Modules and Representation Invariants

COS 326

David Walker
Princeton University

Last Time

Introduction to OCaml mechanisms for defining modules:

– *signatures* (interfaces)
– *structures* (implementations)
– *functors* (functions from modules to modules)

Uses the module system

– provides support for *name-spaces*
– provides support for *hiding information*
  • hide type & value definitions
  • helps to isolate changes to small parts of an application
– provides support for *code reuse*
  • interfaces reuseable on multiple modules
  • modules reuseable with multiple interfaces
  • functors: parameterized modules; body reused with many arguments

An 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
```

Implementing Sets

```
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
```

Implementing Sets

```
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
```

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

Implementing Sets

```
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
```

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

Implementing Sets

```
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
```

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

Implementing Sets

```
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.

Implementing Sets

```
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.

Implementing Sets

```
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.

Implementing Sets

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

OCAML COLLECTION LIBRARIES

Collection Data Types

- Many different kinds of collections:
  - Sets
  - Maps (Symbol Tables)
  - Queues
  - Graphs
  - ...
- Modules with two abstract types:
  - element type
  - collection type
- We often build collections using functors

A Common Structure for Collection Modules

```
module MyCollection =
 struct

  module type Element =
    sig  type t  ... end

  module type S =
    sig ... end

  module Make (Arg : Element) : S with type t = Arg.t =
    struct ... end

 end
```

