CS 11 Ocaml track: lecture 4

- Today:
  - modules

The idea of modules

- What's the idea behind a "module"?
- Package up code so other people/programs can use it
- Control what parts of code are
  - part of the interface
  - part of the implementation only
- and *hide* the implementation part

Implementation hiding

- Why hide the implementation?
- Might want to change later
  - change fundamental data structures
  - different efficiency tradeoffs
- Don't want all code that uses module to require major changes
  - or, ideally, *any* changes

Interface vs. implementation

- In Ocaml:
  - interface goes into .mli files
  - implementation goes into .ml files
- Simple example: lists
  - We'll create our own list module
  - Call it Newlist

Interface of Newlist module

- In newlist.mli
- Must specify:
  - publicly visible types
  - publicly visible functions
  - publicly visible exceptions
- Don't have to *define* any of these!
- Except: need to define types if want users to pattern-match on them

newlist.mli (types)

```
type 'a newlist =
   | Nil
   | Cons of 'a * 'a newlist
```

- If didn't need ability to pattern-match, could do just

```
type 'a newlist
```

- This would be an *abstract* type

newlist.mli (exceptions)

```
exception List_error of string
```

newlist.mli (functions)

```
val hd : 'a newlist -> 'a
val tl : 'a newlist -> 'a newlist
val append : 'a newlist -> 'a newlist -> 'a newlist
val ( @@ ) : 'a newlist -> 'a newlist -> 'a newlist
val length : 'a newlist -> int
```

- Just the signature of the functions
- Any functions not mentioned here are hidden
- Note: operator @@ is exported

Implementation of Newlist

- In newlist.ml
- Must define:
  - all types
  - all functions
  - all exceptions
    - whether exported by module or not
- Copying of code from .mli file sometimes unavoidable

newlist.ml (types)

```
type 'a newlist =
   | Nil
   | Cons of 'a * 'a newlist
```

- Here, had to copy code in newlist.mli
- No way to avoid this!

newlist.ml (exceptions)

```
exception List_error of string
```

- Again, had to copy code in newlist.mli

newlist.ml (functions) (1)

```
let hd nl =
  match nl with
    | Nil -> raise (List_error "head of empty list")
    | Cons (h, _) -> h
let tl nl =
  match nl with
    | Nil -> raise (List_error "tail of empty list")
    | Cons (_, t) -> t
```

newlist.ml (functions) (2)

```
let rec append nl1 nl2 =
  match nl1, nl2 with
    | Nil, _ -> nl2
    | Cons (h, t), _ -> Cons (h, append t nl2)

let ( @@ ) = append
```

newlist.ml (functions) (3)

```
let rec length nl =
  match nl with
    | Nil -> 0
    | Cons (_, t) -> 1 + length t
```

Compiling Newlist files (1)

- To compile the .mli file, just do:

```
ocamlc -c newlist.mli
```

- This will give a compiled interface file (.cmi file)
- The .cmi file is required to compile any file that uses the Newlist module
- The same .cmi file can be used for both bytecode and native-code compilation

Compiling Newlist files (2)

- To compile the .ml file, just do:

ocamlc -c newlist.ml

- (for bytecode compilation), or:

ocamlopt.opt -c newlist.ml

- (for native-code compilation)
- We'll mostly use the bytecode compiler

Compiling Newlist files (3)

- To compile an .ml file that uses the Newlist module, just do:

```
ocamlc -c foobar.ml    (* bytecode compilation *)
```

- Note that don't have to put .cmi file in command line
  - compiler searches for it automatically

Using the Newlist module

- In a file named *e.g.* testlist.ml:

```
open Newlist
let test1 = Cons (1,
   Cons (2, Cons (3, Cons (4, Nil))))
let test2 = Cons (11,
   Cons (12, Cons (13, Cons (14, Nil))))
```

Using the Newlist module

- Without the open declaration:

```
let test1 = Newlist.Cons (1,
  Newlist.Cons (2,
  Newlist.Cons (3,
  Newlist.Cons (4,
  Newlist.Nil))))
(* etc. *)
```

