# Modules and Abstract Data Types

COS 326
David Walker
Princeton University

# Welcome Back!

Assignment #5 is out! Get started! Partners allowed!

Precept this week: Midterm analysis

Why go? There's still a final!

The proofs will be more difficult!

# Before the Break

Design for change using the ML module system

- Structures implement new types & operations over them
- Signatures provide the interfaces
- Functors are functions from modules to modules
  - they allow you to define parameterized modules
- ML also has dynamic, first-class modules

# ANOTHER EXAMPLE OF FUNCTORS

# A Bigger Example

```
module type SET =
  sig
    type elt
    type set
    val empty : set
    val is_empty : set -> bool
    val insert : elt -> set -> set
    val singleton : elt -> set
    val union : set -> set -> set
    val intersect : set -> set -> set
    val remove : elt -> set -> set
    val member : elt -> set -> bool
    val choose : set -> (elt * set) option
    val fold : (elt -> 'a -> 'a) -> 'a -> set -> 'a
  end
```

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
             :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
             :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

ListSet is a parameterized module – given a module argument for Elt, it generates a new module.

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
              :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

This is a very simple, anonymous signature (it just specifies there's some type t) for the argument to ListSet

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
               :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

This is the signature of the resulting module – we have a set plus the knowledge that the Set's elt type is equal to Elt.t

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
              :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end


module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

These are two SET modules that I created with the ListSet functor.

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
             :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set =
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

In this case, I'm passing in an anonymous module for Elt that defines t to be int.

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
              :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end


module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

We know that IntListSet.elt = int.

# Our Set Implementation is a Functor:

```
module ListSet (Elt : sig type t end)
             :   (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = [
  let is_empty (s:set
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:el
...
end
```

```
module type SET =
  sig
    type elt = int
    type set
    val empty : set
    val is_empty : set -> bool
    val insert : elt -> set -> set
    ...
  end
```

equal to int so we can actually build a set using insertions!

```
module IntListSet = ListSet(struct type t =
module StringListSet = ListSet(struct type t = string end)
```

# Let’s Write the Rest of the Functor

```
module ListSet (Elt : sig type t end)
              : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  ...
end
```

# Let's Write the Rest of the Functor

```
module ListSet (Elt : sig type t end)
               :(SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set = ???
end
```

# Let's Write the Rest of the Functor

```
module ListSet (Elt : sig type t end)
              : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
      s1 @ s2
  ...
end
```

Ugh. Wastes space if s1 and s2 have duplicates. (Also, makes remove harder…)

# Let's Write the Rest of the Functor

```
module ListSet (Elt : sig type t end)
              : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
      List.fold_right insert s1 s2
  ...
end
```

Gets rid of the duplicates. Now remove can stop once it finds the element.

# Let's Write the Rest of the Functor

```
module ListSet (Elt : sig type t end)
               : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
      List.fold_right insert s1 s2
  ...
end
```

But List.mem and List.fold_right take time proportional to the length of the list. So union is quadratic.

Gets rid of the duplicates. Now remove can stop once it finds the element.

# Let's Write the Rest of the Functor

```
module ListSet (Elt : sig type t end)
              : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
      List.fold_right insert s1 s2
  ...
end
```

If we knew that s1 and s2 were *sorted* we could use the merge from mergesort to compute the sorted union in linear time.

# A Sorted List Set Functor

```
module type COMPARATOR = sig
  type t
  val compare : t -> t -> Order.order
end

module SortedListSet (Elt : COMPARATOR)
                   : (SET with elt = Elt.t) =
struct ...
  let rec insert (x:elt) (s:set) : set =
      match s with
      | [] -> [x]
      | h::t -> (match Elt.compare x h with
                 | Less -> x::s
                 | Eq -> s
                 | Greater -> h::(insert x t)) ...
end
```

# A Sorted List Set Functor

```
module type COMPARATOR = sig
  type t
  val compare : t -> t -> Order.order
end

module SortedListSet (Elt
                  : (SET with e
struct ...
  let rec insert (x:elt) (s:set) .
      match s with
      | [] -> [x]
      | h::t -> (match Elt.compare x h with
                 | Less -> x::s
                 | Eq -> s
                 | Greater -> h::(insert x t)) ...
end
```

To support the sorting, I'm passing in a comparison operation to go with the element type.

