Modules and Representation Invariants

COS 326
David Walker
Princeton University

slides copyright 2018 David Walker
permission granted to reuse these slides for non-commercial educational purposes

Efficient Data Structures

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

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 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:

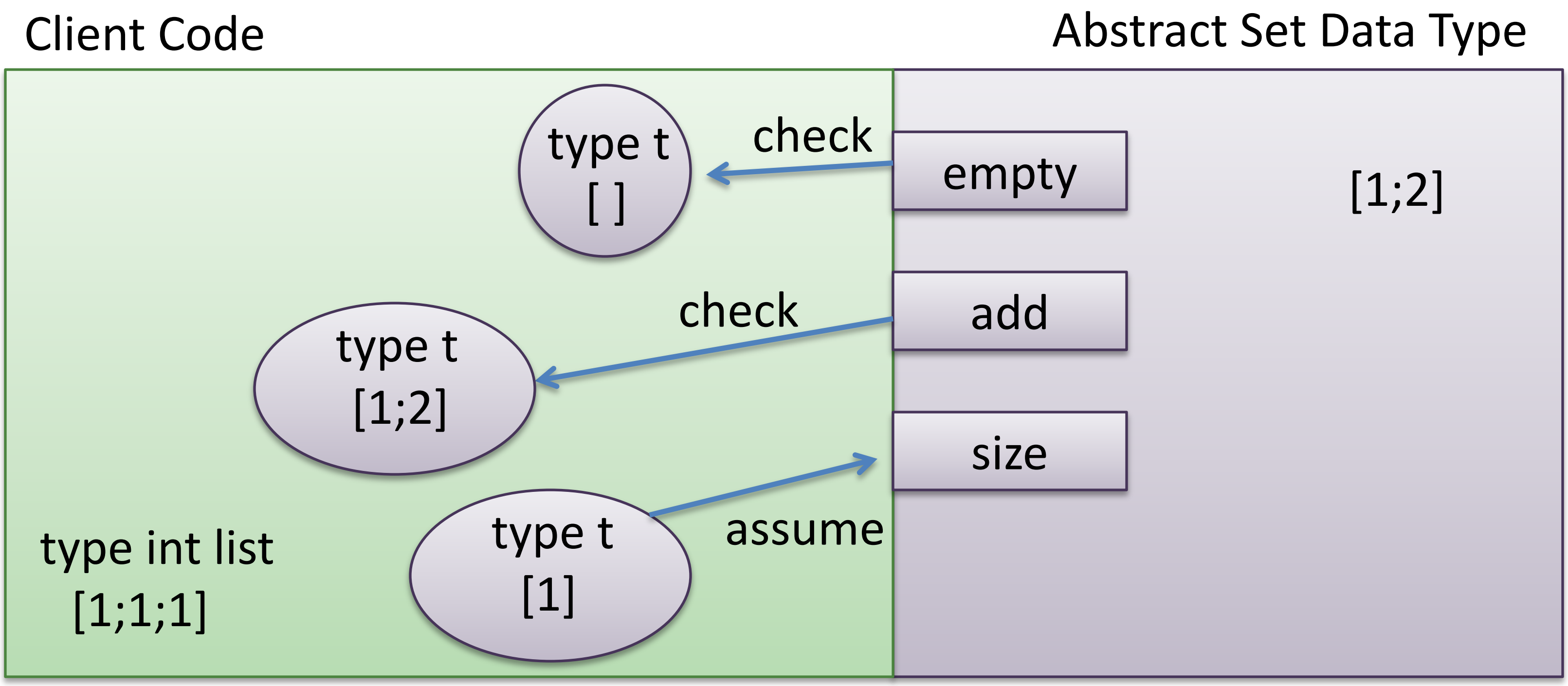
```
(* add x to set s *)
let add x s =
  let s = if mem x s then s else x::s in
  check s "add: bad set output"
```

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
```

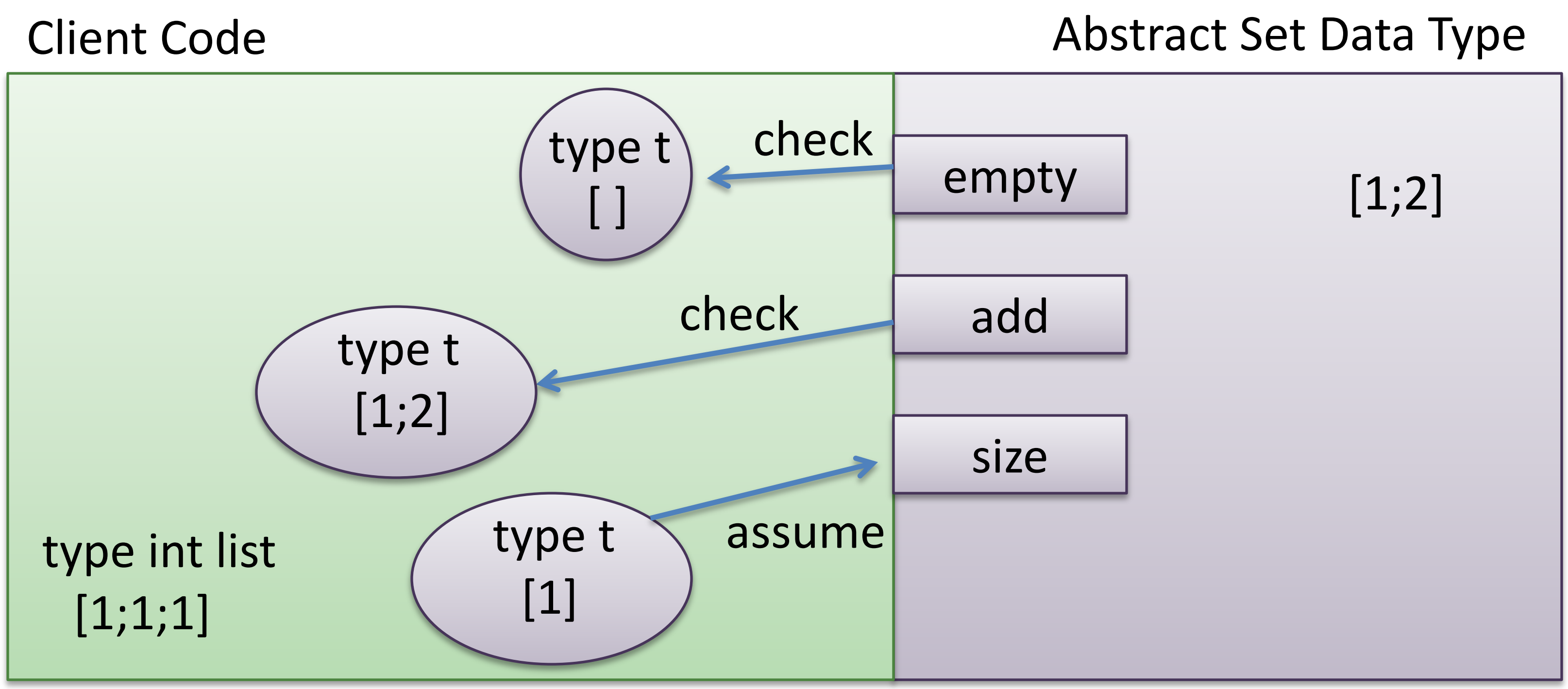
Suppose we check all the red values satisfy our invariant leaving the module, do we have to check the blue values entering the module satisfy our invariant?

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 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:

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

(inv(l) ← assume invariant on input)

Aside: Universal Theorems

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

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:

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.

Such conditionals are actually logical implications:

P(x) ==> 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.

Representation Invariants

```
let rec inv (l : 'a
    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 into 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
    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), eval)
== 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 inv(l) then inv (add x l)