Making lists abstract

- Define a new module called Newlist2
  - files: newlist2.ml, newlist2.mli
- We make one change: want the type to be completely *abstract*
- Downside: can't pattern-match on values of new list type
- Upside: can change implementation without affecting code that uses it
  - BIG win!

newlist2.mli

```
exception List_error of string
type 'a t  (* abstract type *)
val empty : 'a t  (* the empty list *)
val cons : 'a -> 'a t -> 'a t
val hd : 'a t -> 'a
val tl : 'a t -> 'a t
val append : 'a t -> 'a t -> 'a t
val ( @@ ) : 'a t -> 'a t -> 'a t
val length : 'a t -> int
```

type 'a t ????

- Abstract types often given names like "t" (for "type")
- Makes sense when using fully-qualified type name: Newlist2.t
- Means "the type t defined in the Newlist2 module"
- More concise than *e.g.* Newlist2.newlist

newlist2.mli (new interface)

```
exception List_error of string
type 'a t  (* abstract type *)
val empty : 'a t  (* the empty list *)
val cons : 'a -> 'a t -> 'a t
val hd : 'a t -> 'a
val tl : 'a t -> 'a t
val append : 'a t -> 'a t -> 'a t
val ( @@ ) : 'a t -> 'a t -> 'a t
val length : 'a t -> int
```

newlist2.mli (new interface)

```
val empty : 'a t  (* the empty list *)
val cons : 'a -> 'a t -> 'a t
val hd : 'a t -> 'a
val tl : 'a t -> 'a t
```

- These values/functions used instead of pattern matching and type constructors
- Can create and pick apart Newlist2.t values
- Much like lists in Scheme
- All vals are functions except for empty value

More modules

- What we've seen is the most common way to use modules
- Module is *implicitly* defined by .ml and .mli files
- It's also possible to *explicitly* define module types (interfaces) and module implementations inside a body of ocaml code
- That's what we'll look at next
  - Will lead us to functors (next week)

A simple Set module

- What are the characteristics of a set?
  - collection of elements
    - no duplicates
  - there is an empty set value
  - can add elements to set
  - can test whether elements are in set (set membership)
  - other operations (union, intersection etc.)
    - we'll ignore these

A simple Set "signature"

- Types:
  - type of elements of the set
  - type of the set itself
- Values:
  - empty: an empty set value
- Functions:
  - add: adds item to the set
  - member: test for membership

A simple Set "signature" in Ocaml

- Use a module type form:

```
module type Set =
  sig
    type elem
    type t
    val empty : t
    val add : elem -> t -> t
    val member : elem -> t -> bool
  end
```

A simple Set "signature" in ocaml

- This defines a Set "type" as a module type
- sig means "signature"
- Can contain
  - type definitions (abstract or not)
  - exceptions
  - val declarations (value or function signatures)
- No actual definitions
  - of values
  - or of functions

A simple Set "signature" in ocaml

- In our case, mostly obvious except for:

```
type t
type elem
```

- These are *abstract* type defintions
- type t is?
    - type of the set itself
- type elem is?
    - type of the elements of the set

Set implementation (1)

- To use Set, must provide an implementation
- To do this, use a module form:

```
module IntListSet =
  struct
    (* implementation goes here *)
  end
```

- Note: this is more specific than Set
- A set of integers implemented using lists

Set implementation (2)

```
module IntListSet =
  struct
    (* Specify the types. *)
    type elem = int
    type t    = int list

    (* continued on next slide *)
```

Set implementation (3)

```
(* continued from previous slide *)
let empty = []            (* empty set *)
let add el s = el :: s  (* allows duplicates *)
let rec member el s =
    match s with
      | [] -> false
      | x :: xs when x = el -> true
      | _ :: xs -> member el xs
end
```

Using the Set implementation

- (We'll assume we're still in the same file)

```
(* Create a set. *)
let set = IntListSet.empty

(* Add an element.  This generates a new set. *)
let set2 = IntListSet.add 1 set1;;

(* Test for membership. *)
IntListSet.member 1 set2;;
- : bool = true
```

Note

- You can use IntListSet without having defined Set
- In that case, *all* types/exceptions/functions/values inside IntListSet exported
- Not necessarily what you want!

