# 15-150 Lecture 9:
# Options; Domain-specific Datatypes

Lecture by Dan Licata

February 14, 2012

> Programs must be written for people to read, and only incidentally for machines to execute. — Abelson and Sussman, Structure and Interpretation of Computer Programs

Programming is about communicating with people: you, when you come back to a piece of code next year; other people on your project. This week we're going to take the kid gloves off, and talk about a few ways to make your ML programs more readable. The first is to use the type system to communicate. We'll talk about *options* and *domain-specific datatypes* as ways of doing this.

## 1 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
```

## 1.1 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. Suppose you define a type `'a tree` representing a dictionary mapping strings to something.

One interface for lookuping up in a dictionary would be

```
val contains : 'a tree -> string -> bool

(* If contains (t,k), then lookup(t,k) == v,
   where (k,v) occured at a node in t
 *)
val lookup : 'a tree -> string -> 'a
```

Then you would write code like

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

   if k is not in d then lookup(d,k) == NONE
*)
val lookup_opt : 'a tree * string -> 'a option
```

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 *)
```

## 1.2 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.

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.*

## 1.3 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.

## 2 Constructive Solid Geometry

Thus far, we've mostly used datatypes for things that feel like general-purpose data structures (lists, trees, orders, . . . ). Another important use of them is *domain-specific datatypes*: you define some datatype that are specific to an application domain. This lets you write clear and readable code.

As an example, we'll draw some pictures using *constructive solid geometry*: we construct pictures by combining certain basic shapes. In graphics, people make three-dimensional models of scenes they plan to turn into pictures or movies. A complicated three-dimensional object (say, Buzz Lightyear) is defined in terms of some basic shapes: spheres, rectangles, etc. This code is inspired by the idea of CSG but is highly simplified: we are only working in two dimensions, and the only thing your constructions will be able to do is report "black" or "white" for a particular point.

In the lab and homework, you will use shapes to make fractals—recursive self-similar shapes. To get started, in class we will consider the following shapes, where all points will be represented by Cartesian *xy* coordinates in the plane:

- a filled rectangle, specified by its lower-left corner and upper-right corner
- the union of two shapes, which contains all points that are either

We can represent these constructions using a datatype:

```
type point = int * int (* in Cartesian x-y coordinate *)

datatype shape =
    Rect of point * point
  | Union of shape * shape
```

### 2.1 Displaying shapes

I'm displaying shapes using a bunch of code for writing out a *bitmap file* that says what color each pixel should be. The interesting bit of this is telling whether a point is in a shape:

```
(* Purpose: contains(s,p) == true if p is in the shape,
   or false otherwise *)
fun contains (s : shape, p as (x,y) : point) : bool =
    case s of
        Rect ((xmin,ymin),(xmax,ymax)) =>
            xmin <= x andalso x <= xmax andalso
            ymin <= y andalso y <= ymax
      | Union(s1,s2) => contains(s1,p) orelse contains(s2,p)
```

This illustrates recursion over domain-specific datatypes: you have one branch for each constructor, and structural recursive calls on all the sub-shapes.

### 2.2 Programming Shapes

Why do we want to represent shapes in a programming language? Thus far, it would probably be easier to draw those pictures by hand. The advantage of having a programmatic representation is that we can program shapes.

For example, you will write some code to compute a bounding box:

```
(* boundingbox s computs a rectangle r (lower-left,upper-right),
   such that if p is in s then p is in r
   *)
fun boundingbox (s : shape) : point * point = ...
```

The bitmap-writing code uses the bounding box code to automatically choose the size of the bitmap.

In the homework, you will look at defining shapes recursively.

## 3 Reading: Polynomials

*Note for Spring, 2012. The following is another example of a domain-specific datatype that we used in past instances of the course.*

If you type a query like $(x+2)^2 = x(x+4)+4$ into Wolfram Alpha, it will say "yes, $(x+2)^2 = x(x+4)+4$". How does it do that?

### 3.1 A Datatype for Polynomials

To illustrate programming with domain-specific datatypes, we're going to represent polynomials and define an equality test for them. Remember from high school that a (univariate) polynomial is something like $x^2+2x+1$, built up using the variable $x$,constants, addition, and multiplication.

How should we represent polynomials?

One option is strings. The problem with this is that we'd then be dealing with the details of strings like `"x^2+2x+1"` all over the code.

Another idea is to use *coefficient lists*, which we'll talk about below. The problem with coefficient lists is that they are too lossy: we want to maintain enough information about what the user typed in that we can respond, "yes, $(x+2)^2 = x(x+4)+4$". Both of these have the same coefficient list, so we would say 'yes, $x^4+4x+4 = x^2+4x+4$".

A middle ground is to use a datatype to capture the important features of the problem domain, abstracting away from the concrete textual representation, but preserving some of the structure of the expression that was typed in. We can represent polynomails with a datatype as follows:

```
datatype poly =
    X
  | K of int
  | Plus of poly * poly
  | Times of poly * poly
```

Unlike lists or trees, this is a *domain-specific datatype*: you'd only use it if you were programming with polynomials.

*Values* are constructed by applying the datatype constructors. For example, $x^2+2x+1$ is represented by