Proof:

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

<u>case 2:</u> 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) : bool =
   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) : 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:

```
let rec inv (l : 'a set) : bool =
   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) : bool =
   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(v)
      then Inv (transform x v)

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” and prove results “valid”
- What it means to be “valid” depends on the *type* of the value

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" and prove results "valid"
- What it means to be "valid" depends on the *type* of the value
- We are going to decide whether "x is valid for type s"

"valid for type t"

What about more complex types?

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

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

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(snd 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 shows 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

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

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 == None or
        (2) op x == 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 = 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

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

Natural Numbers

```
module type NAT =
  sig

    type t

    val from_int : int -> t

    val to_int : t -> int

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

  end
```

Natural Numbers

```
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
```

Natural Numbers

```
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
```

```
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
```

Look to the signature to figure out what to verify

```
module type NAT =
 sig

  type t

  val from_int : int -> t

  val to_int : t -> int

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

 end
```

```
type t = int
```

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()

```
let inv n : bool =
  n >= 0
```

Verifying The Invariant

In general, we use a type-directed proof methodology:

- Let t be the abstract type and 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
    - inv(v)
  - (v1, v2) is valid for s1 * s2 if
    - v1 is valid for s1, and
    - v2 is valid for s2
  - 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 s1 -> s2 if
    - for all arguments a, if a is valid for s1, then v a is valid for s2
  - v is valid for int if
    - always
  - [v1; ...; vn] is valid for type s list if
    - v1 ... vn are all valid for type s

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
```

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

    ...

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
```

Must prove:

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

```
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
```

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.

Natural Numbers

```
module type NAT =
 sig

  type t

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

  ...

 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.

```
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))

Natural Numbers

```
module type NAT =
 sig

   type t

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

   ...

 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.

```
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)

Natural Numbers

```
module type NAT =
 sig

   type t

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

   ...


 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.

```
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.

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 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

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
```

Hooray! n is never negative so we don't infinite loop

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

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
```

```
let inv n : bool =
  n >= 0
```

```
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
```

One More example

```
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 n 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) ?

