Parametric Polymorphism
Lecture Notes for cs173, Fall 2001

`sk`, `rob`, and `dbtucker`

October 31, 2001

I am Lambda Man!
—**Lambda Man** (Don Blaheta), in a special cs173 guest appearance

So far we've only been working with number lists. What other types of lists might we want? Surely a list of bool's would be useful. How about a list of functions? A list of continuations?

Clearly we want the ability to create a list of any type. We would like to be able to declare, for example, something is of type A list, where A is a type variable. In addition, we want to write generic functions that work on these lists.

Let's assume we have a Scheme with type annotations and static type checking. In this typed Scheme, we can write *map* as follows:

```
(define (map  f : (num → num)
              l : nlist) : nlist
  (cond
    [(empty? l) empty]
    [else (cons (f (first l))
                (map f (rest l)))]))
```

This function *map* will not work generically, since we've hardcoded the types inside the function. Let's apply a standard CS technique: If we aren't sure of a value, we can parameterize it and supply it later. We will let the *type variables A* and *B* be arguments to *map*:

```
(define map
  (lambda (A B)
    (lambda (f : (A → B)  L : A list) : B list
      (cond
        [(empty? L) empty]
        [else (cons (f (first L))
                    (map f (rest L)))]))))
```

This function won't parse, since *A* and *B* aren't annotated with types. Furthermore, even if it did parse, it isn't legal because types (e.g., *num*, *bool*) are not values in the language. We'll address these problems later, and continue with the example for now.

The function *map* takes types *A* and *B* and returns a function which operates on lists of the specified types. Let's see how we could use *map* to square a list of numbers:

```
(define ls (list 1 2 3))

(define (square x : num) : num (* x x))

((map num num) square ls)
```

The above definition of *map* isn't right—in the recursive call to *map*, it doesn't pass the types as arguments. We can fix that problem by simply passing *A* and *B* along:

```
(define map
  (lambda (A B)
    (lambda (f : (A → B)  L : A list) : B list
      (cond
        [(empty? L) empty]
        [(cons? L) (cons (f (first L))
                         ((map A B) f (rest L)))]))))
```

We have another problem—what is the type of *cons*? It takes an element of type *A* and a list of type *A list*, then returns a list of type *A list*. So *cons* is parameterized over the type *A* just as *map* is. In fact, all the constructors, predicates, and selectors created when we defined the *A list* datatype are parameterized over *A*. The contracts to these functions follow:

cons : $A \rightarrow (A \times A$ list $\rightarrow A$ list)
empty : $A \rightarrow A$ list
first : $A \rightarrow (A$ list $\rightarrow A)$
rest : $A \rightarrow (A$ list $\rightarrow A$ list)
cons? : $A \rightarrow (A$ list $\rightarrow$ bool)
empty? : $A \rightarrow (A$ list $\rightarrow$ bool)

```
(define map
  (lambda (A B)
    (lambda (f : (A → B)  L : A list) : B list
      (cond
        [((empty? A) L) (empty B)]
        [((cons? A) L) ((cons B) (f ((first A) L))
                                 ((map A B) f ((rest A) L)))]))))
```

This treatment of parameterized types raises several questions:

1. Are types values?
2. Since we can create functions over types, and apply functions to types, can't we just compute everything with types?

3. Since *A* and *B* are specified as parameters to *map*, and all function parameters must be annotated with their types, what are the types of *A* and *B* themselves?

We don't want types to be values in our language, since it breaks the distinction we have made between a static universe of types, and a dynamic one of values. Once we conflate the two, what good do types do us? If we can involve them in computations, then type checking could become undecidable, in which case they become useless.

The solution is to evaluate the code in two phases. In the first phase, we do all the necessary computations over types. To mark this difference, we denote functions over types (or, *type functions*) using Λ. The function *map* below is one example of a type function. Each of the functions with contracts listed above is also a type function. We use the syntax $\{f_t\, t_1\, ...\, t_n\}$ where f_t is a type function and t_i are actual types to denote *type application*:

```
(define map
  (Λ (A B)
    (lambda (f : (A → B)  L : A list) : B list
      (cond
        [({empty? A} L) {empty B}]
        [({cons? A} L) ({cons B} (f ({first A} L))
                                  ({map A B} f ({rest A} L)))]))))
```

The first phase evaluates the type functions and type applications which perform type elaboration. After this phase, all types are explicitly known. The second phase evaluates the resulting program.

There's something subtle going on in phase one. At run-time, actual types are passed into *map* as variables *A* and *B*. Phase one then elaborates {*empty? A B*}, {*empty B*}, {*cons? A*}, etc with type information. But what about when we get to the recursive call, {*map A B*}? In order to elaborate this with type information, we need to pass in the explicit types assigned to *A* and *B* back to *map*. This could go on forever if we aren't careful. It turns out that the compiler needs to bail us out. The compiler will (hopefully) recognize that each recursive call to *map* is exactly the same, and thus phase one will terminate.

We really have two different languages here, one for type expansion, and one for the regular evaluation. How do we prevent the evaluation of the type expansion language from diverging? One solution is to impose a second type system on top of that language, where this second type system is strongly normalizing.

Note that we have expanded the legal set of types: We now allow type *variables* (e.g., *A* and *B* in *map* above), which were not previously in our grammar. We do have type variables in our proof rules, but these are used so that we don't have to write typing rules with specific types hardcoded in the rules.

The kind of polymorphism we've seen here is called *parametric polymorphism* which is a kind of *explicit polymorphism*. It seems to be a real pain to program in this language, since we have to explicitly write down a type every time we use a function. Who would want to program in such a language? In fact, many people do—this is effectively what C++ template programmers write, especially when they use the STL. (They don't always realize this, since many of these type abstractions and applications are hidden in the STL routines, and anyway the syntaxes—function application vs. template specialization—are quite different.)