Modules and Representation Invariants

COS 326

David Walker

Princeton University
In previous classes:

Reasoning about individual OCaml expressions.

Now:

Reasoning about Modules (abstract types + collections of values)
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
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!
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
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
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 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"
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**

**Client Code**

- `type t []`
- `type t [1;2]`
- `type int list [1;1;1]`
- `type t [1]`

**Abstract Set Data Type**

- `empty`
- `add`
- `size`
- `[1;2]`

---

*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.
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!
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
The interesting operation:

\[
\begin{align*}
(* \text{ size: list length is number of unique elements } *) \\
\text{let size (l:'a set) : int = List.length l}
\end{align*}
\]

Why does this work? It depends on an invariant:

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

How is this invariant enforced? By using abstract types. Every value of the abstract type `'a set satisfies the invariant. Internally, the module knows that the `a set is `a list and can establish the invariant, but externally clients don’t know that and can’t mess with established invariants.
A representation invariant \( \text{inv}(v) \) for abstract type \( t \) is a property of all data values \( v \) with abstract data type \( t \).

**Client Code**

**Abstract Set Data Type**

- Invariants on abstract types are *local* to the ADT because they talk about the representation. Client code doesn’t know or care what the invariant is.
- However, client code *preserves the invariant* because it can’t mess with values of abstract type directly.
A representation invariant $\text{inv}(v)$ for abstract type $t$ is a property of all data values $v$ with abstract data type $t$.

- Because Clients can’t mess with the invariants on abstract types, ADT code gets to assume the invariant for inputs with abs. type provided it proves the invariant for outputs with abs. type.
- These proofs are modular: Done in isolation in the ADT module.
Establishing Representation Invariants

E.g., when it comes to the size function:

```ocaml
(* signature *)
val size : 'a set -> int

(* implementation: length is # of distinct elements *)
let size l = List.length l
```

If we want to assume all arguments to size have no duplicates, then:

– we have to ensure that our client can only pass us a list with no dups
– clients get their values of type ‘a set from our module, hence we have to ensure other functions in our module only produce lists with no duplicates
  • empty, add, rem, union, intersect
– typically the proof that a function produces elements that satisfy \textit{inv} depend on assumptions that function inputs satisfy \textit{inv}
  • add, rem, union, intersect
PROVING THE REP INVARIANT FOR THE SET ADT
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail

inv (empty) == true
```

Definition of empty:

```ocaml
let empty : 'a set = []
```

Proof Obligation:

```ocaml
inv (empty) == true
```

Proof:

```ocaml
inv (empty)
== inv []
== match [] with [] -> true | hd::tail -> ...
== true
```
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail
```

Checking add:

```ocaml
let add (x:'a) (l:'a set) : 'a set =
  if mem x l then l else x::l
```

Proof obligation:

for all x:'a and for all l:'a set,

if inv(l) then inv (add x l)
Theorem: for all \( x: \text{'a} \) and for all \( l: \text{'a set} \), if \( \text{inv}(l) \) then \( \text{inv}(\text{add } x \ l) \)

Proof:

(1) pick an arbitrary \( x \) and \( l \). (2) assume \( \text{inv}(l) \).

Break in to two cases:

-- one case when \( \text{mem } x \ l \) is true

-- one case where \( \text{mem } x \ l \) is false
Theorem: for all $x:a$ and for all $l:a$ set, if $\text{inv}(l)$ then $\text{inv}(\text{add } x \ l)$

Proof:

(1) pick an arbitrary $x$ and $l$.  (2) assume $\text{inv}(l)$.

\[
\text{case 1: assume (3): mem } x \ l \iff \text{true:}
\]

\[
\begin{align*}
\text{inv } (\text{add } x \ l) & \iff \text{inv } (\text{if mem } x \ l \ \text{then } l \ \text{else } x::l) \\
& \iff \text{inv } l \\
& \iff \text{true}
\end{align*}
\]

(by (2))
Theorem: for all \( x: 'a \) and for all \( l: 'a \) set, if \( \text{inv}(l) \) then \( \text{inv}(\text{add}(x, l)) \)

Proof:

(1) pick an arbitrary \( x \) and \( l \). (2) assume \( \text{inv}(l) \).