Type and necessary operations on collection element

Type and necessary operations of collection as a whole

Functor to create the collection

Modules in the Wild

- The OCaml Map Module
  - creates polymorphic symbol tables

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Map.html

REASONING ABOUT MODULES

Back to Sets (with a cut-down signature)

```
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 -> bool
    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 = List.length
    (* 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:

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

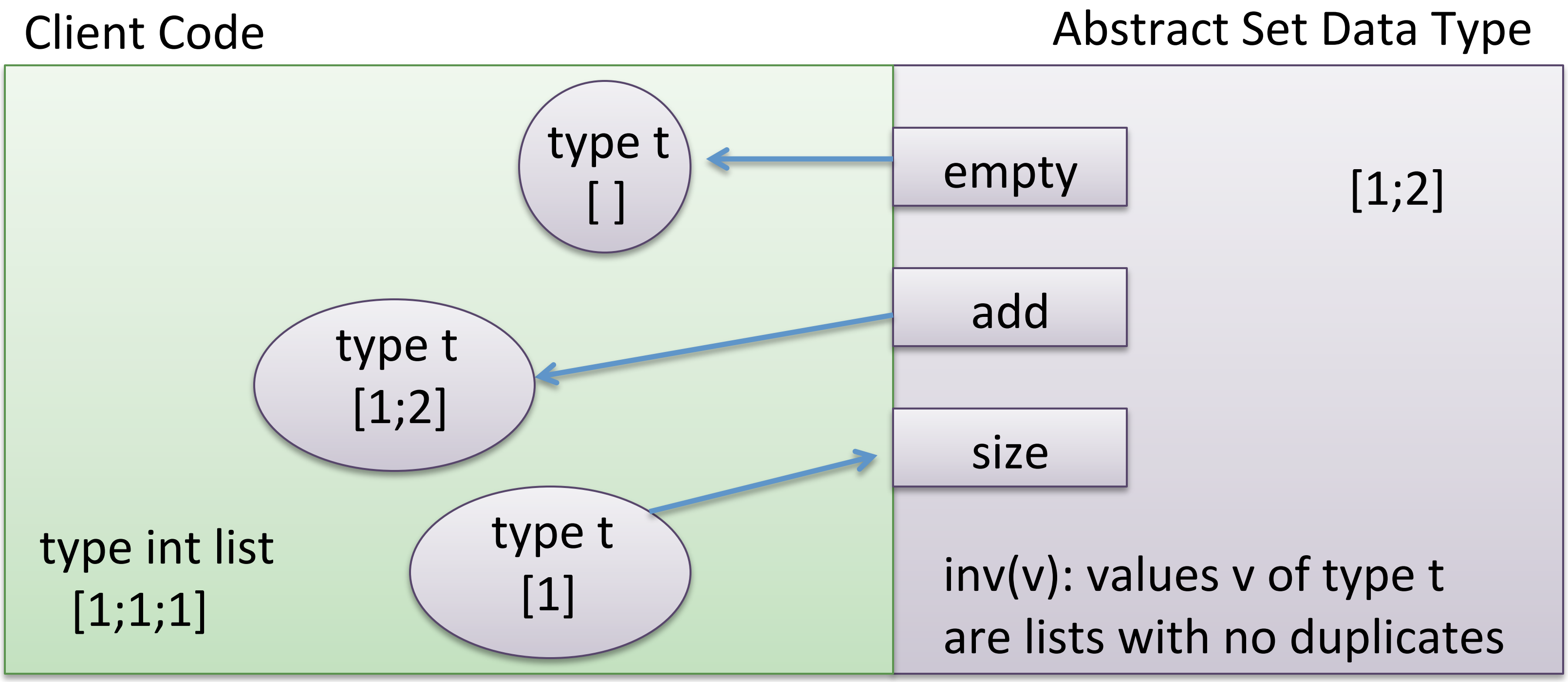
Why does this work? It depends on an invariant:

All lists supplied as an argument contain no duplicates.

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.

Representation Invariants

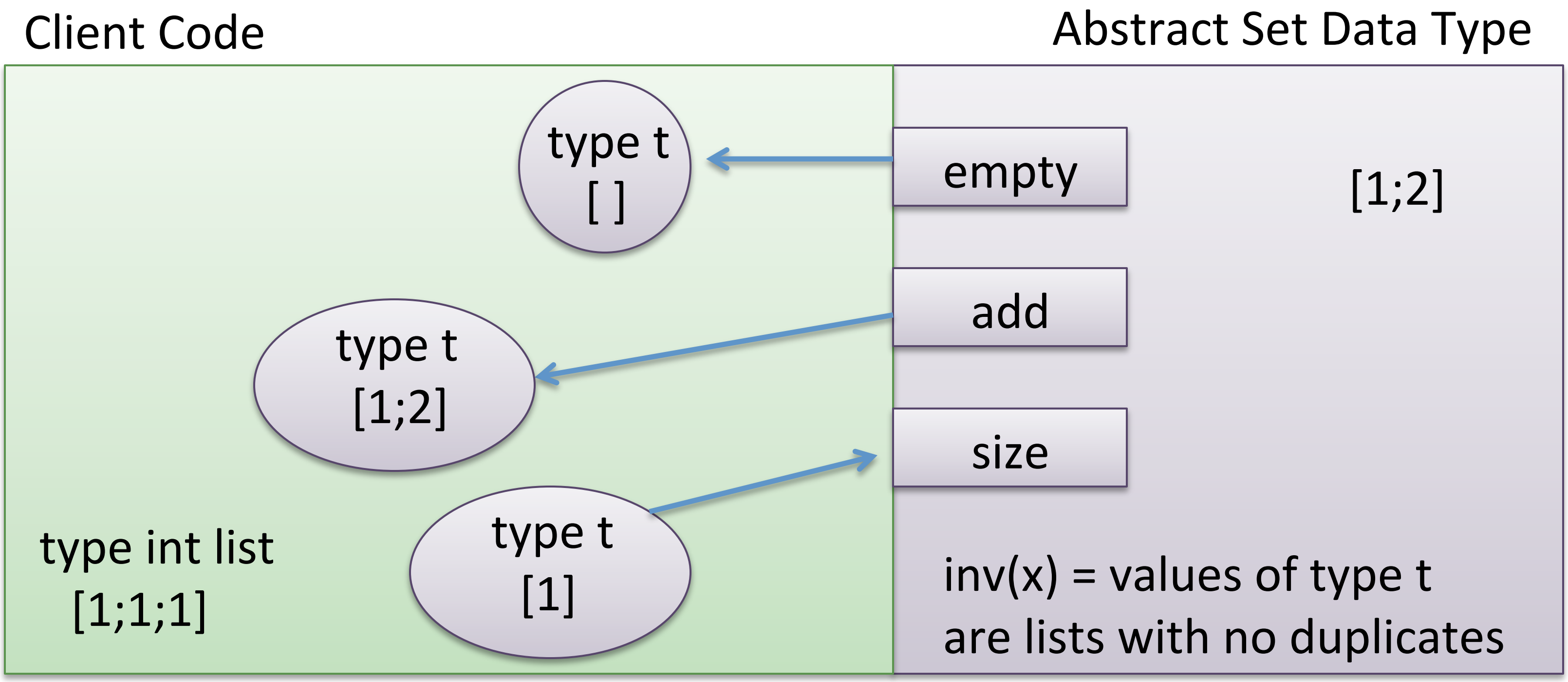
A *representation invariant inv(v) for abstract type t* is a property of all data values v with abstract data type t



- Invariants on abstract types are *local* to the ADT. 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.

Representation Invariants

A *representation invariant 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* provided it *proves the invariant for outputs*
- These proofs are *modular*: Done in isolation in the ADT module

Establishing Representation Invariants

Eg, when it comes to the size function:

