Chapter 29

Explicit Polymorphism

29.1 Motivation

Earlier, we looked at examples like the length procedure (from now on, we'll switch to Scheme with imaginary type annotations):

```
(define lengthNum
  (lambda (l : numlist) : number
    (cond
      [(numEmpty? l) 0]
      [(numCons? l) (add1 (lengthNum (numRest l)))])))
```

If we invoke *lengthNum* on (list 1 2 3), we would get 3 as the response.

Now suppose we apply *lengthNum* to (list 'a 'b 'c). What do we expect as a response? We might *expect* it to evaluate to 3, but that's not what we're going to get! Instead, we are going to get a type error (before invocation can even happen), because we are applying a procedure expecting a numlist to a value of type symlist (a list of symbols).

We can, of course, define another procedure for computing the length of lists of symbols:

```
(define lengthSym
  (lambda (l : symlist) : number
    (cond
      [(symEmpty? l) 0]
      [(symCons? l) (add1 (lengthSym (symRest l)))])))
```

Invoking *lengthSym* on (list 'a 'b 'c) will indeed return 3. But look closely at the difference between *lengthNum* and *lengthSym*: what changed in the code? Very little. The changes are almost all in the *type* annotations, not in the code that executes. This is not really surprising, because there is only one *length* procedure in Scheme, and it operates on all lists, no matter what values they might hold.

This is an unfortunate consequence of the type system we have studied. We introduced types to reduce the number of errors in our program (and for other reasons we've discussed, such as documentation), but in the process we've actually made it more difficult to write some programs. This is a constant tension in the design of typed programming languages. Introducing new type mechanisms proscribes certain programs,[1] but in return it invalidates some reasonable programs, making them harder to write. The *length* example is a case in point.

Clearly computing the length of a list is very useful, so we might be tempted to somehow add *length* as a primitive in the language, and devise special type rules for it so that the type checker doesn't mind what kind of list is in use. This is a bad idea! There's a principle of language design that says it's generally unadvisable for language designers to retain special rights for themselves that they deny programmers who use their language. It's unadvisable because its condescending and paternalistic. It suggests the language designer somehow "knows better" than the programmer: trust us, we'll build you just the primitives you need. In fact, programmers tend to always exceed the creative bounds of the language designer. We can already see this in this simple example: Why *length* and not *reverse*? Why *length* and *reverse* but not *append*? Why all three and not *map*? Or *filter* or *foldl* and *foldr* or.... Nor is this restricted to lists: what about trees, graphs, and so forth? In short, special cases are a bad idea. Let's try to do this right.

29.2 Solution

To do this right, we fall back on an old idea: abstraction. The two length functions are nearly the same except for small differences; that means we should be able to parameterize over the differences, define a procedure once, and instantiate the abstraction as often as necessary. Let's do this one step at a time.

Before we can abstract, we should identify the differences clearly. Here they are, boxed:

(**define** *length*$\boxed{Num}$
  (**lambda** (*l* : $\boxed{\text{num}}$list) : number
    (**cond**
      [($\boxed{\text{num}}$*Empty?* *l*) 0]
      [($\boxed{\text{num}}$*Cons?* *l*) (*add1* (*length*$\boxed{Num}$ ($\boxed{\text{num}}$*Rest* *l*)))])))

(**define** *length*$\boxed{Sym}$
  (**lambda** (*l* : $\boxed{\text{sym}}$list) : number
    (**cond**
      [($\boxed{\text{sym}}$*Empty?* *l*) 0]
      [($\boxed{\text{sym}}$*Cons?* *l*) (*add1* (*length*$\boxed{Sym}$ ($\boxed{\text{sym}}$*Rest* *l*)))])))

Because we want only one *length* procedure, we'll drop the suffixes on the two names. We'll also abstract over the num and sym by using the parameter τ, which will stand (of course) for a type:

(**define** *length*
  (**lambda** (*l* : $\boxed{\tau}$list) : number
    (**cond**
      [($\boxed{\tau}$*Empty?* *l*) 0]
      [($\boxed{\tau}$*Cons?* *l*) (*add1* (*length* ($\boxed{\tau}$*Rest* *l*)))])))

[1]It had better: if it didn't prevent some programs, it wouldn't catch any errors!