_**Case 2:** assume (3) not (\( \text{mem}(x, l) \)) => true:

\[
\begin{align*}
\text{inv}(\text{add}(x, l))
= &\ \text{inv}(\text{if } \text{mem}(x, l) \text{ then } l \text{ else } x::l) \\
= &\ \text{inv}(x::l) \\
= &\ \text{not}(\text{mem}(x, l)) \&\& \text{inv}(l) \\
= &\ \text{true} \&\& \text{inv}(l) \\
= &\ \text{true} \&\& \text{true} \\
= &\ \text{true}
\end{align*}
\]
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail
```

Checking rem:

```ocaml
let rem (x:'a) (l:'a set) : 'a set =
  List.filter ((<>) x) l
```

Proof obligation?

for all x:'a and for all l:'a set,

if inv(l) then inv (rem x l)  
prove invariant on output

assume invariant on input
Representation Invariants

Representation Invariant for sets without duplicates:

```
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail
```

Checking size:

```
let size (l:'a set) : int =
  List.length l
```

Proof obligation?

no obligation – does not produce value with type ‘a set
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
    match l with
    [] -> true
    | hd::tail -> not (mem hd tail) && inv tail
```

Checking union:

```ocaml
let union (l1:'a set) (l2:'a set) : 'a set =
  ...
```

Proof obligation?

for all l1:'a set and for all l2:'a set,

if inv(l1) and inv(l2) then inv (union l1 l2)

assume invariant on input
prove invariant on output
Representation Invariants

Representation Invariant for sets without duplicates:

\[
\begin{align*}
\text{let rec } \text{inv} \ (l : \ 'a \ \text{set}) : \ \text{bool} = \\
\ & \text{match} \ l \ \text{with} \\
\ & \quad [] \to \text{true} \\
\ & \quad \text{hd::tail} \to \ \text{not} \ \text{(mem hd tail)} \ \&\& \ \text{inv tail}
\end{align*}
\]

Checking inter:

\[
\begin{align*}
\text{let } \text{inter} \ (l1:'a\ \text{set}) \ (l2:'a\ \text{set}) : \ 'a\ \text{set} = \\
\ & \ldots
\end{align*}
\]

Proof obligation?

for all \(l1:'a\ \text{set}\) and for all \(l2:'a\ \text{set}\),

if \(\text{inv}(l1)\) and \(\text{inv}(l2)\) then \(\text{inv} \ (\text{inter} \ l1 \ l2)\)

\[
\begin{align*}
\text{assume invariant on input} & \quad \text{prove invariant on output}
\end{align*}
\]
Representation Invariants: a Few Types

- Given a module with abstract type t
- Define an invariant Inv(x)
- Assume arguments to functions satisfy Inv
- Prove results from functions satisfy Inv

```plaintext
sig
type t

val value : t

val constructor : int -> t

val transform : int -> t -> t

val destructor : t -> int

prove: Inv (value)
prove: for all x:int, Inv (constructor x)
prove: for all x:int, for all v:t, if Inv(v) then Inv (transform x v)
assume Inv(t))
```
REPRESENTATION INVARIANTS FOR HIGHER TYPES
What about more complex types?

eg: for abstract type \( t \), consider: \( \text{val op : } t \times t \rightarrow t \text{ option} \)

Basic concept: Assume arguments are “valid”; Prove results “valid”

We know what it means to be a “valid” value \( v \) for abstract type \( t \):

- \( \text{Inv}(v) \) must be true

What is a valid pair? \( v \) is valid for type \( s_1 \times s_2 \) if

- (1) \( \text{fst } v \) is valid for type \( s_1 \), and
- (2) \( \text{snd } v \) is valid for type \( s_2 \)

Equivalently: \( (v_1, v_2) \) is valid for type \( s_1 \times s_2 \) if

- (1) \( v_1 \) is valid for type \( s_1 \), and
- (2) \( v_2 \) is valid for type \( s_2 \)
Representa/on Invariants: More Types

