OCaml Modules Signatures Information Hiding Functors Separate Compilation

Funktionale Programmierung (Technische Universität München)

OCaml: Modules, Signatures, Information Hiding, Functors, Separate Compilation

- **Modules:** same concept as in other programming languages (i.e. classes in Python or Java) in order to organize larger software systems in OCaml
    - allow to do several declarations of values and functions (as every value is also a function), as for other programming languages applying modules and corresponding signatures is a better software development approach than simply writing function after function in one file
    - define module structure with keywords `module` and `struct`

```
module Pairs =
  struct
    type 'a pair = 'a * 'a
    let pair (a,b) = (a,b)
    let first (a,b) = a
    let second (a,b) = b
  end
```

- **Signatures:** compiler answers with type of module identified by `signature` keyword
    - define signature of existing module with keywords `module` and `sig`

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end

(* different implementation than defined by module Pairs,  *)
module Pairs2 =
  struct
    type 'a pair = bool -> 'a
    let pair (a,b) = fun x -> if x then a else b
    let first ab = ab true
    let second ab = ab false
  end;;
```

    - note that OCaml doesn't make connection between type pair declaration and val pair, thus its types remain as general as possible using different polymorphic types `'a` and `'b` as OCaml doesn't know a pair consists of the same type
    - accessing declaration of a module using dot-notation

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

    - opening a module allows to access its declarations directly using `open`

```
# open Pairs2;;
# pair;;
- : 'a * 'a -> bool -> 'a = <fun>
# pair (4,3) true;;
- : int = 4
```

    - definitions of one module can be included into another using `include`

```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A (* not added to B, but allows access from within B *)
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
```

```
    include A (* capabilities of A added to C "*)
    include B
  end;;
module C : sig val x : int val y : int end
```

- modules can be nested into another module alongside other declarations (functions, values, and types)

```
module Quads  = struct
      module Pairs = struct
          type 'a pair = 'a * 'a
          let pair (a,b) = (a,b)
          let first (a,_) = a
          let second (_,b) = b
      end
  type 'a quad = 'a Pairs.pair Pairs.pair
  let quad (a,b,c,d) = Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
  let first q = Pairs.first (Pairs.first q)
  let second q = Pairs.second (Pairs.first q)
  let third q = Pairs.first (Pairs.second q)
  let fourth q = Pairs.second (Pairs.second q)
end

# Quads.quad (1,2,3,4);;
- : (int * int) * (int * int) = ((1,2),(3,4))
# Quads.Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

- **Module Types or Signatures**
  - signatures can restrict variables and types that shall be allowed to be exported from a module

```
(* sorting algorithm, note there exist one in the OCaml API *)
module Sort = struct
  let single list = map (fun x->[x]) list (* mapping every element of a list into another list, making a singleton list i.e.
  let rec merge l1 l2 = match (l1,l2) (* merge two lists *)
      with ([],_) -> l2
      | (_,[]) -> l1
      | (x::xs,y::ys) -> if x<y then x :: merge xs l2
                                else y :: merge l1 ys
  let rec merge_lists = function (* merging lists of lists together, assumes a list consisting of n other lists *)
          [] -> [] (* empty list of lists *)
          | [l] -> [l] (* list with only one element *)
          | l1::l2::ll -> merge l1 l2 :: merge_lists ll (* otherwise one merges the first two lists, and continues with the r
  let sort list = let list = single list (* get singleton *)
      in let rec doit = function
         [] -> [] (* empty list already sorted *)
         | [l] -> l (* one element list, returning element which then should only be one list *)
         | l -> doit (merge_lists l)
  in doit list
end
```

    - to hide functions single and merge_list one can create its own module type

```
module type Sort = sig
  val merge : 'a list -> 'a list -> 'a list
  val sort : 'a list -> 'a list
end
```

