CMSC 330: Organization of Programming Languages

Functional Programming with OCaml, con't.

Modules

- So far, most everything we've defined has been at the "top-level" of OCaml
  - This is not good software engineering practice
- A better idea: use *modules* to group associated types, functions, and data together
  - Avoid polluting the top-level with unnecessary stuff
- For lots of sample modules, see the OCaml standard library

Creating a Module

```
module Shapes =
  struct
    type shape =
       Rect of float * float    (* width * length *)
     | Circle of float          (* radius *)

    let area = function
       Rect (w, l) -> w *. l
     | Circle r -> 3.14 *. r *. r

    let unit_circle = Circle 1.0
  end;;

unit_circle;;    (* not defined *)
Shapes.unit_circle;;
Shapes.area (Shapes.Rect (3.0, 4.0));;
open Shapes;;    (* import all names into current scope *)
unit_circle;;    (* now defined *)
```


Modularity and Abstraction

- Another reason for creating a module is so we can *hide* details
  - For example, we can build a binary tree module, but we may not want to expose our exact representation of binary trees
  - This is also good software engineering practice
    - Prevents clients from relying on details that may change
    - Hides unimportant information
    - Promotes local understanding (clients can't inject arbitrary data structures, only ones our functions create)

Module Signatures

Entry in signature

Supply function types

```
module type MOD =
  sig
    val add : int -> int -> int
  end;;

module MyModule : MOD =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;;

MyModule.add 3 4;;     (* OK *)
MyModule.mult 3 4;;    (* not accessible *)
```

Give type to module

Module Signatures (cont'd)

- The convention is for signatures to be all capital letters
  - This isn't a strict requirement, though
- Items can be omitted from a module signature
  - This provides the ability to hide values
- The default signature for a module hides nothing
  - You'll notice this is what OCaml gives you if you just type in a module with no signature at the top-level

Abstract Types in Signatures

```
module type SHAPES =
  sig
    type shape
    val area : shape -> float
    val unit_circle : shape
    val make_circle : float -> shape
    val make_rect : float -> float -> shape
end;;

module Shapes : SHAPES =
  struct
    ...
    let make_circle r = Circle r
    let make_rect x y = Rect (x, y)
  end
```

- Now the definition of **`shape`** is hidden

Abstract Types in Signatures

```
# Shapes.unit_circle
- : Shapes.shape = <abstr>  (* OCaml won't show impl *)
# Shapes.Circle 1.0
Unbound Constructor Shapes.Circle
# Shapes.area (Shapes.make_circle 3.0)
- : float = 29.5788
# open Shapes;;
# (* doesn't make anything abstract accessible *)
```

- How does this compare to modularity in...
  - C?
  - Java?
  - Ruby?

.ml and .mli files

- Put the signature in a my_module.mli file, the struct in a my_module.ml file
  - Use the same names
  - Omit the sig...end and struct...end parts
  - The OCaml compiler will make a MyModule module from these

Example

shapes.mli

```
type shape
val area : shape -> float
val unit_circle : shape
val make_circle : float -> shape
val make_rect : float -> float -> shape
```

shapes.ml

```
type shape =
  Rect of ...
...
let make_circle r = Circle r
let make_rect x y = Rect (x, y)
```

```
% ocamlc shapes.mli   # produces shapes.cmi
% ocamlc shapes.ml    # produces shapes.cmo
ocaml
# #load "shapes.cmo"  (* load Shapes module *)
```


Functors

- Modules can take other modules as arguments
  - Such a module is called a *functor*
  - You're mostly on your own if you want to use these
- Example: **`Set`** in standard library

```
module type OrderedType = sig
  type t
  val compare : t -> t -> int
end

module Make(Ord: OrderedType) =
struct ... end

module StringSet = Set.Make(String);;
(* works because String has type t,
   implements compare *)
```


So Far, only Functional Programming

- You haven't seen *any* way so far to change something in memory
  - All you can do is create new values from old
- This actually makes programming *easier* !
  - Don't care whether data is shared in memory
    - Aliasing is irrelevant
  - Provides strong support for compositional reasoning and abstraction
    - Example: calling a function f with argument x always produces the same result

Imperative OCaml

- There are three basic operations on memory:
  - `ref : 'a -> 'a ref`
    - Allocates an updatable reference
  - `! : 'a ref -> 'a`
    - Returns the value stored in a reference
  - `:= : 'a ref -> 'a -> unit`
    - Updates a reference

```
let x = ref 3   (* x : int ref *)
let y = !x
x := 4
```


Comparison to lvalues and rvalues

- Recall that in Java or C, there's a strong distinction between lvalues and rvalues
  - An *rvalue* refers to just a value, like an integer
  - An *lvalue* refers to a location that can be written
- A variable's meaning depends on where it appears
  - In places like the right-hand side of an assignment, it's an rvalue, and it refers to the contents of the variable
  - In places like the left-hand side of an assignment, it's an lvalue, and it refers to the location the variable is stored in

Lvalues and rvalues (cont'd)

Store 3 in location x

```
int x, y;

x = 3;

y = x;

3 = x;
```

Read contents of x and store in location y

Makes no sense

- Notice that x, y, and 3 all have type **`int`**

Comparison to OCaml

```
int x, y;

x = 3;

y = x;

3 = x;
```

```
let x = ref 0;;
let y = ref 0;;

x := 3;;   (* x : int ref *)

y := (!x);;

3 := x;;   (* 3 : int; error *)
```

- In OCaml, an updatable location and the contents of the location have different types
  - The location has a **`ref`** type

Capturing a ref in a Closure

- We can use refs to make things like counters that produce a fresh number “everywhere”

```
let next =
  let count = ref 0 in
    function () ->
      let temp = !count in
        count := (!count) + 1;
        temp;;

# next ();;
- : int = 0
# next ();;
- : int = 1
```


Semicolon Revisited; Side Effects

- Now that we can update memory, we have a real use for ; and () : unit
  - e1; e2 means evaluate e1, throw away the result, and then evaluate e2, and return the value of e2
  - () means “no interesting result here”
  - It’s only interesting to throw away values or use () if computation does something besides return a result
- A *side effect* is a visible state change
  - Modifying memory
  - Printing output or reading input
  - Writing to disk

Grouping with begin...end

- If you’re not sure about the scoping rules, use begin...end to group together statements with semicolons

```
let x = ref 0

let f () =
  begin
    print_string "hello";
    x := (!x) + 1
  end
```


The Trade-Off of Side Effects

- Side effects are absolutely necessary
  - That’s usually why we run software! We want something to happen that we can observe
- They also make reasoning harder
  - Order of evaluation now matters
  - Calling the same function in different places may produce different results
  - Aliasing is an issue
    - If we call a function with refs r1 and r2, it might do strange things if r1 and r2 are aliased

OCaml Language Choices

- Implicit or explicit declarations?
  - Explicit – variables must be introduced with let before use
  - But you don't need to specify types

- Static or dynamic types?
  - Static – but you don't need to state types
  - OCaml does *type inference* to figure out types for you
  - Good: less work to write programs
  - Bad: easier to make mistakes, harder to find errors