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

ML modules
topics for today

- making dependencies explicit
- functorization
- controlling client's view of a module
- signature ascription
  - functors: functions from modules to modules
  - signatures: types for modules
  - structs: modules, export types and values
- elements of ML module language
Set implementation def and use

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

one possible type for the module

```ocaml
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
  type 'a t = 'a list
  sig
    module Type ManifestSet =
  end
end
```

```
let member s e = List.mem e s
let add s e = e :: s
let empty () = []
let mem empty () = List.empty
struct
module SetImpl : ManifestSet =
```

Another type for the same module

 endemic
d Vall member: a t -> a -> bool
 Vall add: a t -> a -> a t
 Vall empty: unit -> a t
 type a t
 sig
 module type opaqueset =

 end

 let member s = e = List.mem e s
 let add s e = e :: s
 let empty () = [] = ()
 type a t = a List
 struct
 module SetImplA: OpaqueSet =

controlling access
end

let union s1 s2 = List.append s1 s2

include SetImpl

module SetWithUnion = struct

extending a module
Suppose we want a set of strings

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

Substructure
module OrderedStringSet = struct

  module Os = OrderedString
  type t = Os.t

  let empty () = []

  let rec add s e =
    match s with
    | [] -> false
    | x :: xs -> if Os.lt x e then false else member xs e

  module OpaqueSet = Opaque

  let rec member s e =
    match s with
    | [] -> false
    | x :: xs -> if Os.lt x e then e else s else x :: add xs e

  module Opaque = OpaqueString

  let rec add s e =
    match s with
    | [] -> false
    | x :: xs -> if Os.lt x e then e else s else x :: add xs e

  let empty () = ()

  let type t = Os.t

module OpaqueSet = OpaqueString

module OrderedStringSet = strict
making it generic

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

module Type Ordered =

making it generic
module OrderedSetValmp = functor (Elt: Ordered) -> struct

  type element = Elt.t
  type set = Elt.t list

  let rec member s e = match s with
    | [] -> false
    | x :: xs -> if Elt.lt x e then false
      else member xs e

  let rec add s e = match s with
    | [] -> [e]
    | x :: xs -> if Elt.lt e x then s :: e
      else e :: add xs e

  let empty = ()

  type set = Elt.t list
  type element = Elt.t

  struct

  module OrderedSetImpl = functor (Elt: Ordered) ->

  end

  end

  end

end

let rec member s e = match s with
  | [] -> false
  | x :: xs -> if Elt.lt x e then false
    else member xs e

let rec add s e = match s with
  | [] -> [e]
  | x :: xs -> if Elt.lt e x then s :: e
    else e :: add xs e

let empty = ()

type set = Elt.t list
type element = Elt.t

struct

module OrderedSetValmp = functor (Elt: Ordered) ->

end

end

end

end

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

a small design problem
use parse/unparse for unmarshal/marshal too

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

4 generic parsable type
module FILEFUN = functor (K: PARSEABLE) -> functor (V: PARSEABLE) -> sig

  type keytype = K.t
  type valuetype = V.t

  type 'k keytype = 'k keytype
  type 'v valuetype = 'v valuetype

  type 'k keytype 'v valuetype 'k keytype 'v valuetype 'k 'v 'k 'v Hashtbl.t

  val read : filetype -> (keytype, valuetype) Hashtbl.t -> unit
  val write : filetype -> (keytype, valuetype) Hashtbl.t -> unit
  val empty : unit -> filetype

end

module type TYPEFILENAME = 
  functor (K: PARSEABLE) -> functor (V: PARSEABLE) ->

  type keytype = K.t
  type valuetype = V.t

  type 'k keytype = 'k keytype
  type 'v valuetype = 'v valuetype

  type 'k keytype 'v valuetype 'k keytype 'v valuetype 'k 'v 'k 'v Hashtbl.t

  val read : filetype -> (keytype, valuetype) Hashtbl.t -> unit
  val write : filetype -> (keytype, valuetype) Hashtbl.t -> unit
  val empty : unit -> filetype

end

module type TYPEFILENAME = 

module type TYPEFILENAME = 

module type TYPEFILENAME = 

module type TYPEFILENAME = 

module File : FILEFUN = functor (K: PARSEABLE) -> functor (V: PARSEABLE) ->
struct
type keytype = K.t
  type valuetype = V.t
end

let cons k v l = (K.unparse k, V.unparse v) :: l

let write file tbl =
  List.iter insert file
  Hashedbl.add tbl (K.parse (fst p), V.parse (snd p))

let insert p =
  let read file tbl = [ ]
  let empty ( ) ret = ()
  let string * string = String.String
  let type keytype = K.t
  let type valuetype = V.t
  let type keytype = K.t
  let type valuetype = V.t

  struct
    functor (K: PARSEABLE) -> functor (V: PARSEABLE) <- functor
    module File : FILEFUN =

  a file implementation
a generic file-backed mapper
names & phone numbers
An implementation of a phonebook:

```ocaml
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
```

A phonebook implementation
using the phonebook
fully functorizing (1)
fully functorizing (2)
Putting it all together

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

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

And doesn’t give readable file. But relies on extra-linguistic mechanism.
Will this work?
Sharing constraints
Discussion

- why?
- one seems to shadow the other in signature and structure with matching or different types
- apparently: yes
- can a Caml module have two components with same name?
- came up in discussion
- why aren't sharing constraints a big deal in Java?
- what does ML offer over Java?