```
Plus(Times(X,X),Plus(Times(K 2 , X), K 1))
```

This represents the *abstract syntax* of an expression as a tree, whose nodes label an operation (plus, times), and whose subtrees are the arguments to the operation.

Just like lists and trees, we can define functions using pattern matching and recursion. Here's how you apply a polynomial to an argument:

```
fun apply (p : poly , a : int) : int =
    case p of
        X => a
      | K c => c
      | Plus (p1 , p2) => apply (p1 , a) + apply (p2 , a)
      | Times (p1 , p2) => apply (p1 , a) * apply (p2 , a)
```

For example, `apply(Times(Plus(X, K 1), Plus(X, K 1)),4)` computes to 25, as you would expect. There are four cases, corresponding to the four *datatype constructors*. There are two recursive calls in the `Plus` and `Times` cases, corresponding to the two recursive references in the datatype definition.

## 3.2 Equality

How would you test whether two polynomials are equal? First, what does "equality" even mean? For example, you know that $(x+1)^2 = x^2 + 2x + 1$. These two polynomials are not syntactically the same—one starts with a `Plus` and the other a `Mult`. What we mean by equality is that they *agree on all arguments.*

We can't test this using just apply: we'd have to apply them to infinitely many arguments.

However, you learned how to do this in high school: put the polynomials in *normal form*

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \dots$$

and then compare the coefficients.

What we're going to do is to write a program to normalize a polynomial.

It is convenient to represent normal forms using a different type than `poly`, as lists of coefficients. E.g. `[c0,c1,c2,c3,...]` means

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \dots$$

```
(* represent c_0 x^0 + c_1 x + c_2 x^2 + ...
   by the list [c_0 , c_1 , c_2, ...]
   empty list is 0
*)
type nf = int list
```

Now, we write

```
val norm : poly -> nf
```

Working backward, we'll write `norm` as follows: we interpret each syntactic constructor as an operation on normal forms.

```
fun norm (p : poly) : nf =
    case p of
        X => xnf
      | K c =>  constnf c
      | Plus (t1 , t2) => plusnf (norm t1 , norm t2)
      | Times (t1 , t2) => timesnf (norm t1 , norm t2)
```

The operations we need are

```
val xnf : nf
val constnf : int -> nf
val plusnf : nf -> nf -> nf
val timesnf : nf -> nf -> nf
```

`norm` is a *ring homomorphism*: it interprets the operations of the syntactic ring given by the datatype `poly` as the corresponding ring operations on normal forms.

Some of these are obvious:

```
val xnf : nf = [0,1]
fun constnf (c : int) : nf = [c]
```

For addition, we just add the coefficients, or return the longer polynomial if one is zero:

```
fun plusnf (n1 : nf , n2 : nf) : nf =
    case (n1, n2) of
        ([] , n2) => n2
      | (n1 , []) => n1
      | (c1 :: cs1 , c2 :: cs2)  => (c1 + c2) :: plusnf (cs1 , cs2)
```

Multiplication is trickier. You can write it out explicitly, though we didn't do so in lecture. However, there is a nicer way to do it once we have *higher-order functions.*

The idea is FOIL:

$$(x+y)(z+w) = xz + xw + yz + yw$$

That is, we take the first summand in the first polynomial times the second polynomial, then the second summand in the first times the second. In general, it's the first summand in the first times the second, then the rest of the first times the second.

Here's how we implement it:

```
local
    (* Purpose: multiply each number in the list by c' *)
    fun multAll (n : nf , c' : int) : nf =
        case n of
            [] => []
          | c :: cs => (c * c') :: multAll (cs , c')

    (* Purpose: compute (c x^e) * n *)
    fun mult1 (n : nf, c : int , e : int) : nf =
        case e of
            0 => multAll (n , c)
               (* correct because c x^0 * n is c * n *)
          | _ => 0 :: mult1 (n , c , e - 1)
               (* correct because
                  (1) c x^e * n = x*(c x^(e-1) * n)
                  (2) for the coefficient list representation
                      x*n can be implemented by 0 :: n
                  *)

    (* if n1 = [c0,c1,c2,...]
       compute (c0x^e + c1 x^(e + 1) + ...) * n2
```

```
        *)
    fun times' (n1 : nf , n2 : nf, e : int) =
        case n1 of
            [] => []
          | c1 :: cs1 => plusnf (mult1 (n2 , c1 , e),
                                 times' (cs1 , n2 , e + 1))
                (* i.e.  (c1 x^e)*n2 + (c1 x^(e+1)*n2 + c2 x^(e+2)*n2 + ...)  *)
in
    fun timesnf (n1 : nf , n2 : nf) = times' (n1 , n2 , 0)
end
```

This function is long, but not difficult, if we break it down: `multAll` just multiplies each thing in a list by a coefficeint. `mult1` multiplies a normal form by a single term $cx^e$. Note that, for the coefficient representation, $x * n$ is implemented by just consing on 0 and shifting everything else down. `times'` just keeps track of the current exponent, and does what we said above: multiple the second polynomial by the first term, and then recursively by the rest. Finally, `times` starts the exponent off at zero.

Caveat: the normal forms we have computed so far are not quite unique: `[1,2,1,0]` and `[1,2,1]` both represent $1 + 2x + x^2$, though the former with a gratuitious $0x^3$. So to finish things off, we'd need to remove trailing zeroes, which we'll leave as an exercise.