

CS558 Programming Languages

Winter 2008

Lecture 10

FUNCTIONAL LANGUAGES

There are several widely-used functional languages, which share many features in common:

- Support first-class functions in a style based on the λ -calculus
- Pure programming style is encouraged (but not necessarily required)
- Good support for recursive data structures (especially **lists**)
- Implicit memory allocation and garbage collection

They differ in certain ways:

- **Scheme** (derived from Lisp): eager, impure, dynamically typed
- **ML**: eager, impure, statically typed
- **Haskell**: lazy, pure, statically typed

What does **functional** mean?

Functions are “**first-class**” values

- Can be passed as parameters or returned as results of other, **higher-order** functions

- Can be stored in data structures

- Supports more abstract programming style

Programs consist of **pure** functions with **no side-effects**

- Input/output description of problems

- Build programs by function composition

- No accidental or hidden coupling between functions

- Evaluation order can be either **eager** (based on call-by-value) or **lazy** (based on call-by-need)

FIRST-CLASS FUNCTIONS: FUNCTIONS AS PARAMETERS IN ML

```
fun map (g:int -> int,u: int list) : int list =
  case u of
    nil => nil
  | (h::t) => (g h)::(map (g,t))
```

```
fun inc x = x + 1
fun dec x = x - 1
val w = map (inc,[1,2,3]) yields [2,3,4]
val z = map (dec,[1,2,3]) yields [0,1,2]
```

ML also supports **anonymous function** values, i.e., functions that can be defined without being named. Could do above example as:

```
val w = map (fn x => x + 1, [1,2,3])
val z = map (fn x => x - 1, [1,2,3])
```

In fact, the following declarations are identical:

```
fun foo x = e
val rec foo = fn x => e -- here rec makes foo (potentially) recursive
```

Noticing that `map` passes along an unchanging parameter at each recursive call, we might refactor it this way using a **nested function**:

```
fun map (g:int -> int, u: int list) : int list =
  let fun f (v :int list) : int list =
        case v of
          nil => nil
        | (h::t) => (g h)::(f t)
      in f u
  end
```

This acts similarly to other nested declarations:

- Parameters and local variables of outer functions are visible within inner functions (using lexical scoping rules).
- Purpose: localize scope of nested functions, and avoid the need to pass auxiliary parameters defined in outer scopes.
- Semantics of a function definition now depend on values of function's **free variables**.

CURRIED FUNCTIONS

ML also provides syntactic sugar for such “**Curried**” functions:

```
fun map' (g:int->int) (u:int list) : int list =
  case u of
    nil => nil
  | (h::t) => (g h)::(map g t)
```

- When defining “multi-argument” functions in ML, have a choice between using a tuple argument and Currying.
- Can apply Curried version `map'` to either one or two arguments.
- Function application associates to the **left**, so

```
map' inc [2,4,6] = (map' inc) [2,4,6]
```

- Function type arrows associate to the **right**, so `map'` has type

```
(int -> int) -> int list -> int list =
(int -> int) -> (int list -> int list)
```

- Note: the “built-in” definition of `map` in the SML standard library is Curried (like `map'`).

In fact, we can simplify still further by rewriting `map` to **return** the nested function:

```
fun map' (g:int->int) : int list -> int list =
  let fun f (v: int list) : int list =
        case v of
          nil => nil
        | (h::t) => (g h)::(f t)
      in f
  end
```

Now we need a slightly different calling convention, passing the two arguments separately:

```
val w = map' inc [1,2,3] yields [2,3,4]
```

But now we can also pass just **one** argument at a time:

```
val minc = map' inc
val w = minc [1,2,3] yields [2,3,4]
val y = minc [4,5,6] yields [5,6,7]
```

CURRIED FUNCTIONS (2)

- Currying is most often useful when passing partially applied functions to **other** higher-order functions, e.g.:

```
fun pow (n:int) (b:int) : int =
  if n = 0 then 1 else b * (pow (n-1) b)
```

```
map (pow 3) [1,2,3] (yields [1,8,27])
```

- We can also store them in data structures:

```
fun each (fl: (int->int) list) (x:int) =
  case fl of
    nil => nil
  | (f:t) => (f x)::(each t x)
```

```
val powers_0_10 = each (map pow [0,1,2,3,4,5,6,7,8,9,10])
```

```
powers_0_10 2 (yields [1,2,4,8,16,32,64,128,256,512,1024])
powers_0_10 3 (yields [1,3,9,27,81,243,729,2187,6561,19683,59049])
```

CAPTURING ANOTHER PATTERN OF ABSTRACTION

Consider the following problems:

Sum a list of integers

```
fun sum l =
  case l of
    nil => 0
  | h::t => h + (sum t)
```

Multiply a list of integers:

```
fun prod l =
  case l of
    nil => 1
  | h::t => h * (prod t)
```

THE PATTERN CONTINUES...

Copy a list (of anything):

```
fun copy l =
  case l of
    nil => nil
  | h::t => h::(copy t)
```

Query: How does `copy` differ from the identity function `fn x => x`?