Problem

- As written, the IntListSet module exports everything inside it
  - and exposes its internal implementation
  - not what we usually want
- Example (in interactive interpreter):

```
# IntListSet.empty;;
- : 'a list = []
```

- Know that empty is an empty list
  - breaks abstraction boundary

Solution

- Note also that the IntListSet module conforms to the Set module type
- We can use these two facts to *restrict* the visible part of the IntListSet to the entities specified in the Set module type

First try (broken)

- You'd think it would be as easy as writing

```
module IntListSet : Set =
  struct
    (* same as before *)
  end
```

- But *noooooooo* !
- This will compile, but will be unusable
- Anybody guess what the problem might be?
  - hint: types inside Set

Problem with first try (1)

- Two types defined in Set module type:

```
type t         (* type of Set as a whole *)
type elem  (* type of Set elements *)
```

- type t is OK -- anyone know why?
- … because always use type t as an abstract type
  - cannot use a raw list as a Set
  - can only use t values returned from functions in Set (or the empty value to start with)

Problem with first try (2)

- Two types defined in Set module type:

```
type t       (* type of Set as a whole *)
type elem  (* type of Set elements *)
```

- type elem is *not* OK -- anyone know why?
- Need to pass in int values as arguments to Set functions *e.g.* member
- member expects arguments of type elem, not type int

Problem with first try (3)

- Might expect that compiler could figure out that type elem = type int in IntListSet module from module definition
- Unfortunately, this isn't the case
- Have to tell the compiler explicitly
  - lame
- Which leads us to…

Second try (working)

```
module IntListSet : Set with type elem = int =
  struct
    (* same as before *)
  end
```

- Problems with modules (and functors!) are very often some variation of this
- Can be a real pain in the ass

Using IntListSet

- Same as before, except…
- Cannot access anything inside IntListSet except those things defined inside Set module type
- Example (in interactive interpreter):

```
# IntListSet.empty;;
- : IntListSet.t = <abstr>
```

- Internal implementation is hidden
  - type t implementation no longer visible
- Abstraction boundary is maintained

Modules with polymorphic types

- Many modules use polymorphic types
- Example: Newlist
- Set could also use polymorphic type for elem
- Will have to change our set type and implementation to make this work
- Call the new set PolySet

Interface

```
(* Polymorphic set type. *)
module type PolySet =
  sig
    type 'a t   (* NOTE: no elem type *)
    val empty : 'a t
    val add : 'a -> 'a t -> 'a t
    val member : 'a -> 'a t -> bool
  end
```

Implementation

```
module PolyListSet : PolySet =
  struct
    type 'a t = 'a list
    let empty = []
    let add el s = el :: s  (* allows duplicates *)
    let rec member el s =
      match s with
       | [] -> false
       | x :: xs when x = el -> true
       | _ :: xs -> member el xs
  end
```

Notice anything odd?

- Didn't have to use funky "with type elem = " syntax
- Why not?
- Polyset will work with *any* element type
- Ironically, this makes module syntax simpler
  - Yay polymorphism!

However…

- Polymorphism is not the cure for all your module problems
- Sometimes want a data structure (like a set) which can take a wide variety of data types as elements, but not *any* data type
- Example: some set implementations require elements to be *orderable* (*i.e.* notion of "less than", "equal", "greater than" is meaningful)
- Why would you want this?

Sets with orderable elements

- Can get better efficiency with set implementation if know that elements are orderable
- Example: set implementation: ordered binary tree
  - left subtree: all elems < node elem
  - right subtree: all elems >= node elem
- member function now O(log n) instead of O(n)
  - HUGE win!

(Preview)

- How do we express the notion of "set whose elements have to be orderable"?
- In ocaml, use *functors*
- Basic idea:
  - you give me an orderable type
  - I compute a module that uses that orderable type in a set implementation
- Like a "function on modules"
- Subject of next week's lecture

Next time

- On to functors!