  - types defined in a signature must align with the exported definitions

```
module type A1 = sig
  val f : 'a -> 'b -> 'b
end
module type A2 = sig
  val f : int -> char -> int
end
(* only module type A2 can be used to restrict module A as types align *)
module A = struct
  let f x y = x
end

(* as visible below *)
# match A1 : A1 = A;;
Signature mismatch:
Modules do not match: sig val f : 'a -> 'b -> 'a end
                               is not included in A1
```

```
Values do not match:
  val f : 'a -> 'b -> 'a
is not included in
  val f : 'a -> 'b -> 'b
# module A2 : A2 = A;;
module A2 : A2
# A2.f;;
- : int -> char -> int = <fun>
```

- Note the **difference between modules and types**, custom defined types always need to be mentioned everywhere, they are constructed within `.ml` file or in `module` `struct` and have to be mentioned but not declared in `module sig`
- **Information Hiding:** as already mentioned above, signatures allow to hide implementations

```
module ListQueue = struct
  type 'a queue = 'a list
  let empty_queue () = []
  let is_empty = function
      [] -> true | _ -> false
  let enqueue xs y = xs @ [y]
  let dequeue (x::xs) = (x,xs)
end

module type Queue = sig
  type 'a queue (* in signatures type s are only mentioned but not necessarily defined as in .mli file*)
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a queue -> 'a -> 'a queue
  val dequeue : 'a queue -> 'a * 'a queue
end

# module Queue : Queue = ListQueue;;
module Queue : Queue
# open Queue;;
# is_empty [];;
This expression has type 'a list but is here used with type
  'b queue = 'b Queue.queue
(* as shown above is_empty [] returns an error as the form
of a queue is ophiscated *)

(* to export datatype with all other constructors, defintion must be
repeated in signature *)
module type Queue =
sig
  type 'a queue = Queue of ('a list * 'a list)
  val empty_queue : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val enqueue : 'a -> 'a queue -> 'a queue
  val dequeue : 'a queue -> 'a option * 'a queue
end
```

  - in order to hide information about the implementation of a certain module the module from which it inherits restricts the access to it by not defining corresponding types for that information

```
(* Example *)

module type Fact = sig (* kind of definition for the public how they
shall perceive RecursiveFact *)
  val fact : int -> int
end

module RecursiveFact : Fact = struct (* if we remove declaration to
Fact, everything is accessible  *)
  let rec fact n =
    if n = 0 then 1 else n * fact (n-1)
end

module TailRecursiveFact : Fact = struct
  let rec fact_aux n acc = if n = 0 then acc
    else fact_aux (n-1) (n*acc)

  let fact n = fact_aux n 1
end

let x = TailRecursiveFact.fact 10
(* we can access fact, but not fact_aux as the module type Fact
restricts access to only those functions mentioned within in its sig,
thus let x = TailRecursiveFact.fact_aux 10 would result in an error,
note that functions/values declared in module type Fact must be
provided *)
```

