ML modules

Daniel Jackson
MIT Lab for Computer Science
6898: Advanced Topics in Software Design
March 20, 2002

topics for today

elements of ML module language

- structs: modules, export types and values
- signatures: types for modules
- functors: functions from modules to modules

signature ascription

- controlling client's view of a module

functorization

- making dependences explicit

set implementation def and use

```
module SetImpl =
   struct
     type 'a t = 'a list
     let empty () = []
     let add s e = e :: s
     let member s e = List.mem e s
   end;;

let s = SetImpl.empty ();;
SetImpl.add s 3;;
```

one possible type for the module

```
module SetImplC: ManifestSet =
  struct
    type 'a t = 'a list
    let empty () = []
    let add s e = e :: s
    let member s e = List.mem e s
  end

module type ManifestSet =
  sig
    type 'a t = 'a list
    val empty: unit -> 'a t
    val add: 'a t -> 'a -> 'a t
    val member: 'a t -> 'a -> bool
  end
```

another type for the same module

```
module SetImplA: OpaqueSet =
  struct
    type 'a t = 'a list
    let empty () = []
    let add s e = e :: s
    let member s e = List.mem e s
  end

module type OpaqueSet =
  sig
    type 'a t
    val empty: unit -> 'a t
    val add: 'a t -> 'a -> 'a t
    val member: 'a t -> 'a -> bool
  end
```

controlling access

```
let s = SetImplC.empty ();;
SetImplC.add s 3;;
4::s;;

let s = SetImplA.empty ();;
SetImplA.add s 3;;
4::s;;   (* type error *)
```

extending a module

```
module SetWithUnion = struct
  include SetImpl
  let union s1 s2 = List.append s1 s2
  end;;
```

substructure

suppose we want a set of strings

› exploiting ordering

```
module OrderedString =
  struct
    type t = string
    let lt a b = a < b
  end;;
```

```
module OrderedStringSet = struct
    module Os = OrderedString
    type t = Os.t
    let empty () = []
    let rec add s e =
        match s with [] -> [e]
        | x :: xs -> if Os.lt e x then e :: s else x :: add xs e
    let rec member s e =
        match s with [] -> false
        | x :: xs -> if Os.lt x e then false else member xs e
    end;;
```

does this satisfy the signature OpaqueSet?

making it generic

```
module type Ordered =
   sig
      type t
      val lt: t -> t -> bool
   end;;
```

a functor

```
module OrderedSetImpl = functor (Elt: Ordered) ->
  struct
    type element = Elt.t
    type set = Elt.t list
    let empty () = []
    let rec add s e =
      match s with [] -> [e]
      | x :: xs -> if Elt.lt e x then e :: s else x :: add xs e
    let rec member s e =
      match s with [] -> false
      | x :: xs -> if Elt.lt x e then false else member xs e
    end;;
```

a small design problem

design a program

- takes names & phone numbers as input
- saves and restores from a file
- does lookup of number given name

a generic parseable type

```
module type PARSEABLE =
sig
    type t
    val parse: string -> t
    val unparse: t -> string
end;;
```

use parse/unparse for unmarshal/marshal too

a file module type

```
module type FILEFUN =
   functor (K: PARSEABLE) -> functor (V: PARSEABLE) ->
sig
   type keytype = K.t
   type valuetype = V.t
   type filetype
   val empty: unit -> filetype
   val read: filetype -> (keytype, valuetype) Hashtbl.t -> unit
   val write: filetype -> (keytype, valuetype) Hashtbl.t -> unit
end
```

a file implementation

```
module File : FILEFUN =
   functor (K: PARSEABLE) -> functor (V: PARSEABLE) ->
struct
   type keytype = K.t
   type valuetype = V.t
   type filetype = (string * string) list ref
   let empty () = ref [ ]
   let read file tbl =
      let insert p =
         Hashtbl.add tbl (K.parse (fst p)) (V.parse (snd p)) in
            List.iter insert !file
   let write file tbl =
      let cons k v l = (K.unparse k, V.unparse v) :: l in
            file := Hashtbl.fold cons tbl [ ]
end
```

