Reasoning about Modules

COS 326
David Walker
Princeton University

Before the Break

Introduction to OCaml mechanisms for defining modules:

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

Representation Invariants: a mechanism for reason about modules

- a property of all values with abstract type
- proof technique (roughly):
  - assume invariant on inputs to a module
  - prove invariants on outputs from the module
  - works because client code can move the *abstract* module outputs around before passing them back in to the module, but can't muck with the internals of abstract types
- proof technique (more precisely):
  - proof obligation based on the type of the value in the module signature
    - prove each value *v is valid for type s*
    - where s is the type in the module signature

REPRESENTATION INVARIANTS: A SIMPLE 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 rep 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 rep 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

let from_int (n:int) : t =
```

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

since function input has type t, assume the output satisfies inv()

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 in to 2 cases.
(1) n > 0, and (2) n <= 0

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 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
```

```
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)

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
```

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.
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

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*

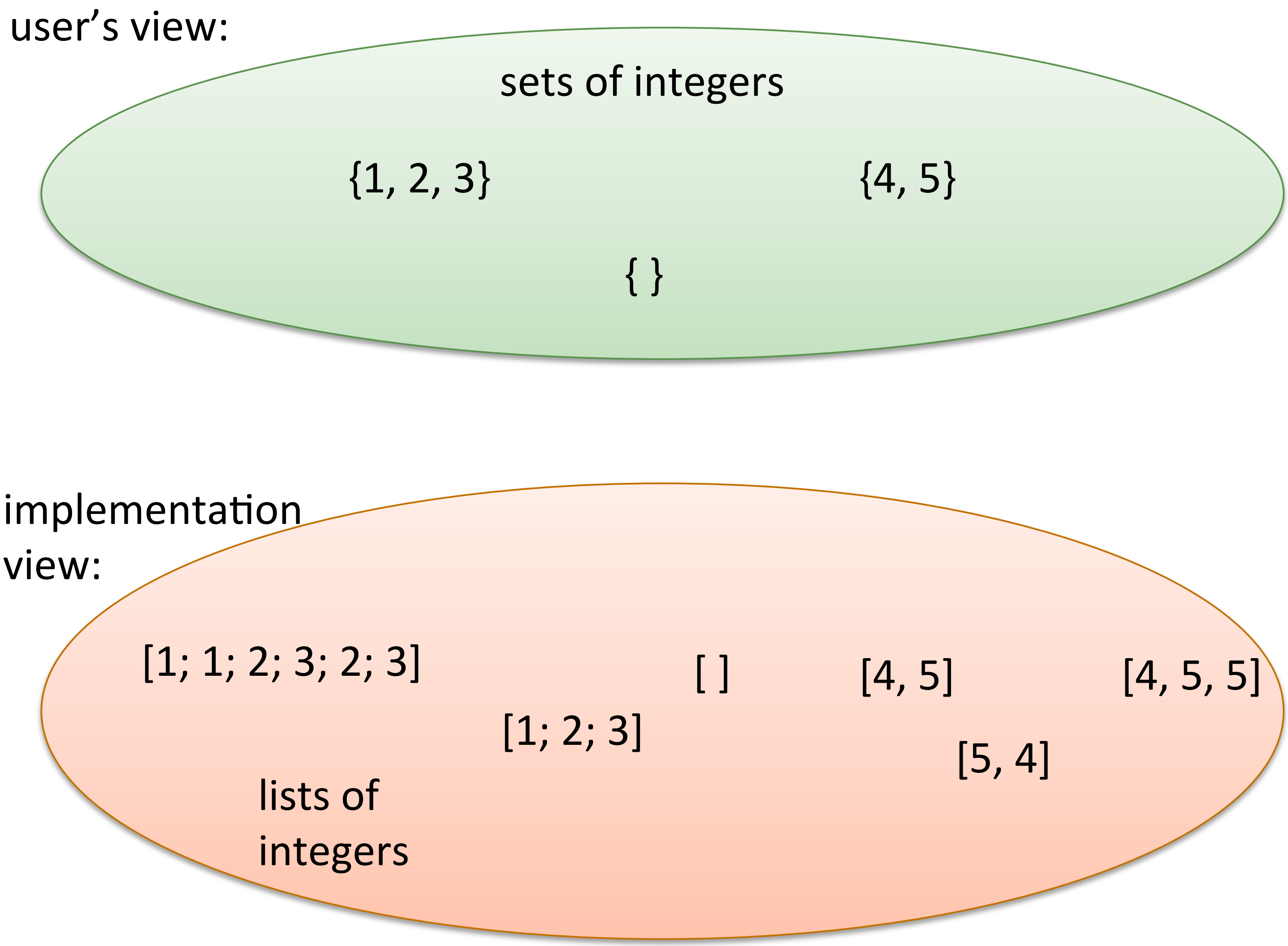
ABSTRACTION FUNCTIONS

Abstraction

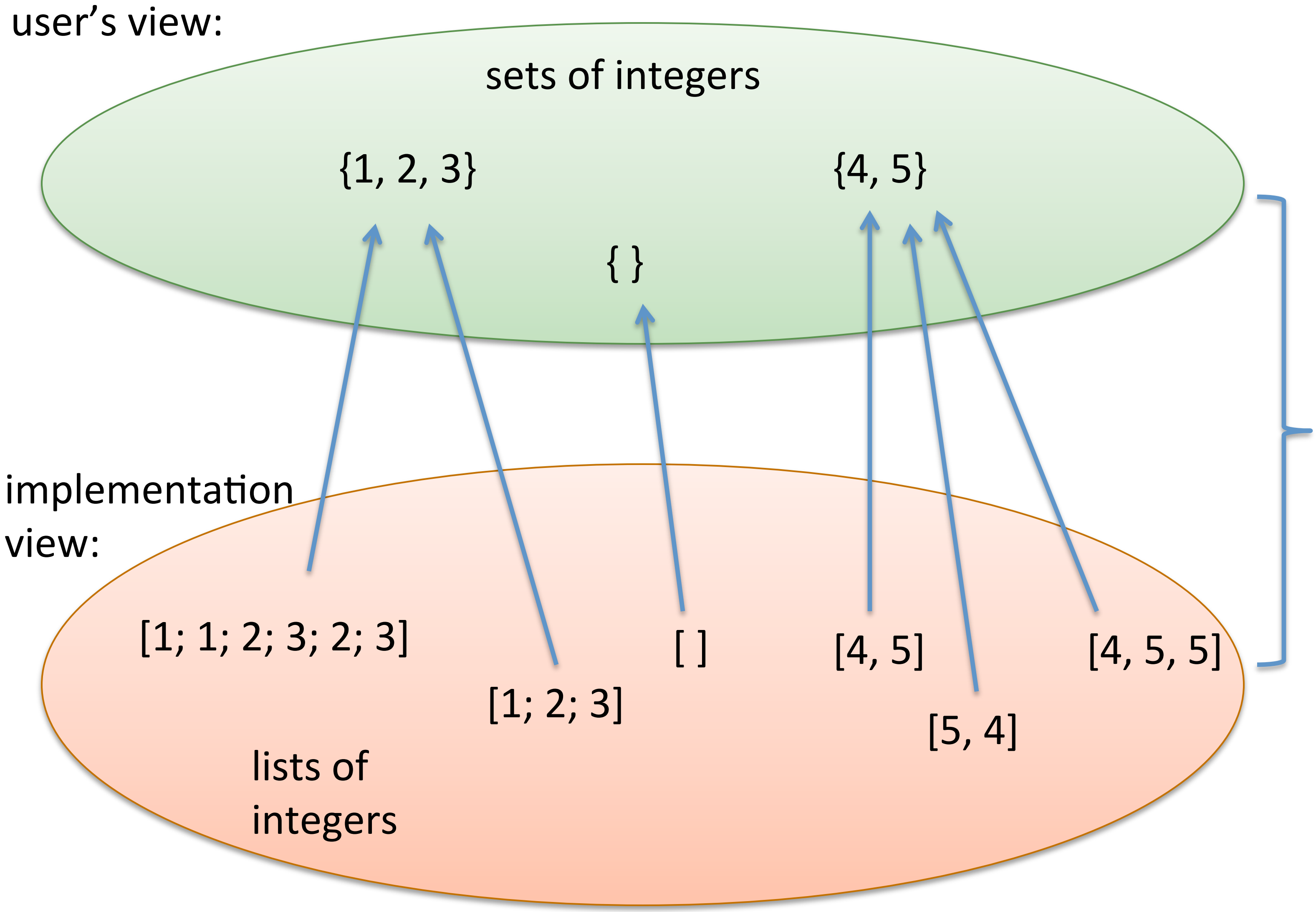
```
module type SET =
  sig
    type 'a set
    val empty : 'a set
    val mem : 'a -> 'a set -> bool
    ...
end
```

- When explaining our modules to clients, we would like to explain them in terms of *abstract values*
  - sets, not the lists (or may be 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



