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`.

4 Option Types

Sometimes, when I'm walking down the street, someone will ask me "do you know what time it is?" If I feel like being a literalist, I'll say "yes." Then they roll their eyes and say "okay, jerk, tell me what time it is!" The downside of this is that they might get used to demanding the time, and start demanding it of people who don't even know it.

It's better to ask "do you know what time is it, and if so, please tell me?"—that's what "what time is it?" usually means. This way, you get the information you were after, when it's available.

Here's the analogy:

```
doyouknowwhattimeitis? : person -> bool
tellmethetime : person -> int
whattimeisit : person -> int option
```

Options are a simple datatype:

```
datatype 'a option =
     NONE
   | SOME of 'a
```

Here `'a` is a type variable. This means there is a type `T option` for every type `T`. The same constructors can be used to construct values of different types. For example

```
val x : int option = SOME 4
val y : string option = SOME "a"
```

This is because `NONE` and `SOME` are polymorphic: they work for any type `'a`.

```
NONE : 'a option
SOME : 'a -> 'a option
```

4.1 Boolean Blindness

Don't fall prey to *boolean blindness*: boolean tests let you *look*, options let you *see*. If you write

```
case (doyouknowwhattimeitis? p) of
  true => tellmethetime p
  false => ...
```

The problem is that there's nothing about the *code* that prevents you from also saying `tellmethetime` in the false branch. The specification of `tellmethetime` might not allow it to be called—but that's in the math, which is not automatically checked.

On the other hand, if you say

```
case (whattimeisit p) of
  SOME t => ...
  NONE => ...
```

you naturally have the time in the `SOME` branch, and not in the `NONE` branch, without making any impolite demands. The structure of the code helps you keep track of what information is available—this is easier than reasoning about the meaning of boolean tests yourself in the specs. And the type system helps you get the code right.

Here's a less silly example. Returning to the `lookup` example from last time, when you say

```
case contains(letters,unknown) of
    true =>
        (* now we know that unknown is in letters *)
        ... lookup(letters,unknown) ...
  | false => (* do something else *) ...
```

there's nothing that prevents you from calling `lookup` in the false branch, except for some spec reasoning, which you can get wrong.

If you write a version of `lookup` using options:

```
(* if k is in d then
   lookup_opt(d,k) == SOME v, where v is the value associated with k in d

   if k is not in d then lookup_opt(d,k) == NONE
*)
fun lookup_opt (d : 'a grades, k : string) : 'a option =
    case d of
        Empty => NONE
      | Node(l,(k',v),r) =>
            (case String.compare(k,k') of
                 EQUAL => SOME v
               | LESS => lookup_opt(l,k)
               | GREATER => lookup_opt(r,k))
```

then you can eliminate the need to first look and then see. We weaken the pre-condition—the function works on any key and database. But this comes at the expense of weakening the post-condition—it no longer definitely returns a grade, but instead returns an `'a option`. In the process, we've fused `contains` and `lookup` into, so we can *see* what we looked at.

When you use lookup_opt, you naturally have the grade in the SOME branch, without doing any spec reasoning.

```
case lookup_opt(letters,unknown) of
   SOME grade => ... grade ...
 | NONE => (* do something else *)
```

4.2 Commitmentophobia

In ML, the type system the type system reminds you that lookup_opt might fail, and forces you to case on a value of option type. You can't treat a T option as a T, and write something like

```
"The grade is " ^ lookup_opt(letters,unknown)
```

when lookup_opt(letters,unknown) is actually NONE. You have to explicitly pass from the T option to the T by case-analyzing, and handle the failure case.

Another metaphor: Some of you might be dating people. Here's an analogy:

```
T = I'll see you tonight
T option = Maybe I'll see you tonight
```

If you say "I'll see you tonight" and then you don't, your boyfriend or girlfriend is gonna be mad. If you say "maybe I'll see you tonight" and you don't... well, they'll still be mad, but at least you won't feel like you did anything wrong. However, while saying maybe might be good when you're playing hard to get at the begining, if you never make any promises, eventually they'll get sick of the drama and break up with you.

Here's the point of this story: Languages like C and Java are for commitmentophobes, in that you can only say "maybe...". The reason for this is the *null pointer*: when you say String in Java, or int* in C, what that means is "maybe there is a string at the other end of this pointer, or maybe not". You'd write lookup_opt but give it the type of lookup, and so you'd never be sure that the grade is actually there. Tony Hoare calls this his "billion dollar mistake"—the cost of not making this distinction, in terms of software bugs and security holes, is huge.

In ML, you can use the type system to track these obligations. If you say T, you know you definitely absolutely have a T. If you say T option, you know that maybe you have one, or maybe not—and the type system forces you to handle this possibility. Thus, the distinction between T and T option lets you make appropriate promises, *using types to communicate.*

4.3 How not to program with lists

This whole discussion about options may seem obvious, but functional languages like Lisp and Scheme are based on the doyouknow/tellme style. For example, the interface to lists is

```
nil? -- check if a list is empty
first (car) -- get the first element of a list, or error if it's empty
rest (cdr) -- get the tail of a list, or error if it's empty
```

and you write code that looks like

```
(if (nil? l) <case for empty> <case for cons, using (first l) and (rest l)>)
```

Of course you can also accidentally call first and rest in the ¡case for empty¿, but they will fail.

On the other hand, if you write

```
case l of
    [] => ...
  | x :: xs => ...
```

then the code naturally lets you see what you looked at.