Today's Lecture Notes for cs173

sk and dbtucker

October 20, 2000

Polymorphism

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? Certainly—we can use it to represent contexts, a list of hash functions, or a jump table for bytecodes (in fact, a vector of functions would make more sense for a jump table).

We want to use lists of any type, and we want to write functions over these lists that don't restrict the element type ahead of time. So, we need a type like `list(`T`)` and `vector(`T`)`, where T can be any type. How do we program in such a language?

We'll write Scheme-like code, and assume that the language is statically type checked, and allows only top-level datatype definitions. In Scheme, we can write *map* as follows:

```
(define map
  (λ (f L)
    (cond
      [(empty? L) empty]
      [else (cons (f (first L))
                  (map f (rest L)))])))
```

Since we're statically type checking this language, we need to specify the types of the function parameters f and L. The parameter L is a list of elements of some type T, and f consumes a value of this type T and returns a value of some other type U:

```
(define map
  (λ (f : (T → U)  L : list(T))
    (cond
      [(empty? L) empty]
      [else (cons (f (first L))
                  (map f (rest L)))])))
```

The list returned by the function contains values produced by f, so its type is $list(U)$. So the type of *map* looks something like this:

$$map : (T \rightarrow U) \times list(T) \rightarrow list(U)$$

The problem is that we haven't said what T and U are. We really mean that for *any* type T and *any* type U, *map* can be given a type of that form:

$$map : \forall T, U.\ (T \to U) \times list(T) \to list(U)$$

What's the difference between the type *nlist* and the type $list(num)$? The former type hardcodes the fact that its elements are numbers, while the latter parameterizes the type of its elements. So $list(T)$ is a type which takes a parameter, and the paremeter is itself a type.

How can we type check a function like *map*, that uses types parameterized over T and U, when we don't know what T and U are? Let's apply a standard CS technique, namely, if we aren't sure of a value, we can parameterize it and supply it later. In this case, we make T and U parameters to *map*:

```
(define map
  (λ (T U)
    (λ (f : (T → U)  L : list(T))
      (cond
        [(empty? L) empty]
        [else (cons (f (first L))
                    (map f (rest L)))]))))
```

This function won't parse, since T and U 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 T and U and returns a function which operates on lists of the specified types. For example, we could use *map* with types *num* and *bool*:

```
({map num bool}
 (λ (x : num) (x > 0))
 (list 0 1 2 3))
```

Note: we'll use '{' and '}' to mark applications of functions to types.

The above version 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 T and U along:

```
(define map
  (λ (T U)
    (λ (f : (T → U)  L : list(T))
      (cond
        [(empty? L) empty]
        [else (cons (f (first L))
                    ({map T U} f (rest L)))]))))
```

Now every time we make a recursive call, the evaluation of $\{map\ T\ U\}$ creates a new closure. If we use *map* on a list with 20,000 items, we'll create 20,000 closures, which is tremendously inefficient!

We have another problem—what is the type of *cons*? It takes an element of type T and a list of type $list(T)$, then returns a list of type $list(T)$. So *cons* is parameterized over the type T just as *map* is. In fact, all the constructors, predicates, and selectors created when we defined the $list(T)$ datatype are parameterized over T. So we need to instantiate each of those functions with a particular type; for example, $\{empty?\ T\}$ is a predicate that only works on lists of type T. Now *map* looks like this:

```
(define map
  (λ (T U)
    (λ (f : (T → U)  L : list(T))
      (cond
        [({empty? T} L) {empty U}]
        [else ({cons U} (f ({first T} L))
                        ({map T U} f ({rest T} 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 T and U are specified as parameters to *map*, and all function parameters must be annotated with their types, what are the types of T and U 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 using Λ:

```
(define map
  (Λ (T U)
    (λ (f : (T → U)  L : list(T))
      (cond
        [({empty? T} L) {empty U}]
        [else ({cons U} (f ({first T} L))
                        ({map T U} f ({rest T} L)))]))))
```

The first phase evaluates the functions (denoted by Λ) and applications (denoted by '{' and '}') which perform type elaboration. The second phase evaluates the resulting program.

Now that we evaluate the function in two phases, how many closures do we create for *map*? Previously, we had to create 20,000 when operating on a list of

length 20,000, since we created a new closure on every recursive call. Now, we only create two, because all the type expansion is done in the first phase.

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.

The kind of polymorphism we've seen here is called *parametric 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.)