What is a valid pair?  v is valid for type s1 * s2 if

- (1) fst v is valid for s1, and
- (2) snd v is valid for s2

eg: for abstract type t, consider:  val op : t * t -> t

must prove to establish rep invariant:
for all x : t * t,
  if Inv(fst x) and Inv(snd x) then
  Inv (op x)

Equivalent Alternative:
must prove to establish rep invariant:
for all x1:t, x2:t
  if Inv(x1) and Inv(x2) then
  Inv (op (x1, x2))
Another Example:

```haskell
val v : t * (t -> t)
```

must prove both to satisfy the rep invariant:

1. valid (fst v) for type t:
   
   ie: inv (fst v)

2. valid (snd v) for type t -> t:
   
   ie: for all v1:t,
   
   if Inv(v1) then
   
   Inv ((snd v) v1)
What is a valid option? \( v \) is valid for type \( s1 \) option if

- (1) \( v \) is None, or
- (2) \( v \) is Some \( u \), and \( u \) is valid for type \( s1 \)

eg: for abstract type \( t \), consider: val op : \( t \times t \rightarrow t \) option

must prove to satisfy rep invariant:

for all \( x : t \times t \),
if Inv(fst \( x \)) and Inv(snd \( x \))
then
  either:
  (1) \( \text{op} \ x \) is None or
  (2) \( \text{op} \ x \) is Some \( u \) and Inv \( u \)
Suppose we are defining an abstract type \( t \).
Consider happens when the type \( \text{int} \) shows up in a signature.
The type \( \text{int} \) does not involve the abstract type \( t \) at all, in any way.

\[
\text{eg: in our set module, consider: val size : t -> int}
\]

When is a value \( v \) of type \( \text{int} \) valid?

\[
\begin{align*}
\text{all values } v \text{ of type int are valid} \\
\text{val size : } t \rightarrow \text{int} & \quad \text{must prove nothing} \\
\text{val const : int} & \quad \text{must prove nothing} \\
\text{val create : } \text{int} \rightarrow t & \quad \text{for all } v: \text{int}, \\
& \quad \text{assume nothing about } v, \\
& \quad \text{must prove } \text{Inv (create } v \text{)}
\end{align*}
\]
What is a valid function? Value \( f \) is valid for type \( t_1 \rightarrow t_2 \) if

- for all inputs \( arg \) that are valid for type \( t_1 \),
- it is the case that \( f \ arg \) is valid for type \( t_2 \)

**eg:** for abstract type \( t \), consider: \( \text{val op : } t \times t \rightarrow t \text{ option} \)

must prove to satisfy rep invariant:

for all \( x : t \times t \),

if \( \text{Inv}(\text{fst } x) \) and \( \text{Inv}(\text{fst } x) \)

then

either:

(1) \( \text{op } x \) is None or
(2) \( \text{op } x \) is Some \( u \) and \( \text{Inv } u \)

valid for type \( t \times t \)

(the argument)

valid for type \( t \text{ option} \)

(the result)
What is a valid function? Value $f$ is valid for type $t_1 \rightarrow t_2$ if

- for all inputs $\text{arg}$ that are valid for type $t_1$,
- it is the case that $f \text{arg}$ is valid for type $t_2$

eg: for abstract type $t$, consider: \text{val op : (t \rightarrow t) \rightarrow t}

must prove to satisfy rep invariant:

for all $x : t \rightarrow t$,

\[
\begin{align*}
\text{if} \\
\{ \text{for all arguments arg:t,} \\
\text{if Inv(arg) then Inv(x arg)} \} \\
\text{then} \\
\text{Inv (op x)}
\end{align*}
\]

valid for type $t \rightarrow t$ (the argument)

valid for type $t$ (the result)
**Representation Invariants: More Types**

**sig**

```
  type t
  val create : int -> t
  val incr : t -> t
  val apply : t * (t -> t) -> t
  val check_t : t -> t
end
```