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
```

```
let inv n :
  n >= 0
```

challenge:
create a program that
loops forever

```
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
```

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*

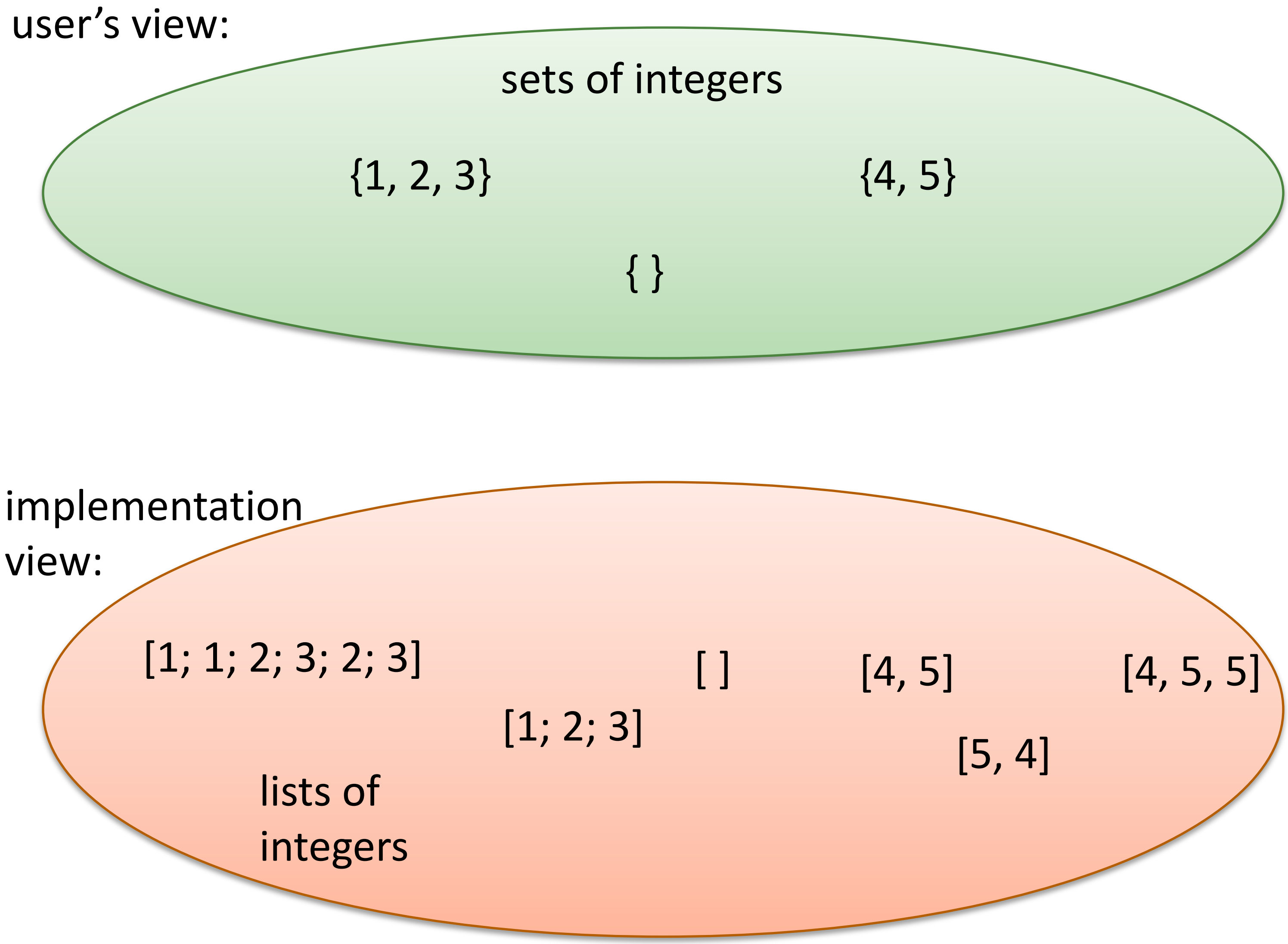
ABSTRACTION FUNCTIONS

Abstraction

