
CMSC 330: Organization of Programming Languages

Basic OCaml Modules

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, e.g., **List**, **Str**, etc.

Creating A Module In OCaml

```
module ISet =
  struct
    type set = EMPTY | INS of int * set
    let empty = EMPTY
    let isEmpty(s) = s = EMPTY
    let insert(s, i) = INS(i, s)
    let rec contains(s, i) =
      match s with
      | EMPTY -> false
      | INS(j, r) -> if i = j then true
                      else contains(r, i)
  end;;
```

Creating A Module In OCaml (cont.)

```
module ISet =
  struct
    type set = ...
    let empty = ...
    let isEmpty = ...
    let insert = ...
    let contains = ...
  end;;

# empty;;
ERROR: unbound value empty
# ISet.empty;;
- : ISet.set = ISet.EMPTY
# ISet.contains (Set.empty, 1);;
- bool = false
# open ISet;; (* add ISet names to curr scope *)
# empty;;
- : ISet.EMPTY (* now defined *)
```

Module Signatures

Entry in signature

Supply function types

```
module type FOO =
  sig
    val add : int -> int -> int
  end;;
module Foo : FOO =
  struct
    let add x y = x + y
    let mult x y = x * y
  end;;
Foo.add 3 4;;      (* OK *)
Foo.mult 3 4;;    (* not accessible *)
```

Give type to module

Module Signatures (cont.)

- Convention: **Signature names in all-caps**
 - 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**
 - This is what OCaml gives you if you just type in a module with no signature at the top-level

Abstraction = Hiding

- Signatures **hide** module implementation details
 - Why do that? Doesn't that reduce flexibility?
- This is good software engineering practice
 - Ensures data structure **invariants maintained**
 - clients can't construct arbitrary data structures, only ones our module's functions create
 - Facilitates **code collaboration**
 - Write code to the interface as implementation worked out
 - Clients do not rely **details that may change**
 - Changing set representation later won't affect clients

Abstract Data Types

Idea: Hide data value's internal representation from its clients

Invented by **Barbara Liskov** in the **CLU** programming language

- Professor at MIT since 1971

Won **Turing Award** for ADTs and other contributions in 2008

http://amturing.acm.org/award_winners/liskov_1108679.cfm



Abstract Data Types In OCaml Sigs

```
module type ISET =
  sig
    type set
    val empty : set
    val isEmpty : set -> bool
    val insert : set * int -> set
    val contains : set * int -> bool
  end;;

module ISet : ISET =
  struct
    type set = EMPTY | INS of int * set
    ...
    let insert (s,i) = INS(i,s)
  end
```

- The definition of `set` is hidden from ISet clients

Quiz 1: Evaluation on ADTs

```
# ISet.empty
- : ISet.set = <abstr> (* OCaml won't show impl *)
# ISet.EMPTY
Unbound Constructor ISet.EMPTY
# ISet.isEmpty (ISet.insert(ISet.empty, 0))
- : bool = false
# open ISet;;
(* doesn't make anything abstract accessible *)
```

```
# ISet.insert (ISet.empty, 0)
```

A. - : ISet.set = <abstr>

B. Type Error

C. - : ISet.INS (0, ISet.EMPTY)

Quiz 1: Evaluation on ADTs

```
# ISet.empty
- : ISet.set = <abstr> (* OCaml won't show impl *)
# ISet.EMPTY
Unbound Constructor ISet.EMPTY
# ISet.isEmpty (ISet.insert(ISet.empty, 0))
- : bool = false
# open ISet;;
(* doesn't make anything abstract accessible *)
```

```
# ISet.insert (ISet.empty, 0)
```

A. - : ISet.set = <abstr>

B. Type Error

C. - : ISet.INS (0, ISet.EMPTY)

Multiple representations

```
module ISetBST : ISET =
  struct
    type set = TIP | BIN of int * set * set
    ...
    let rec insert (s,i) =
      match s with
      | TIP -> BIN(i,TIP, TIP)
      | BIN(j,l,r) ->
        if i = j then s
        else if i < j then BIN(j,insert(l,i),r)
        else BIN(j,l,insert(r,i))
    end
```

- Now **set** is a binary search tree (why?)

Quiz 2: Mixing ADTs?

```
# ISetBST.empty
- : ISetBST.set = <abstr> (* OCaml won't show impl *)
# ISetBST.insert (ISetBST.empty, 0)
- : ISetBST.set = <abstr>
# ISetBST.contains (ISetBST.empty, 0)
- : bool = false
```

```
# ISet.insert (ISetBST.empty, 0)
```

A. - : ISet.set = <abstr>

B. - : ISetBST.set = <abstr>

C. Type Error

D. - : ISetBST.INS (0, ISet.EMPTY)

Quiz 2: Mixing ADTs?

```
# ISetBST.empty
- : ISetBST.set = <abstr> (* OCaml won't show impl *)
# ISetBST.insert (ISetBST.empty, 0)
- : ISetBST.set = <abstr>
# ISetBST.contains (ISetBST.empty, 0)
- : bool = false
```

```
# ISet.insert (ISetBST.empty, 0)
```

A. - : ISet.set = <abstr>

B. - : ISetBST.set = <abstr>

C. Type Error

D. - : ISetBST.INS (0, ISet.EMPTY)

Different ADTs are ... different

- The `set` type of `ISet` and `ISetBST` look the same, but are not
 - Both modules are an instance of the `ISET` signature
 - But because the type's definition is hidden, it is not safe to mix them
- This distinction is enforced by the type system
 - set `type` is an `abstract type`
 - the instances of `modules` having the `ISET` signature (which has an abstract type) are called `abstract data types` (ADTs)

Other Module Systems

- How OCaml's approach compare to modularity in...
 - Java?
 - C?
 - Ruby?

Modules In Java

- Java **classes** are like modules
 - Provide implementations for a group of functions
 - But classes can also
 - Instantiate objects
 - Inherit attributes from other classes
- Java **interfaces** are like module signatures
 - Defines a group of functions that may be used
 - Implementation is hidden
 - But: **Objects and modules/ADT not the same**
 - Next lecture topic!

Modules In C

- **.c** files are like modules
 - Provides implementations for a group of functions
- **.h** files are like module signatures
 - Defines a group of functions that may be used
 - Implementation is hidden
- Usage is not enforced by C language
 - Can put C code in .h file



Modules In Ruby

- Ruby explicitly supports modules
 - Modules defined by `module ... end`
 - Modules cannot
 - Instantiate objects
 - Derive subclasses

```
puts Math.sqrt(4)      # 2
puts Math::PI          # 3.1416

include Math           # open Math
puts sqrt(4)           # 2
puts PI                # 3.1416
```