15-150 Lecture 8:
Polymorphism; Datatypes; Options

Lecture by Dan Licata

February 9, 2012

Thus far, we've been building a foundation for writing parallel functional programs (key tool: recursion), analyzing work and span (key tools: recurrences, big-O), and proving correctness (key tools: induction, equivalence). These are the basic ingredients of functional programming. But to a large extent, we haven't been taking advantage of the things that make ML fun and elegant to program in. Over the next few lectures, we're going to introduce some new features of ML that will make code a lot more concise and pretty.

1 Type Constructors

The idea of a list is not specific to integers. Here's a list of strings:

```
val s : string list = "a" :: ("b" :: ("c" :: []))
```

Here's a list of lists of integers:

```
val i2 : (int list) list = [1,2,3] :: [4,5,6] :: []
```

Note that `(int list) list` can also be written `int list list`.

In fact, there is a type `T list` for every type `T`. And we can *reuse* `[]` and `::` for a list with any type of elements. This abstracts over having different nils and conses for different types of lists.

2 Polymorphism

Some functions work just as well for any kind of list:

```
fun length (l : int list) : int =
    case l of
        [] => 0
      | x :: xs => 1 + length xs

fun length (l : string list) : int =
    case l of
        [] => 0
      | x :: xs => 1 + length xs
```

What's the difference between this code and the above? Nothing! Just the type annotation.

You can express that a function is *polymorphic* (works for any type) by writing

```
fun length (l : 'a list) : int =
    case l of
        [] => 0
      | x :: xs => 1 + length xs
```

This says that `length` works for a list of `'a`'s. Here `'a` (which is pronounced α) is a type variable, that stands for any type `'a`. You can apply length to lists of any type:

```
val 5 = length (1 :: (2 :: (3 :: (4 :: (5 :: [])))))
val 5 = length ("a" :: ("b" :: ("c" :: ("d" :: ("e" :: [])))))
```

The type of `length` is

```
length : 'a list -> int
```

and it's implicit in this that it means "for all `'a`".

Here's another example, `zip` from the last HW:

```
fun zip (l : int list, r : string list) : (int * string) list =
   case (l,r) of
      ([],_) => []
    | (_,[]) => []
    | (x::xs,y::ys) => (x,y)::zip(xs,ys)
```

Does it depend on the element types? No: it just shuffles them around. So we can say

```
fun zip (l : 'a list, r : 'b list) : ('a * 'b) list =
   case (l,r) of
      ([],_) => []
    | (_,[]) => []
    | (x::xs,y::ys) => (x,y)::zip(xs,ys)
```

instead.

That is, we can abstract over the pattern of zipping together two lists, and do it for all element types at once! This saves you from having to write out zip every time you have two kinds of lists that you want to zip together, which would be bad: (1) it's annoying to write that extra code, and (2) it's hard to maintain, because when you find bugs you have to make sure you fix it in all the copies. This kind of code reuse is very important for writing maintainable programs.

Note that `[]` and `::` are polymorphic:

```
[] : 'a list
:: : 'a * 'a list -> 'a list
```

3 Parametrized Datatypes

You can definite your own parametrized datatypes and polymorphic functions on them.

Let's represent the course grades database as a `datatype`:

```
datatype letter_grades =
    LEmpty
  | LNode of letter_grades * (string * string) * letter_grades
```

```
(* invariant: sorted according to andrew id, which is the first string
              at each node *)

val letters = Node(Node(Empty,("drl","B"),Empty),("iev","A"),Empty)

fun lookup_letter (d : letter_grades, k : string) : string =
    case d of
        LEmpty => raise Fail "not found"
      | LNode (l,(k',v),r) =>
            (case String.compare(k,k') of
                 EQUAL => v
               | LESS => lookup_letter(l,k)
               | GREATER => lookup_letter(r,k))
```

`letter_grades` is a *binary search tree*, where at each node we store a string like `"drl` and a grade like `"B"`.

Now suppose we switch to number grades:

```
datatype number_grades =
    NEmpty
  | NNode of number_grades * (string * int) * number_grades
(* invariant: sorted according to andrew id, which is the string
              at each node *)

val numbers = Node(Node(Empty,("drl",89),Empty),("iev",90),Empty)

fun lookup_number (d : number_grades, k : string) : int =
    case d of
        NEmpty => raise Fail "not found"
      | NNode (l,(k',v),r) =>
            (case String.compare(k,k') of
                 EQUAL => v
               | LESS => lookup_number(l,k)
               | GREATER => lookup_number(r,k))
```

You should abstract the repeated pattern, and write

```
datatype 'a grades =
    Empty
  | Node of 'a grades * (string * 'a) * 'a grades
(* invariant: sorted according to andrew id, which is the string
              at each node *)

(* if k is in d then
   lookup(d,k) returns the grades associated with k in d

   (don't call lookup when k is not in d)
*)
fun lookup (d : 'a grades, k : string) : 'a =
```

```
    case d of
        Empty => raise Fail "not found"
      | Node(l,(k',v),r) =>
            (case String.compare(k,k') of
                 EQUAL => v
               | LESS => lookup(l,k)
               | GREATER => lookup(r,k))

val letters : string grades =
   Node(Node(Empty,("drl","B"),Empty),("iev","A"),Empty)
val "B" = lookup(letters,"drl")

val numbers : int grades =
   Node(Node(Empty,("drl",89),Empty),("iev",90),Empty)
val 89 = lookup(numbers,"drl")
```

To parametrize a datatype, you put the type variables before the type's name, and use them that way in the types of the constructors (type constructors are applied postfix).

You can use `Node` and `Empty` to create dictionaries of different types. When you do type inference on `lookup`, the value component is underconstrained, so the function is polymorphic: it works for any `'a grades` database with grades of type `'a`.