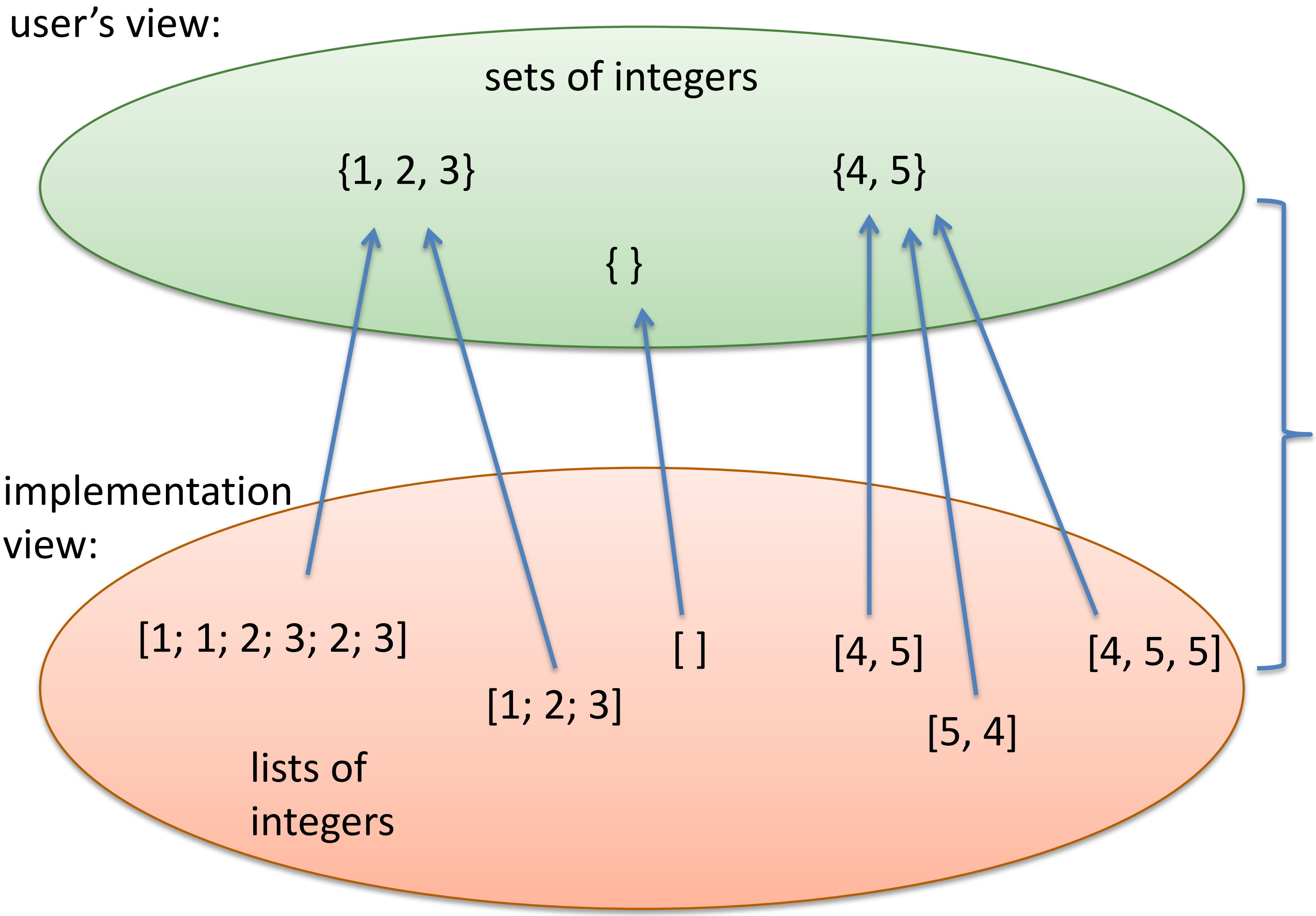
```
module type SET =
  sig
    type 'a set
    val empty : 'a set
    val mem : 'a -> 'a set -> bool
    ...
end
```

- 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?*

Abstraction



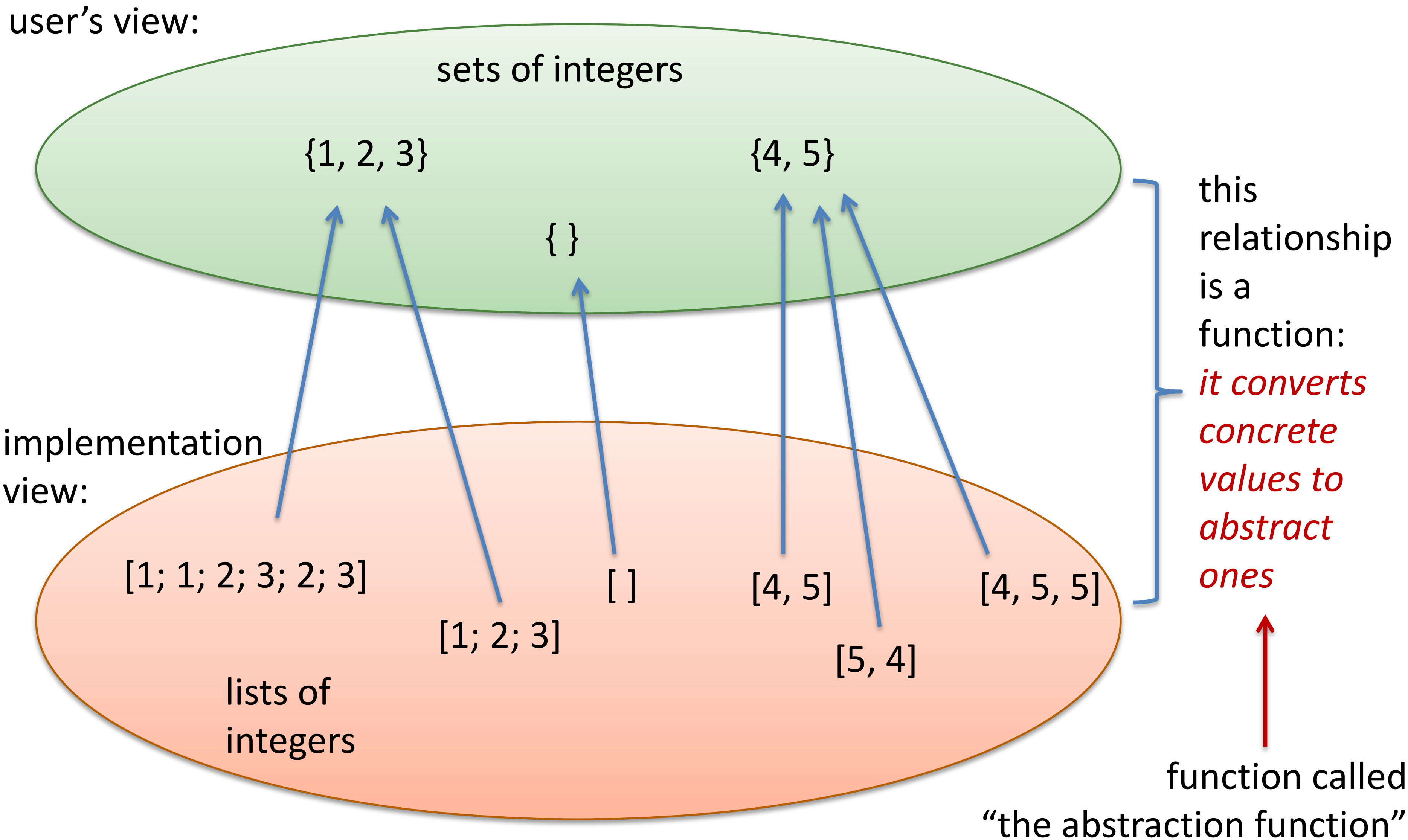
Abstraction



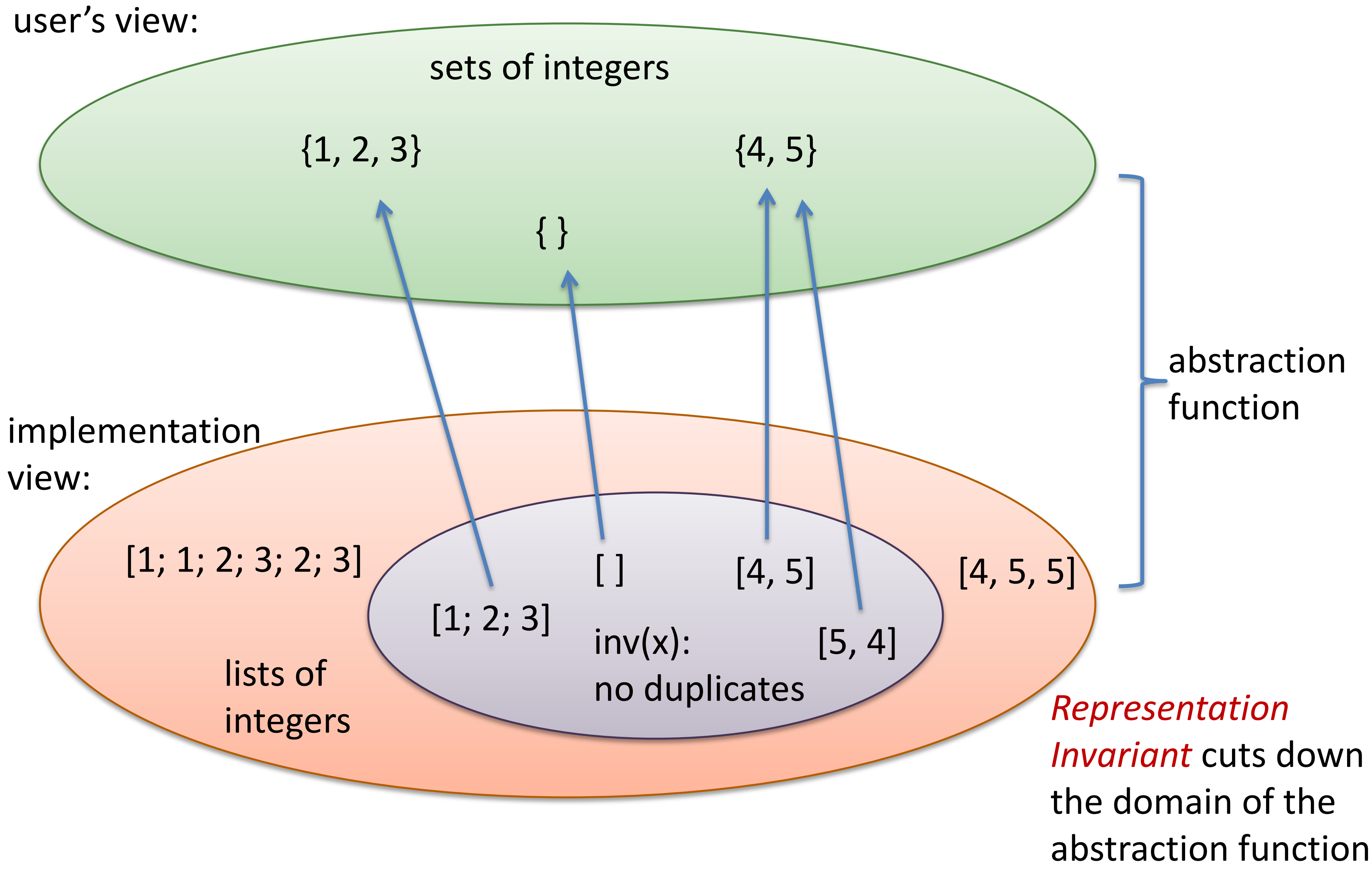
