CS558 Programming Languages

Winter 2013

Lecture 5

FUNCTIONAL PROGRAMMING

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)

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

FIRST-CLASS FUNCTIONS: FUNCTIONS AS PARAMETERS IN OCAML

```
let rec map ((g:int -> int),(u:int list)) : int list =
    match u with
      [] -> []
    | h::t -> (g h)::(map (g,t))

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

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

```
let w = map ((fun x -> x + 1), [1;2;3])
let z = map ((fun x -> x - 1), [1;2;3])
```

In fact, the following declarations are identical:

```
let rec foo x = e
let rec foo = fun x -> e
```

NESTED FUNCTION DECLARATIONS

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

```
let rec map ((g:int -> int), (u: int list)) : int list =
  let rec f (v :int list) : int list =
    match v with
      [] -> []
    | h::t -> (g h)::(f t) in
  f u
```

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**.

FUNCTIONS AS RETURN VALUES

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

```
let map' (g:int->int) : int list -> int list =
  let rec f (v: int list) : int list =
    match v with
      [] -> []
    | h::t -> (g h)::(f t) in
  f
```

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

```
letl w = map' inc [1;2;3]
```
yields [2;3;4]

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

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

Curried Functions

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

```
let rec map' (g:int->int) (u:int list) : int list =
  match u with
    [] -> []
  | h::t -> (g h)::(map' g t)
```

• When defining “multi-argument” functions in OCaml, have a choice between using a tuple argument and Currying. Latter is better style.

• 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 Ocaml standard library is Curried (like `map'`).

Curried Functions (2)

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

```
let rec 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 functions in data structures:

```
let rec each (fl: (int->int) list) (x:int) =
    match fl with
      [] -> []
    | (f:t) -> (f x)::(each t x)

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

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

Python: Anonymous Functions

Python supports similar techniques, but sometimes with more awkward syntax.

Anonymous functions can be written using the `lambda` form, e.g. `(lambda x :  x + 1)`.

• Name derives from the Alonzo Church's "lambda calculus," originally invented to study computability theory, in which anonymous functions would be written using the Greek λ character, e.g. $\lambda x.x + 1$.

• Python restricts the bodies of lambda expressions to be expressions (not statements).

PYTHON: FIRST-CLASS FUNCTIONS

Functions can be nested, passed, returned, and stored. Here is the analog of our Curried `map` definition:

```
def map(g) :
    def f(v) :
       if v == None:
           return None
       else:
           h,t = v
           return (g(h), f(t))
    return f

def main() : print(map(lambda x: x+1)((1,(2,(3,None)))))
```

• No syntactic sugar available for Curried definitions.

• To make this example parallel to OCaml, we represented linked lists by nested tuples, rather than using Python's usual lists.

PYTHON: LIST OPERATIONS

More idiomatic Python would use built-in lists, and the built-in `map` function, which returns an **iterator**, which can in turn be fed to the `list` constructor.

```
list(map(lambda x: x+1, [1,2,3]))
```

Better still would be to use a **list comprehension**:

```
[x+1 for x in [1,2,3]]
```

or

```
[x+1 for x in range(1,4)]
```

which lets us build a list by mapping over the values of an iterator.

Python borrowed the comprehension syntax from Haskell, another well-known functional language.

CAPTURING ANOTHER PATTERN OF ABSTRACTION

Consider the following problems:

Sum a list of integers

```
let rec sum l =
    match l with
      [] -> 0
    | h::t -> h + (sum t)
```

Multiply a list of integers:

```
let rec prod l =
    match l with
      [] -> 1
    | h::t -> h * (prod t)
```

The Pattern Continues...

Copy a list (of anything):

```
let rec copy l =
   match l with
     [] -> []
   | h::t -> h::(copy t)
```

Query: How does `copy` differ from the identity function `fun x -> x` ?

Calculate the length of a list (of anything):

```
let rec len l =
   match l with
     [] -> 0
   | h::t -> 1 + (len t)
```

FOLDS

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

```
let rec foldr f n l =
  match l with
    [] -> n
  | h::t -> f h (foldr f n t)

let sum l = foldr (fun x y -> x+y) 0 l
let prod l = foldr ( * ) 1 l
let copy l = foldr (fun x y -> x::y) [] l
let len l = foldr (fun _ y -> 1+y) 0 l
```

Function `foldr` computes a value working from the tail of the list to the head (from **r**ight to left). Argument `n` is the value to return for the empty list. Argument `f` is the (Curried) 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 `[]` constructor with n. For example:

```
l = x1 :: (x2 :: (... :: (xn :: [])))
foldr (+) 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.