**representation invariant:**

```
let inv x = x >= 0
```

**function apply, must prove:**

```
  for all x:t,
  for all f:t -> t
    if x valid for t
    and f valid for t -> t
  then f x valid for t
```

**struct**

```
  type t = int
  let create n = abs n
  let incr n = if n<maxint then n + 1
                else raise Overflow
  let apply (x, f) = f x
  let check_t x = assert (x >= 0); x
end
```

**function apply, must prove:**

```
  for all x:t,
  for all f:t -> t
    if (1) inv(x)
    and (2) for all y:t, if inv(y) then inv(f y)
  then inv(f x)
```

**Proof:** By (1) and (2), `inv(f x)`. 
ANOTHER EXAMPLE
module type NAT =
  sig
    type t
    val from_int : int -> t
    val to_int : t -> int
    val map : (t -> t) -> t -> t list
  end
module type NAT =
  sig
    type t
    val from_int : int -> t
    val to_int : t -> int
    val map : (t -> t) -> t -> t list
  end

module Nat : NAT =
  struct
    type t = int
    let from_int (n:int) : t =
      if n <= 0 then 0 else n
    let to_int (n:t) : int = n
    let rec map f n =
      if n = 0 then []
      else f n :: map f (n-1)
  end
module type NAT = 
  sig 
    type t 
    val from_int : int -> t 
    val to_int : t -> int 
    val map : (t -> t) -> t -> t list 
  end 

module Nat : NAT = 
  struct 
    type t = int 
    let from_int (n:int) : t = 
      if n <= 0 then 0 else n 
    let to_int (n:t) : int = n 
    let rec map f n = 
      if n = 0 then [] 
      else f n :: map f (n-1) 
  end 

let inv n : bool = 
  n >= 0
Look to the signature to figure out what to verify

```ocaml
module type NAT = sig
  type t
  val from_int : int -> t
  val to_int : t -> int
  val map : (t -> t) -> t -> t list
end
```

let inv n : bool = n >= 0

since function result has type t, must prove the output satisfies inv()

can assume inv(x) for all inputs; don't need to prove anything of the outputs with type int

for map f x, assume:
(1) inv(x), and
(2) f’s results satisfy inv() when it’s inputs satisfy inv().

then prove that all elements of the output list satisfy inv()
In general, we use a type-directed proof methodology:

- Let \( t \) be the abstract type and \( \text{inv()} \) the representation invariant.
- For each value \( v \) with type \( s \) in the signature, we must check that \( v \) is valid for type \( s \) as follows:
  - \( v \) is valid for type \( t \) if
    - \( \text{inv}(v) \)
  - \( (v_1, v_2) \) is valid for \( s_1 * s_2 \) if
    - \( v_1 \) is valid for \( s_1 \), and
    - \( v_2 \) is valid for \( s_2 \)
  - \( v \) is valid for type \( s \) \( \text{option} \) if
    - \( v \) is \( \text{None} \) or,
    - \( v \) is \( \text{Some} \) \( u \) and \( u \) is valid for type \( s \)
  - \( v \) is valid for type \( s_1 \rightarrow s_2 \) if
    - for all arguments \( a \), if \( a \) is valid for \( s_1 \), then \( v \ a \) is valid for \( s_2 \)
  - \( v \) is valid for int if
    - always
  - \( [v_1; \ldots; v_n] \) is valid for type \( s \) \( \text{list} \) if
    - \( v_1 \ldots v_n \) are all valid for type \( s \)
module type NAT =
sig
  type t
  val from_int : int -> t
  ...
end

module Nat : NAT =
struct
  type t = int
  let from_int (n:int) : t =
    if n <= 0 then 0 else n
  ...
end

let inv n : bool =
  n >= 0

Must prove:
for all n,
  inv (from_int n) == true

Proof strategy: Split into 2 cases.
(1) n > 0, and (2) n <= 0
Natural Numbers

module type NAT =
  sig
    type t
    val from_int : int -> t
    ...
  end

module Nat : NAT =
  struct
    type t = int
    let from_int (n:int) : t =
      if n <= 0 then 0 else n
    ...
    let inv n : bool =
      n >= 0
    end