there's a relationship here, of course!

we are trying to *implement* the *abstraction*

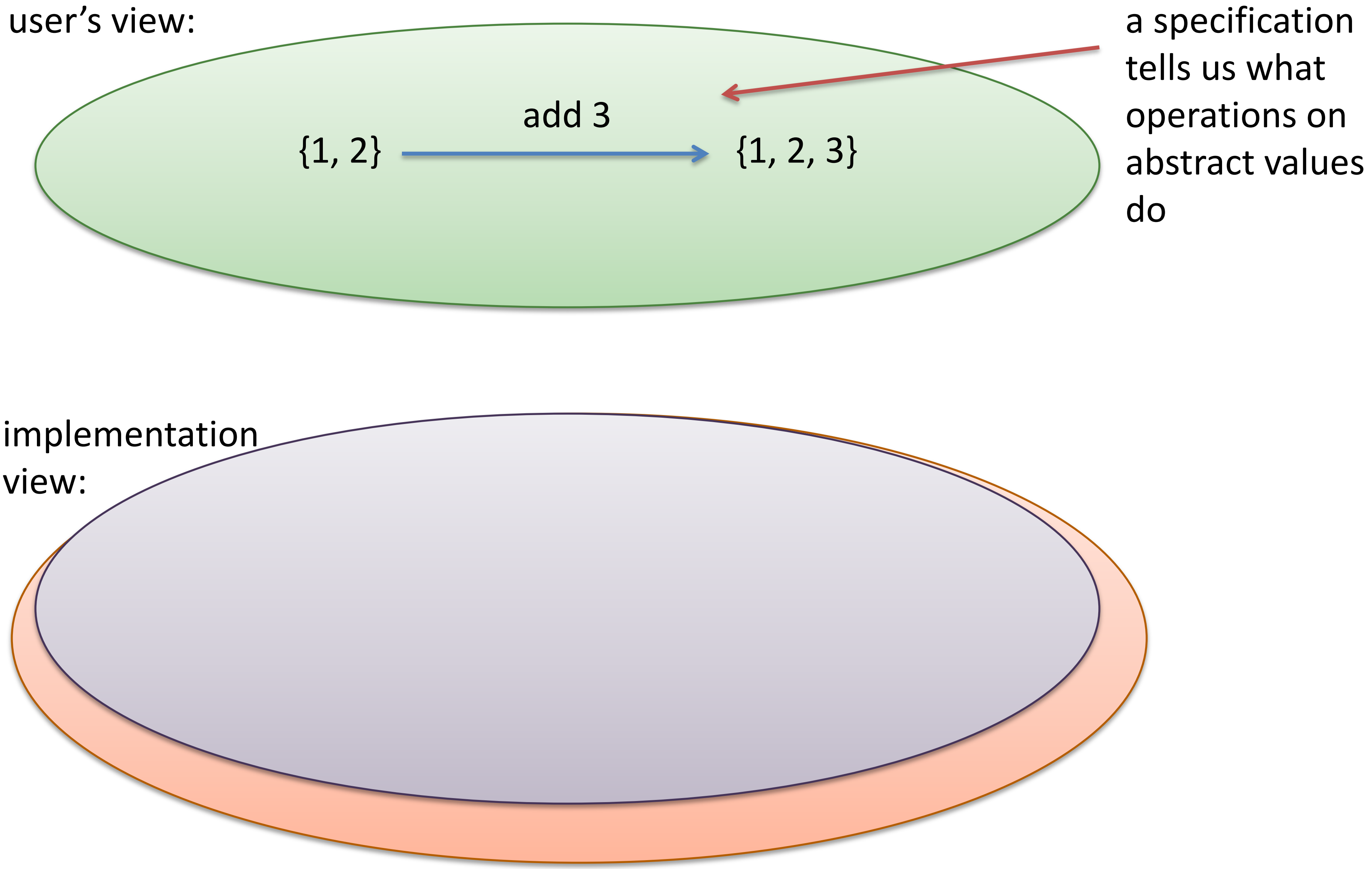
Abstraction



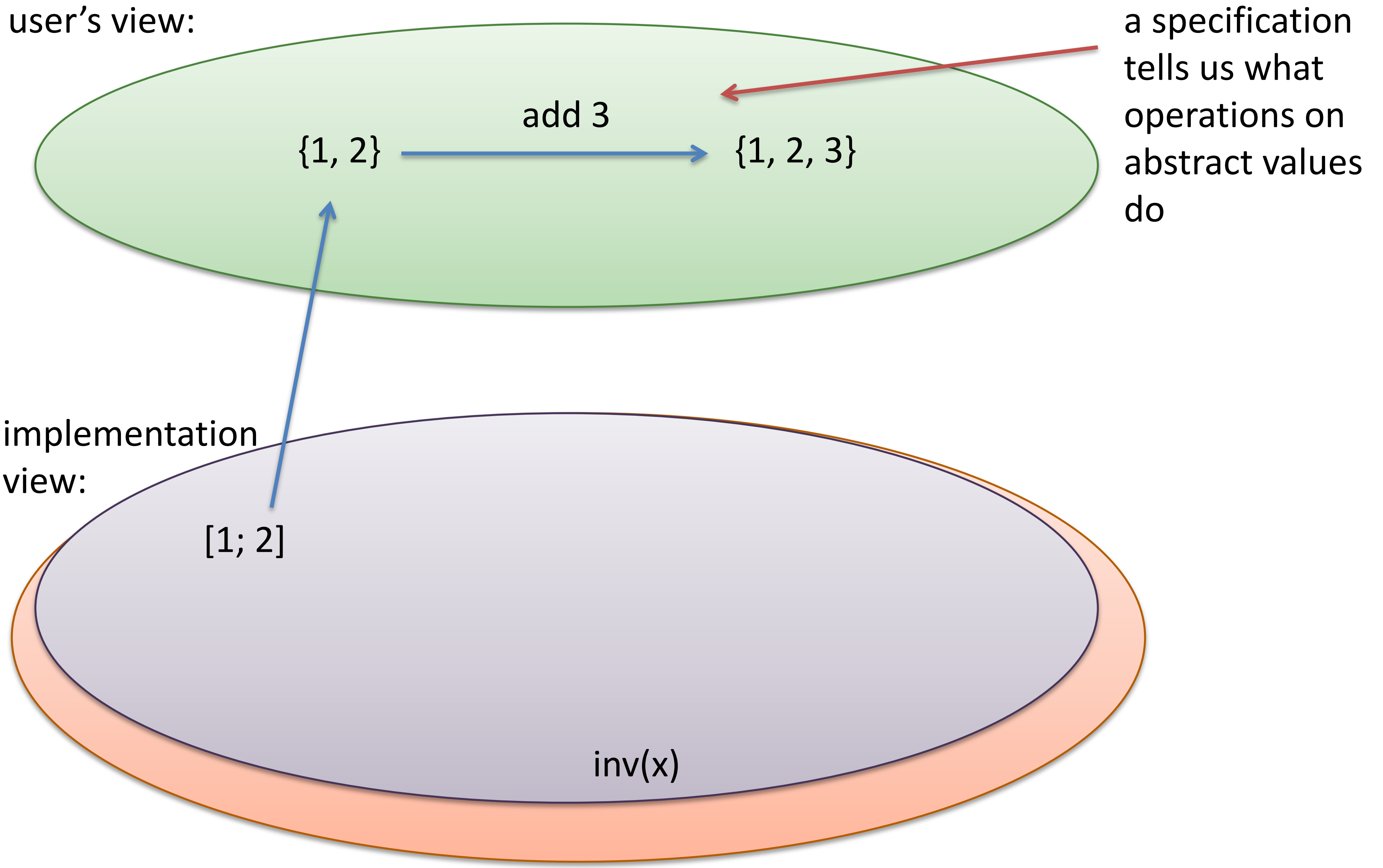
Abstraction



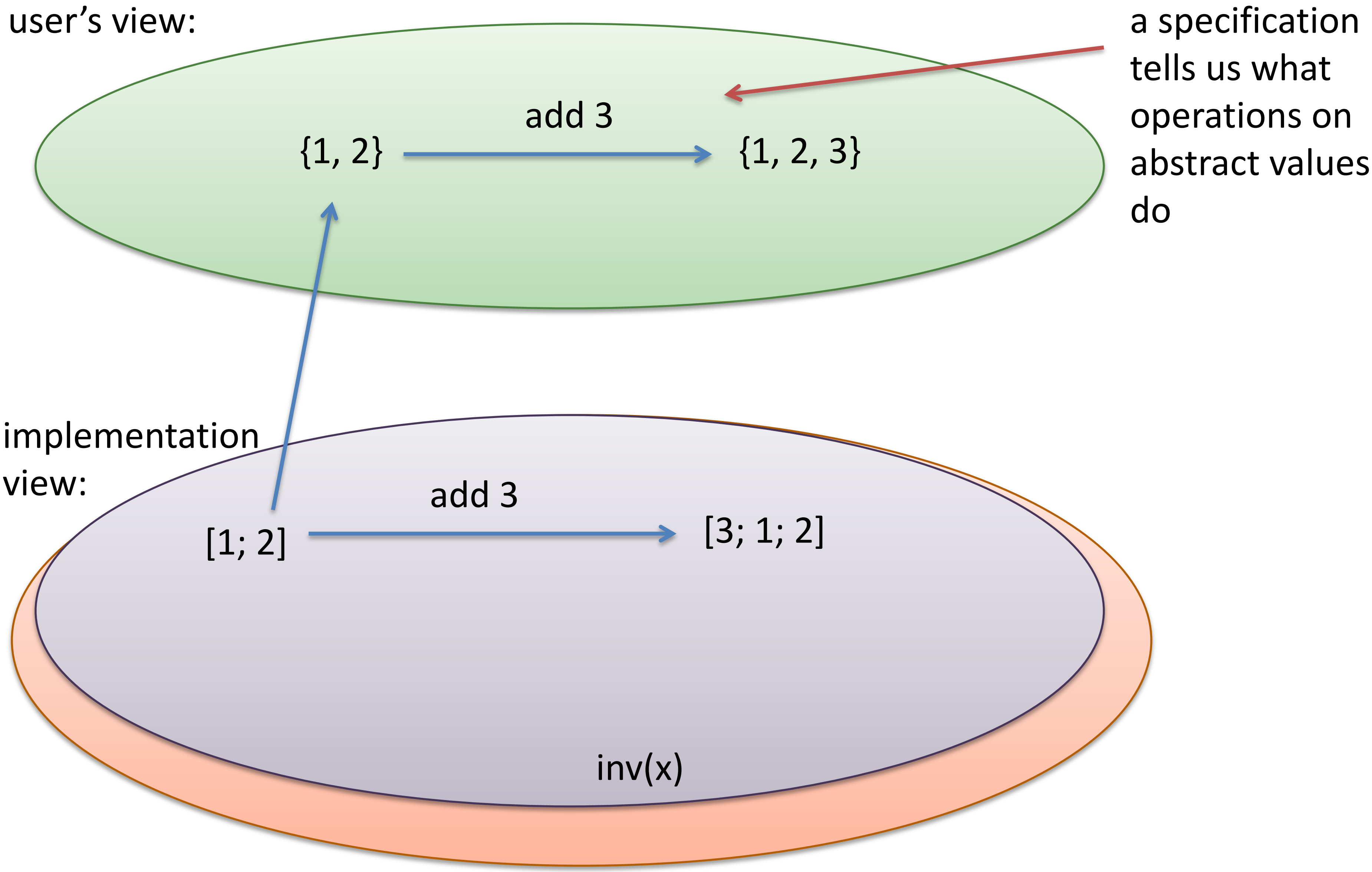
Specifications



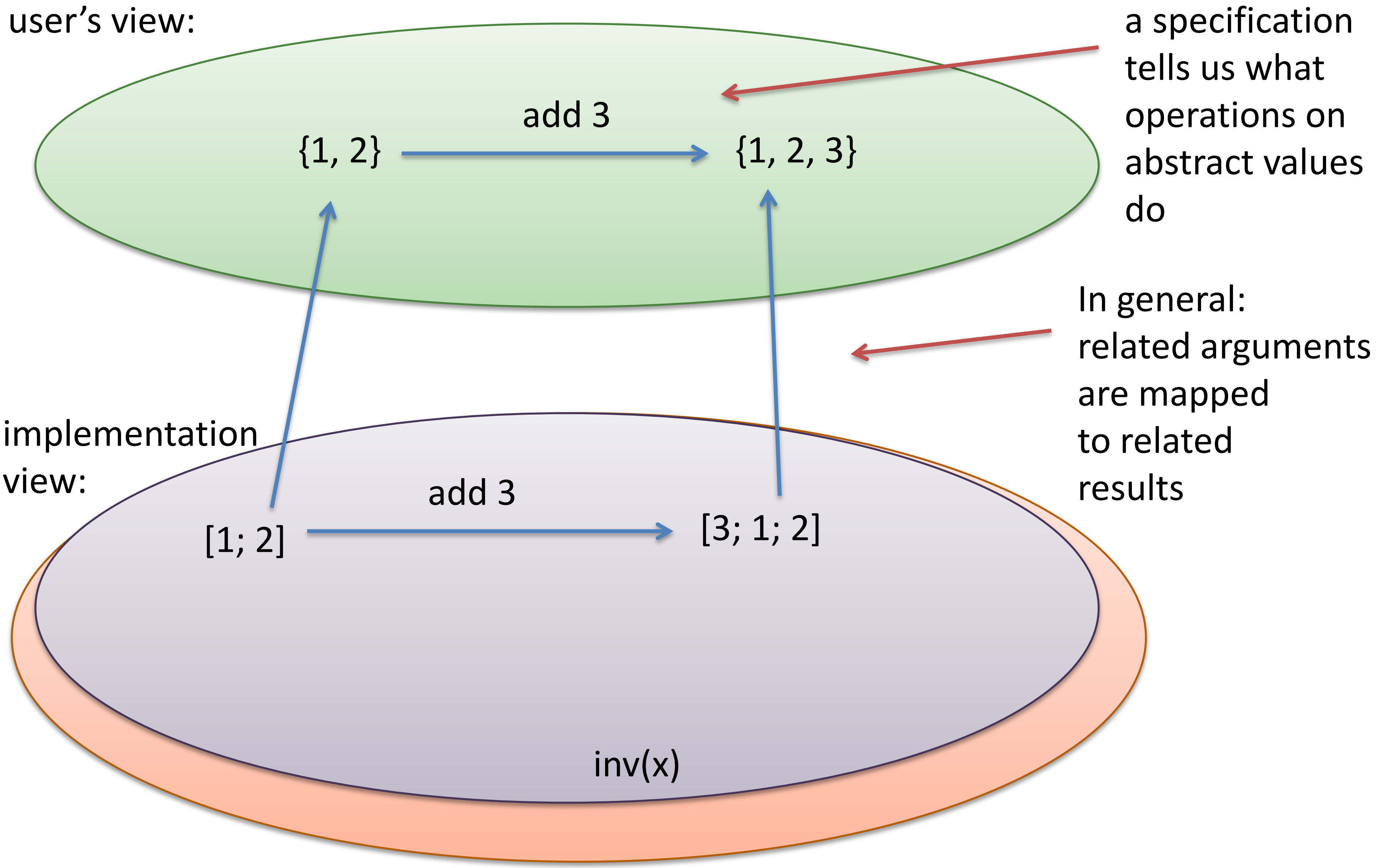
Specifications



