# The OCaml Module System

Dipl.-Inf. Quang M. Tran
tranqm@mcmaster.ca

McMaster University
Department of Computing and Software
Hamilton, ON, Canada

October 26, 2010

# Outline

# Module is an Abstract Concept

- Module is an abstract concept of modular programming.
- Modular programming: decomposition of a program into separate and replaceable modules [1].
- Each module represents a separation of concern, i.e. features or behaviors of a software.

http://www.leistungen.city-map.de/de/18/ausbau_module/

# Abstract View of Modules

- Module interface or specification: declarations of visible elements and promises of dynamic behaviors.
- Module implementation: a concrete implementation for realizing the module interface.
- Interface is publicly visible. Implementation is hidden and thus can evolve. (cf. Information hiding).

# Various Implementations of the Concept Module

- Different paradigms and languages have different approaches to implementing the concept module.

## Example

- OO languages such as Java and C#: separation of concerns into packages, objects etc.
- Model-View-Controller (MVC) design pattern: separation of content from presentation into layers.
- Service-oriented design: separation of concerns into services.
- ML-family languages such as Haskell and OCaml: **separation of concerns into modules.**

# OCaml's Module

- OCaml support the module concept with the construct module: sig ≅ interface, struct ≅ implementation.
- A module groups types, functions and exceptions etc.

## Syntax for Signature and Structure

```
module type NAME =
sig
interface declarations: types,functions etc.
end
module Name  =
struct
implementation definitions:types, functions etc.
end
```

# Example: Stack Signature [3]

## Example

```
module type STACK =
sig
  type 'a t
  exception Empty
  val create : unit -> 'a t
  val push : 'a -> 'a t -> unit
  val pop  : 'a t -> 'a
end
```

- The data structure for storing elements of the stack is not specified (type abstraction).

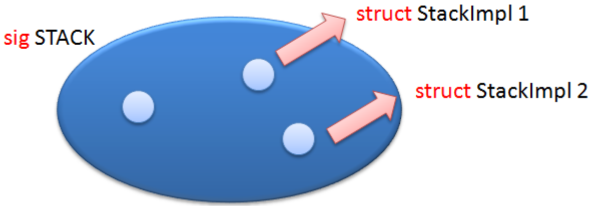
# Example: A Stack Structure/Module [3]

## Example

```
module StandardStack =
struct
type 'a t={mutable sp:int; mutable c:'a array}
exception Empty
let create() =  sp = 0; c = [||]
let push x s =
  if s.sp >= Array.length s.c then increase s 0;
  s.c.(s.sp) <- x;
  s.sp <- succ s.sp
let increase s x = ...
let pop s = ...
end
```

# Perspective 1: A Signature is a Type Specification

- A sig is a type specification.
- A realizing struct is an element of that type.



# Perspective 2: A Signature is a View on a Structure

## Example

```
module type PUSHONLYSTACK =
sig
  type 'a t
  exception Empty
  val create : unit -> 'a t
  val push : 'a -> 'a t -> unit
  (* NO pop *)
end
```

- PUSHONLYSTACK is a partial view on StandardStack.
- module PushOnlyStack = (StandardStack : PUSHONLYSTACK) exposes push but hides pop.

# Type Sharing Between Modules [3]

- Abstracted types in different modules are distinct.
- StandardStack.t and PushOnlyStack.t are incompatible.
- Use type equality constraints to force type equality.

**Example**

```
module PushOnlyStack =
(StandardStack : PUSH_ONLY_STACK
with type 'a t = 'a StandardStack.t )
```

# Parameterized Modules

- A parameterized module or functor builds a new module from input modules.
- Parameterized modules allow generic programming [2].

## Syntax for Parameterized Module

```
module Name = functor (M1:sig1) ->...->
              functor (Mn:sign) ->
struct
end
(* Or syntactic sugar: *)
module Name (M1:sig1)...(Mn:sign) =
struct
end
```

# Example: Find Least Element Functor

- Input: any module implementing a totally ordered data type (*"Tell me how to compare two elements"*).
- Output: module implementing a function that finds the least element in a list of elements of that type.

**Example**

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

# Example: Find Least Element Functor

## Example

```
module FindLeastElem (Ord : ORDERED_TYPE) =
 struct
   type elt = Ord.t
   let rec leastElemRec l le =
     match l with
     | (x :: xs) -> if Ord.compare x le = -1
                    then leastElemRec xs x
                    else leastElemRec xs le
     | [] -> le
     ...
 end
```

# OCaml Module Language is Simply Typed $\lambda$-calculus

- Signatures are types.
- Non-functor structures are constants.
- Functors are function abstractions with bound variables the parameterized arguments.
- Module instantiations are function applications with substitution as their semantics.

**Syntax for Simply Typed $\lambda$-calculus**

(* Types *)

$\tau \; ::= \; T \mid \tau \to \tau$

(* Lambda terms *)

t ::= c | x | $\lambda\ x : \tau.\ t \mid t_1 t_2$

# OCaml Functors Are Applicative

- In SML: functors are generative, i.e each functor application generates distinct abstract types for the same input [5].
- In OCaml: functors are applicative, i.e functor application generates compatible abstract types for the same input.

**Example**

```
module M1 = FindLeastElem(OrderedPairInt)
module M2 = FindLeastElem(OrderedPairInt)
```

- In OCaml: M1.elt and M2.elt are the same type (*applicative*).
- In SML: M1.elt and M2.elt are distinct types (*generative*).

# Conclusion

- Module is an abstract concept from modular programming.
- OCaml supports the module concept with the module construct.
- sig is module interface, struct is module implementation.
- Parameterized modules or functors create modules from other modules and thus allow generic programming.
- Type equality constraints allows type sharing between modules.
- OCaml functors are applicative, i.e. producing the same abstract types for the same input.
- *Dr. Kahl, are OCaml modules first-class?*. My tentative answer: No.

# Acknowledgement


- My grateful thanks go to Gordon, Eden and Pouya (ITB 206) for their invaluable feedback.
- Gordon occationally helped me with clarifying some details.
- Eden and Pouya patiently listened to my dry-run and gave suggestions for improvement.

# References


- [1] Wikipedia entry on modular programming.
- [2] Wikipedia entry on generic programming.
- [3] E. Chailloux, P. Manoury, N. Pagano
  *Developing Apps with OCaml*.
  O'Reilly, 2000.
- [4] OCaml Authors
  *Official OCaml documentation and user's manual*.
- [5] X. Leroy
  *Applicative functors and fully transparent higher-order modules*.