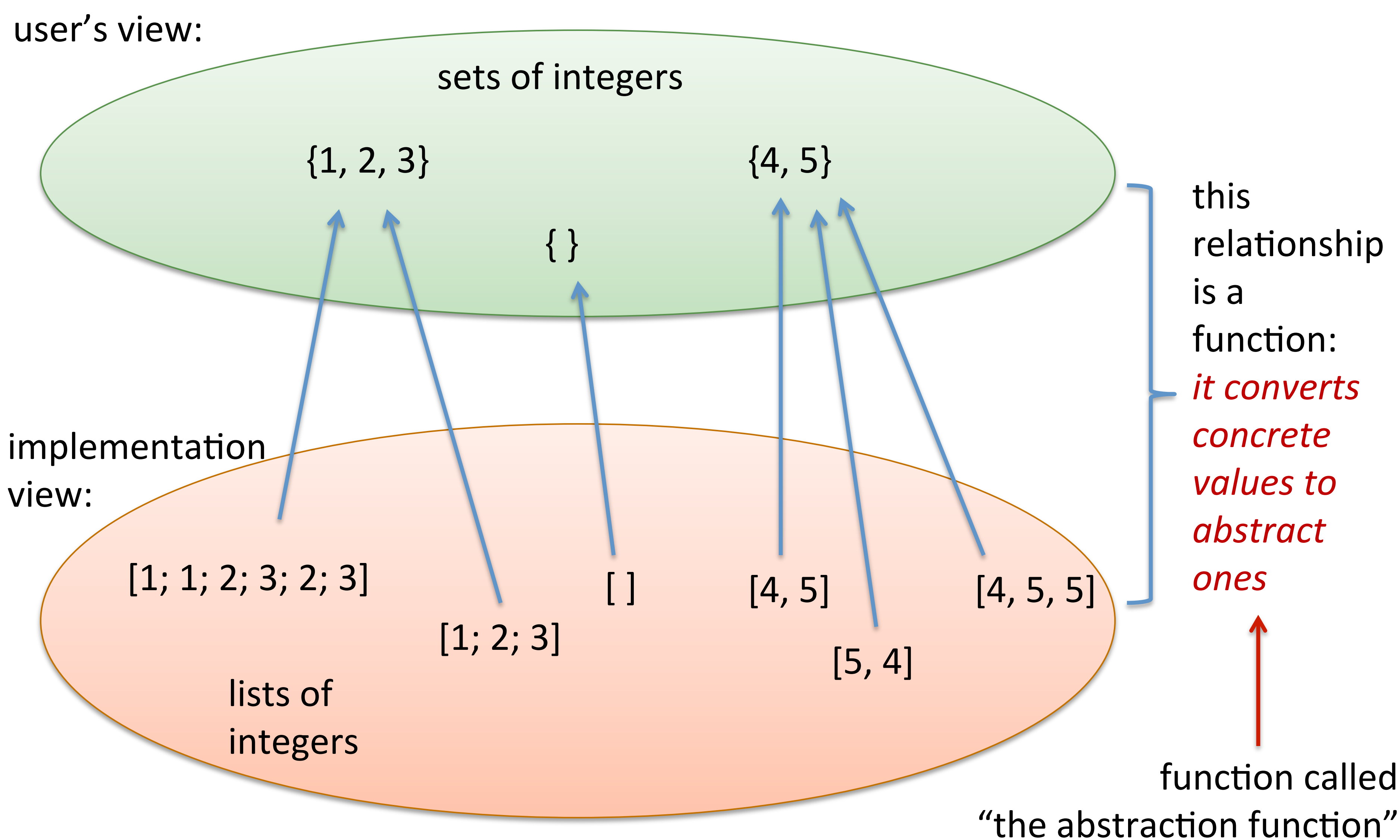
Abstraction



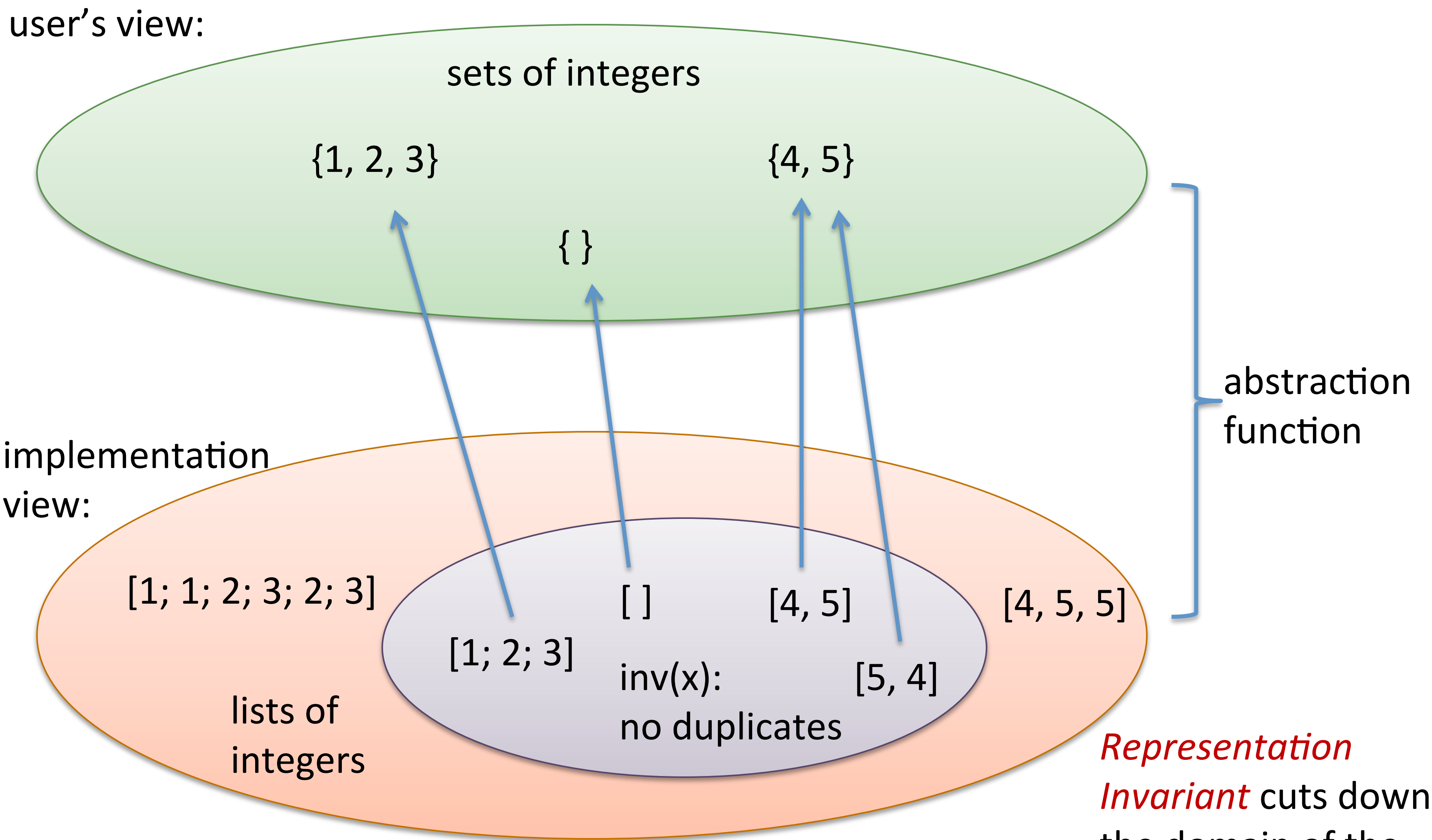
there's a relationship here, of course!