Must prove:

for all n,
  inv (from_int n) == true

Case: n > 0

  inv (from_int n)
  == inv (if n <= 0 then 0 else n)
  == inv n
  == true
Natural Numbers

module type NAT =
  sig
    type t
    val from_int : int -> t
    ...
  end

module Nat : NAT =
  struct
    type t = int
    let from_int (n:int) : t =
      if n <= 0 then 0 else n
    ...
    end

Must prove:
for all n,
  inv (from_int n) == true

Case: n <= 0
  inv (from_int n)
  == inv (if n <= 0 then 0 else n)
  == inv 0
  == true
Natural Numbers

module type NAT = 
  sig
    type t
    val to_int : t -> int
    ...
  end

module Nat : NAT =
  struct
    type t = int
    let to_int (n:t) : int = n
    ...
  end

let inv n : bool = n >= 0

Must prove:

for all n,
  if inv n then
  we must show ... nothing ...
  since the output type is int
Natural Numbers

module type NAT = sig
    type t
    val map : (t -> t) -> t -> t list
    ...
end

module Nat : NAT = struct
    type t = int
    let rep map f n =
        if n = 0 then []
        else f n :: map f (n-1)
    ...
end

let inv n : bool = n >= 0

Must prove:
for all f valid for type t -> t
for all n valid for type t
map f n is valid for type t list

Proof: By induction on n.
module type NAT =
  sig
    type t
    val map : (t -> t) -> t -> t list
    ...
  end

module Nat : NAT =
  struct
    type t = int
    let rep map f n =
      if n = 0 then []
      else f n :: map f (n-1)
    ...
  end

Must prove:

for all f valid for type t -> t
for all n valid for type t
map f n is valid for type t list

Proof: By induction on nat n.

Case: n = 0
map f n  == []
(Note: each value v in [ ] satisfies inv(v))
Natural Numbers

module type NAT =
  sig
    type t
    val map : (t -> t) -> t -> t list
  end

module Nat : NAT =
  struct
    type t = int
    let rep map f n =
      if n = 0 then []
      else f n :: map f (n-1)
  end

Must prove:
for all f valid for type t -> t
for all n valid for type t
map f n is valid for type t list

Proof: By induction on nat n.

Case: n > 0
map f n == f n :: map f (n-1)
module type NAT =
  sig
    type t
    val map : (t -> t) -> t -> t list
  end

module Nat : NAT =
  struct
    type t = int
    let rep map f n =
      if n = 0 then []
      else f n :: map f (n-1)
    ...
  end

Must prove:
for all f valid for type t -> t
for all n valid for type t
map f n is valid for type t list

Proof: By induction on nat n.

Case: n > 0
map f n  ==  f n :: map f (n-1)

By IH, map f (n-1) is valid for t list.
Natural Numbers

module type NAT =
  sig
    type t
    val map : (t -> t) -> t -> t list
  end

module Nat : NAT =
  struct
    type t = int
    let rep map f n =
      if n = 0 then []
      else f n :: map f (n-1)
    ...
  end

Must prove:
for all f valid for type t -> t
for all n valid for type t
map f n is valid for type t list

Proof: By induction on nat n.

Case: n > 0
map f n == f n :: map f (n-1)

By IH, map f (n-1) is valid for t list.
Since f valid for t -> t and n valid for t
f n::map f (n-1) is valid for t list

let inv n : bool = n >= 0
module type NAT = 
sig
   type t
   val map : (t -> t) -> t -> t list
   ...
end

module Nat : NAT = 
struct
   type t = int

   let rep map f n = 
     if n = 0 then []
     else f n :: map f (n-1)

   ...
end

End result: We have proved a strong property \((n \geq 0)\) of every value with abstract type Nat.t

Hooray! \(n\) is never negative so we don’t infinite loop
Summary for Representation Invariants

• The signature of the module tells you what to prove

• Roughly speaking:
  – assume invariant holds on values with abstract type on the way in
  – prove invariant holds on values with abstract type on the way out
ABSTRACTION FUNCTIONS
Abstraction

