

Script generated by TTT

Title: Seidl: Functional Programming and Verification (21.12.2018)

Date: Fri Dec 21 08:29:53 CET 2018

Duration: 89:31 min

Pages: 33

6.2 Module Types or Signatures

Signatures allow to restrict what a module may export.

Explicit indication of the signature allows

- to restrict the set of exported variables;
- to restrict the set of exported types ...

... an Example

293

```
module Sort = struct
  let single list = map (fun x->[x]) list
  let rec merge l1 l2 = match (l1,l2)
    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
    [] -> [] | [l] -> [l]
  | l1::l2::l1 -> merge l1 l2 :: merge_lists l1
  let sort list = let list = single list
    in let rec doit = function
      [] -> [] | [l] -> l
      | l -> doit (merge_lists l)
    in doit list
end
```

294

The implementation allows to access the auxiliary functions `single`, `merge` and `merge_lists` from the outside:

```
# Sort.single [1;2;3];;
- : int list list = [[1]; [2]; [3]]
```

In order to hide the functions `single` and `merge_lists`, we introduce the signature

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

295

```

module Sort = struct
  let single list = map (fun x->[x]) list
  let rec merge l1 l2 = match (l1,l2)
    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
    [] -> [] | [l] -> [l]
  | l1::l2::l1 -> merge l1 l2 :: merge_lists l1
  let sort list = let list = single list
    in let rec doit = function
      [] -> [] | [l] -> l
      | l -> doit (merge_lists l)
    in doit list
end

```

294

The functions `single` and `merge_lists` are no longer exported:

```

# module MySort : Sort = Sort;;
module MySort : Sort
# MySort.single;;
Unbound value MySort.single

```

296

Sort

```

module Sort = struct
  let single list = map (fun x->[x]) list
  let rec merge l1 l2 = match (l1,l2)
    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
    [] -> [] | [l] -> [l]
  | l1::l2::l1 -> merge l1 l2 :: merge_lists l1
  let sort list = let list = single list
    in let rec doit = function
      [] -> [] | [l] -> l
      | l -> doit (merge_lists l)
    in doit list
end

```

294

6.3 Information Hiding

For reasons of modularity, we often would like to prohibit that the structure of exported types of a module are visible from the outside.

Example

```

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

```

299

```
# 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
```



The restriction via signature is sufficient to obfuscate the [true nature](#) of the type queue.

301

6.3 Information Hiding

For reasons of modularity, we often would like to prohibit that the structure of exported types of a module are visible from the outside.

Example

```
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
```

299

If the datatype should be exported together with all constructors, its definition is [repeated](#) in the 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
```

302

6.4 Functors

Since (almost) everything in [Ocaml](#) is higher order, it is no surprise that there are modules of higher order: [Functors](#).

- A functor receives a sequence of modules as parameters.
- The functor's body is a module where the functor's parameters can be used.
- The result is a new module, which is defined relative to the modules passed as parameters.

303

First, we specify the functor's argument and result by means of signatures:

```
module type Decons = sig
  type 'a t
  val decons : 'a t -> ('a * 'a t) option
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
...
```

304

```
...
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;
```

Now, we can [apply](#) the functor to the module to obtain a new module ...

305

```
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
```

306

```
module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)
```

By that, we may define

```
let sort list = FoldAVL.list_of (
  AVL.from_list list)
```

Caveat

A module satisfies a signature whenever it implements it !

It is not required to [explicitly](#) declare that !!

307

```

...
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;;

```

Now, we can [apply](#) the functor to the module to obtain a new module ...

305

```

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

```

306

```

...
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;;

```

Now, we can [apply](#) the functor to the module to obtain a new module ...

305

```

module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)

```

By that, we may define

```

let sort list = FoldAVL.list_of (
AVL.from_list list)

```

Caveat

A module satisfies a signature whenever it implements it !

It is not required to [explicitly](#) declare that !!

307

```
module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)
```

By that, we may define

```
include Queue
include
let sort list = FoldAVL.list_of (
    AVL.from_list list)
```

Caveat

A module satisfies a signature whenever it implements it !

It is not required to **explicitly** declare that !!

307

```
module GQ = struct
include Queue
include
```

```
module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)
```

By that, we may define

```
let sort list = FoldAVL.list_of (
    AVL.from_list list)
```

Caveat

A module satisfies a signature whenever it implements it !

It is not required to **explicitly** declare that !!

307