It's cleaner to think of list as a *type constructor*, analogous to how variants define value constructors: that is, list is a constructor in the type language whose argument is a type. We'll use an applicative notation for constructors in keeping with the convention in type theory. This avoids the odd "concatenation" style of writing types that our abstraction process has foisted upon us. This change yields

```
(define length
  (lambda (l : list(τ)) : number
    (cond
      [([τ]Empty? l) 0]
      [([τ]Cons? l) (add1 (length ([τ]Rest l)))])))
```

At this point, we're still using concatenation for the list operators; it seems to make more sense to make those also parameters to *Empty* and *Cons*. To keep the syntax less cluttered, we'll write the type argument as a subscript:

```
(define length
  (lambda (l : list(τ)) : number
    (cond
      [(Empty?_τ l) 0]
      [(Cons?_τ l) (add1 (length (Rest_τ l)))])))
```

The resulting procedure declaration says that *length* consumes a list of any type, and returns a single number. For a given type of list, *length* uses the type-specific empty and non-empty list predicates and rest-of-the-list selector.

All this syntactic manipulation is hiding a great flaw, which is that we haven't actually defined τ anywhere! As of now, τ is just a free (type) variable. Without binding it to specific types, we have no way of actually providing different (type) values for τ and thereby instantiating different typed versions of *length*.

Usually, we have a simple technique for eliminating unbound identifiers, which is to bind them using a procedure. This would suggest that we define *length* as follows:

```
(define length
  (lambda (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?_τ l) 0]
        [(Cons?_τ l) (add1 (length (Rest_τ l)))]))))
```

but this is horribly flawed! To wit:

1. The procedure *length* now has the wrong form: instead of consuming a list as an argument, it consumes a value that it will bind to τ, *returning* a procedure that consumes a list as an argument.

2. The program isn't even syntactically valid: there is no designation of argument and return type for the procedure that binds τ![2]

[2] You might wonder why we don't create a new type, call it type, and use this as the type of the type arguments. This is trickier than it seems: is type also a type? What are the consequences of this?

3. The procedure bound to *length* expects one argument which is a *type*. This violates our separation of the static and dynamic: types are supposed to be static, whereas procedure arguments are values, which are dynamic.

So on the one hand, this seems like the right sort of idea—to introduce an abstraction—but on the other hand, we clearly can't do it the way we did above. We'll have to be smarter.

The last complaint above is actually the most significant, both because it is the most insurmountable and because it points the way to a resolution. There's a contradiction here: we *want* to have a type parameter, but we *can't* have the type be a value. So how about we create procedures that bind *types*, and execute these procedures during type checking, not execution time?

As always, name and conquer. We don't want to use **lambda** for these type procedures, because **lambda** already has a well-defined meaning: it creates procedures that evaluate during execution. Instead, we'll introduce a notion of a type-checking-time procedure, denoted by Λ (capital λ). A Λ procedure takes only types as arguments, and its arguments do not have further type annotations. We'll use angles rather than parentheses to denote their body. Thus, we might write the *length* function as follows:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?_τ l) 0]
        [(Cons?_τ l) (add1 (length (Rest_τ l)))]))>)
```

This is a lot better than the previous code fragment, but it's still not quite there. The definition of *length* binds it to a type procedure of one argument, which evaluates to a run-time procedure that consumes a list. Yet *length* is applied in its own body to a list, not to a type.

To remedy this, we'll need to *apply* the type procedure to an argument (type). We'll again use the angle notation to denote application:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?_τ l) 0]
        [(Cons?_τ l) (add1 (length<τ> (Rest_τ l)))]))>)
```