- **Functors:** modules of higher order allow parameterizing modules
  - functor receives sequence of modules as parameters, elements of modules given as parameters can then be used in functor's body (if given as parameter to functor)
  - functor's arguments must be defined beforehand through signatures
  - functors, module level function, take modules as input and produce modules as output; functor is a function mapping values from one module to the other

```
(* defining types *)
module type Decons = sig
  type ’a t
  val decons : ’a t -> (’a * ’a t) option (* must be optional in case empty *)
end
module type GenFold = functor (X:Decons) -> sig
  val fold_left : (’b -> ’a -> ’b) -> ’b -> ’a X.t -> ’b
  val fold_right : (’a -> ’b -> ’b) -> ’a X.t -> ’b -> ’b
  val size : ’a X.t -> int
  val list_of : ’a X.t -> ’a list
  val iter : (’a -> unit) -> ’a X.t -> unit
end

(* defining declarations of functor Fold *)
module Fold : GenFold = functor (X:Decons) -> struct
  let rec fold_left f b t = match X.decons t
      with None -> b
      | Some (x,t) -> fold_left f (f b x) t
  let rec fold_right f t b = match X.decons t
      with None -> b
      | Some (x,t) -> f x (fold_right f t b)
  let size t = fold_left (fun a x -> a+1) 0 t
  let list_of t = fold_right (fun x xs -> x::xs) t []
  let iter f t = fold_left (fun () x -> f x) () t
end;;

module MyQueue = struct open Queue
  type ’a t = ’a queue
  let decons = function
      Queue([],xs) -> (match rev xs
                          with [] -> None
                          | x::xs -> Some (x, Queue(xs,[])))
      | Queue(x::xs,t) -> Some (x, Queue(xs,t))
end
module MyAVL = struct open AVL
  type ’a t = ’a avl
  let decons avl = match extract_min avl
      with (None,avl) -> None
      | Some (a,avl) -> Some (a,avl)
end

(* using created modules and module types to create new ones *)
module FoldAVL = Fold (MyAVL)
(* FoldQueue equals the module Fold using MyQueue as parameter which
provides the definition for t and decons *)
module FoldQueue = Fold (MyQueue)

(* a more convenient way of declaring a functor *)
(* defining original module type to use as parameter *)
# module type A = sig val x: int end;;
module type A = sig val x : int end

(* defining functor using module type A as parameter where result should
satisfy the signature of A *)
# module F (X:A) : A = struct let x = X.x + 1 end;;
module F : functor (X : A) -> A

# module Y = struct let x = 42 end;;
module Y : sig val x : int end

(* possible to apply F to Y as Y has the same signature as module type A *)
# module Y' = F (F (Y));; (* incrementing 42 twice *)
module Y' : sig val x : int end

# Y'.x;;
- : int = 44
```

  - Caveat: module satisfies signature whenever it implements it, but mustn't specifically declare that

```
(* YouTube Tutorial Example *)
module type X = sig
  val x : int
end

module A : X = struct
  let x = 0
end

(* module Incx accesses module type X as functor *)
module Incx = functor (M : X) -> struct
  let x = M.x + 1
end

(* syntactic sugar for expression above *)
module Incx (M : X) = struct
  let x = M.x + 1
end

(* in utop modules can only be applied by binding them to another module,
because it is defined as a function which means it's definitions
take inputs only through the means of another module *)
 # module B = IncX(A);;
module B : sig val x : int end
# B.x;;
- : int = 1
# A.x;;
- : int = 0
# module C = IncX(B);;
module C : sig val x : int end
# C.x;;
- : int = 2
```

```
(* More examples from YouTube Channel *)
module ListStack = struct
  type 'a stack = 'a list
  let empty = []
  let push x s = x::s
  let peek = function
    | [] -> failwith "Empty"
    | x :: _ -> x
  let pop = function
    | [] -> failwith "Empty"
    | _ :: s -> s
end

(* Following Code is the same *)
let x = ListStack.peek (ListStack.push 42 ListStack.empty)
let x = ListStack.(peek (push 42 empty)
let x = ListStack.(empty |> push 42 |> peek)
```

- Creating a module that implements (inherits) the signature of another module type in which the type of a variable is not defined ("polymorphic") requires the module structure to define its type when assigning it to the newly created module, e.g.

```
module IntRing : Ring with type t = int = struct
  type t = int
  t
  let zero = 0
  t
  let one = 1
  'a -> 'a -> t
  let compare = compare
  t -> string
  let to_string = string_of_int
  t -> t -> t
  let add a b = a + b
  t -> t -> t
  let mul a b = a * b
end
sig
  type t = int
  val zero : t
  val one : t
  val compare : t -> t -> int
  val to_string : t -> string
  val add : t -> t -> t
  val mul : t -> t -> t
end
```

- **Separate Compilation** : in order to compile a sequence of definitions of a module the compiler uses the file `.ml` which represents the sequence of definitions of a module
  - the compiler `ocamlc` generates files based on these

`Test.cmo`	bytecode for the module
`Test.cmi`	bytecode for the signature
`a.out`	executable program

  - if file `.mli` exists it is interpreted as signature for module of the same name, then compiler called as, e.g. `ocamlc Test.mli Test.ml` otherwise called `ocamlc Test.ml`
  - BUT ALL THIS IS HANDLED BY DUNE: important is the difference between the files `.ml` and `.mli` where the last defines the signature of the first