```
(* 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 inv depend on assumptions that function inputs satisfy inv
  - add, rem, union, intersect

PROVING THE REP INVARIANT FOR THE SET ADT

Representation Invariants

Representation Invariant for sets without duplicates:

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

Definition of empty:

```
let empty : ‘a set = []
```

Proof Obligation:

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

Proof:

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

Representation Invariants

Representation Invariant for sets without duplicates:

```
let rec inv (l : 'a set) : 'a set =
   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 inv(l) then inv (add x l) ← prove invariant on output

(inv(l) ← assume invariant on input)

Representation Invariants

```
let rec inv (l : 'a set) : 'a set =
   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 inv(l) then inv (add x l)

Proof:

(1) pick an arbitrary x and l. (2) assume inv(l).

Break in to two cases:

-- one case when mem x l is true

-- one case where mem x l is false

Representation Invariants

```
let rec inv (l : 'a set) : 'a set =
  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 inv(l) then inv (add x l)

Proof:

(1) pick an arbitrary x and l. (2) assume inv(l).

<u>case 1:</u> assume (3): mem x l == true:

inv (add x l)	
== inv (if mem x l then l else x::l)	(eval)
== inv (l)	(by (3))
== true	(by (2))

Representation Invariants

```
let rec inv (l : 'a set) : 'a set =
  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 inv(l) then inv (add x l)

Proof:

(1) pick an arbitrary x and l. (2) assume inv(l).

case 2: assume (3) not (mem x l) == true:

inv (add x l)	
== inv (if mem x l then l else x::l)	(eval)
== inv (x::l)	(by (3))
== not (mem x l) && inv (l)	(by eval)
== true && inv(l)	(by (3))
== true && true	(by (2))
== true	(eval)

Representation Invariants

Representation Invariant for sets without duplicates:

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

Checking rem:

```
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) : 'a set =
   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:

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

Checking union:

```
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:

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

Checking inter:

```
let inter (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 (inter l1 l2)

assume invariant on input

prove invariant on output

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

```
sig
  type t

  val value : t

  val constructor : int -> t

  val transform : int -> t -> t

  val destructor : t -> int

end
```

prove: Inv (value)

prove: for all x:int, Inv (constructor x)

prove:
for all x:int,
for all v:t,
if Inv(t)
then Inv (constructor x)

assume Inv(t))

REPRESENTATION INVARIANTS FOR HIGHER TYPES

Representation Invariants: More Types

What about more complex types?

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

- Inv(v) must be true

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

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

Equivalently: (v1, v2) is valid for type s1 * s2 if

- (1) v1 is valid for type s1, and
- (2) v2 is valid for type s2

Representation 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))

Representation Invariants: More Types

Another Example:

```
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)

Representation Invariants: More Types

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 * t -> t option

must prove to satisfy rep invariant:
for all x : t * t,
if Inv(fst x) and Inv(fst x)
then
either:
(1) op x is None or
(2) op x is Some u and Inv u

Representation Invariants: More Types

Suppose we are defining an abstract type t.

Consider happens when the type int to show up in a signature.

The type int does not involve the abstract type t at all, in any way.

eg: in our set module, consider: val size : t -> int

When is a value v of type int valid?

all values v of type int are valid

val size : t -> int — must prove nothing

val const : int — must prove nothing

val create : int -> t — for all v:int,
  assume nothing about v,
  must prove Inv (create v)

Representation Invariants: More Types

What is a valid function? Value f is valid for type t1 -> t2 if

- for all inputs arg that are valid for type t1,
- it is the case that f arg is valid for type t2

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

must prove to satisfy rep invariant:
for all x : t * t,
if Inv(fst x) and Inv(fst x)
then
either:
(1) op x is None or
(2) op x is Some u and Inv u

valid for type t * t
(the argument)

valid for type t option
(the result)

Representation Invariants: More Types

What is a valid function? Value f is valid for type t1 -> t2 if

- for all inputs arg that are valid for type t1,
- it is the case that f arg is valid for type t2

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

```
must prove to satisfy rep invariant:
 for all x : t -> t,
    if
      for all arguments arg:t,
         if Inv(arg) then Inv(x arg)
    then
       Inv (op x)
```

valid for type t -> 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
```

```
struct
  type t = int
  let create n = abs n
  let incr n = n + 1
  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

function apply, must prove:
- for all x:t,
- for all f:t -> t
  - if (1) inv(t)
  - 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)

Debugging with Representation Invariants

```
struct
  type t = int



  let create n = abs n
  let incr n = n + 1
  let apply (x, f) = f x
end
```

→

```
struct
  type t = int
  let inv x : bool = x >= 0
  let check_t x = assert (inv x); x

  let create n = check(abs n)
  let incr n = check ((check n) + 1)
  let apply (x, f) = check (f (check x))
end
```

check output produced

check input assumption

- It's good practice to implement your representation invariants
- Use them to check your assumptions about inputs
  - find bugs in other functions
- Use them to check your outputs
  - find bugs in your function

Representation Invariants Proof Summary

If a module M defines an abstract type t

- Think of a representation invariant inv(x) for values of type t
- Prove each value of type s provided by M is *valid for type s* relative to the representation invariant

If v : s then prove v is valid for type s as follows:

- if s is the abstract type t then prove inv(v)
- if s is a base type like int then v is always valid
- if s is s1 * s2 then prove:
  - fst v is valid for type s1
  - snd v is valid for type s2
- if s is s1 option then prove:
  - v is None, or
  - v is Some u and u is valid for type s1
- if s is s1 -> s2 then prove:
  - for all x:s1, if x is valid for type s1 then v x is valid for type s2

Aside: This kind of proof is known as a proof using *logical relations*. It lifts a property on a basic type like inv( ) to a property on higher types like t1 * t2 and t1 -> t2

Summary

- Representation invariants define the valid implementations of an abstract data type
- Assume the invariant on all inputs; prove it on all outputs
  - to debug, implement the invariant function
    - apply it on inputs and outputs with abstract type to check for errors
- "Lift" the invariant to higher types like function types, pairs, options, lists and data types in a mechanical way
  - the type tells you what to verify
    - eg: for pairs, verify both components
    - eg: for functions, assume verified inputs & use that to verify outputs
  - technically, we have defined a "logical relation"

END