Calculate the length of a list (of anything):

```
fun len l =
  case l of
    nil => 0
  | h::t => 1 + (len t)
```

FOLDS

We can **abstract** over the common inductive pattern displayed by these examples:

```
fun foldr f n l =
  case l of
    nil => n
  | h::t => f(h, foldr f n t)
```

```
fun sum l = foldr (fn (x,y) => x+y) 0 l
fun prod l = foldr (op *) 1 l
fun copy l = foldr (op ::) nil l
fun len l = foldr (fn (_,y) => 1+y) 0 l
```

Function `foldr` computes a value working from the tail of the list to the head (from right to left). Argument `n` is the value to return for the `nil` list. Argument `f` is the function to apply to each element and the previously computed result.

FOLDS (2)

Can view `foldr f n l` as replacing each `::` constructor in `l` with `f` and the `nil` constructor with `n`. For example:

```
l = x1 :: (x2 :: (... :: (xn :: nil)))
foldr (op *) 1 l =
  x1 * (x2 * (... (xn * 1)))
```

SEMANTICS OF FIRST-CLASS FUNCTIONS

What is the “value” of a first-class function f ?

Roughly speaking, it is just f 's definition (parameters and body).

But nested functions can have free variables, defined in the enclosing scope. It is clear that the value of the function depends on the values of its free variables. How are they found?

Semantically, it suffices to know the static environment surrounding the **declaration** of f was encountered.

An **interpreter** can simply attach the current variable environment to its description of f when it encounters f 's declaration and records it in the function environment.

- When the interpreter applies f , it evaluates its body in an initial environment taken from the recorded description, which is then extended with f 's parameters.
- When the interpreter looks up a variable while executing f , it looks first among f 's locals and parameters, and then in the lexically-enclosing environment.

ASIDE: NOT QUITE FIRST-CLASS FUNCTIONS

Many languages support functions as values, but not in a fully first-class way.

For example, it is possible to pass functions as parameters to other functions in Pascal, Ada, ML, and C/C++ (though not directly by Java).

C/C++ also permit functions to be returned as results or stored in data structures.

The basic implementation idea is to represent each function value as a **pointer** to the compiled code of the function body.

FORMALIZING FUNCTION ENVIRONMENTS

Here are appropriate dynamic semantic rules (for eager evaluation):

$$\overline{\langle \text{fn } x \Rightarrow e, E, S \rangle} \Downarrow \overline{\langle [x, e, E], S \rangle} \quad (\text{Fn})$$

$$\frac{\langle e_1, E, S \rangle \Downarrow \langle [x, e', E'], S' \rangle \quad \langle e_2, E, S' \rangle \Downarrow \langle v', S'' \rangle \quad \langle e', E' + \{x \mapsto v'\}, S'' \rangle \Downarrow \langle v, S''' \rangle}{\langle (@ e_1 e_2), E, S \rangle \Downarrow \langle v, S''' \rangle} \quad (\text{Appl})$$

Can this semantics be implemented efficiently on real machines?

EXAMPLE: PARAMETERIZED ALGORITHMS IN C

```
typedef int (* leqfn) (int,int);

void isort(int n, int a[], leqfn leq) {
    int i,j,t;
    for (i = n-1; i >= 0; i--) {
        t = a[i];
        for (j = i; j < n-1 && leq(a[j+1],t); j++)
            a[j] = a[j+1];
        a[j] = t;
    }
}

int up(int p,int q) { return p <= q; }
int down(int p, int q) { return p >= q; }

int a[] = {2,1,3};
isort(3, a, up); /* a = {1,2,3} */
isort(3, a, down); /* a = {3,2,1} */
```

NESTED FUNCTIONS: IMPLEMENTATION ISSUES

If the language (e.g. Pascal) supports nested functions, and hence possible free variables, the generated code needs a way to access the values of these variables.

- If we use conventional activation records, the free variables for a function `p` live in the activation record of some **statically enclosing** function `q`.

Assuming `p`'s lifetime is contained within `q`'s lifetime, then can access `q`'s variables via a pointer to its activation record.

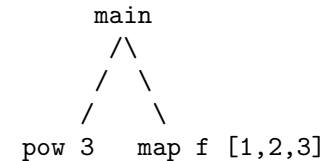
- Usually done by maintaining (at runtime) a **chain of static links** from each activation to the lexically enclosing function's activation.
- To access a free variable, the generated code de-references one or more links in the chain and then uses a known offset relative to link target. This has (modest) runtime cost.
- To pass `p` as a parameter to another function, we package its code address together with its own static link.

But what happens if the **lifetime** of `p` outlives that of `q`?

PROBLEMS WITH RETURNING FUNCTION VALUES

Consider activation tree for **map (pow 3)** example:

```
fun pow (n:int) (b:int) : int = ...
let val f = pow 3
in map f [1,2,3]
```



The activation of `pow` is no longer live when `map` is called!

If `n` is stored in a stack-allocated activation record for `pow`, it will be gone at the point where `f` needs it!

HEAP STORAGE FOR FUNCTION ACTIVATIONS

To avoid this problem:

- Pascal prohibits “upward funargs;” function values can only be passed downward, and can't be stored.
- Some other languages only permit “top-level” functions to be manipulated as values (in C, this means **all** functions!).

Functional languages supporting first-class nested functions must solve this problem by using the **heap** to store variables like `n`.

- Simple solution: Just put all activation records in the heap to begin with! Efficient garbage collection is a must! (SML/NJ does this.)
- More refined solution: Represent function values by a heap-allocated **closure** record, containing the function's code pointer and values of its **free** variables.
- Building closures involves taking **copies** of the values of the free variables, so only works when values are **immutable**. (Can always introduce extra level of indirection to achieve this.)