we are trying to *implement* the *abstraction*

Abstraction

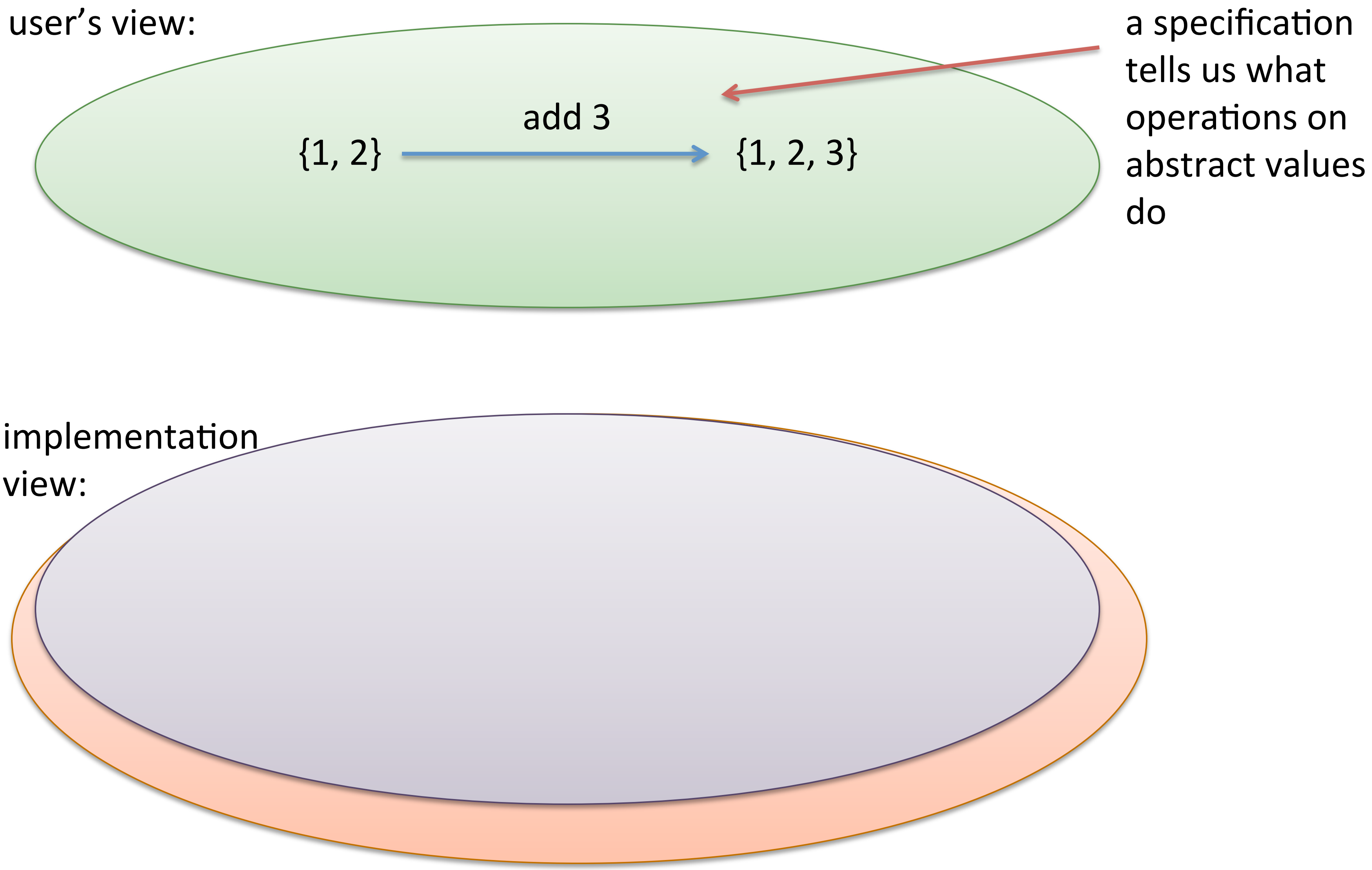


Abstraction

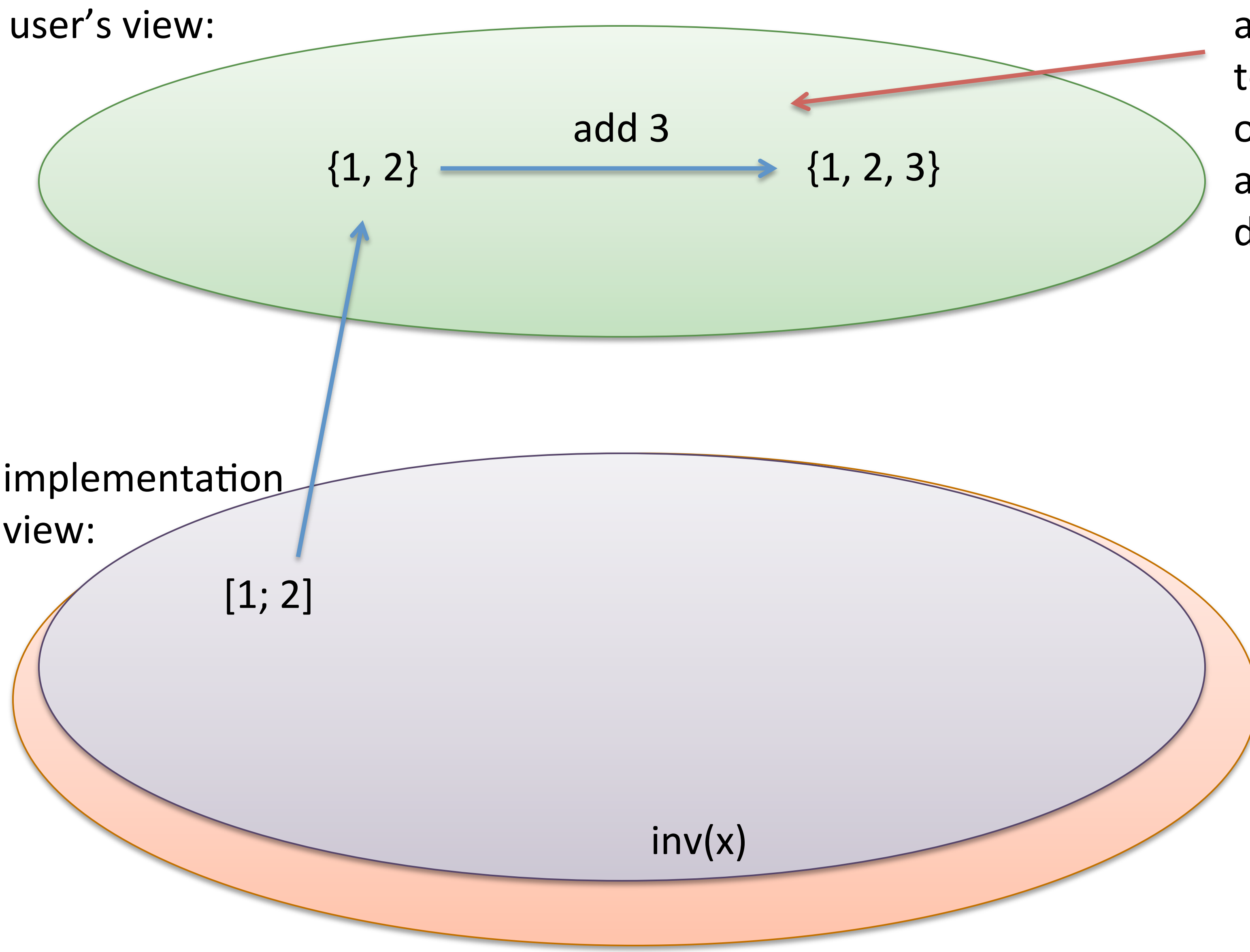