```
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
```

306

```

module GQ = struct
  include Queue
  include FoldQueue
end

```

```

module FoldAVL = Fold (MyAVL)
module FoldQueue = Fold (MyQueue)

```

module F (X, S) = struct

By that, we may define

```

let sort list = FoldAVL.list_of (
  AVL.from_list list)

```

T

Caveat

end A module satisfies a signature whenever it implements it !
 It is not required to explicitly declare that !!

6.5 Separate Compilation

- In reality, deployed **Ocaml** programs will not run within the interactive shell.
- Instead, there is a compiler `ocamlc ...`

```
> ocamlc Test.ml
```

that interpretes the contents of the file `Test.ml` as a sequence of definitions of a module `Test`.

- As a result, the compiler `ocamlc` generates the files

<code>Test.cmo</code>	bytecode for the module
<code>Test.cmi</code>	bytecode for the signature
<code>a.out</code>	executable program

- If there is already a file `Test.mli` this is interpreted as the signature for `Test`. Then we call


```
> ocamlc Test.mli Test.ml
```
- Given a module `A` and a module `B`, then these should be compiled by


```
> ocamlc B.mli B.ml A.mli A.ml
```
- If a re-compilation of `B` should be omitted, `ocamlc` may receive a pre-compiled file


```
> ocamlc B.cmo A.mli A.ml
```
- For practical management of required re-compilation after modification of files, `Linux` offers the tool `make`. The script of required actions then is stored in a `Makefile`.
- ... alternatively, `dune` can be used.

309

This fragment of `Ocaml` is called `MiniOcaml`.

Expressions in `MiniOcaml` can be described by the grammar

$$E ::= \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \text{fun name} \rightarrow E \mid E E_1$$

$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

312

This fragment of `Ocaml` is called `MiniOcaml`.

Expressions in `MiniOcaml` can be described by the grammar

$$E ::= \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid (E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid \text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid \text{fun name} \rightarrow E \mid E E_1$$

$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

Short-cut

$$\text{fun } x_1 \rightarrow \dots \text{fun } x_k \rightarrow e \equiv \text{fun } x_1 \dots x_k \rightarrow e$$

313

7.2 A Semantics for MiniOcaml

Question

Which `value` is returned for the expression `E` ??

A `value` is an expression that cannot be further evaluated.

The set of all values can also be specified by means of a grammar:

$$V ::= \text{const} \mid \text{fun name}_1 \dots \text{name}_k \rightarrow E \mid (V_1, \dots, V_k) \mid [] \mid V_1 :: V_2$$

316

This fragment of **Ocaml** is called **MiniOcaml**.

Expressions in **MiniOcaml** can be described by the grammar

```

$$E ::= \text{const} \mid \text{name} \mid \text{op}_1 E \mid E_1 \text{op}_2 E_2 \mid$$

$$(E_1, \dots, E_k) \mid \text{let name} = E_1 \text{ in } E_0 \mid$$

$$\text{match } E \text{ with } P_1 \rightarrow E_1 \mid \dots \mid P_k \rightarrow E_k \mid$$

$$\text{fun name} \rightarrow E \mid E E_1$$

```

```

$$P ::= \text{const} \mid \text{name} \mid (P_1, \dots, P_k) \mid P_1 :: P_2$$

```

Short-cut

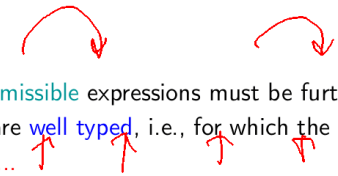
```

$$\text{fun } x_1 \rightarrow \dots \text{fun } x_k \rightarrow e \equiv \text{fun } x_1 \dots x_k \rightarrow e$$

```

313

Caveat

- The set of **admissible** expressions must be further restricted to those which are **well typed**, i.e., for which the **Ocaml** compiler infers a type ...

 - $(1, [\text{true}; \text{false}])$ **well typed**
 - $(1 [\text{true}; \text{false}])$ **not well typed**
 - $([1; \text{true}], \text{false})$ **not well typed**
- We also rule out `if ... then ... else ...`, since it can be simulated by `match ... with true -> ... | false -> ...`
- We could also have omitted `let ... in ...` (why?)

314