Modular Programming

CS3100 Fall 2019

Review

- Previously
  - How to build small programs
- This lecture
  - How to build at scale: Structures, Signatures, Functors.

Scale

- **Assignment 1 & 2** : ~100 lines of code
- **OCaml** : 375000 lines of code
- **Hubble space telescope** : 2 million lines of code
- **Facebook** : 60 million lines of code (estimated)
- **Google (all services)** : 2 billion lines of code (estimated)

https://informationisbeautiful.net/visualizations/million-lines-of-code/ (https://informationisbeautiful.net/visualizations/million-lines-of-code/)

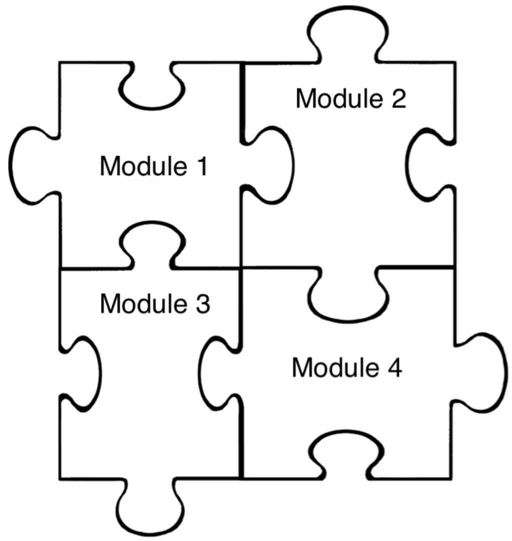
...can't be done by one person

...no individual programmer can understand all the details

...too complex to build with OCaml we've seen so far

Modularity

- Code comprises of individual **Modules**
- Developed separately
  - Reason locally, not globally.
- Clearly specified **Interfaces** for using the modules



Features of modularity

- **Namespacing**
  - Provide way to name a collection of related features (List, Set)
  - Avoid name clashes with similarly named features from different collections (insert in List and Set)
- **Abstraction**
  - Hide the details of implementation
  - Avoid the user breaking invariants implicit in the code.
    - Using a list as a stack (FIFO): User sorts the list which has no meaning in stacks.
  - Transparently change the implementation without breaking the code of the client (i.e. user) of the module.
- **Code reuse**
  - Avoid reimplementing features that are already present.
  - Using a list as a stack (FIFO): Reuse functions such as `is_empty` and `length`.

OCaml Features for Modularity

- **Namespacing:** Structure
- **Abstraction:** Signature
- **Code Reuse:** Functors, includes, sharing constraints.

Structure

- A collection of **definitions**.

- Evaluated in order
- Structure value can be bound to a name

The syntax is

```
module Module_name = struct
  (* collection of definitions *)
end
```

Structure

- Let us implement a purely functional stack using OCaml's built-in list data structure.
- In a purely functional data structure, "update" operations return a new version of the data structure.
  - In a stack, push and pop return a new version of the stack.

In [37]:

```
module Stack = struct
  let empty = []
  let push v s = v::s
  let pop s = match s with
  | [] -> None
  | x::xs -> Some (x, xs)
  let depth s = List.length s
end
```

Out[37]:

```
module Stack :
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
  end
```

Structures provide namespacing

In [38]:

```
Stack.empty
```

Out[38]:

```
- : 'a list = []
```

In [39]:

```
empty
```

Out[39]:

```
- : 'a list = []
```

Signature

- A collection of **declarations**.
- No evaluation, only used for type-checking.
- Signature type can also be bound to a name.

```
module type Module_type_name = sig
  (* collection of declaration *)
end
```

Stack Signature

In [40]:

```
module type StackType = sig
  val empty : 'a list
  val push : 'a -> 'a list -> 'a list
  val pop : 'a list -> ('a * 'a list) option
  val depth : 'a list -> int
end
```

Out[40]:

```
module type StackType =
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
  end
```

Handy Jupyter Functions

Jupyter (really the OCaml top-level power Jupyter) has the following handy functions to display the modules and module signatures.

In [41]:

```
#show_module Stack
```

In [42]:

```
#show_module_type StackType
```

```
module Stack :
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
  end
module type StackType =
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
  end
```

Explicit Signatures

In [43]:

```
module M : StackType = Stack
```

Out[43]:

```
module M : StackType
```

Hiding functionality with explicit signatures

In [44]:

```
module M : sig
  val empty : 'a list
end = Stack
```

Out[44]:

```
module M : sig val empty : 'a list end
```

Opening modules

- Use `open` to make available all the definitions from the module

In [45]:

```
open Stack
```

In [46]:

```
empty
```

Out[46]:

```
- : 'a list = []
```

Opening shadows earlier defintions

In [47]:

```
module M = struct let x = 10 end
module N = struct let x = 20 end
open M
open N
```

Out[47]:

```
module M : sig val x : int end
```

Out[47]:

```
module N : sig val x : int end
```

In [48]:

```
x
```

Out[48]:

```
- : int = 20
```

Including module functionality

- `include` allows new modules to be constructed by extending earlier modules (Code reuse).