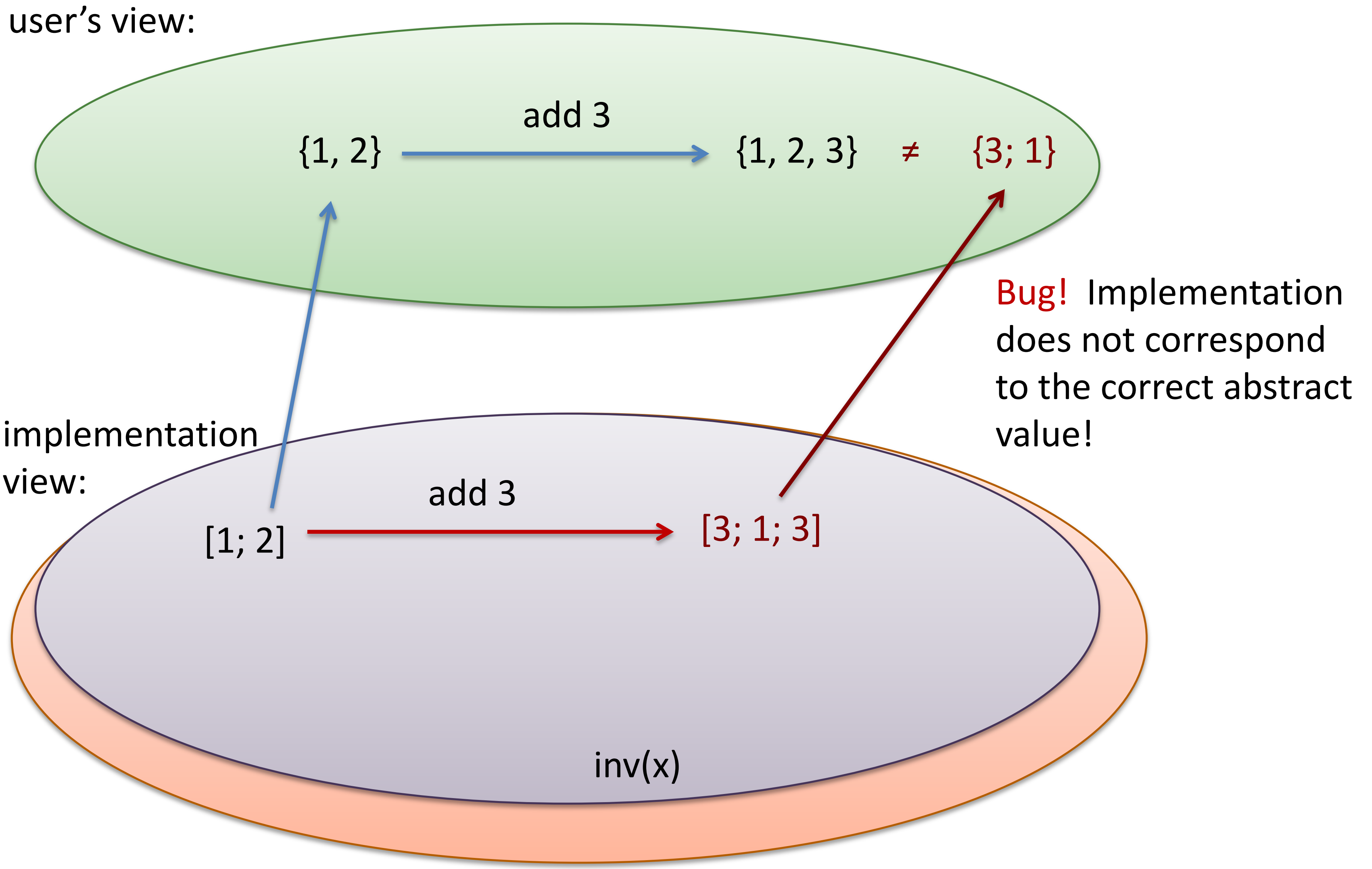
Specifications



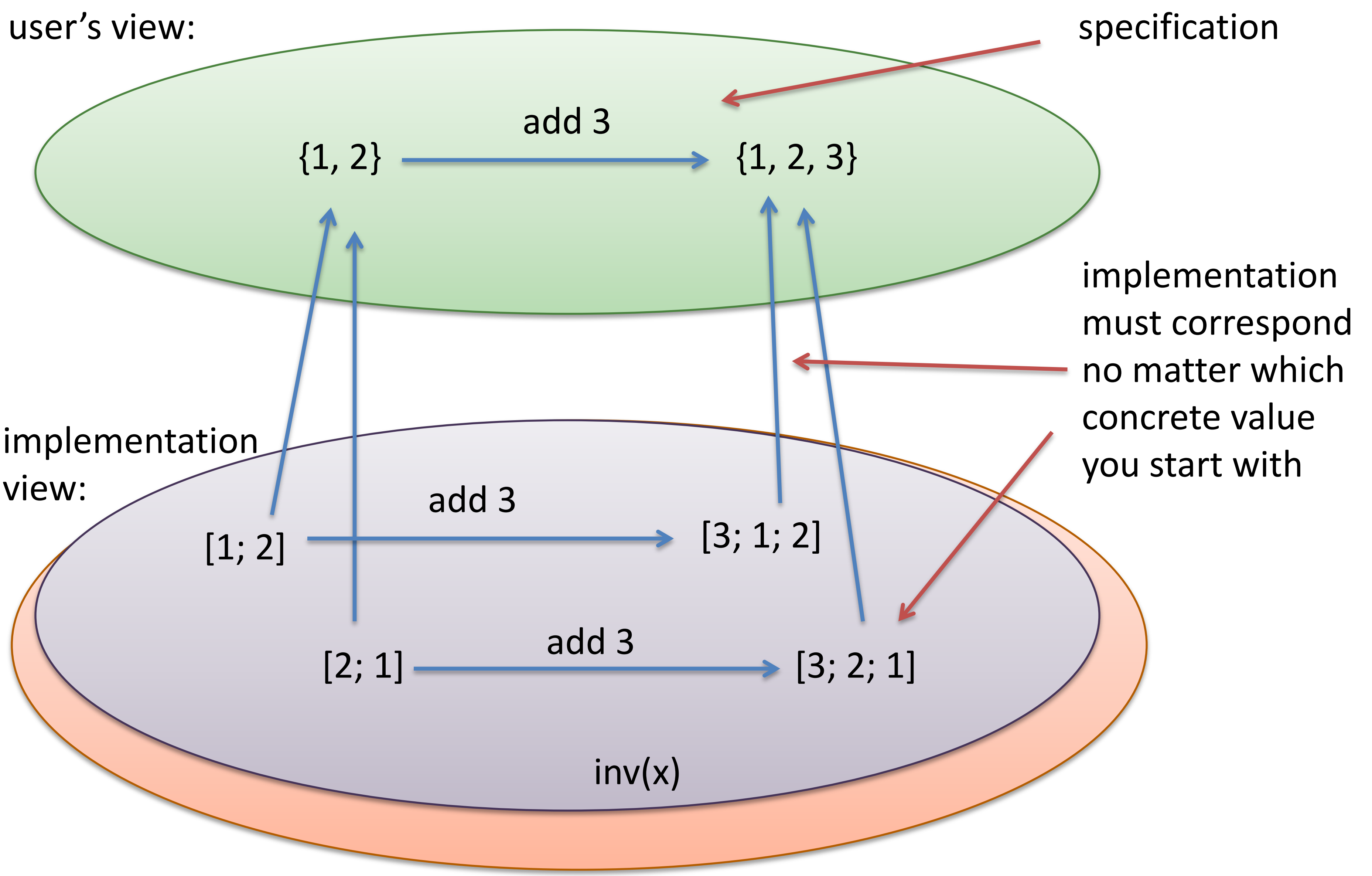
Specifications



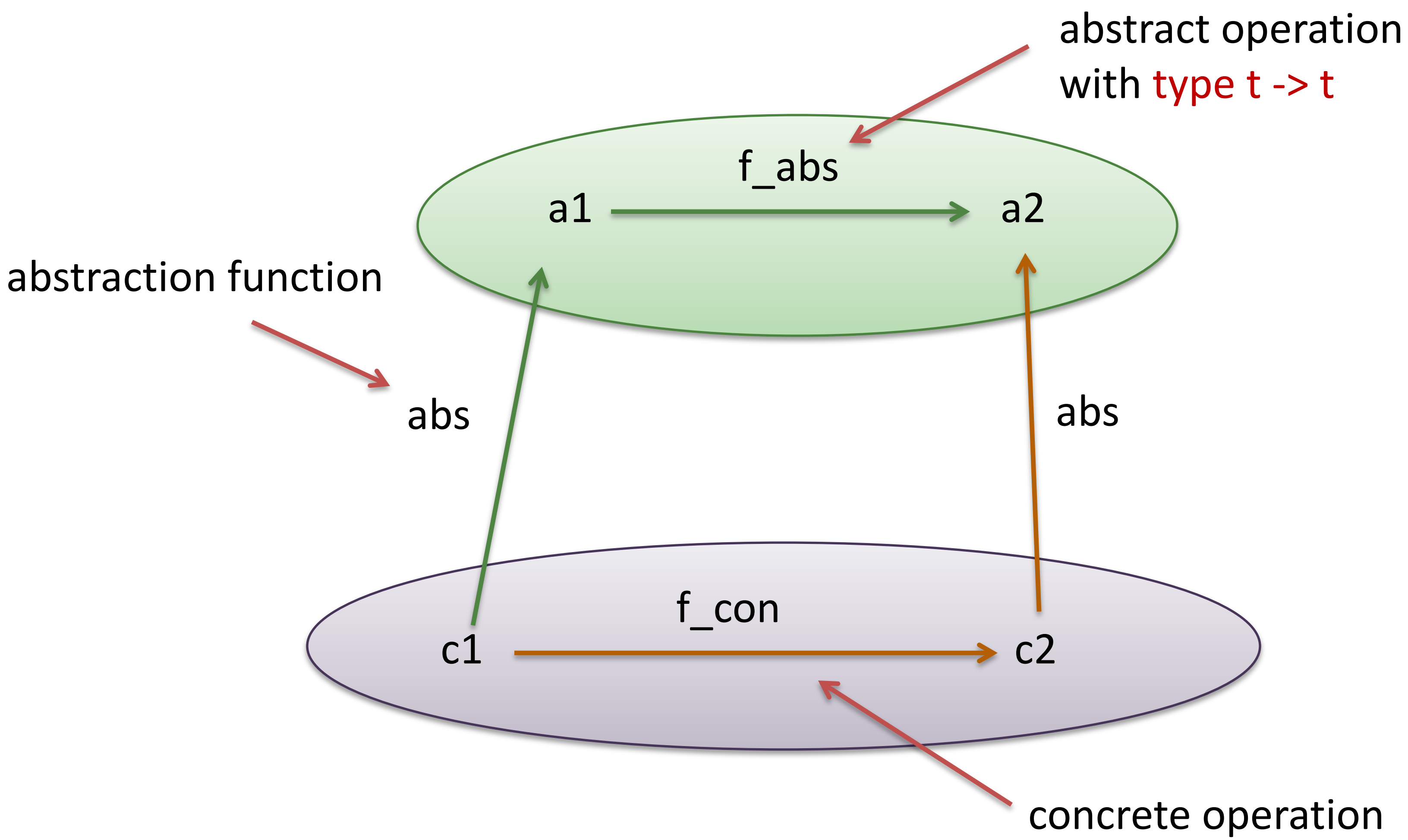
Specifications



Specifications



A more general view



to prove:

for all c1:t, if inv(c1) then f_abs (abs c1) == abs (f_con c1)

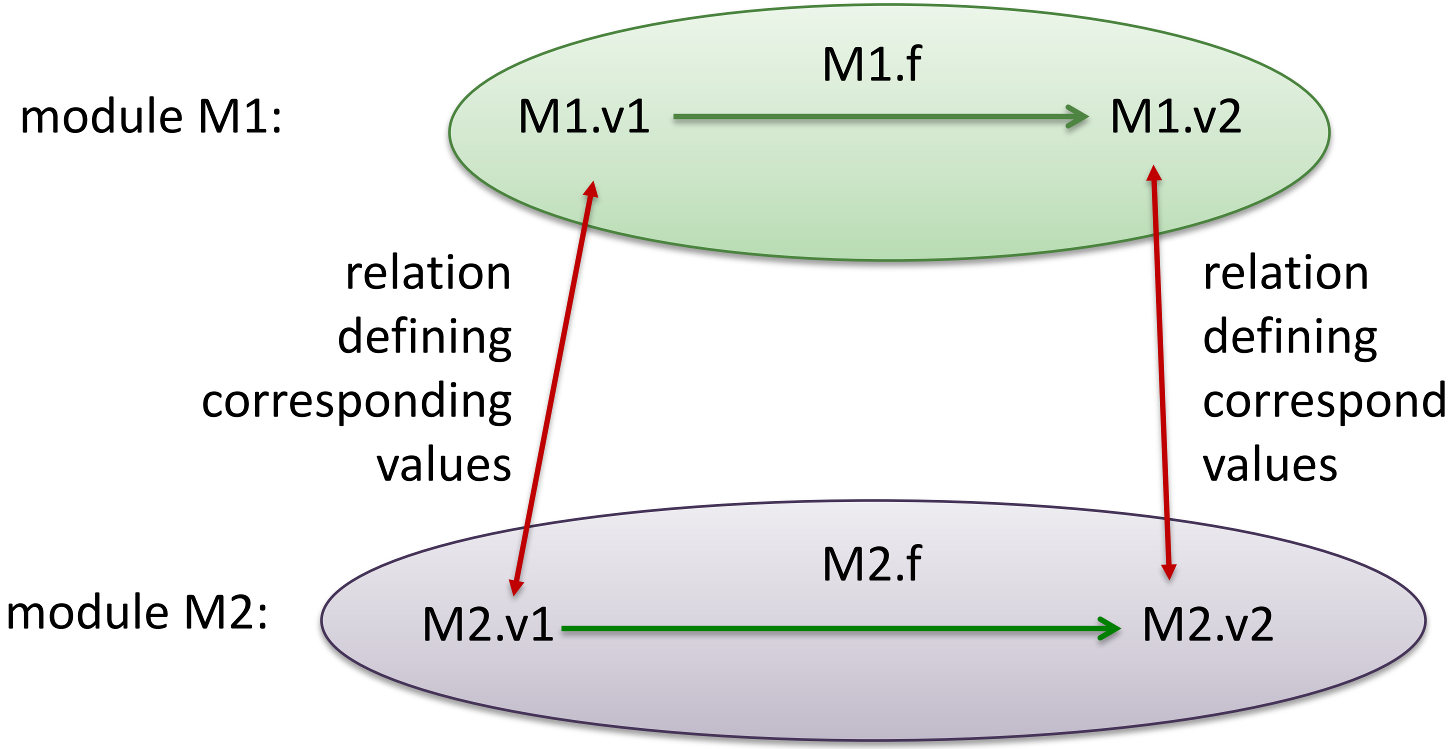