# A Sorted List Set Functor

```
module SortedListSet (Elt : COMPARATOR)
                   : (SET with elt = Elt.t) =
struct ...
  let rec union (s1:set) (s2:set) : set =
      match s1, s2 with
      | [], _ -> s2
      | _, [] -> s1
      | h1::t1, h2::t2 ->
         (match Elt.compare h1 h2 with
            | Less -> h1::(union t1 s2)
            | Eq -> h1::(union t1 t2)
            | _ -> h2::(union s1 t2))
  …
end
```

# Simpler

```
module SortedListSet (Elt : COMPARATOR)
                    : (SET with elt = Elt.t) =
struct ...
  let rec union (s1:set) (s2:set) : set = ...

  let insert (x:elt) (s:set) : set = union [x] s ;;

end
```

# Another Alternative:  Bit Vectors

```
module BitVectorSet (Elt : sig type t
                               val index : t -> int
                               val max : int
                           end)
                    : (SET with elt = Elt.t) =
struct
  type set = bool array
  let empty = Array.create Elt.max false
  let member x s = s.(Elt.index x)
  let union s1 s2 =
       Array.init Elt.max
         (fun i -> s1.(i) || s2.(i))
  let intersect s1 s2 =
       Array.init Elt.max
         (fun i -> s1.(i) && s2.(i))
  ...
```

# Another Alternative: Binary Search Trees

```
module BSTreeSet(Elt : sig type t
                           val compare : t -> t -> Order.order
                       end) : (SET with elt = Elt.t) =
struct
  type set = Leaf | Node of set * elt * set
  let empty() = Leaf
  let rec insert (x:elt) (s:set) : set =
      match s with
      | Leaf -> Node(Leaf,x,Leaf)
      | Node(left,e,right) ->
        (match Elt.compare x e with
             | Eq -> s
             | Less -> Node(insert x left, e, right)
             | Greater -> Node(left, e, insert x right))
  let rec member (x:elt) (s:set) : bool =
      match s with
      | Leaf -> false
      | Node(left,e,right) ->
       (match Elt.compare x e with
           | Eq -> true
           | Less -> member x left
           | Greater -> member x right)
  ... end
```

# SIGNATURE SUBTYPING

# Subtyping

A module matches any interface as long as it provides *at least* the definitions (of the right type) specified in the interface.

But as we saw earlier, the module can have more stuff.

- e.g., the deq function in the Queue modules

Basic principle of subtyping for modules:

- wherever you are expecting a module with signature S, you can use a module with signature S', as long as all of the stuff in S appears in S'.
- That is, S' is a bigger interface.

# Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

# Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

RING is a sub-type of GROUP.

# Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

There are *more* modules matching the GROUP interface than the RING one.

# Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one  : t
    val add  : t -> t -> t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

Any module expecting a GROUP can be passed a RING.

# Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    include GROUP
    val one  : t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

The **include** primitive is like cutting-and-pasting the signature's content here.

# Groups versus Rings

```
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    include GROUP
    val one  : t
    val mul  : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
```

That *ensures* we will be a sub-type of the included signature.

# ASSIGNMENT #5
# CODE WALKTHROUGH

# MODULE EVALUATION

# The Structure of ML

An ML program is a sequence of modules.

module m1 = module expression ...

module m2 = module expression ...

module m3 = module expression ...

To evaluate an ML program, we must evaluate this sequence of module expressions.  How?

# Evaluating the contents of a module

A module expression is a series of declarations

- How does one evaluate a type declaration?  We'll ignore it.
- How does one evaluate a let declaration?

```
let x = e
```

evaluate the expression e
bind the value to x

How does one evaluate an entire structure?

