

CIS 500
Software Foundations
Fall 2004
27 September

Administrivia

- ◆ Second homework assignment was due at noon.
- ◆ Third homework assignment is due one week from today.
- ◆ Should already be reading TAPL Chapter 5.

Programming with Functions in OCaml

Functions as Data

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.

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 curry' = curry bar;;
val curry' : 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

Anonymous functions may appear, syntactically, in the same places as values of any other types.

For example, 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
```

Or (slightly more usefully):

```
# (if 5*5 > 20 then (fun x -> x*2) else (fun x -> x+3)) 5;;  
- : int = 10
```

The conditional yields a function on the basis of some boolean test, and its result is then applied to 5.

Quick Check

What is the type of 1?

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

Applying 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>]; <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]
```

Another useful higher-order function: fold

```
# let rec fold f l acc =  
  match l with  
  | a::l -> f a (fold f l acc);;  
  val fold : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

For example:

```
# fold (fun a b -> a + b) [1; 3; 5; 100] 0;;  
- : int = 109
```

In general:

```
f [a1; ...; an] b  
is  
f a1 (f a2 (... (f an b) ...)).
```


Using fold

Most of the list-processing functions we have seen can be defined compactly in terms of `fold`:

```
# let listSum l =  
  fold (fun a b -> a + b) 1 0;;  
val listSum : int list -> int = <fun>  
  
# let length l =  
  fold (fun a b -> b + 1) 1 0;;  
val length : 'a list -> int = <fun>
```

Using fold

```
# let map f l =  
  fold (fun a b -> (f a) :: b) l [];;  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>  
# let filter p l =  
  fold (fun a b -> if p a then (a::b) else b) l [];;
```

Using fold

And even:

```
# (* List of numbers from m to n, as before *)
let rec fromTo m n =
  if n < m then []
  else m :: fromTo (m+1) n;;
val fromTo : int -> int -> int list = <fun>

# let fact n =
  fold (fun a b -> a * b) (fromTo 1 n) 1;;
val fact : int -> int = <fun>
```

Quick Check

What is the type of this function?

```
# let foo l =  
  fold (fun a b -> List.append b [a]) l [];;
```

What does it do?

Forms of fold

The OCaml `List` module actually provides two folding functions:

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b
```

The one we're calling `fold` (here and in the homework assignment) is

`List.fold_right`.

`List.fold_left` performs the same basic operation but takes its arguments in a different order.

The unit type

OCaml provides another built-in type called `unit`, with just one inhabitant, written `()`.

```
# let x = ();;  
val x : unit = ()  
  
# let f () = 23 + 34;;  
val f : unit -> int = <fun>  
  
# f ();;  
- : int = 57
```

Why is this useful?

Uses of unit

A function from `unit` to `'a` is a **delayed computation** of type `'a`.

When we define the function...

```
# let f () = <long and complex calculation>;  
val f : unit -> int = <fun>
```

... the **long and complex calculation** is just boxed up in a **closure** that we

can save for later (by binding it to a variable, e.g.).

When we actually need the result, we apply `f` to `()` and the calculation

actually happens:

```
# f () ;;  
- : int = 57
```

Thunks

A function accepting a `unit` argument is often called a `thunk`.
Thunks are widely used in functional programming.

A typical example...

Suppose we are writing a function where we need to make sure that some “finalization code” gets executed, even if an exception is raised.

```
# let read file =
  let chan = open_in file in
  try
    let nbytes = in_channel_length chan in
    let string = String.create nbytes in
    really_input chan string 0 nbytes;
    close_in chan;
    string
  with exn ->
    (* finalize channel *)
    close_in chan;
    (* re-raise exception *)
    raise exn;;
```

We can avoid duplicating the finalization code by wrapping it in a thunk:

```
# let read file =  
  let chan = open_in file in  
  let finalize () = close_in chan in  
  try  
    let nbytes = in_channel_length chan in  
    let string = String.create nbytes in  
    really_input chan string 0 nbytes;  
    finalize();  
    string  
  with exn ->  
    (* finalize channel *)  
    finalize();  
    (* re-raise exception *)  
    raise exn;;
```

In fact, we can go further...

```
# let unwind_protect body finalize =
  try
    let res = body() in
      finalize();
    res
  with exn ->
    finalize();
    raise exn;;
# let read file =
  let chan = open_in file in
    unwind_protect
      (fun () ->
        let nbytes = in_channel_length chan in
          let string = String.create nbytes in
            really_input chan string 0 nbytes;
            string)
      (fun () -> close_in chan);;
```