In [49]:

```
module Stack = struct
  include Stack
  let is_empty s = match s with
  | [] -> true
  | _ -> false
end
```

Out[49]:

```
module Stack :
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
    val is_empty : 'a list -> bool
  end
```

Include on signatures

`include` also works on signatures to define other signatures.

In [50]:

```
module type MT = sig
  include StackType
  val is_empty : 'a list -> bool
end
```

Out[50]:

```
module type MT =
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
    val is_empty : 'a list -> bool
  end
```

Difference between Open and Include

In [51]:

```
module M1 = struct module Stack = Stack end;;
M1.Stack.empty;;
module M2 = struct include Stack end
```

Out[51]:

```
module M1 : sig module Stack = Stack end
```

Out[51]:

```
- : 'a list = []
```

Out[51]:

```
module M2 :
  sig
    val empty : 'a list
    val push : 'a -> 'a list -> 'a list
    val pop : 'a list -> ('a * 'a list) option
    val depth : 'a list -> int
    val is_empty : 'a list -> bool
  end
```

Abstract types

- So far we have only seen how to collect definitions under a common name (Namespacing)
  - The implementation details are still visible.

In [52]:

```
let s = Stack.empty |> Stack.push 1 |> Stack.push 2 |> Stack.push 3
```

Out[52]:

```
val s : int list = [3; 2; 1]
```

- `s` is of type `int list`
  - Can do non-sensical operations for a stack such as `List.sort`.
- **Abstract** the type of stack such that only those operations allowed by the interface are applicable.

Abstract stack

Define the stack signature with abstract stack type.

In [53]:

```
module type AbsStackType = sig
  type 'a t
  val empty    : 'a t
  val push     : 'a -> 'a t -> 'a t
  val pop      : 'a t -> ('a * 'a t) option
  val depth    : 'a t -> int
  val is_empty : 'a t -> bool
end
```

Out[53]:

```
module type AbsStackType =
  sig
    type 'a t
    val empty : 'a t
    val push : 'a -> 'a t -> 'a t
    val pop : 'a t -> ('a * 'a t) option
    val depth : 'a t -> int
    val is_empty : 'a t -> bool
  end
```

Abstract stack

In [54]:

```
module AbsStack : AbsStackType = struct
  type 'a t = 'a list
  include Stack
end
```

Out[54]:

```
module AbsStack : AbsStackType
```

In [55]:

```
let s = AbsStack.empty |> AbsStack.push 1 |> AbsStack.push 2 |> AbsStack
```

Out[55]:

```
val s : int AbsStack.t = <abstr>
```

Abstraction

- Interfaces with abstract types are **contracts** between the *implementer* and the *user*.
- User cannot violate the API
  - No way to sort the internal list anymore.
- Implementer can transparently change the internals of the implementation
  - As long as the same API is preserved, the user code does not break.

Stack using variants

In [56]:

```
module VariantStack = struct
  type 'a t = Nil | Cons of 'a * 'a t
  let empty = Nil
  let rec depth_aux acc l = match l with
  | Nil -> acc
  | Cons (x, xs) -> depth_aux (1+acc) xs
  let depth l = depth_aux 0 l
  let push v s = Cons (v,s)
  let pop s = match s with Nil -> None | Cons (x,xs) -> Some (x,xs)
  let is_empty s = match s with Nil -> true | _ -> false
end
```

Out[56]:

```
module VariantStack :
  sig
    type 'a t = Nil | Cons of 'a * 'a t
    val empty : 'a t
    val depth_aux : int -> 'a t -> int
    val depth : 'a t -> int
    val push : 'a -> 'a t -> 'a t
    val pop : 'a t -> ('a * 'a t) option
    val is_empty : 'a t -> bool
  end
```

Abstracting the variant stack

In [57]:

```
module AbsVariantStack : AbsStackType = VariantStack
```

Out[57]:

```
module AbsVariantStack : AbsStackType
```

Whereever the user uses the `AbsStack` , we can replace that with `AbsVariantStack` and the user code wouldn't be able to tell the difference.

Two languages

- OCaml is a stratified language.
  - Values + Expressions and Types at term level.
  - Structures and Signatures at module level.
- Structures are (generally) not first-class in OCaml.

    - OCaml has **first-class modules**, which we will not cover in this class.
- What is the equivalent of functions at the module level?
    - And why would we need it?
- **Functors**
    - Functions that take structures and return other structures.

Identity functor

The simplest function is the identity function

In [58]:

```
let id x = x
```

Out[58]:

```
val id : 'a -> 'a = <fun>
```

At module level, we can correspondingly define

In [59]:

```
module type T = sig
  type t
  val v : t
end
module Id (X : T) : T = X
```

Out[59]:

```
module type T = sig type t val v : t end
```

Out[59]:

```
module Id : functor (X : T) -> T
```

Applying Identity functor

In [60]:

```
id 5
```

Out[60]:

```
- : int = 5
```

Similarly:

In [61]:

```
module M = struct
  type t = int
  let v = 10
end
```

Out[61]:

```
module M : sig type t = int val v : int end
```

In [62]:

```
module M' = Id(M)
```

Out[62]:

```
module M' : sig type t = Id(M).t val v : t end
```

Type equality under abstraction

- We know `M` and `M'` are the same modules.
  - Let's ask whether `M.v = M'.v`.

