15-150 Lecture 8:
Polymorphism; Specs and Checks; Datatypes

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.

Today's focus is on two things:

- You should abstract repeated patterns of code. We'll learn a couple of new tools for doing this: type constructors, polymorphism, and type inference.
- Along the way, we'll have an opportunity to discuss how to use specs and checks appropriately.

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
```

2.1 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.

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 Specs vs. checks

For more on specs vs. checks, there is a great GCD example in Chapter 25 of PSML.

When you have a specification for a function, the call sites:

- must *ensure* that the preconditions (assumptions about the inputs) hold before calling the function
- but *learn* that the postconditions (statements about the outputs) hold

For example, before you call `lookup`, you need to *know* that `k` is in `d`, and once you call it, you *learn* that `k`'s grade in `d` is `v` . This all happens at the specification level, in your mathematical reasoning about your code. For example, you might know that `k` is in `d` because you just inserted it.

What if you want to be lazy, and don't want to do a proof? Or what if you can't do a proof? E.g. suppose you want to call

```
lookup(d, <student you typed in>)
```

This is the point at which you need to do a run-time check:

> The purpose of a run-time check is to establish a spec.

Specs come first; checks fill in gaps in your knowledge.

Let's look at both sides of this: using specs to establish pre-conditions, and using specs to establish post-conditions.

Bullet-proofing *Bulletproofing* is doing a runtime check to establish a precondition.

For example, you could write a function

```
(* contains(d,k) == true if k is in d
                  == false if k is not in d *)
fun contains (d : 'a grades, k : string) : bool =
    case d of
        Empty => false
      | Node(l,(k',v),r) =>
            (case String.compare(k,k') of
                 EQUAL => true
               | LESS => contains(l,k)
               | GREATER => contains(r,k))
```

Then you can write code like

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

In the `true` branch, you know that `unknown` is in the database. In the `false` case, you have to handle the failure somehow.

In this case, it's not so hard to change the spec on `lookup` to say that it raises an exception when the key is not found. But in many cases it's much easier to write code under some pre-conditions (and let it fail in weird ways if they're violated), and then to do checks at the outside, if you can't prove the specs.

Another example of bulletproofing is `sublist_check` on HW3. This bulletproofed version is like defensive driving: no matter what the other guy does, you don't crash, or at least you crash in a well-defined and previously agreed-upon manner.

Doublechecking *Doublechecking* is doing a runtime check to ensure that a postcondition holds. You can use doublechecking to establish a *partial correctness guarantee*, even if the function you're calling is sometimes buggy. The failure of the check tells you to blame the implementation of the function. An example is `subset_sum_cert` and `subset_sum_dc` on HW3: `subset_sum_cert` produces a certificate, which `subset_sum_dc` can check independetly of the code for `subset_sum_cert`. This, whe the doublechecked version returns, we know that the result is correct—even if it is sometimes buggy.

Specs are more general than checks Mathematical specifications are *more general* than runtime checks: you can prove things that are undecidable, or computationally intractible, to check. For example, recall

```
raiseBy(l,a + b) == raiseBy(raiseBy(l,a),b)
```

There's no way to check at run-time that `raiseBy(l,s)` produced the same value as a succession of raises by any two numbers that sum to `s`.

As another example, recall

```
for all l, reverse(l) == fastReverse l
```

It's good to *prove* this equivalence (at compile-time), but using it as a run-time check would defeat the point of writing `fastReverse` in the first place, since you'd have to run the quadratic reverse algorithm in the check it.

Also, it's important to keep in mind that specs are more general than contracts (pre- and post-conditions): you might write a theorem that relates two calls to a function, or two or more functions:

```
raiseBy(l,a + b) == raiseBy(raiseBy(l,a),b)
zip(unzip l) == l
reverse(treeToList t) == treeToList(revT t)
```