abstract then apply the abstract op == *apply concrete op then abstract*

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.*



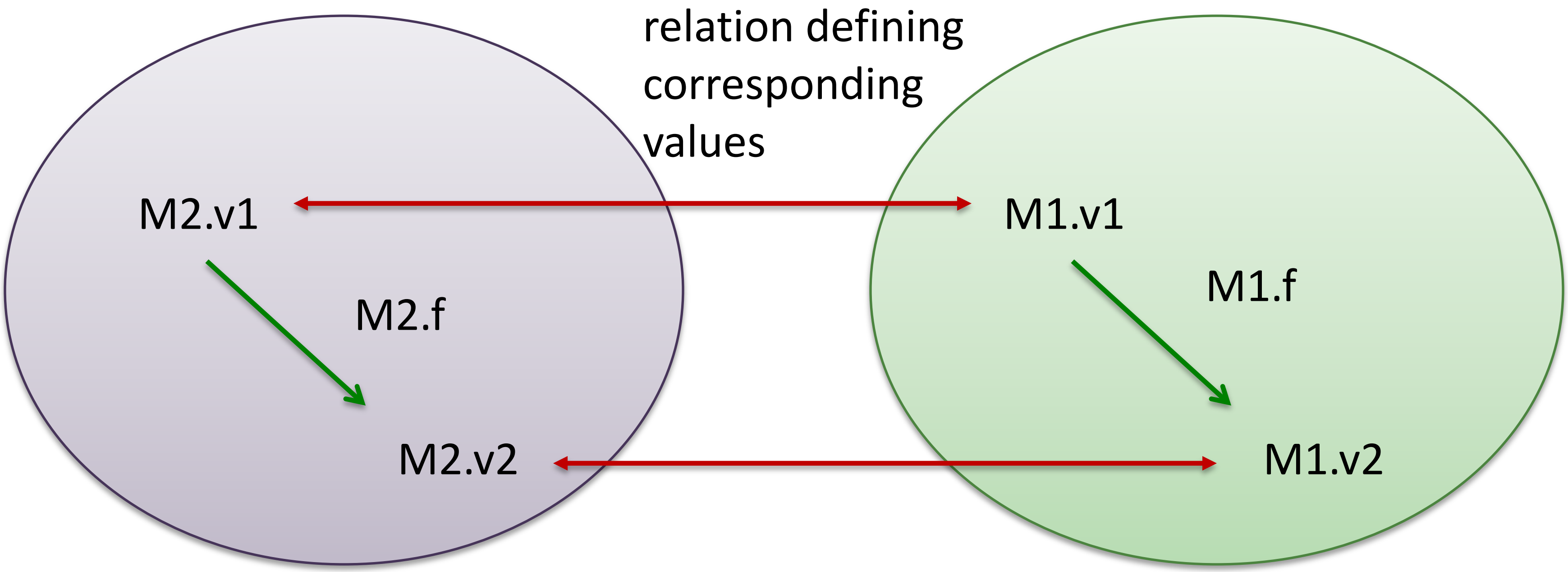
We ask: Do operations like f take related arguments to related results?

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.*



One Signature, Two Implementations

