CSE341: Programming Languages

Lecture 7
Functions Taking/Returning Functions

Dan Grossman

Fall 2011

On to first-class functions

“Functional programming” can mean a few different things:

1. Avoiding mutation in most/all cases (done and ongoing)

2. Using functions as values (the next week)

…

Recursion?

Mathematical definitions?

Not OO?

Laziness (later)?

First-class functions

- Functions are (first-class) values: Can use them *wherever* we use values
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, …
- Most common use is as an argument / result of another function
  - The other function is called a *higher-order function*
  - Powerful way to *factor out* common functionality
- 3-ish lectures on how and why to use first-class functions

Example

Can reuse **`n_times`** rather than defining many similar functions

- Computes **`f(f(…f(x)))`** where number of calls is **`n`**

```
fun n_times (f,n,x) =
   if n=0
   then x
   else f (n_times(f,n-1,x))

fun double x = x + x
fun increment x = x + 1
val x1 = n_times(double,4,7)
val x2 = n_times(increment,4,7)
val x3 = n_times(tl,2,[4,8,12,16,20])

fun double_n_times (n,x) = n_times(double,n,x)
fun nth_tail (n,x) = n_times(tl,n,x)
```

Types

- **`val n_times : ('a -> 'a) * int * 'a -> 'a`**

- Two of our examples *instantiated* **`'a`** with **`int`**
- One of our examples *instantiated* **`'a`** with **`int list`**
- This *polymorphism* makes **`n_times`** more useful

- Type is *inferred* based on how arguments are used (later lecture)
  - Describes which types must be exactly something (e.g., **`int`**) and which can be anything but the same (e.g., **`'a`**)

Polymorphism and higher-order functions

- Many higher-order functions are polymorphic because they are so reusable that some types, "can be anything"
- But some polymorphic functions are not higher-order
  - Example: **`length : 'a list -> int`**
- And some higher-order functions are not polymorphic
  - Example: **`times_til_0 : (int -> int) * int -> int`**

```
fun times_til_0 (f,x) =
   if x=0 then 0 else 1 + times_til_0(f, f x)
```

* Would be better with tail-recursion

Toward anonymous functions

- Definitions unnecessarily at top-level are still poor style:

```
fun triple x = 3*x
fun triple_n_times (f,x) = n_times(triple,n,x)
```

- So this is better (but not the best):

```
fun triple_n_times (f,x) =
  let fun trip y = 3*y
  in
     n_times(trip,n,x)
  end
```

- And this is even smaller scope
  - It makes sense but looks weird (poor style; see next slide)

```
fun triple_n_times (f,x) =
  n_times(let fun trip y = 3*y in trip end, n, x)
```

Anonymous functions

- This does not work: A function *binding* is not an *expression*

```
fun triple_n_times (f,x) =
  n_times((fun trip y = 3*y), n, x)
```

- This is the best way we were building up to: an expression form for *anonymous functions*

```
fun triple_n_times (f,x) =
  n_times((fn y => 3*y), n, x)
```

  – Like all expression forms, can appear anywhere
  – Syntax:
    - **`fn`** not **`fun`**
    - **`=>`** not **`=`**
    - no function name, just an argument pattern

Using anonymous functions

- Most common use: Argument to a higher-order function
  - Don't need a name just to pass a function

- But: Cannot use an anonymous function for a recursive function
  - Because there is no name for making recursive calls
  - If not for recursion, **`fun`** bindings would be syntactic sugar for **`val`** bindings and anonymous functions

```
fun triple x = 3*x

val triple = fn y => 3*y
```

A style point

Compare:

```
if x then true else false
```

With:

```
(fn x => f x)
```

So don't do this:

```
n_times((fn y => tl y),3,xs)
```

When you can do this:

```
n_times(tl,3,xs)
```

Map

```
fun map (f,xs) =
   case xs of
     [] => []
   | x::xs' => (f x)::(map(f,xs'))
```

```
map :  ('a -> 'b) * 'a list -> 'b list
```

Map is, without doubt, in the higher-order function hall-of-fame

- The name is standard (for any data structure)
- You use it *all the time* once you know it: saves a little space, but more importantly, *communicates what you are doing*
- Similar predefined function: **`List.map`**
  - But it uses currying (lecture 9)

Filter

```
fun filter (f,xs) =
   case xs of
     [] => []
   | x::xs => if f x
              then x::(filter(f,rest))
              else filter(f,rest)
```

```
filter :  ('a -> bool) * 'a list -> 'a list
```

Filter is also in the hall-of-fame

- So use it whenever your computation is a filter
- Similar predefined function: **`List.filter`**
  - But it uses currying (lecture 9)

Returning functions

- Remember: Functions are first-class values
  - For example, can return them from functions

- Silly example:

```
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

Has type `(int -> bool) -> (int -> int)`

But the REPL prints `(int -> bool) -> int -> int`
because it never prints unnecessary parentheses and
`t1 -> t2 -> t3 -> t4` means `t1->(t2->(t3->t4))`

Other data structures

- Higher-order functions are not just for numbers and lists
- They work great for common recursive traversals over your own data structures (datatype bindings) too
  - Example of a higher-order *predicate*:

    Are all constants in an arithmetic expression even numbers?

    Use a more general function of type
    `(int -> bool) * exp -> bool`

    And call it with **`(fn x => x mod 2 = 0)`**