Representation Invariant cuts down the domain of the abstraction function

Specifications

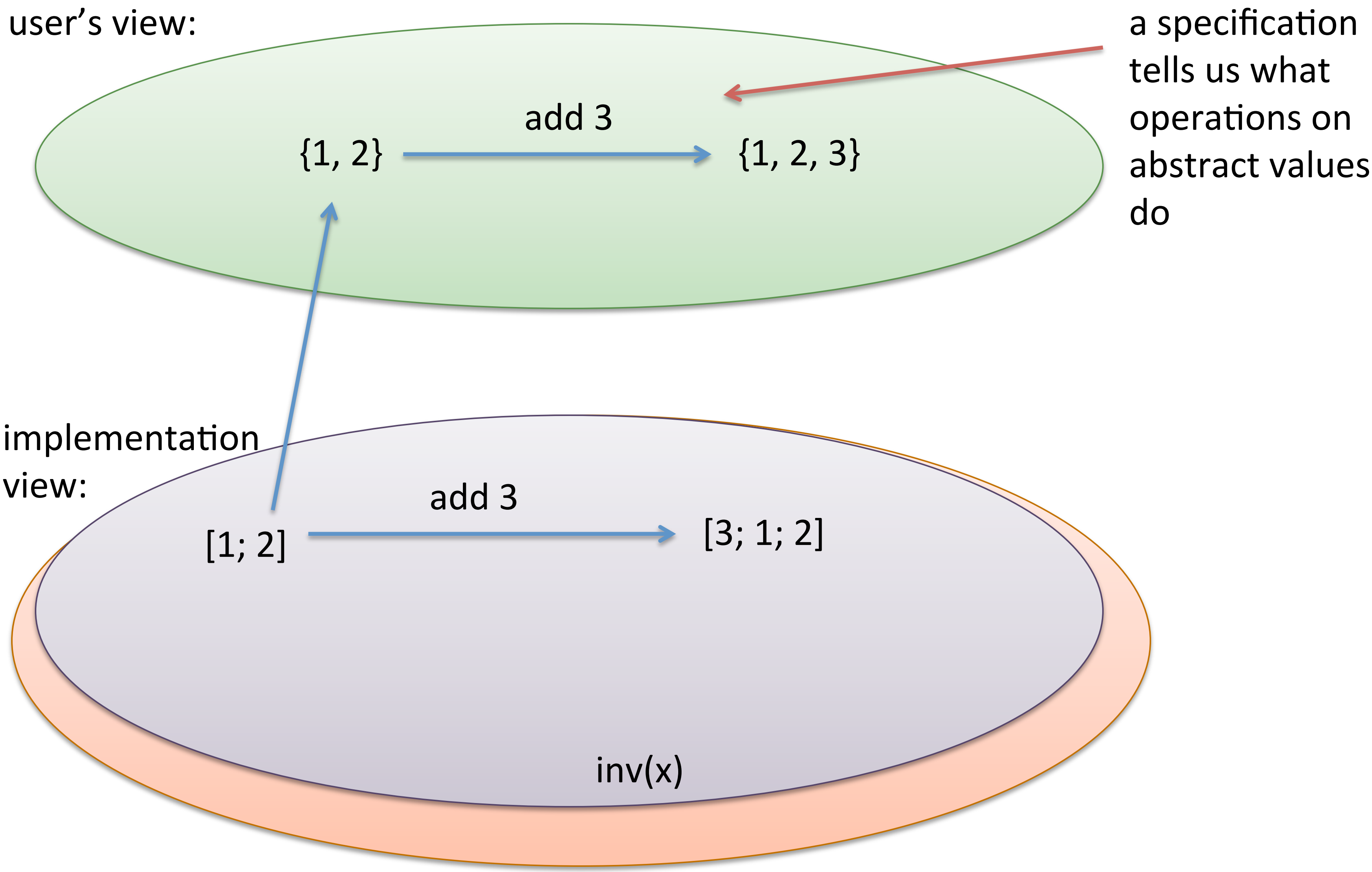