```
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
```

Consider a client that might use the module:

```
let x1 = M1.bump (M1.bump (M1.zero)
```

```
let x2 = M2.bump (M2.bump (M2.zero)
```

What is the relationship?

```
is_related (x1, x2) =
  x1   ==   x2/2 - 1
```

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

How do we prove it?

One Signature, Two Implementations

```
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: A representation invariant is a property that holds for all values of abs. type:

- if M.v has abstract type t,
  - we want 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 is_related(M1.v, M2.v) to be true

It's just a relation between two modules instead of one

One Signature, Two Implementations

```
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

One Signature, Two Implementations

```
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
```

Consider zero, which has abstract type t.

```
is_related (x1, x2) =
  x1   ==   x2/2 - 1
```

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         (subsitution)
```

One Signature, Two Implementations

```
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
```

```
is_related (x1, x2) =
  x1   ==   x2/2 - 1
```

Consider bump, which has abstract type t -> t.

Must prove for all v1:int, v2:int
if is_related(v1,v2) then is_related (M1.bump v1, M2.bump v2)

Proof:
(1) Assume is_related(v1, v2).
(2) v1 == v2/2 – 1 (by def)

Next, prove:
(M2.bump v2)/2 – 1 == M1.bump v1

(M2.bump v2)/2 - 1	
== (v2 + 2)/2 – 1	(eval)
== (v2/2 – 1) + 1	(math)
== v1 + 1	(by 2)
== M1.bump v1	(eval, reverse)

One Signature, Two Implementations

```
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
```

```
is_related (x1, x2) =
  x1   ==   x2/2 - 1
```

Consider reveal, which has abstract type t -> int.

Must prove for all v1:int, v2:int
if is_related(v1,v2) then M1.reveal v1 == M2.reveal v2

Proof:
(1) Assume is_related(v1, v2).
(2) v1 == v2/2 – 1 (by def)

Next, prove:
(M2.reveal v2 == M1.reveal v1

(M2.reveal v2)	
== v2/2 – 1	(eval)
== v1	(by 2)
== M1.reveal v1	(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