- When explaining our modules to clients, we would like to explain them in terms of abstract values
  - sets, not the lists (or maybe trees) that implement them
- From a client’s perspective, operations act on abstract values
- Signature comments, specifications, preconditions and post-conditions in terms of those abstract values
- *How are these abstract values connected to the implementation?*

```ml
module type SET =
  sig
    type ‘a set
    val empty : ‘a set
    val mem : ‘a -> ‘a set -> bool
  end
```
Abstraction

user’s view:
sets of integers
{1, 2, 3} {4, 5}
{}{}

implementation view:
lists of integers
[1; 1; 2; 3; 2; 3] [1; 2; 3] [ ] [4, 5] [4, 5, 5] [5, 4]
Abstraction

user’s view:

sets of integers

\{1, 2, 3\} \rightarrow \{4, 5\}

\{\}\rightarrow \{\}

implementation view:

lists of integers

\[1; 1; 2; 3; 2; 3\] \rightarrow \[4, 5, 5\]

\[1; 2; 3\] \rightarrow \[4, 5\]

\[5, 4\] \rightarrow \[\]

there’s a relationship here, of course!

we are trying to implement the abstraction
user’s view:

sets of integers

\{1, 2, 3\}  \{4, 5\}  \{\} 

implementation view:

lists of integers

[1; 1; 2; 3; 2; 3]  [1; 2; 3]  [4, 5]  [4, 5, 5]  [5, 4] 

this relationship is a function: it converts concrete values to abstract ones

function called “the abstraction function”
Abstraction

user’s view:

sets of integers

\{1, 2, 3\} \quad \{4, 5\}

\{\}\n
implementation view:

lists of integers

\[1; 1; 2; 3; 2; 3\] \quad \[\\] \quad \[4, 5\] \quad \[4, 5, 5\]

\[1; 2; 3\]

inv(x): no duplicates

Representation Invariant cuts down the domain of the abstraction function
Specifications

user’s view:

{1, 2} \[\rightarrow\] add 3 \[\rightarrow\] {1, 2, 3}

implementation view:

a specification tells us what operations on abstract values do
Specifications

User’s view:

\{1, 2\} \xrightarrow{\text{add 3}} \{1, 2, 3\}

Implementation view:

\[1; 2\]

inv(x)

A specification tells us what operations on abstract values do.
a specification tells us what operations on abstract values do.
Specifications

User’s view:
- \{1, 2\} → add 3 → \{1, 2, 3\}

Implementation view:
- \[1; 2\] → add 3 → \[3; 1; 2\]

In general: related arguments are mapped to related results.

A specification tells us what operations on abstract values do.

inv(x)
User’s view:

{1, 2} \rightarrow \text{add 3} \rightarrow \{1, 2, 3\} \neq \{3; 1\}

Implementation view:

[1; 2] \rightarrow \text{add 3} \rightarrow [3; 1; 3]

Bug! Implementation does not correspond to the correct abstract value!
Specifications

user’s view:

implementation view:

add 3

{1, 2} → {1, 2, 3}

[1; 2] → [3; 1; 2]

[2; 1] → [3; 2; 1]

inv(x)

implementation must correspond no matter which concrete value you start with
A more general view

To prove:
for all \( c1 : t \), if \( \text{inv}(c1) \) then \( \text{f_abs} (\text{abs} \; c1) = \text{abs} (\text{f_con} \; c1) \)

abstract then apply the abstract op == apply concrete op then abstract
Another Viewpoint

A specification is really just another implementation (in this viewpoint) – but it’s often simpler ("more abstract")

We can use similar ideas to compare *any two implementations of the same signature*. Just come up with a *relation between corresponding values of abstract type*.

We ask: Do operations like $f$ take related arguments to related results?
What is a specification?