Specifications

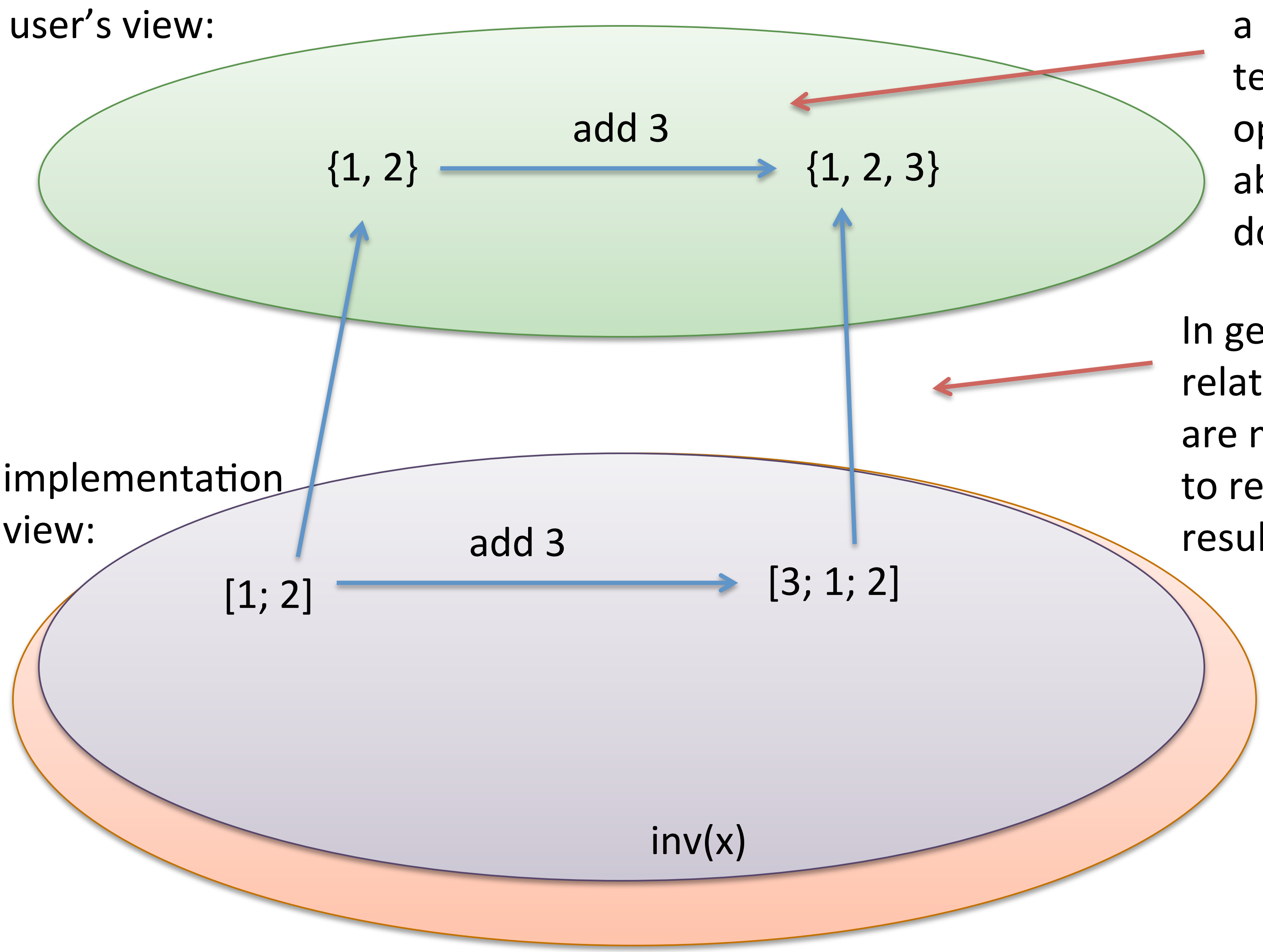


a specification tells us what operations on abstract values do

Specifications



