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

Lecture by Dan Licata

February 14, 2012

1 Datatypes

Lists are built in to ML, and we have said that

> A list of integers (`int list`) is either
>
> > `[]`, or
> >
> > `x :: xs` where `x : int` and `xs : int list`.
>
> And that's it!

However, aside from some syntactic issues, lists could be defined by the following datatype definition:

```
datatype intlist =
    []
  | :: of int * intlist
```

Caveats: you can't actually use `[]` as a constructor name; you can say that constructors should be used in infix notation (to write `x :: xs`), but this doesn't; the bracket notation `[1,2,3]` is special to lists. But other than that, lists are just a datatype.

This generates *constructors*, `[]` and `::`, which can be used both to construct lists and to case-analyze them, as we have been doing.

Trees are defined as follows:

> A `tree` is either
>
> > `Empty`, or
> >
> > `Node(l,x,r)` where `x : int` and `l : tree` and `r : tree`.
>
> And that's it!

Trees are constructed by applying constructors:

```
val example = Node (Empty , 1 , Node (Empty , 3 , Empty))
```

However, trees are not built-in, so we have to define them. We do this using a datatype declaration:

```
datatype tree =
    Empty
  | Node of (tree * int * tree)
```

2 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.

This is defined by a parametrized datatype declaration:

```
datatype 'a list =
    []
  | :: of 'a * 'a list
```

3 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
```

3.1 Type inference

Here's another way to make your code easier to read: leave off unnecessary type annotations (we'll talk about what's necessary in a minute). Then *type inference* will fill in the types for you.

For example:

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

To figure out the type of this function, we (1) annotate with type variables, and (2) generate and solve constraints.

For example:

```
fun length (l : 'a1) : 'a2 =
    case l of
        [] => 0
      | x :: xs => 1 + length xs
```

We can reason as follows: the argument l has some type α_1 and the result is some type α_2. Because l gets case-analyzed with nil and cons patterns, it must be some kind of list, so we get the constraint $\alpha_1 = \alpha list$ for some type α. Because 0 gets returned from the function, we get the constraint $\alpha_2 = int$. Thus,

$$\alpha_1 = \beta list$$
$$\alpha_2 = int$$

This system of equations is *underconstrained*: these equations do not constrain α. So we make `length` polymorphic.

On the other hand, if we do the same for `sum`:

```
fun add(x:int,y:int) : int = x + y

fun sum l =
    case l of
        [] => 0
      | x :: xs => add (x , sum xs)
```

Then we get the constraints

$$\alpha_1 = \alpha list$$
$$\alpha_2 = int$$
$$\alpha = int$$

The last equation comes from the fact that in the second branch `x` has type α, and the `+` function is applied to `x`. These equations have a unique solution, where $\alpha_1 = intlist$, so `sum` does not have a polymorphic type.

On the third hand, if we screwed up the base case:

```
fun sum l =
    case l of
        [] => "hi"
      | x :: xs => add (x , sum xs)
```

$$\alpha_1 = \alpha list$$
$$\alpha_2 = string$$
$$\alpha_2 = int$$
$$\alpha = int$$

These constraints have no solution, so the code is ill-typed.

Now. Just because you can leave off types, doesn't mean you should: writing types is good documentation, and it will give you better error messages too (in math, there is no one equation to blame for a system not having a solution; so too in ML, there is no mathematically well-defined way to say who to blame for an unsatisfiable system of constraints). So, we will start allowing you to leave off types on `val` bindings (in a let, in test cases); however, at this point, we will require you to still write the types (and follow the rest of the methodology) for all functions. As the people who have to grade your code, we can definitely say that this makes it easier to read.

4 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
```

4.1 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.

4.2 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.

5 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?

5.1 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.

5.2 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*

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

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

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

```
(* 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.