CIS 500

Software Foundations

Fall 2003

10 September

Study Groups

[Send mail to cis500 if you want us to help you get into one.]

[Plans: one more lecture on ML, then on to TAPL.

We'll be talking today about programming with functions as data.

In a sense, this is what the whole course is about — functions [programs] as objects of study in their own right.]

Multi-parameter functions

We have seen two ways of writing functions with multiple parameters:

```
# let foo x y = x + y;;
val foo : int -> int -> int = <fun>

# let bar (x,y) = x + y;;
val bar : int * int -> int = <fun>
```

The first takes its two arguments separately; the second takes a tuple and uses a pattern to extract its first and second components.

The syntax for applying these two forms of function to their arguments differs correspondingly:

```
# foo 2 3;;
- : int = 5

# bar (4,5);;
- : int = 9

# foo (2,3);;
This expression has type int * int
but is here used with type int

# bar 4 5;;
This function is applied to too many arguments
```


Partial Application

One advantage of the first form of multiple-argument function is that such functions may be partially applied.

```
# let foo2 = foo 2;;
val foo2 : int -> int = <fun>

# foo2 3;;
- : int = 5

# foo2 5;;
- : int = 7

# List.map foo2 [3;6;10;100];;
- : int list = [5; 8; 12; 102]
```


Currying

Obviously, these two forms are closely related — given one, we can easily define the other.

```
# let foo' x y = bar (x,y);;
val foo' : int -> int -> int = <fun>

# let bar' (x,y) = foo x y;;
val bar' : int * int -> int = <fun>
```


Currying

Indeed, these transformations can themselves be expressed as (higher-order) functions:

```
# let curry f x y = f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

# let foo'' = curry bar;;
val foo'' : int -> int -> int = <fun>

# let uncurry f (x,y) = f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

# let bar'' = uncurry foo;;
val bar'' : int * int -> int = <fun>
```

A Closer Look

The type `int -> int -> int` can equivalently be written `int -> (int -> int)`.

That is, a function of type `int -> int -> int` is actually a function that, when applied to an integer, yields a function that, when applied to an integer, yields an integer.

Similarly, an application like `foo 2 3` is actually shorthand for `(foo 2) 3`.

Formally: `->` is right-associative and application is left-associative.

Anonymous Functions

It is fairly common in OCaml that we need to define a function and use it just once.

```
# let timesthreeplustwo x = x*3 + 2;;
val timesthreeplustwo : int -> int = <fun>

# List.map timesthreeplustwo [4;3;77;12];;
- : int list = [14; 11; 233; 38]
```

To save making up names for such functions, OCaml offers a mechanism for writing them in-line:

```
# List.map (fun x -> x*3 + 2) [4;3;77;12];;
- : int list = [14; 11; 233; 38]
```


Anonymous Functions

The following let-bindings are completely equivalent:

```
# let double x = x*2;;
val double : int -> int = <fun>

# let double' = (fun x -> x*2);;
val double' : int -> int = <fun>

# double 5;;
- : int = 10

# double' 5;;
- : int = 10
```


Anonymous Functions

We can even write:

```
# (fun x -> x*2) 5;;
- : int = 10
```

First-class functions

Functions in OCaml are first class — they have the same rights and privileges as values of any other types. E.g., they can be

- passed as arguments to other functions
- returned as results from other functions
- stored in data structures such as tuples and lists
- etc.

E.g...

Quick Check

What is the type of `l`?

```
# let l = [ (fun x -> x + 2);
            (fun x -> x * 3);
            (fun x -> if x > 4 then 0 else 1) ];;
```


A list of functions

```
# let l = [ (fun x -> x + 2);
            (fun x -> x * 3);
            (fun x -> if x > 4 then 0 else 1) ];;
val l : (int -> int) list = [<fun>; <fun>; <fun>]

# let applyto x f = f x;;
val applyto : 'a -> ('a -> 'b) -> 'b = <fun>

# List.map (applyto 10) l;;
- : int list = [12; 30; 0]

# List.map (applyto 2) l;;
- : int list = [4; 6; 1]
```


[supplement with fold (and show how to write the rest using fold - note, for later, that fold is a kind of universal iterator for lists)

write length in terms of map and fold

write factorial in terms of fold

write forall and exists, straight and in terms of map and fold]

[Lazy streams - infinite-length lists of numbers

infinitely ascending sequence

sieve of E.

One more nice stream example (e.g., from Bird and Wadler, or Thompson, or something)

efficiency: memoization (data structures with a purely functional interface but stateful guts]

[maybe: the compose function]


```
continuations
  review of tail recursion
  more general idea of a tail call
  if a call is not a tail call, then we can wrap up its ‘‘continuation’’
    as an explicit function that is applied to the call’s result
  in fact, we can go further -- we can pass the continuation to the
    function, giving *it* the responsibility for calling it at the
    appropriate moment
  note that the original call becomes a tail call
  indeed, we can rewrite *any* program in ‘‘continuation passing style,’’
    making all calls into tail calls
  this is a generalization of the transformation that we did on fact to
    make it tail recursive
  ==> look up Friedman et al article on continuations for concurrency

other interesting topics
  refs, memoization, delay/force, lazy functional programming
    (or is this the advanced section??)
```