

CSE 341: Programming Languages

Hal Perkins

Spring 2011

Lecture 8— Function Closures

Today

- Continue examples of functions taking and returning other functions
- Discuss *free variables* in function bodies
- In general, discuss environments and lexical scope
- See key idioms using first-class functions (more next time)

If you remember one thing...

We evaluate expressions in an environment, and function bodies in an environment extended to map arguments to values.

But which one? *The environment in which the function was defined!*

An equivalent description:

- Functions are values, but they're not just code.
- `fun f p = e` and `fn p => e` evaluate to values with two parts (a "pair"): the code and the current environment
- Function application evaluates the "pair"'s function body in the "pair"'s environment (extended)
- This "pair" is called a *(function) closure*.

There are *lots* of good reasons for this semantics.

For hw, exams, and competent programming, you must "get this"

Example 1

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x+y)
```

Example 2

```
val x = 1
```

```
fun f y = let val x = 2 in fn z => x + y + z end
```

```
val x = 3
```

```
val g = f 4
```

```
val y = 5
```

```
val z = g 6
```

Example 3

```
fun f g = let val x = 3 in g 2 end
```

```
val x = 4
```

```
fun h y = x + y
```

```
val z = f h
```

Scope

A key language concept: how are user-defined things *resolved*?

We have seen that ML has *lexically scoped* variables

Another (more-antiquated-for-variables, sometimes-useful) approach is *dynamic scope*

Example of dynamic scope: Exception handlers (where does `raise` (in Java `throw`) transfer control?)

Why lexical scope?

1. Functions can be reasoned about (defined, type-checked, etc.) where defined
2. Function meaning not related to choice of variable names
3. “Closing over” local variables creates private data; function definer *knows* function users cannot affect it

Example:

```
fun add_2x x = fn z => z + x + x
```

```
fun add_2x x = let val y = x + x in fn z => z + y end
```


Key idioms with closures

- Create similar functions
- Combine functions
- Pass functions with private data to iterators (map, *fold*, ...)
- Provide an ADT
- Partially apply functions (“currying”)
- As a *callback* without the “wrong side” specifying the environment.

(Will go through these today and Friday. See `lec8.sm1` for examples for first three idioms.)

In all cases, a closure’s “private fields” (i.e., free variables) are *essential*.

Why Google cares about functional iterators

Remember MapReduce? (fold is a slightly cleaner variant of reduce.)

Often the client of fold does not care what order the data is combined

- True for all 3 examples using `fold` in `lec8.sml`.
- Not true in general (e.g., is list sorted)

So what if we had *huge* arrays of data and 1000s of computers.

- Provide map for huge arrays of data; run in parallel
- Provide reduce for combining results; run in parallel then combine results across computers

Example: How many web pages have the phrase “Converse hightops”?

Example: Is “hightop” or “high-top” more common on the Web?

Key separation: MapReduce is a sophisticated fault-tolerant distributed system. Users just call map and reduce on some data.

Fault-Tolerance

At data-center scales, computers fail or become disconnected or start running too slow very often.

- If 1 computer has a hardware crash once every year on average, how long before 1 out of 10000 computers crash?

So part of MapReduce is redoing the computation parts that were given to a computer that fails.

But when can you take a computation, run it more than once, and know that's the same as running it exactly once?

When you do not use mutation!

A “new” style of programming: Computation in terms of maps and folds instead of sequences of assignment statements.

- Not new at all of course, just fairly new to BusinessWeek. :-)