Modules and Representation Invariants

COS 326
Andrew Appel
Princeton University
In COS 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*:

- eg: keys are in order in the tree

Operations such as look-up, *depend upon* those invariants to be correct. *All inputs to look-up must satisfy the in-order invariant.*
Efficient Data Structures

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

**Key Question:** How do you arrange for that to happen when client code is using your interface & calling your functions?

**Answer:** Use abstract types & representation invariants.
REPRESENTATION INVARIANTS
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 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) 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:

```ocaml
(* 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
```
Debugging with Representation Invariants

As a precondition on input sets:

```ocaml
(* 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:

```ocaml
(* 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?
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 \( \text{inv}(i) \) for module inputs \( i \) with abstract type

provided you

prove the invariant \( \text{inv}(o) \) 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 representation 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
- easier to check, even informally, that each function and value you write satisfies the invariants once you have written them
- great documentation for others
- great debugging tool if you implement your invariant
- you’ll need them to prove to yourself that your code is correct
PROVING THE REP INVARIANT FOR THE SET ADT
Representation Invariants

Representation Invariant for sets without duplicates:

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

Definition of empty:

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

Proof Obligation:

```ml
inv (empty) == true
```

Proof:

```ml
inv (empty)
== inv []
== match [] with [] -> true | hd::tail -> ...
== true
```
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 add:

```
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 \( \text{inv}(l) \) then \( \text{inv}(\text{add } x \ l) \)
Lots of theorems (like the one we just saw) have the form:

\[ \text{forall } x: t. \ P(x) \]

To prove such theorems, we often pick an arbitrary representative \( r \) of the type \( t \) and then prove \( P(r) \) is true.

(Often times we just use “\( x \)” as the name of the representative. This just helps prevent a proliferation of names.)

If we can’t do the proof by picking an arbitrary representative, we may want to split values of type \( t \) into cases or use induction.
Aside: Conditional Theorems

Lots of theorems (also like the one we just saw) have the form:

if $P(x)$ then $Q(y)$

To prove such theorems, we typically assume $P(x)$ is true and then under that assumption, prove $Q(y)$ is true.
Aside: Conditional Theorems

Lots of theorems (also like the one we just saw) have the form:

\[
\text{if } P(x) \text{ then } Q(y)
\]

To prove such theorems, we typically assume $P(x)$ is true and then under that assumption, prove $Q(y)$ is true.

Such conditionals are actually logical implications:

\[
P(x) \implies Q(y)
\]
Aside: Conditional Theorems

Putting ideas together, proving:

for all x:t, y:t', if P(x) then Q(y)

will involve:
(1) picking arbitrary x:t, y:t'
(2) assuming P(x) is true and then using that assumption to
(3) prove Q(y) is true.
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 \( \text{inv}(l) \).

Break into two cases:

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

Proof:

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

**case 1:** assume (3): $\text{mem } x \text{ } l \Rightarrow \text{true}$:

$\text{inv}(\text{add } x \text{ } l)$

$= \text{inv}(\text{if } \text{mem } x \text{ } l \text{ then } l \text{ else } x::l)$ (eval)

$= \text{inv}(l)$ (by (3), eval)

$= \text{true}$ (by (2))
Representation Invariants

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

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

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 (mem x l) == true:

\[
\begin{align*}
\text{inv}(\text{add} \ x \ l) &= \text{inv}(\text{if mem x l then l else x::l}) \quad \text{(eval)} \\
&= \text{inv}(x::l) \quad \text{(by (3))} \\
&= \text{not (mem x l) && inv (l)} \quad \text{(by eval)} \\
&= \text{true && inv(l)} \quad \text{(by (3))} \\
&= \text{true && true} \quad \text{(by (2))} \\
&= \text{true} \quad \text{(eval)}
\end{align*}
\]
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 \( \text{inv}(l) \) then \( \text{inv}(\text{rem } x \ l) \)

Assume invariant on input

Prove invariant on output
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 size:

```ocaml
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:

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

Checking union:

```ml
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:

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

Checking inter:

```plaintext
let inter (l1:'a set) (l2:'a set) : 'a set =
  ...
```

Proof obligation?

for all \( l1: 'a \) set and for all \( l2: 'a \) set, if \( \text{inv}(l1) \) and \( \text{inv}(l2) \) then \( \text{inv} \) (inter \( l1 \) \( l2 \))
Representation Invariants: a Few Types

Given a module with abstract type t

Define an invariant \( \text{Inv}(x) \)

Assume arguments to functions satisfy \( \text{Inv} \)

Prove results from functions satisfy \( \text{Inv} \)