In [63]:

```
M.v = M'.v
```

```
File "[63]", line 1, characters 6-10:
Error: This expression has type M'.t = Id(M).t
       but an expression was expected of type int
   1: M.v = M'.v
```

- While the compiler knows that `M.v` has type `int`, the type `M'.t` (return type of functor) is **abstract**
  - Compiler does not know that `M.t` is the same type as `M'.t`.

Sharing constraints

- Use **sharing constraints** to let the compiler know about type equalities.
- Sharing constraints make the types less abstract / more concrete.

In [64]:

```
module Id2 (X : T) : T with type t = X.t = X
```

Out[64]:

```
module Id2 : functor (X : T) -> sig type t = X.t val v : t
end
```

In [65]:

```
module M2 = Id2(M)
```

Out[65]:

```
module M2 : sig type t = M.t val v : t end
```

In [66]:

```
M2.v = M.v
```

Out[66]:

```
- : bool = true
```

Multiple sharing constraints

In [67]:

```
module type T2 = sig
  type t
  type u
end

module Id2 (X2 : T2) : T2 with type t = X2.t
                             and type u = X2.u = X2
```

Out[67]:

```
module type T2 = sig type t type u end
```

Out[67]:

```
module Id2 : functor (X2 : T2) -> sig type t = X2.t type u
= X2.u end
```

Functor Example: Serializable List

- Write a `string_of_list` and `list_of_string` functions for a list.
- For an `int list`, we can write:

In [68]:

```
let string_of_list l =
  let rec loop acc l = match l with
    | [] -> acc
    | [x] -> acc ^ (string_of_int x)
    | x::xs -> loop (acc ^ (string_of_int x) ^ ";") xs
  in
  "[" ^ (loop "" l) ^ "]"
```

Out[68]:

```
val string_of_list : int list -> string = <fun>
```

In [69]:

```
string_of_list [1;2;3]
```

Out[69]:

```
- : string = "[1;2;3]"
```

Functor Example: Serializable List

In [70]:

```
let list_of_string s =
  let s = String.sub s 1 (String.length s - 2) in
  List.map int_of_string (String.split_on_char ';' s)
```

Out[70]:

```
val list_of_string : string -> int list = <fun>
```

In [71]:

```
list_of_string "[1;2;3]"
```

Out[71]:

```
- : int list = [1; 2; 3]
```

Generalise

What about float, tuples, records, etc? Use **Functors**.

In [72]:

```
module type Serializable = sig
  type t
  val t_of_string : string -> t
  val string_of_t : t -> string
end

module SerializableList (C : Serializable) = struct
  type t = C.t list
  let string_of_t l =
    let rec loop acc l = match l with
      | [] -> acc
      | [x] -> acc ^ (C.string_of_t x)
      | x::xs -> loop (acc ^ (C.string_of_t x) ^ ";") xs
    in
    "[" ^ (loop "" l) ^ "]"

  let t_of_string s =
    let s = String.sub s 1 (String.length s - 2) in
    List.map C.t_of_string (String.split_on_char ';' s)
end
```

Out[72]:

```
module type Serializable =
  sig type t val t_of_string : string -> t val string_of_t
: t -> string end
```

Out[72]:

```
module SerializableList :
  functor (C : Serializable) ->
    sig
      type t = C.t list
      val string_of_t : C.t list -> string
      val t_of_string : string -> C.t list
    end
```

SerializableFloatList

In [73]:

```
module SerializableFloatList = SerializableList (struct
  type t = float
  let t_of_string = float_of_string
  let string_of_t = string_of_float
end)
```

Out[73]:

```
module SerializableFloatList :
  sig
    type t = float list
    val string_of_t : float list -> string
    val t_of_string : string -> float list
  end
```

SerializableFloatList

In [74]:

```
SerializableFloatList.string_of_t [1.4;2.3;3.4]
```

Out[74]:

```
- : string = "[1.4;2.3;3.4]"
```

In [75]:

```
SerializableFloatList.t_of_string "[1.4;2.3;3.4]"
```

Out[75]:

```
- : float list = [1.4; 2.3; 3.4]
```

Behold the power of abstraction

Observe that the signature of `SerializableFloatList` is also `Serializable` .

In [76]:

```
module SerializableFloatListList = SerializableList (SerializableFloatLi
```

Out[76]:

```
module SerializableFloatListList :
  sig
    type t = SerializableFloatList.t list
    val string_of_t : SerializableFloatList.t list -> strin
g
    val t_of_string : string -> SerializableFloatList.t lis
t
  end
```

In [77]:

```
SerializableFloatListList.string_of_t [[1.1]; [2.1;2.2]; [3.1;3.2;3.3]]
```

Out[77]:

```
- : string = "[[1.1];[2.1;2.2];[3.1;3.2;3.3]]"
```

Fin.