Modules

Ziyan Maraikar

September 19, 2014

Lecture Outline

1. Ocaml Module language
2. Structures: module implementation
3. Signatures: module interfaces
4. ADTs: type abstraction
5. Building and executing Ocaml programs

Modularity

We can make a software system *modular* by structuring it as a set of units that are *loosly coupled*, that is, relatively independent of each other.

This aids

- ★ understanding large systems.
- ★ making modifications without affecting the rest of the system.
- ★ reuse of existing code.
- ★ independent development of modules in parallel.

Language support

Language support for modularity must address several aspects:

namespaces Prevent clashes due to use of the same name in different parts of the program or in different libraries. Supported via *packages* in OO languages.

interface A specfication containing the *operations* supported by a particular module.
Module signatures in Ocaml and *interfaces* in OO languages provide interface specification.

implementation The actual code to implement the operations specified by an interface.
Module structures in Ocaml and *classes* in OO languages provide implementation specification.

C and Javascript lack some of these features, making it difficult to structure large programs.

Lecture Outline

Structures

Structures define the implementation details of a module. It can contain types, functions and, nested modules.

Structure syntax

`module` *Name* = `struct`
implementation definitions
`end`

Example:

```
module Circle = struct
  let pi = 3.14159
  let circum r = 2.0 *. pi *. r
  let sq x = x *. x
  let area r = pi *. (sq r)
end
```

Qualified names

- To reference functions and types in another module, prefix them with the module name separated by a dot.
  e.g. `Circle.area 1.0`.
- Alternatively, you may `open` a module, and use unqualified names.

```
let open Circle in
  sq pi
```

Opening two modules defining the same names causes confusion, so always use the *local open* shown here.

Nested modules

content...

Lecture Outline

Implementation hiding

- The key to constructing larger systems is defining clear interfaces between modules.
- A *signature* (module type) can be used to restrict the visibility of implementation details.
- A module can implement many signatures. Conversely, a signature may be implemented by many modules.

Signature syntax

module type *Name* = sig
interface definitions
end

Default signature

A module has a *default signature* that exposes all implementation details.

UTop echos back this default signature when you enter a module definitions.

```
module Circle: sig
  val pi : float
  val circum : float -> float
  val sq : float -> float
  val area : float -> float
end
```

Note the similarity with C function prototypes.

Implementation hiding

To *hide* the function such as `sq`, we simply omit it from the signature.

```
module type CIRCLE = sig
  val pi : float
  val circum : float -> float
  val area : float -> float
end

module Circle :CIRCLE = struct
  ...
end
```

The `sq` function is no longer visible.

```
Circle.sq 6. ;;
Error: Unbound value Circle.sq
```

Lecture Outline

Abstract data types

- We have seen several examples of data structures, e.g. lists and trees.
- Data structures *concretely* specify how they are represented using a type definition.
- In contrast, an *abstarct data type* (ADT) hides its representation and specifies just the available operations on its type.
- We perform *type abstraction* in module signatures when specifying ADTs in Ocaml.

Set ADT signature

Sets are mathematical objects that support various operations.

```
module type SET = sig
  type 'a set
  (** the empty set **)
  val empty : 'a set
  (** insert element into given set **)
  val insert : 'a -> 'a set -> 'a set
  (** is element a member of given set? **)
  val member : 'a -> 'a set -> bool
end
```

Note that type `set` is *abstract*, we have not specified how it is represented.

This gives us the flexibility to implement `Set` using any suitable data structure.

Defining Set ADT operations

```
(* Set ADT implemented as a sorted list of unique elements *)
module ListSet: SET = struct
  type 'a set = 'a list
  let empty = []

  let rec member x l =
    match l with
    | [] -> false
    | hd::tl -> (x = hd) || ((x > hd) && (member x tl))

  let rec insert x l =
    match l with
    | [] -> [x]
    | hd::tl -> if x < hd then x::l else if x = y then l
                else hd::(insert hd tl)
end
```

Benefits of type abstraction

- We can only manipulate `set`s using the operations in `ListSet`.

  ```
  # ListSet.insert 1  ListSet.empty ;;
  - : int ListSet.set = <abstr>
  ```

  The internal use of `list`s to represent sets is completely hidden.
- Invalid manipulations like violating the sorted order, or uniqueness of a `list` that represents a set, are automatically prevented.
- We can define multiple implementations of the `Set` interface, e.g. a more efficient `BSTset` based on a binary search tree. The *client code* that uses `Set` remains unchanged even when changing implementations.
- An ADT implementation and its clients can be coded independently once an interface has been agreed on.

Pitfalls of depending on concrete types

Suppose we implement `ListSet` without bothering to define a signature for the `SET` ADT.

```
module ListSet
  ...
end
```

We lose all the benefits of type abstraction with this implementation.

```
# ListSet.insert 1 ListSet.empty ;;
- : int list = [1]
```

Pitfalls of depending on concrete types

We are prone to errors due to `list` being visible when dealing with sets.

```
# let s1 = ListSet.insert 1 ListSet.empty ;;
- : int list = [1]
# let s2 = 2::s1 ;;
- : int list = [2; 1]
# ListSet.insert 1 s2;;
- : int list = [1; 2; 1]
```

This is the hallmark of a lack of modularity!

Lecture Outline

Source files, structures and signatures

An Ocaml program is structured as a collection of source files.

Module implementations A source file with a `.ml` extension defines a structure. Its name is the same as the file, with the first character uppercased.
e.g. the file `binarytree.ml` defines a module called `Binarytree`.

Module interfaces A source file with a `.mli` extension defines a signature. Its name is the same as the file, with the first character uppercased.
e.g. the file `set.mli` defines a signature called `Set`.

Building code

Building Ocaml programs is a two-step process, similar to how C code is built.

Compilation
: Each source file is a separate *compilation unit*, compiled individually using the `-c` switch to `ocamlc`,

```
ocamlc -c binarytree.ml
ocamlc -c set.mli
```

This produces two *object files* named `binarytree.cmo` and `set.cmi`.

Linking
: The resulting object files are then linked together using `ocamlc` again.

```
ocamlc -o example binarytree.cmo set.cmi
```

This entire process can be automated using the `ocamlbuild` tool.

Program execution

- At runtime definitions (types, functions etc.) in the entire program are evaluated.
- There is no unique `main` function. Any function application (call) at the top level is evaluated.
- To call a function producing a *side-effect* like printing we precede the call with `let () =`. This pattern matches the function result against `unit` type.

```
let () = Printf.printf "hello world\n" ;;
```