```
sig
  type t

  val value : t
  val constructor : int -> t
  val transform : int -> t -> t
  val destructor : t -> int
end
```

prove: \( \text{Inv}(value) \)

prove: for all \( x: \text{int} \), \( \text{Inv}(\text{constructor } x) \)

prove: for all \( x: \text{int} \), for all \( v: \text{t} \), if \( \text{Inv}(v) \) then \( \text{Inv}(\text{transform } x v) \)

assume \( \text{Inv}(t) \)
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” and prove results “valid”
• What it means to be “valid” depends on the \textit{type} of the value
What about more complex types?

eg: for abstract type \( t \), consider: 

   val op : \( t \times t \rightarrow t \) option

Basic concept:

- Assume arguments are “valid” and prove results “valid”
- What it means to be “valid” depends on the \textit{type} of the value
- We are going to decide whether “\( x \) is valid for type \( s \)”
What about more complex types?

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

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

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

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

- must prove to establish rep invariant:
  - for all \( x : t \times t \),
    - if \( \text{Inv(fst x)} \) and \( \text{Inv(snd x)} \) then
      - \( \text{Inv(op x)} \)

**Equivalent Alternative:**

- must prove to establish rep invariant:
  - for all \( x_1 : t, x_2 : t \)
    - if \( \text{Inv(x1)} \) and \( \text{Inv(x2)} \) then
      - \( \text{Inv(op (x1, x2))} \)
What is a valid option? \( v \) is valid for type \( s_1 \ \text{option} \) if

1. \( v \) is \textbf{None}, or
2. \( v \) is \textbf{Some} \( u \), and \( u \) is valid for type \( s_1 \)

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

\text{must prove to satisfy rep invariant: for all} \ x : t \times t,
\begin{align*}
\text{if Inv(fst } x) \ \text{and} \ \text{Inv(snd } x) \\
\text{then}
\text{either:}
\begin{align*}
\text{(1) } \text{op } x \text{ is None or} \\
\text{(2) } \text{op } x \text{ is Some } u \ \text{and Inv } u
\end{align*}
\end{align*}
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.

eg: in our set module, consider: \( \text{val size : } t \to \text{int} \)

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

all values \( v \) of type \( \text{int} \) are valid

\( \text{val size : } t \to \text{int} \)

must prove nothing

\( \text{val const : int} \)

must prove nothing

\( \text{val create : int} \to t \)

for all \( v : \text{int} \), assume nothing about \( v \),
must prove \( \text{Inv (create} v \text{)} \)
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 $t_2$

**Note:** We’ve been using this idea all along for all operations!

**eg:** for abstract type $t$, consider: val $\text{op} : t \ast t \rightarrow t \text{ option}$

must prove to satisfy rep invariant:

for all $x : t \ast t$,

- if $\text{Inv}(\text{fst } x)$ and $\text{Inv}(\text{fst } x)$
- then
- either:
  - (1) $\text{op } x == \text{None}$
  - (2) $\text{op } x == \text{Some } u$ and $\text{Inv } u$

valid for type $t \ast t$ (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: val $\text{op} : (t \rightarrow t) \rightarrow t$

must prove to satisfy rep invariant:

\[
\text{for all } x : t \rightarrow t, \\
\text{if} \\
\{ \text{for all arguments } \text{arg}:t, \\
\text{if } \text{Inv}(\text{arg}) \text{ then } \text{Inv}(x \ \text{arg}) \} \\
\text{then} \\
\text{Inv} (\text{op } x)
\]
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
```

```
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
```

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

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
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

Look to the signature to figure out what to verify

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 \( t \) if
    - \( \text{inv}(v) \)
  - \( (v_1, v_2) \) is valid for \( s_1 \times s_2 \) if
    - \( v_1 \) is valid for \( s_1 \), and
    - \( v_2 \) is valid for \( s_2 \)
  - \( v \) is valid for type \( s \) option if
    - \( v \) is None or,
    - \( v \) is 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 \( s_2 \)
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
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

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

  let inv n : bool =
    n >= 0

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
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
Module type \texttt{NAT} = 

\begin{verbatim}
  sig

  type t

  val map : (t -> t) -> t -> t list

  ...

  end
\end{verbatim}

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

let inv n : bool = n >= 0

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

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.
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.
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

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

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.
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 >= 0) of every value with abstract type Nat.t

Hooray! n is never negative so we don’t infinite loop
One More example

module type NAT =
  sig
    type t
    val from_int : int -> t
    val to_int : t -> int
    val map : (t -> t) -> t -> t list
    val foo : (t -> t) -> t
  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)
    let foo f = f (-1)
  end

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

module Nat : NAT =
  struct
    ...
    let foo f = f (-1)
  end

let inv n : bool =
  n >= 0

Must prove:
for all f valid for type t -> t
foo f is valid for type t

Proof?
Consider any f valid for type t -> t
for all arguments v, if inv (v) then inv (f v).
What can we prove about f (-1)?
module type NAT =
  sig
      type t
      val from_int : int -> t
      val to_int : t -> int
      val map : (t -> t) -> t -> t list
      val foo : (t -> t) -> t
  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)
      let foo f = f (-1)
  end

let inv n :
  n >= 0

challenge:
create a program that
loops forever
• The signature of the module tells you what to prove

• Roughly speaking:
  – assume invariant holds on values with abstract type \textit{on the way in}
  – prove invariant holds on values with abstract type \textit{on the way out}