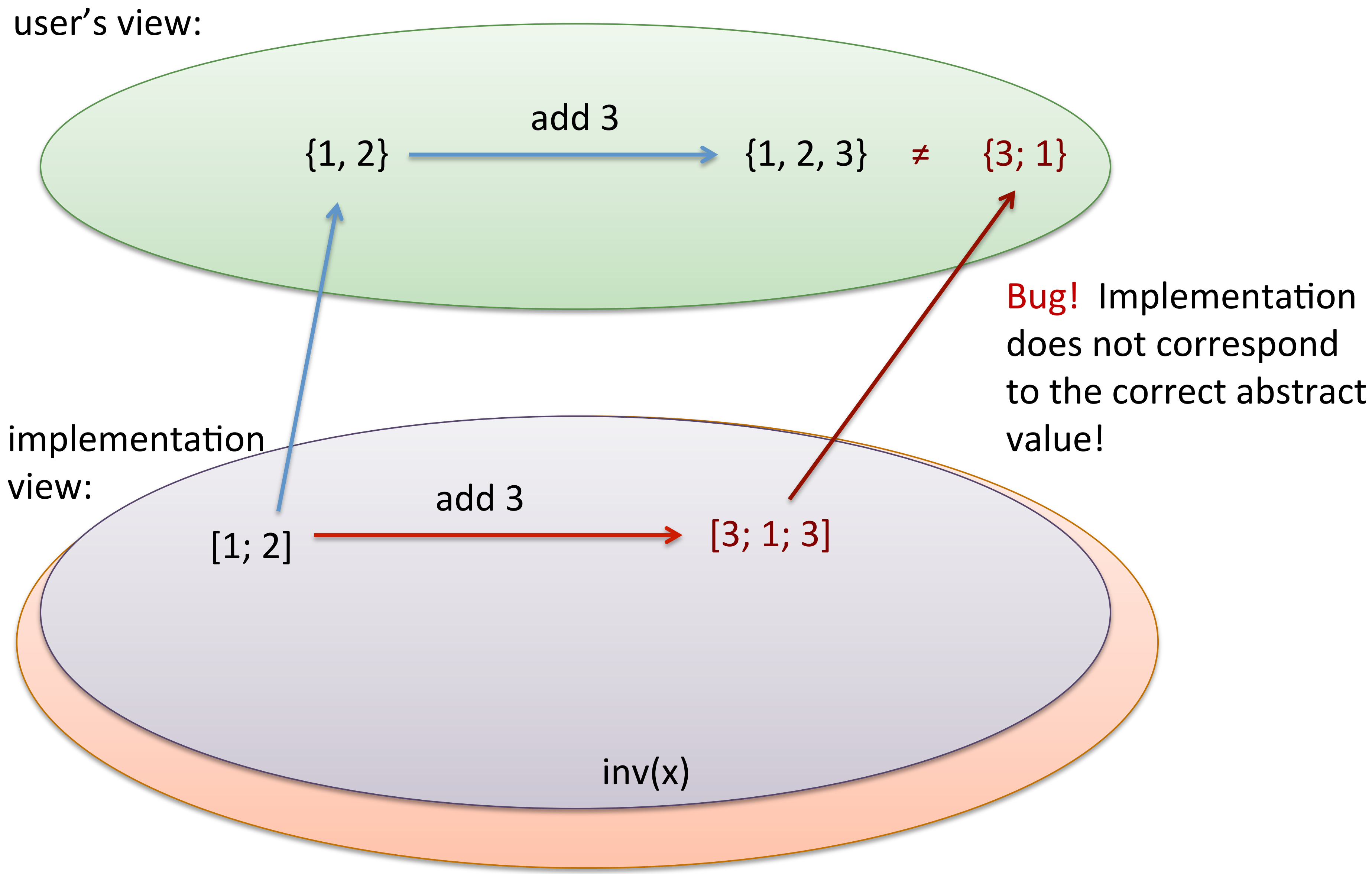
Specifications



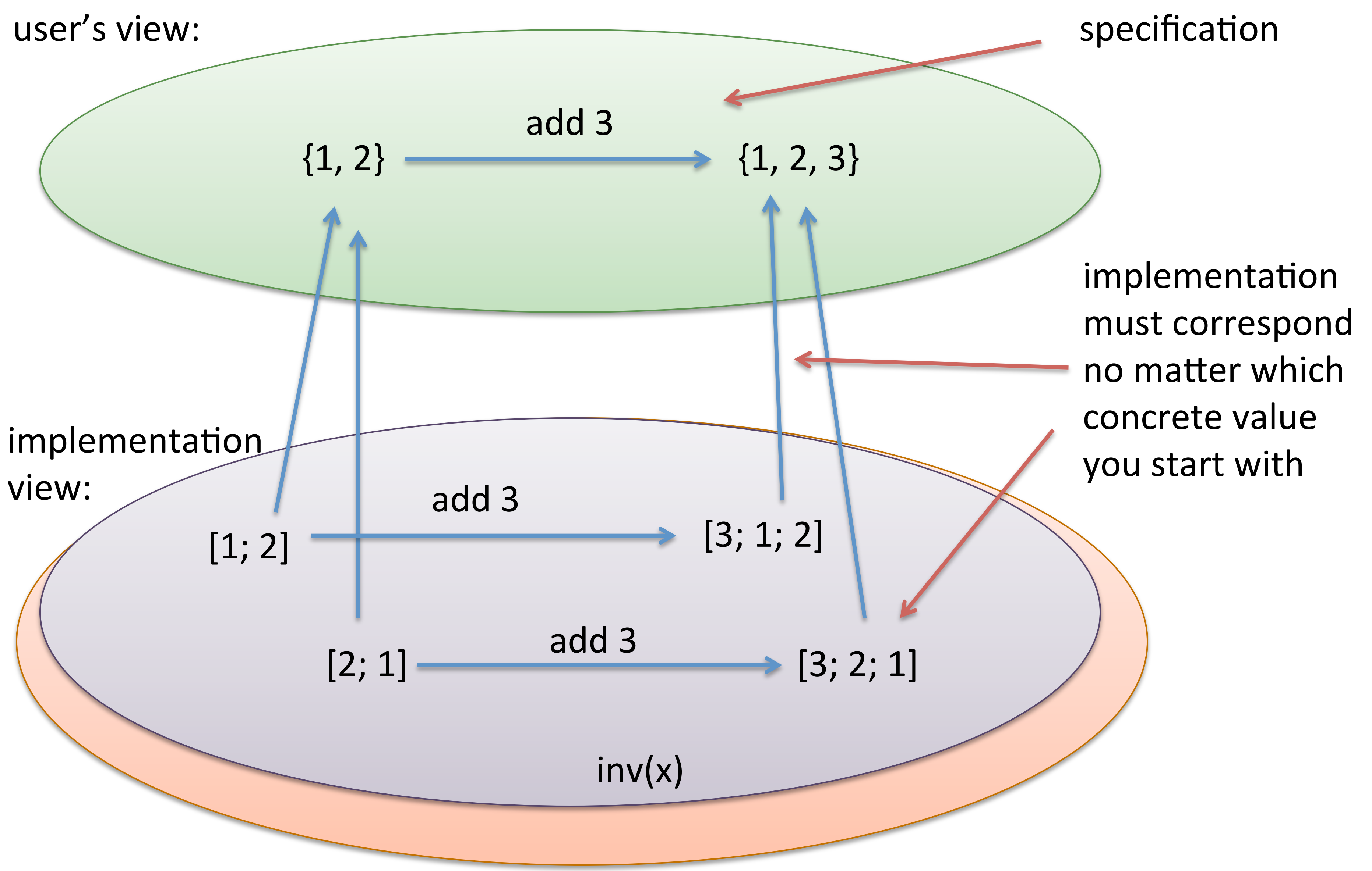
a specification tells us what operations on abstract values do