- evaluate each declaration in order from first to last

# Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf “Hello COS %d\n” x

let foo =
   Printf.printf “Byeee!\n”

let _ =
    main ()
```

# Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf “Hello COS %d\n” x

let foo =
  Printf.printf “Byeee!\n”

let _ =
   main ()
```

<u>Step 1:</u>
evaluate the 1st declaration

but the RHS (326)
is already a value so there’s
nothing to do except
remember that x is bound
to the integer 326

# Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf "Hello COS %d\n" x

let foo =
  Printf.printf "Byeee!\n"

let _ =
   main ()
```

Step 2:
evaluate the 2nd declaration
this is slightly trickier:

let main () = ...

really declares a function.
It's equivalent to:

let main = fun () -> ...

"fun () -> ..." is already
a value, like 326.
So there's nothing to do again.

# Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
   Printf.printf "Hello COS %d\n" x

let foo =
   Printf.printf "Byeee!\n"

let _ =
    main ()
```

Step 3:
evaluate the 3rd declaration

let foo = ...

evaluation of this expression has an effect – it prints out "Byeee!\n" to the terminal.

the resulting value is () which is bound to foo

# Evaluating the contents of a module

main.ml

```
let x = 326

let main () =
  Printf.printf “Hello COS %d\n” x

let foo =
  Printf.printf “Byeee!\n”

let _ =
   main ()
```

Step 4:
evaluate the 4th declaration

let _ = ...

evaluation main ()
causes another effect.

“Hello ...” is printed

the resulting value is () again.
the “_” indicates we don’t
care to bind () to any variable

# A Variation

main.ml

```
let x = 326

let main =
  (fun () ->
    Printf.printf "Hello COS %d\n" x)

let foo =
  Printf.printf "Byeee!\n"

let _ =
  main ()
```

This evaluates exactly the same way

We just replaced

let main () = ...

with the equivalent

let main = fun () -> ...

# A Variation

main.ml

```
let x = 326

let main =
  Printf.printf "Hello COS %d\n" x;
  (fun () -> ())

let foo =
   Printf.printf "Byeee!\n"

let _ =
    main ()
```

This rewrite does something different.

On the 2nd step, it prints because that's what evaluating this expression does:

```
Printf.printf "Hello COS %d\n" x;
(fun () -> ())
```

The result of the expression is:

```
fun () -> ()
```

which is bound to main.
This is a pretty silly function.

# A Variation

main.ml

```
module C326 =
struct
  let x = 326

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let _ =
  Printf.printf "Done\n"
```

Now what happens?

# A Variation

main.ml

```
module C326 =
struct
  let x = 326

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

# A Variation

main.ml

```
module C326 =
struct
  let x = 326

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

Executing the module declaration has the effect of executing every declaration within it in order.

Executing let done = ... is as before

# A Variation

main.ml

```
module C326 =
struct
  exception Unimplemented
  let x = raise Unimplemented

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

# A Variation

main.ml

```
module C326 =
struct
  exception Unimplemented
  let x = raise Unimplemented

  let main =
    Printf.printf “Hello COS %d\n” x;
    (fun () -> ())

  let foo = Printf.printf “Byeee!\n”

  let _ =  main ()
end

let done =
  Printf.printf "Done\n"
```

Now what happens?

The entire file contains 2 decls:

- module C326 = ...
- let done = ...

We execute both of them in order.

Executing the module declaration has the effect of executing every declaration within it in order.

The first declaration within it raises an exception which is not caught! That is the only result.

# A Variation

main.ml

```
module type S =
sig
  type t = int
  val x : t
end

module F (M:S) : S =
struct
  let wow = Printf.printf "%d\n" M.x
  let t = M.t
  let x = M.x
end

let done = Printf.printf "Done\n"
```

Now what happens?

The entire file contains 3 decls:

- module type = ...
- module F (M:S) : S = ...
- let done = ...

# A Variation

main.ml

```
module type S =
sig
  type t = int
  val x : t
end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let done = Printf.printf "Done\n"
```

The signature declaration has no (run-time) effect.

The functor declaration is like declaring a function value.

The body of the functor is not executed until it is applied.

The functor is not applied here so M.x is not printed.

Only “Done\n” is printed.

# A Variation

main.ml

```
module type S = sig ... end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let module M1 = F (
      struct
        type t = int
        val x = 3
      end)

let done = Printf.printf "Done\n"
```

What happens now?

# A Variation

main.ml

```
module type S = sig ... end

module F (M:S) : S =
struct
  let wow = Printf.printf “%d\n” M.x
  let t = M.t
  let x = M.x
end

let module M1 = F (
      struct
         type t = int
         val x = 3
      end)

let done = Printf.printf "Done\n"
```

What happens now?

When M1 is declared,
F is applied to an argument.

This creates a new structure and its components are executed.

This has the effect of printing 3.

# FIRST CLASS MODULES

# Limitations of the Module language

The module language contains:

- structures
- functions from structures to structures

This is like a programming with just typed functions and simple data d. Possible expressions:

d

f d

f (f d)

f (d, f d)

etc

# But there is no dynamic decision making

We can’t do this:

```
module Set =
  if parse_command_line () = ”big_set” then
    HashSet
  else
    ListSet
```

# First-class Modules

There is a way of including modules in the expression language.

And then to get it back out.

