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)
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

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

```haskell
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

- `[ ] ⊢ □` - The environment □ is valid
- `[ ] ⊢ A` - Given environment □, A is a valid type
- `[ ] ⊢ M:A` - Given environment □, expression M has type A

Type rules

- `⊥ ⊢ A`, `⊥ ⊢ B`:
  `⊥ ⊢ A ⊔ B`
- `x:A ⊢ M:B`:
  `⊥ ⊢ (\(\overline{x:A}. M\) : A ⊔ B)`
Types in $F_1$

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

Recursive Types

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

Recursive Types

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