a generic file-backed mapper

```
module Mapper =
    functor (K: PARSEABLE) -> functor (V: PARSEABLE) ->
struct
    module KVF = File (K) (V)
    type keytype = K.t
    type valuetype = V.t
    let file = KVF.empty ()
    let tbl = Hashtbl.create (10)
    let save () = KVF.write file tbl
    let restore () = KVF.read file tbl
    let put k v = Hashtbl.add tbl k v
    let get k = Hashtbl.find tbl k
    let remove k = Hashtbl.remove tbl k
    let has k = Hashtbl.mem tbl k
end
```

names & phone numbers

```
module Name: PARSEABLE =
struct
   type t = string
   let parse x = x
   let unparse x = x
end;;

module PhoneNumber: PARSEABLE =
struct
   type t = {areacode: string; rest: string}
   let parse s =
     {areacode = String.sub s 0 3; rest = String.sub s 4 7}
   let unparse n = String.concat "." [n.areacode ; n.rest]
end;;
```

a phonebook implementation

```
module PB =
struct
   module M = Mapper (Name) (PhoneNumber)
   include M
   let enter name num =
     M.put (Name.parse name) (Num.parse num)
   let lookup name =
     let n = Name.parse name in
        if M.has n then PhoneNumber.unparse (M.get n)
        else "missing"
end
```

using the phonebook

```
# PB.enter "home" "617 9644620";;
- : unit = ()
# PB.lookup "home";;
- : string = "617.9644620"
# PB.save ();;
# PB.enter "office" "617 2588471";;
# PB.lookup "office";;
- : string = "617.2588471"
# PB.enter "home" "617 9999999";;
# PB.restore ();;
# PB.lookup "home";;
- : string = "617.9644620"
```

fully functorizing (1)

```
module type MAPPERFUN =
   functor (K: Parseable) -> functor (V: Parseable) ->
sig
   type keytype = K.t
   type valuetype = V.t
   val save: unit -> unit
   val restore: unit -> unit
   val put: keytype -> valuetype -> unit
   val get: keytype -> valuetype
   val has: keytype -> bool
   val remove: keytype -> unit
end
```

fully functorizing (2)

```
module PBFun =
   functor (Name: PARSEABLE) ->
   functor (Num: PARSEABLE) ->
   functor (MF: MAPPERFUN) ->
struct
   module M = MF (Name) (Num)
   include M
   let enter name num =
      M.put n (Name.parse name) (Num.parse num)
   let lookup name =
      let n = Name.parse name in
      if M.has n then
         Num.unparse (M.get n)
      else "missing"
end
```

putting it all together

```
module MyPB = PBFun (Name) (PhoneNumber) (Mapper);;

MyPB.enter "home" "617 9644620";;
MyPB.lookup "home";;
```

notes

functorizing

- eliminates global references
- makes dependences explicit
- but parameter proliferation can be cumbersome

Mapper

- is a singleton
- probably a bad design

object-oriented solution

- would require a separate factory class
- using serialization avoids this
  but relies on extra-linguistic mechanism
  and doesn't give readable file

will this work?

```
module MarriageRegFun =
   functor (Man: PARSEABLE) ->
   functor (Woman: PARSEABLE) ->
   functor (MF: MAPPERFUN) ->
struct
   module M = MF (Man) (Woman)
   include M
   let enter a b=
      let a' = Man.parse a and b' = Woman.parse b in
         M.put a' b' ; M.put b' a'
   let lookup name =
      let n = Man.parse name in
      if M.has n then
         Woman.unparse (M.get n)
      else "missing"
```

sharing constraints

```
module MarriageRegFun =
   functor (Man: PARSEABLE) ->
   functor (Woman: PARSEABLE with type t = Man.t) ->
   functor (MF: MAPPERFUN) ->
struct
   module M = MF (Man) (Woman)
   include M
   let enter a b=
      let a' = Man.parse a and b' = Woman.parse b in
         M.put a' b' ; M.put b' a'
   let lookup name =
      let n = Man.parse name in
      if M.has n then
         Woman.unparse (M.get n)
      else "missing"
```

discussion

› what does ML offer over Java?
› why aren't sharing constraints a big deal in Java?

came up in discussion

› can a Caml module have two components with same name?
› apparently: yes
  with matching or different types
  in signature and structure
  one seems to shadow the other
› why?