```
module type Box = sig val x : int end

module Three : Box = struct let x = 3 end
module Four : Box = struct let x = 4 end

let three = (module Three : Box)
let four = (module Four : Box)
```

# First-class Modules

There is a way of including modules in the expression language.

And then to get it back out.

```
module type Box = sig val x : int end

module Three : Box = struct let x = 3 end
module Four : Box = struct let x = 4 end

let three = (module Three : Box)
let four = (module Four : Box)
```

three and four are ordinary values
that can be passed around
like other ordinary values

```
three : module Box
```

# First-class Modules

... and then getting them back out ....

```
module type Box = sig val x : int end

module Three : Box = struct let x = 3 end
module Four : Box = struct let x = 4 end

let three = (module Three : Box)
let four = (module Four : Box)

let three_or_four : (module Box) =
  if command_line () then three else four

module Three_or_Four = (val three_or_four : Box)
```

keyword val brings it back into the module language

# DESIGN CONSIDERATIONS FROM REAL WORLD OCAML

Expose Concrete Types Rarely

```
type ‘a tree =
    Leaf
  | Node of ‘a * ‘a tree * ‘a tree
```

```
match t with
   Leaf -> ...
 | Node (v, left, right) -> ...
```

```
type 'a tree

val empty : 'a tree
val node  : 'a -> 'a tree -> 'a tree -> 'a tree
val top   : 'a tree -> ('a * 'a tree * 'a tree) option
```

```
match top t with
   None -> ...
 | Some (v, left, right) -> ...
```

```
type 'a tree

type 'a view =
  Empty
| Single of 'a
| Children of 'a tree * 'a tree

val split : 'a tree -> 'a view
```

```
match split t with
   Empty -> ...
 | Single v -> ...
 | Children (t1, t2) -> ...
```

# Design for Call Sites

Use techniques that make it easier to read uses of your module, not just the signature

Use techniques that make it easier to read uses of your module, not just the signature

Basic rules: Choose good names for types, record labels, values

Good names aren't always long: `fun x -> x * 2`

Rule of thumb:

- names with small scope are short
- names with large scope are long (eg: names in module interfaces)

Tradeoff:

- long, descriptive names for uncommon values
- shorter names for common values ("map")

Use techniques that make it easier to read uses of your module, not just the signature

Use techniques that make it easier to read uses of your module, not just the signature

```
substring “0123456” 2 3
```

Use techniques that make it easier to read uses of your module, not just the signature

```
substring "0123456" 2 3
```

vs

```
substring "0123456" ~pos:2 ~len:3
```

Defining functions with labelled arguments

```
let divide ~num ~denom = num / denom

val divide : num:int -> denom:int -> int
```

# Optional Arguments

```
let concat ?sep x y =
  let sep =
    match sep with
      None -> ""
    | Some x -> x
  in
  x ^ sep ^ y

val concat : ?sep:string -> string -> string -> string
```

```
concat "foo" "bar";;

concat ~sep:":" "foo" "bar";;
```

# Optional Arguments

```
let concat ?sep x y =
  let sep =
    match sep with
      None -> ""
    | Some x -> x
  in
  x ^ sep ^ y
```

more concise ("syntactic sugar")

```
let concat ?(sep="") x y = x ^ sep ^ y
```

# Optional Arguments

```
let concat ?(sep="") x y = x ^ sep ^ y
```

Pitfalls of optional arguments:

- Users might not realize there is an optional argument
- Easy to accidentally get a “bad default”

Rules of thumb:

- Avoid in functions that are rarely used
- Avoid in functions internal to a module
- Avoid in situations where you have to “think carefully” about that argument
- Avoid just to save a couple of characters of typing
- Use it to make your code clearer

More on labelled and optional arguments: Real World OCaml Chap 2

# Create Uniform Interfaces

# Jane St Core

An alternate to the standard core libraries:

```
opam install core
```

More features

More uniform:

- a module for every type: Int, Float, ...
  - useful for instantiating functors
- the primary type in a module is called t:  Int.t
- put t first:
  - the functions that take t should take t first:
- functions that routinely throw an exception end in _exn
  - otherwise, return an option

# Interfaces before Implementations

# Interfaces before Implementations

<u>When writing functions:</u>

- start with the type of the function
- the types drive the structure of the code
  - eg: 1 case per element of a data type

<u>When writing modules:</u>

- start with the type of the module (the signature)
- is there a primary type t?
- what accessors do you need?  what types?
- interfaces allow you to flesh out a design

# DESIGN CONSIDERATIONS: REPRESENTATION INVARIANTS

# Efficient Data Structures

In 226, you learned about all kinds of clever data structures:

- red-black trees
- union-find sets
- tries, ...

Not just any tree is a red-black tree. In order to be a red-black tree, you need to obey several *invariants*:

- keys are in order in the tree
- # of black nodes to the root is constant along all paths,...

Operations such as look-up, *depend upon* those invariants to be correct. All inputs to look-up must satisfy the red-black invariant

Such invariants are often called *representation invariants*

# A Signature for Sets

```
module type SET =
  sig
    type ‘a set
    val empty : ‘a set
    val mem : ‘a -> ‘a set -> bool
    val add : ‘a -> ‘a set -> ‘a set
    val rem : ‘a -> ‘a set -> ‘a set
    val size : ‘a set -> int
    val union : ‘a set -> ‘a set -> ‘a set
    val inter : 'a set -> ‘a set -> 'a set
  end
```

# Sets as Lists

```
module Set1 : SET =
  struct
    type 'a set = 'a list
    let empty = []
    let mem = List.mem
    let add x l = x :: l
    let rem x l = List.filter ((<>) x) l
    let rec size l =
      match l with
      | [] -> 0
      | h::t -> size t + (if mem h t then 0 else 1)
    let union l1 l2 = l1 @ l2
    let inter l1 l2 = List.filter (fun h -> mem h l2) l1
  end
```

Very slow in many ways!

# Sets as Lists without Duplicates

```
module Set2 : SET =
  struct
    type 'a set = 'a list
    let empty = []
    let mem = List.mem
    (* add:  check if already a member *)
    let add x l = if mem x l then l else x::l
    let rem x l = List.filter ((<>) x) l
    (* size:  list length is number of unique elements *)
    let size l = List.length l
    (* union: discard duplicates *)
    let union l1 l2 = List.fold_left
          (fun a x -> if mem x l2 then a else x::a) l2 l1
    let inter l1 l2 = List.filter (fun h -> mem h l2) l1
  end
```

# Back to Sets

The interesting operation:

```
(* size:  list length is number of unique elements *)
let size (l:'a set) : int = List.length l
```

Why does this work?  It depends on an invariant:

*All lists supplied as an argument contain no duplicates.*

A *representation invariant* is a property that holds of all values of a particular (abstract) type.

# Implementing Representation Invariants

For lists with no duplicates:

```
(* checks that a list has no duplicates *)
let rec inv (s : 'a set) : bool =
    match s with
      [] -> true
    | hd::tail -> not (mem hd tail) && inv tail

let rec check (s : 'a set) (m:string) : 'a set =
  if inv s then
    s
  else
    failwith m
```

# Debugging with Representation Invariants

As a precondition on input sets:

```
(* size:  list length is number of unique elements *)
let size (s:'a set) : int =
  ignore (check s "size:  bad set input");
  List.length s
```

# Debugging with Representation Invariants

As a precondition on input sets:

```
(* size:  list length is number of unique elements *)
let size (s:'a set) : int =
  ignore (check s "size:  bad set input");
  List.length s
```

As a postcondition on output sets:

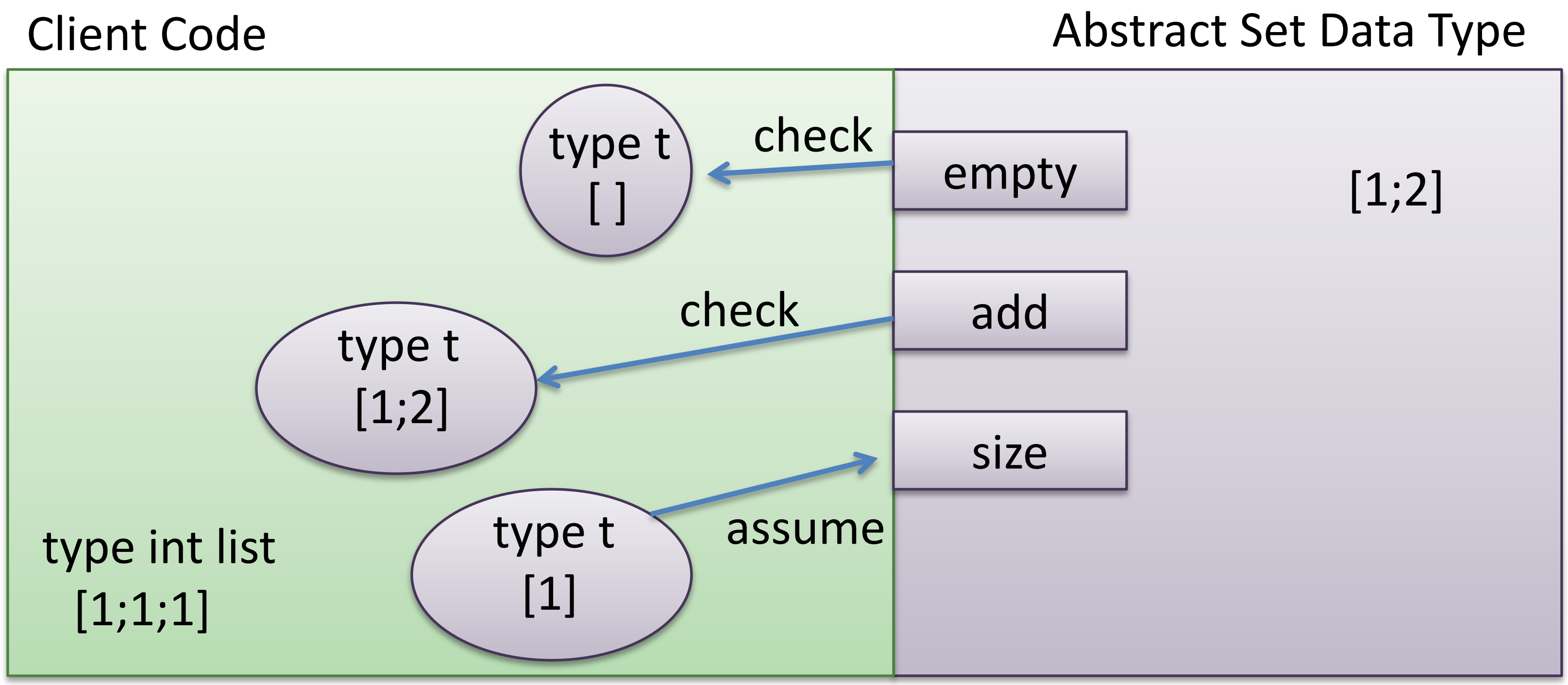
```
(* add x to set s *)
let add x s =
  let s = if mem x s then s else x::s in
  check s "add: bad set output"
```

# A Signature for Sets

