

CMSC 631

O'Caml Separate Compilation

- Put interface in .mli file
- mySet.mli:
type 'a t
val empty : 'a t
val mem : 'a -> 'a t -> bool
val insert : 'a -> 'a t -> 'a t

Implementation

- mySet.ml:
type 'a t = 'a list
let empty : 'a t = []
let rec mem x s = match s with
| [] -> false
| hd :: tail -> if (hd = x) then true
else mem x tail
let insert x s = if (mem x s) then s
else x :: s

Imperative test

```
let set = ref MySet.empty in
try
  while true do
    output_string stdout "set> ";
    flush stdout;
    let line = input_line stdin in
      if MySet.mem line !set then
        Printf.printf "%s is already in the set\n" line
      else
        (Printf.printf "adding %s\n" line; set := MySet.insert line !set)
done
with
  End_of_file -> ();
```

Tail recursive test

```
let rec once set =
  output_string stdout "set> ";
  flush stdout;
  let line = input_line stdin in
    if MySet.mem line set then
      (Printf.printf "%s is already in the set\n" line; once set )
    else
      (Printf.printf "adding %s\n" line; once (MySet.insert line set) );

try
  once MySet.empty
with
  End_of_file -> ()
```

ML modules

- Collection of types and values
- Used to define interfaces
 - not objects; can't have an instance of a module

ML Module signatures

```
module type FsetSig = sig
  type 'a t
  val empty : 'a t
  val mem : 'a -> 'a t -> bool
  val insert : 'a -> 'a t -> 'a t
end
```

ML Module implementation

```
module Fset = struct
  type 'a t = 'a list
  let empty = []
  let rec mem x s = match s with
    | [] -> false
  | hd :: tail -> if (hd = x) then true else mem x tail
  let insert x s = if (mem x s) then s else x :: s;
end
```

Use

```
# Fset.empty;;
- : 'a Fset.t = []
# Fset.insert 5 Fset.empty;;
- : int Fset.t = [5]
#
```

Information hiding/abstraction

```
# module Fset' : FsetSig = Fset ;;
module Fset' : FsetSig
# let s = Fset'.empty;;
val s : 'a Fset'.t = <abstr>
# let t = Fset'.insert 5 s;;
val t : int Fset'.t = <abstr>
# Fset'.mem 3 t;;
- : bool = false
# Fset'.mem 5 t;;
- : bool = true
```

Functors

- A Functor is used to define a module parameterized by another module
- A Function takes a module as an argument and returns a module

Parameterized over EltSig

```
module type EltSig =
  sig
    type elt
    val same : elt -> elt -> bool
  end
```

```

module type MySetSig =
  sig
    type elt
    type t
    val empty : t
    val mem : elt -> t -> bool
    val insert : elt -> t -> t
  end

```

The Functor

```

module MakeMySet (Elt : EltSig) : MySetSig = struct
  type elt = Elt.elt
  type t = elt list

  let empty = []
  let rec mem x s = match s with
    [] -> false
  | hd :: tail -> if (Elt.same hd x) then true else mem x tail
  let insert x s = if (mem x s) then s else x :: s
end

```

Using the functor

```

module Int = struct
  type elt = int
  let same = (=)
end;;

module IntSet = MakeMySet(Int);;

```

A problem

```

# IntSet.empty;;
- : IntSet.t = <abstr>
# IntSet.insert 5 IntSet.empty;;
This expression has type int but is here used
with type
IntSet.elt = MakeMySet(Int).elt

```

Need with clause

- Tell system we don't want to abstract over element type

Modified Functor

```

module MakeMySet (Elt : EltSig) : MySetSig with type elt = Elt.elt
= struct
  type elt = Elt.elt
  type t = elt list

  let empty = []
  let rec mem x s = match s with
    [] -> false
  | hd :: tail -> if (Elt.same hd x) then true else mem x tail
  let insert x s = if (mem x s) then s else x :: s
end

```

Now it works

```
open IntSet;;
let s = empty;;
let t = insert 5 s;;
mem 3 t;;
mem 5 t;;
```

Programming assignment 1

- Use the built in Set functor
- Write a Graph module or functor
- Due Friday, Sept 20th

Type Systems

Luca Cardelli

You should have questions

- When there is assigned reading, you are expected to read it and have questions if you aren't prepared for an exam on the material

Type judgements

- $\Gamma \vdash \square$
 - The environment Γ is valid
- $\Gamma \vdash A$
 - Given environment Γ , A is a valid type
- $\Gamma \vdash M:A$
 - Given environment Γ , expression M has type A

Type rules

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \ \square \ B}$$
$$\frac{\Gamma, x:A \vdash M:B}{\Gamma \vdash (\lambda x:A. M) : A \ \square \ B}$$

Types in F_1

- Unit, Bool, Nat
- Product types
- Union types
- Records and Variants
- References

Recursive Types

- $\mu X.A$ is the type equivalent to let $\text{rec } X = A$
- In other words, the type of $\mu X.A$ is A
 - but A may contain occurrences of X
 - which should be interpreted as being recursively defined as A
- $\mu X.A$ unfolds to $(\mu X.A / X) A$
 - replace all free occurrences of X in A with $\mu X.A$

Recursive Types

(Env x)

$$\frac{}{\Gamma \vdash \square} \quad \begin{array}{l} X \text{ not in domain}(\Gamma) \\ \Gamma \vdash \square \end{array}$$

(Type Rec)

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \mu X.A}$$

Recursive Types

(Val Fold)

$$\frac{\Gamma \vdash M : (\mu X.A / X) A}{\Gamma \vdash \text{fold}_{\mu X.A} M : \mu X.A}$$

(Val UnFold)

$$\frac{\Gamma \vdash M : \mu X.A}{\Gamma \vdash \text{unfold}_{\mu X.A} M : (\mu X.A / X) A}$$

Recursive Types

- $\text{List}_A = \mu X. \text{Unit} + (A \times X)$
- unfolds to
 - $\text{Unit} + (A \times \mu X. \text{Unit} + (A \times X))$