we have access to the body, we can successfully perform this substitution, and check for the absence of errors. (Later we saw how type closures achieve the same effect while offering several advantages, but the principle remains the same.)

The last example above shows the danger in generalizing the type of **lambda**-bound identifiers: without knowing what they will actually receive as a value (which we cannot know until run-time), we cannot be sure that they are in fact polymorphic. Because we have to decide at type-checking time whether or not to treat an identifier polymorphically, we are forced to treat them monomophically, and extend the privilege of polymorphism only to **let**-bound identifiers. Knowing exactly which value will be substituted turns out to be a gigantic advantage for the type system!

6 The Structure of ML Programs

While our type inference algorithm inferred types with type variables, we could not actually exploit this power directly. We could use such a value several times in the same type contexts, and the same expression elsewhere several times in a different type context, but not combine the two copies of the code through a binding. Let-based polymorphism earned us this power of abstraction.

Let-based polymorphism depends fundamentally on having access to the bound value when checking the scope of the binding. As a result, an ML program is typically written as a series of **let** expressions; the ML evaluator interprets this as a sequence of nested **let**s. It treats the initial environment similarly as one long sequence of **let**s, so for instance, if a programmer uses *map* in a top-level expression, the evaluator effectively puts the use of *map* in the body of the definition of *map*. Therefore, the uses of *map* benefit from the polymorphic nature of that function.

Puzzle What is the complexity of the polymorphic type inference algorithm that uses type closures?