```
module type SET =
  sig
    type ‘a set
    val empty : ‘a set
    val mem : ‘a -> ‘a set -> bool
    val add : ‘a -> ‘a set -> ‘a set
    val rem : ‘a -> ‘a set -> ‘a set
    val size : ‘a set -> int
    val union : ‘a set -> ‘a set -> ‘a set
    val inter : 'a set -> ‘a set -> 'a set
  end
```

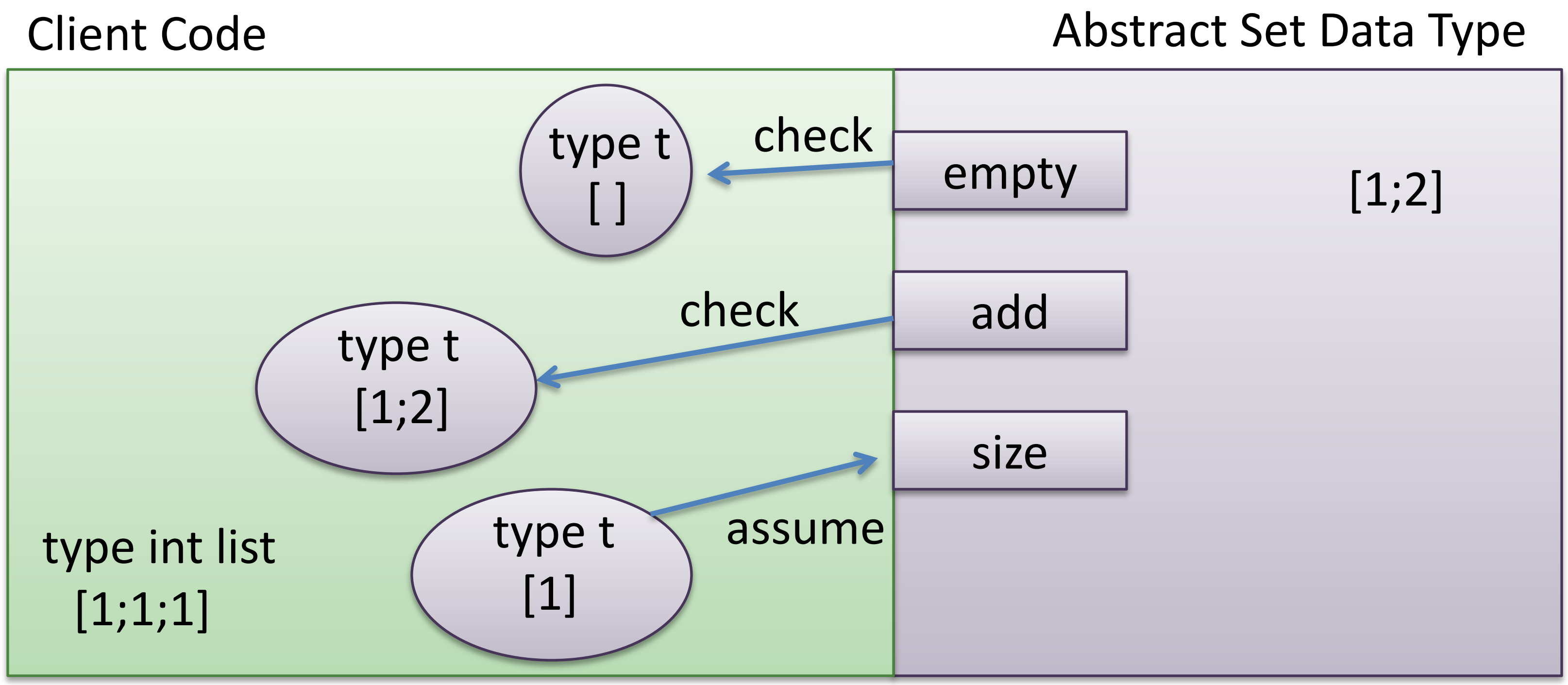
Suppose we check all the red values satisfy our invariant leaving the module, do we have to check the blue values entering the module satisfy our invariant?

# Representation Invariants Pictorially




*When debugging*, we can check our invariant each time we construct a value of abstract type. We then get to assume the invariant on input to the module.

# Representation Invariants Pictorially




*When proving*, we prove our invariant holds each time we construct a value of abstract type and release it to the client. We *get to assume* the invariant holds on input to the module.

Such a proof technique is *highly modular*: Independent of the client!

# Repeating myself

You may

*assume the invariant inv(i) for module inputs i with abstract type*

provided you

*prove the invariant inv(o) for all module outputs o with abstract type*

# Design with Representation Invariants

A key to writing correct code is understanding your own invariants very precisely

Try to write down key invariants

- if you write them down then you can be sure you know what they are yourself!
- you may find as you write them down that they were a little fuzzier than you had thought
- great documentation for others
- great debugging tool
- you'll need them to prove to yourself that your code is correct

# SUMMARY

# Summary

OCaml's linguistic mechanisms include:

- *signatures* (interfaces)
- *structures* (implementations)
- *functors* (functions from modules to modules)
- *first-class modules* (modules as expressions

We can use the module system

- provides support for *name-spaces*
- *hiding information* (types, local value definitions)
- *code reuse* (via functors, reuseable interfaces, reuseable modules)

There's lots, lots, lots more to learn about how to design collections of modules effective. Real World OCaml has some good hints.