If we're going to apply *length* to τ, we might as well assume *Empty?*, *Cons?* and *Rest* are also type-procedures, and supply τ explicitly through type application rather than through the clandestine subscript currently in use:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?<τ> l) 0]
        [(Cons?<τ> l) (add1 (length<τ> (Rest<τ> l)))]))>)
```

Thus, an expression like (*Rest*$<\tau>$ *l*) first applies *Rest* to τ, resulting in an actual *rest* procedure that applies to lists of values of type τ; this procedure consumes *l* as an argument and proceeds as it would in the type-system-free case. In other words, every type-parameterized procedure, such as *Rest* or *length*, is a generator of infinitely many procedures that each operate on specific types. The use of the procedure becomes

(*length*$<$num$>$ (list 1 2 3))
(*length*$<$sym$>$ (list 'a 'b 'c))

We call this language *parametrically polymorphic with explicit type parameters*. The term *polymorphism* means "having many forms"; in this case, the polymorphism is induced by the type parameters, where each of our type-parameterized procedures is really a representative of an infinite number of functions that differ only in the type parameter. The "explicitly" comes from the fact that our language forces the programmer to write the Λ's and type application.

29.3 The Type Language

As a result of these ideas, our type language has grown considerably richer. In particular, we now permit *type variables* as part of the type language. These type variables are introduced by type procedures (Λ), and discharged by type applications. How shall we write such types? We may be tempted to write

$$\textit{length} : \textit{type} \to (\mathsf{list}(\textit{type}) \to \mathsf{number})$$

but this has two problems: first, it doesn't distinguish between the two kinds of arrows ("type arrows" and "value arrows", corresponding to Λ and **lambda**, respectively), and secondly, it doesn't really make clear which type is which, a problem if there are multiple type parameters:

$$\textit{map} : \textit{type}, \textit{type} \to \mathsf{list}(\textit{type}) \times (\textit{type} \to \textit{type}) \to \mathsf{list}(\textit{type})$$

Instead, we adopt the following notation:

$$\textit{length} : \forall\alpha.\ \mathsf{list}(\alpha) \to \mathsf{number}$$

where it's understood that every $\forall$ parameter is introduced by a type procedure (Λ).[3] Here are the types for a few other well-known polymorphic functions:

$$\textit{filter} : \forall\alpha.\ \mathsf{list}(\alpha) \times (\alpha \to \mathsf{boolean}) \to \mathsf{list}(\alpha)$$
$$\textit{map} : \forall\alpha,\beta.\ \mathsf{list}(\alpha) \times (\alpha \to \beta) \to \mathsf{list}(\beta)$$

The type of *map*, in particular, makes this type notation is superior to our previous proposal: when multiple types are involved, we must give each one a name to distinguish between them.

[3]It's conventional to use the beginning of the Greek alphabet—α, β and so on—as the canonical names of polymorphic types, rather than begin from τ. This has two reasons. First, τ is conventionally a *meta-variable*, whereas α and β are type *variables*. Second, not many people know what Greek letter comes after τ....

29.4 Evaluation Semantics and Efficiency

While we have introduced a convenient *notation*, we haven't entirely clarified its meaning. In particular, it appears that every type function application actually happens during program execution. This seems extremely undesirable for two reasons:

- it'll slow down the program, in comparison to both the typed but non-polymorphic programs (that we wrote at the beginning of the section) and the non-statically-typed version, which Scheme provides;
- it means the types must exist as values at run-time.

Attractive as it may seem to students who see this for the first time, we really *do not* want to permit types to be ordinary values. A type is an abstraction of a value; conceptually, therefore, it does not make any sense for the two to live in the same universe. If the types were not supplied until execution, the type checker not be able to detect errors until program execution time, thereby defeating the most important benefit that types confer.

It is therefore clear that the type procedures must accept arguments and evaluate their bodies before the type checker even begins execution. By that time, if all the type applications are over, it suffices to use the type checker built earlier, since what remains is a language with no type variables remaining. We call the phase that performs these type applications the *type elaborator*.

The problem with any static procedure applications is to ensure they will lead to terminating processes! If they don't, we can't even begin the next phase, which is traditional type checking. In the case of using *length*, the first application (from the procedure use) is on the type `num`. This in turn inspires a recursive invocation of *length* also on type `num`. Because this latter procedure application is no different from the initial invocation, *the type expander does not need to perform the application*. (Remember, if the language has no side-effects, computations will return the same result every time. Type application has no side-effects.)

This informal argument suggests that only one pass over the body is necessary. We can formalize this with the following type judgments:

$$\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e < \tau' >: \tau[\alpha \leftarrow \tau']}$$

This judgment says that on encountering a type application, we substitute the quantified type with the type argument replacing the type variable. The program source contains only a fixed number of type applications (even if each of these can execute arbitrarily many times), so the type checker performs this application only once. The corresponding rule for a type abstraction is

$$\frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash < \Lambda\ (\alpha)\ e >: \forall\alpha.\tau}$$

This says that we extend Γ with a binding for the type variable α, but leave the associated type unspecified so it is chosen nondeterministically. If the choice of type actually matters, then the program must not type-check.

Observe that the type expander conceptually creates many monomorphically typed procedures, but we don't really want most of them during execution. Having checked types, it's fine if the length function that actually runs is essentially the same as Scheme's *length*. This is in fact what most evaluators do. The static type system ensures that the program does not violate types, so the program that runs doesn't need type checks.

29.5 Perspective

Explicit polymorphism seems extremely unwieldy: why would anyone want to program with it? There are two possible reasons. The first is that it's the only mechanism that the language designer gives for introducing parameterized types, which aid in code reuse. The second is that the language includes some additional machinery so you don't have to write all the types every time. In fact, C++ introduces a little of both (though much more of the former), so programmers are, in effect, manually programming with explicit polymorphism virtually every time they use the STL (Standard Template Library). Similarly, the Java 1.5 and C# languages support explicit polymorphism. But we can possibly also do better than foist this notational overhead on the programmer.