March 20, 2002
6898: Advanced Topics in Software Design
ML Modules
ML

Daniel Jackson
topics for today

- making dependencies explicit
- functionization
- controlling client's view of a module
- signature ascription
- functors: functions from modules to modules
- signatures: types for modules
- structures: modules, export types and values
- elements of ML module language
```ocaml
SetImpl.add s 3
let s = SetImpl.empty ()
end
let member s e = List.mem e s
let add s e = e :: s
let empty () = []
```

Set implementation def and use
one possible type for the module
module OpaqueSet =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
  type sig

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

  let member s e = List.mem e s
  let add s e = e :: s
  let empty () = []

  type 'a t

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

  another type for the same module
controlling access

let s = SetImplA.empty ();
SetImplA.add s 3;

let s = SetImplA.empty ()

(* type error *)

(* type error *)

let s = SetImplA.empty ()

SetImplA.add s 3;

let s = SetImplA.empty ()
module SetWithUnion = struct
  include SetImpl

  let union s1 s2 = List.append s1 s2

end;;
substructure

end;

let lt a b = a < b

type t = string

struct

module OrderedString = ...

suppose we want a set of strings
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

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

end;

end;

does this satisfy the signature OpaqueSet?
module Ordered = sig
  type t
  val lt: t -> t -> bool
end;;

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

type element = Elt.t

type set = Elt.t list

let empty () = []

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

let rec map f s =
  match s with
  | [] -> []
  | x :: xs -> f x :: map f xs

let rec fold f s =
  match s with
  | [] -> ()
  | x :: xs -> f x (fold f xs)

end

module OrderedSet = functor (Elt: Ordered) -> struct

end

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

a small design problem
module TYPEPARSEABLE =

end

val unparse: t -> string
val parse: string -> t

type t

module TYPEPARSEABLE =

a generic parseable type
module type FILEFUN = functor (K: PARSEABLE) -> functor (V: PARSEABLE) ->
sig
  val read: filetype -> (keytype, valuetype) Hashtbl.t -> unit
  val write: filetype -> (keytype, valuetype) Hashtbl.t -> unit
  val empty: unit -> filetype
  type keytype = K.t
  type valuetype = V.t
end
a file implementation
A generic file-backed mapper

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

module PhoneNumber: PARSEABLE

end:

let unparse n = String.concat "\n" [n.areacode, ";", n.rest]

let parse s = String.sub s 0 3; rest = String.sub s 4

Strict

module PhoneNumber: PARSEABLE

end:

let unparse x = x

let parse x = x

Strict

module PhoneNumber: PARSEABLE
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

end

let lookup name = let n = Name.parse name in
if M.has n then
  PhoneNumber.unparse (M.get n)
else
  "missing"
end

let enter name num = let lookup name = Let
include M

module M = Mapper (Name) (PhoneNumber)

strict
module PB =

a phonebook implementation
using the phonebook
end

val remove: keytype -> unit
val has: keytype -> bool
val get: keytype -> valuetype
val put: keytype -> valuetype -> unit
val restore: unit -> unit
val save: unit -> unit

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

sig

  functor (K: Parseable) -> functor (V: Parseable) -> module type MAPPERFUN

fully functorizing (1)
module PBFun = functor (Name: PARSEABLE) ->
  functor (Num: PARSEABLE) ->
  functor (MF: MAPPERFUN) ->
  struct
    module M = MF (Name) (Num)
    strict
      functor (MF: MAPPERFUN) <-
      functor (Num: PARSEABLE) <-
      functor (Name: PARSEABLE) <-
      module PBFun

fully functorizing (2)
module MyPB = PBFun (Name) (PhoneNumber) (Mapper);;
MyPB.enter "home" "617 964 4620";;
MyPB.lookup "home";;
and doesn’t give readable file
but relies on extra-linguistic mechanism
using serialization avoids this
would require a separate factory class
object-oriented solution
probably a bad design
is a singleton
Mapper
but parameter proliferation can be cumbersome
makes dependencies explicit
eliminates global references
functorizing

notes
Will this work?

else "missing"

Woman.unparse (M.get u)

if M.has in then

let in = Woman.parse name in

let lookup name =

let put a, b = M.put b a

let a' = Man.parse a and b' = Woman.parse b in

let enter a = b

include M

module M = MF (Man) (Woman)

strict

functor (MF: MAPPERFUN) <-

functor (Woman: PARSERABLE) <-

functor (Man: PARSERABLE) <-

module MarriageRegFun =
module MarriageRegFun =

functor (Man: PARSEABLE) ->

functor (Woman: PARSEABLE with type t = Man.t) ->

functor (MF: ...

let lookup name =

let n = Man.parse name in

if M has n then

let a = Man.parse a and b = Woman.parse b in

let enter a b =

include M

module M = MF (Man) (Woman)

strict

functor (MF: MAPPERFUN) <-

functor (Woman: PARSEABLE with type t = Man.t) <-

functor (Man: PARSEABLE) <-

module MarriageRegFun

sharing constraints
Why?

One seems to shadow the other in signature and structure with matching or different types.

Apparent? 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?

Discussion