15-150 Lecture 9:
Options; Domain-specific Datatypes; Functions as Arguments

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. Today we're going to take the kid gloves off, and talk about a few ways to make your ML programs more readable:

- Use the type system to communicate. We'll talk about *options* and *domain-specific datatypes* as ways of doing this.
- Abstract repeated patterns of code. We'll see our first examples of *higher-order functions*, which are an extremely powerful way 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. 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 *)
```

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.

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.*

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.

For example, say that we want to represent 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
- a disc, specified by its center and radius
- the intersection of two shapes, which contains all points that are in both
- the union of two shapes, which contains all points that are either
- the subtraction of two shapes, which contains all points in the first that are not in the second

We can represent these constructions using a datatype:

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

datatype shape =
    Rect of point * point
  | Disc of point * int
  | Intersection of shape * shape
  | Union of shape * shape
  | Without of shape * shape
```

For example, if we make a shape

```
Union(Rect((0,0),(100,100)),
      Union(Rect((100,100),(200,200)),
            Disc((100,100),40)))
```

it looks kind of like a bow tie:

Here's a more complicated shape:

```
val cat =
    let val body = Rect((100,50),(250,150))
        val tail = Rect((250,120),(375,135))
        val head = Disc((100,175),60)
        val earspace = Rect((75,205),(125,250))
        val lefteye = Disc((75,175),5)
        val eyes = Union(lefteye,
                         Translate (lefteye,(50,0)))
    in
```

```
        Union(Union(body,
                    tail),
              Without(head,
                      Union(earspace,eyes)))
    end
```

and here's how it renders:

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
      | Disc ((cx,cy),r) =>
            square(x - cx) + square(y - cy) <= square r
      | Intersection(s1,s2) => contains(s1,p) andalso contains(s2,p)
      | Union(s1,s2) => contains(s1,p) orelse contains(s2,p)
      | Without (s1,s2) => contains (s1,p) andalso (not (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

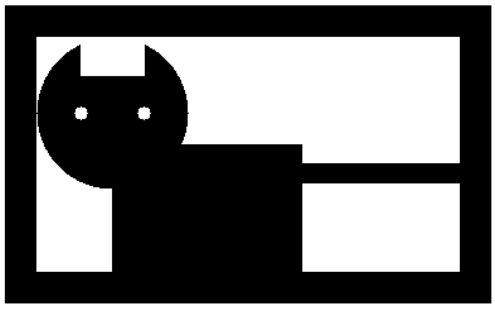
Why do we want to represent shapes in a programming language? Thus far, it would probably be easier to draw those pictures by hand. The advantage of having a programmatic representation is that we can program shapes.

For example, I wrote 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 = ...
```

We can use this to draw Schröedinger's cat:

```
Union (cat , rectb(boundingbox cat,25))
```



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

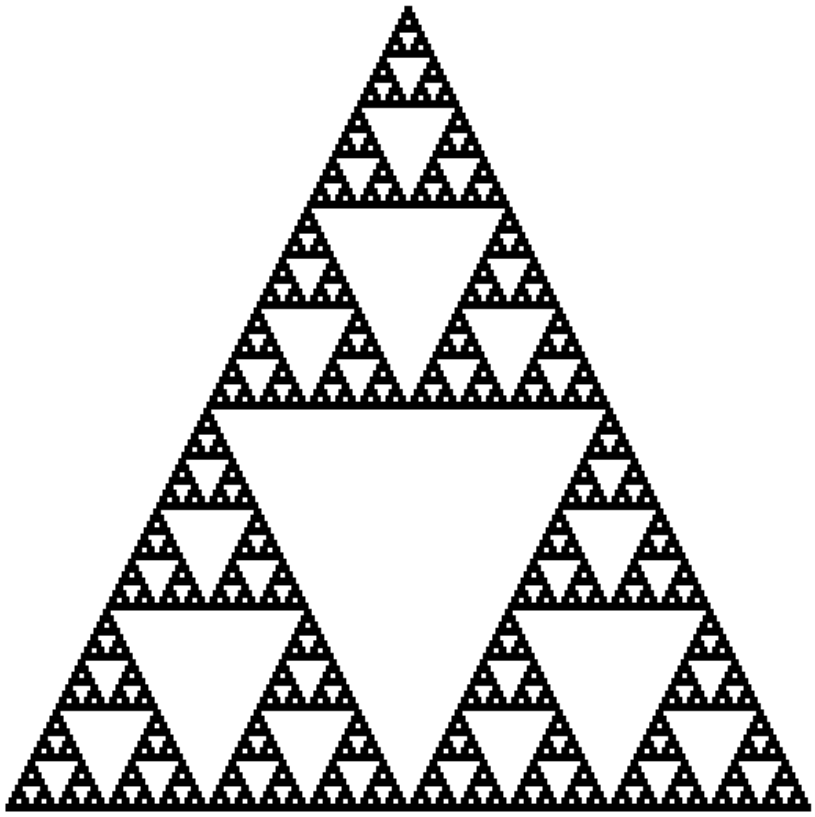
We can also define shapes recursively. The following example uses two new shape constructors, `Translate(s,(x,y))` (translate `s` by `(x,y)`) and `ScaleDown(s,(xf,xf))` (shrink `s` by the scale factor `xf` for the width and `yf` for the height); it's simple to extend `contains` for these:

```
(* assumes s is square and fills the bottom
   edge of its bounding box
```

```
 *)
fun sierp_step (s : shape) : shape =
    let val scaled = ScaleDown(s,(2,2))
        val side = let val ((minx,_),(maxx,_)) = boundingbox scaled
                   in maxx - minx
                   end
        val t1 = scaled
        val t2 = Translate(scaled,(side,0))
        val t3 = Translate(scaled,(side div 2,side))
    in
        Union(Union(t1,t2),t3)
    end

fun sierp (n : int) : shape =
    let
        fun loop i =
            case i of
                0 => Rect((0,0),(512,512))
              | _ => sierp_step (loop (i - 1))
    in
        Translate(loop n,(64,64))
    end
```

`sierp 7` renders like this:



3 Functions as Arguments

Next, we're going to talk about the thing that makes functional programming *functional*:

functions are values that can be passed to, and returned from, other functions

Consider the following two functions:

```
fun double (x : int) : int = x * 2
fun doubAll (l : int list) : int list =
    case l of
        [] => []
      | x :: xs => double x :: doubAll xs

fun raiseBy (l : int list , c : int) : int list =
    case l of
        [] => []
      | x :: xs => (x + c) :: raiseBy (xs,c)
```

How do they differ? They both do something to each element of a list, but they do different things (adding one, multiplying by `c`).

We want to write one function that expresses the pattern that is common to both of them:

```
fun map (f : int -> int , l : int list) : int list =
    case l of
        [] => []
      | x :: xs => f x :: map (f , xs)
```

The idea with `map` is that it takes a function `f : int -> int` as an argument, which represents what you are supposed to do to each element of the list.

`int -> int` is a type and can be used just like any other type in ML. In particular, a function like `map` can take an argument of type `int -> int`; and, as we'll see next time, a function can return a function as a result.

For example, we can recover `doubAll` like this:

```
fun doubAll l = map (double , l)
```

If you substitute `add1` into the body of `map`, you see that it results in basically the same code as before.

Anonymous functions Another way to instantiate `map` is with an anonymous function, which is written `fn x => 2 * x`:

```
fun doubAll l = map (fn x => 2 * x , l)
```

The idea is that `fn x => 2 * x` has type `int -> int` because assuming `x:int`, the body `2 * x : int`. To evaluate an anonymous function applied to an argument, you plug the value of the argument in for the variable. E.g. `(fn x => 2 * x) 3 |-> 2 * 3`. **Functions are values:** The value of `fn x => x + (1 + 1)` is `fn x => x + (1 + 1)`. You don't evaluate the body until you apply the function.

`doubAll` can be defined anonymously too:

```
val doubAll : int list -> int list = fn l => map (fn x => 2 * x , l)
```

Closures A somewhat tricky, but very very useful, fact is that anonymous functions can refer to variables bound in the enclosing scope. This gets used when we instantiate `map` to get `raiseBy`:

```
fun raiseBy (l , c) = map (fn x => x + c , l)
```

The function `fn x => x + c` adds `c` to its argument, where `c` bound as the argument to `raiseBy`. For example, in

```
    raiseBy ( [1,2,3] , 2)
|-> map (fn x => x + 2 , [1,2,3])
|-> (fn x => x + 2) 1 :: map (fn x => x + 2 , [2,3])
|-> 1 + 2 :: map (fn x => x + 2 , [2,3])
|-> 3 :: map (fn x => x + 2 , [2,3])
|-> 3 :: (fn x => x + 2) 2 :: map (fn x => x + 2 , [3])
 == 3 :: 4 :: map (fn x => x + 2 , [3])
 == 3 :: 4 :: (fn x => x + 2) 3 :: map (fn x => x + 2 , [])
 == 3 :: 4 :: 5 :: map (fn x => x + 2 , [])
 == 3 :: 4 :: 5 :: []
```

the `c` gets specialized to `2`. If you keep stepping, the function `fn x => x + 2` gets applied to each element of `[1,2,3]`. The important fact, which takes some getting used to, is that the function `fn x => x + 2` is *dynamically generated*: at run-time, we make up new functions, which do not appear anywhere in the program's source code!

Here's a puzzle: what does

```
let val x = 3
    val f = fn y => y + x
    val x = 5
 in
    f 10
 end
```

evaluate to?

Well, you know how to evaluate `let`: you evaluate the declarations in order, substituting as you go. So, you get

```
let val x = 3
    val f = fn y => y + 3
    val x = 5
 in
    f 10
 end
```

The fact that `x` is shadowed below is irrelevant; the result is `13`. This is one of the reasons why we've been teaching you the substitution model of evaluation all semester; it explains tricky puzzles like this in a natural way.

Polymorphism Another instance of the same pattern is:

```
fun showAll (l : int list) : string list =
    case l of
        [] => []
      | x :: xs => Int.toString x :: showAll xs
```

However, this is not an instance of `map`, because it transforms an `int list` into a `string list`.

We can fix this by giving `map` a polymorphic type:

```
fun map (f : 'a -> 'b , l : 'a list ) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map (f , xs)
```

Then both `doubAll` and `showAll` are instances:

```
fun doubAll l = map (fn x => 2 * x , l)
fun showAll l = map (Int.toString , l)
```

`map` is an example of what is called a *higher-order function*, a function that takes a function as an argument.

4 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?

4.1 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.

4.2 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*

$$c_0 + c_1x + c_2x^2 + c_3x^3 + \ldots$$

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

$$c_0 + c_1x + c_2x^2 + c_3x^3 + \ldots$$

```
(* 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.