It is really just another implementation
— but it’s often simpler (“more abstract”)

We can use similar ideas to compare any two implementations of the same signature. Just come up with a relation between corresponding values of abstract type.
Consider a client that might use the module:

\[
\begin{align*}
&\text{let } x_1 = \text{M1.bump (M1.bump (M1.zero))} \\
&\text{let } x_2 = \text{M2.bump (M2.bump (M2.zero))}
\end{align*}
\]

What is the relationship?

\[
\text{is\_related}(x_1, x_2) = x_1 \equiv x_2/2 - 1
\]

And it persists: Any sequence of operations produces related results from M1 and M2!

How do we prove it?
Recall: A representation invariant is a property that holds for all values of abs. type:
• if \( M.v \) has abstract type \( t \),
  • we want \( \text{inv}(M.v) \) to be true

Inter-module relations are a lot like representation invariants!
• if \( M1.v \) and \( M2.v \) have abstract type \( t \),
  • we want \( \text{is\_related}(M1.v, M2.v) \) to be true

It’s just a relation between two modules instead of one
module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
    val reveal : t -> int
  end

module M1 : S =
  struct
    type t = int
    let zero = 0
    let bump n = n + 1
    let reveal n = n
  end

module M2 : S =
  struct
    type t = int
    let zero = 2
    let bump n = n + 2
    let reveal n = n/2 - 1
  end

Recall: To prove a rep. inv., assume it holds on inputs & prove it holds on outputs:
  • if M.f has type t -> t, we prove that:
    • if inv(v) then inv(M.f v)

Likewise for inter-module relations:
  • if M1.f and M2.f have type t -> t, we prove that:
    • if is_related(v1, v2) then
    • is_related(M1.f v1, M2.f v2)

related functions produce related results from related arguments
Consider zero, which has abstract type t.

Must prove: is_related (M1.zero, M2.zero)

Equivalent to proving: M1.zero == M2.zero/2 – 1

Proof:
M1.zero
== 0 (substitution)
== 2/2 – 1 (math)
== M2.zero/2 – 1 (substitution)
Consider bump, which has abstract type \( t \rightarrow t \).

Must prove for all \( v1:\text{int}, v2:\text{int} \)
if \( \text{is\_related}(v1,v2) \) then \( \text{is\_related}(\text{M1.bump} v1, \text{M2.bump} v2) \)

Proof:
1. Assume \( \text{is\_related}(v1, v2) \).
2. \( v1 = v2/2 - 1 \) (by def)

Next, prove:
\( \text{is\_related}(\text{M1.bump} v1, \text{M2.bump} v2) \)

\begin{align*}
\text{is\_related}(x1, x2) &= x1 == x2/2 - 1 \\
\text{M2.bump v2)/2 - 1} &= (v2 + 2)/2 - 1 \\
&= (v2/2 - 1) + 1 \\
&= v1 + 1 \\
&= \text{M1.bump v1}
\end{align*}
Consider reveal, which has abstract type \( t \rightarrow \text{int} \).

Must prove for all \( v_1: \text{int}, v_2: \text{int} \)
if \( \text{is\_related}(v_1, v_2) \) then \( \text{M1\_reveal} \ v_1 = \text{M2\_reveal} \ v_2 \)

Proof:
(1) Assume \( \text{is\_related}(v_1, v_2) \).
(2) \( v_1 = v_2/2 - 1 \) (by def)

Next, prove:
(\( \text{M2\_reveal} \ v_2 = \text{M1\_reveal} \ v_1 \) (eval, reverse)
Summary of Proof Technique

To prove M1 == M2 relative to signature S,

- Start by defining a relation “is_related”:
  - is_related (v1, v2) should hold for values with abstract type t when v1 comes from module M1 and v2 comes from module M2

- Extend “is_related” to types other than just abstract t. For example:
  - if v1, v2 have type int, then they must be exactly the same
    - ie, we must prove: v1 == v2
  - if v1, v2 have type s1 -> s2 then we consider arg1, arg2 such that:
    - if is_related(arg1, arg2) then we prove
    - is_related(v1 arg1, v2 arg2)
  - if v1, v2 have type s option then we must prove:
    - v1 == None and v2 == None, or
    - v1 == Some u1 and v2 == Some u2 and is_related(u1, u2) at type s

- For each val v:s in S, prove is_related(M1.v, M2.v) at type s