Formalizing Function Environments

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

$$\frac{}{\langle \texttt{fun}\ x\ \texttt{->}\ e, E, S\rangle \Downarrow \langle [x, e, E], S\rangle}\ \text{(Fun)}$$

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

Can this semantics be implemented efficiently on real machines?

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.

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:

```
let rec pow (n:int) (b:int) : int  =  ...
let f = pow 3
in map f [1;2;3]
```

```
     main
      /\
     /  \
    /    \
pow 3   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 allocate all activation records in the heap instead of the stack, and pass static links as in Pascal! Efficient garbage collection is a must! (Standard ML of New Jersey 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. (Most compiled language implementations, including OCaml, do this.)

Closure Example

Consider this very simple Curried function definition and use:

```
let foo x =
  let y = 1 in
  let bar z = x + y + z in
  bar
let f = foo 2
let g = f 3
```

The OCaml compiler will turn this into the equivalent of:

```
let bar' (clos,z) = clos.x + clos.y + z
let foo' x =
  let y = 1 in
  {func=bar',x=x,y=y}
let f = foo' 2
let g = f.func(f,3)
```

• Note that the closure includes **copies** of the values of the free variables, so must do some extra work if these variables are allowed to be **mutable**.

TAIL-CALLS REVISITED

Consider again the non-tail-recursive `len` function. We wrap it in a function that prints out the result:

```
let rec len l =
  match l with
    [] -> 0
  | h::t -> len t + 1 in
print_int (len ["a";"b"])
```

We can write this in a tail-recursive way like this:

```
let rec klen l k =
  match l with
    [] -> k 0
  | h::t -> klen t (fun i -> k (i+1)) in
klen ["a";"b"] print_int
```

This rather odd code was constructed by giving `klen` an additional argument, `k`, of type `int`$\rightarrow$`unit`. Instead of returning its "result" value, `klen` passes it to `k`.

Comparison of Computations

$\texttt{print\_int (len ["a";"b"])} \rightarrow$
$\texttt{print\_int (len ["b"] + 1)} \rightarrow$
$\texttt{print\_int ((len [] + 1) + 1)} \rightarrow$
$\texttt{print\_int ((0 + 1) + 1)} \rightarrow$
$\texttt{print\_int (1 + 1)} \rightarrow$
$\texttt{print\_int 2}$

$\texttt{klen ["a";"b"] print\_int} \rightarrow$
$\texttt{klen ["b"] (fun i}_1\texttt{ -> print\_int(i}_1\texttt{ + 1))} \rightarrow$
$\texttt{klen [] (fun i}_2\texttt{ -> (fun i}_1\texttt{ -> print\_int(i}_1\texttt{ + 1)) (i}_2\texttt{ + 1))} \rightarrow$
$\texttt{(fun i}_2\texttt{ -> (fun i}_1\texttt{ -> print\_int(i}_1\texttt{ + 1))(i}_2\texttt{ + 1)) 0} \rightarrow$
$\texttt{(fun i}_1\texttt{ -> print\_int(i}_1\texttt{ + 1))(0 + 1)} \rightarrow$
$\texttt{(fun i}_1\texttt{ -> print\_int(i}_1\texttt{ + 1)) 1} \rightarrow$
$\texttt{print\_int (1 + 1)} \rightarrow$
$\texttt{print\_int 2}$

Continuation-Passing Style

Notice that **every** call is now a **tail-call**. Note too that `klen` only returns after `print_int` is invoked; in essence, it needn't return at all. If it were the whole program, it wouldn't need to return at all. This is because `k` is serving the same role as a return address: saying "what to do next."

This means we can evaluate `klen` **without a stack**!

Functions like `k` are called **continuations** and programs written using them are said to be in **continuation-passing style (CPS)**.

We may choose to write (parts or all of) programs explicitly in CPS because it makes it easy to express a particular algorithm or because it clarifies the control structure of the program.

Note that CPS programs are just a subset of ordinary functional programs that happens to make heavy use of the (existing) enormous power of first-class functions. Remarkably, we can also systematically convert **any** functional program into an equivalent CPS program. (Details omitted.)

CONTINUATIONS

More broadly speaking, the term **continuation** means any representation of the program's state at a particular point during execution. This state must contain exactly the information needed to "continue" execution of the program from that point until the program completes.

CPS programs represent each continuation as a **function**, which, given the **value** of the expression being computed at that point, returns the result of the entire program.

If we are describing execution in terms of a low-level machine, we can think of a continuation for a program point as the machine state at that point: the values of the pc, return stack, etc. These contain the same information that would go into a closure for the continuation function.