In general: related arguments are mapped to related results

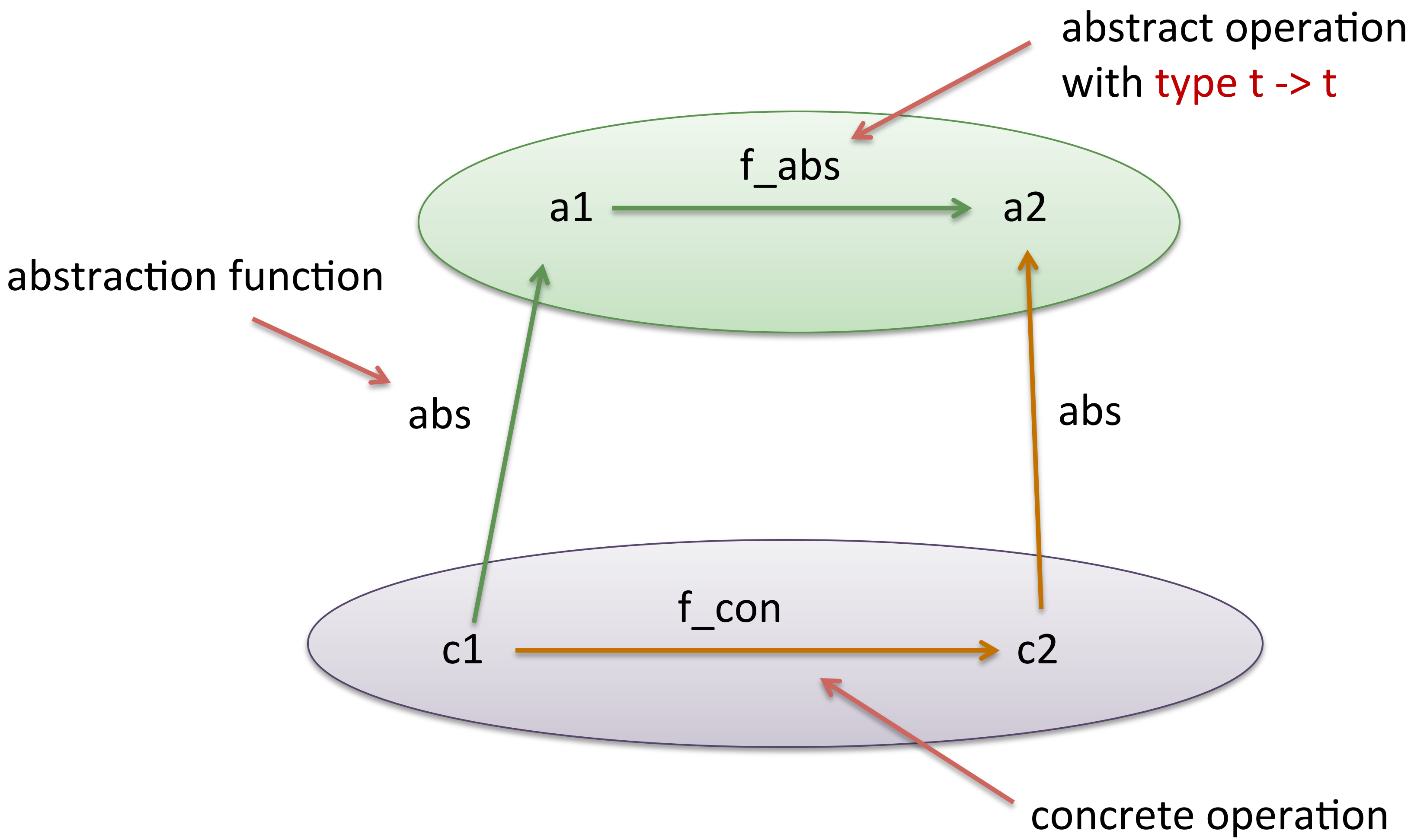
Specifications



Specifications



A more general view



to prove:

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

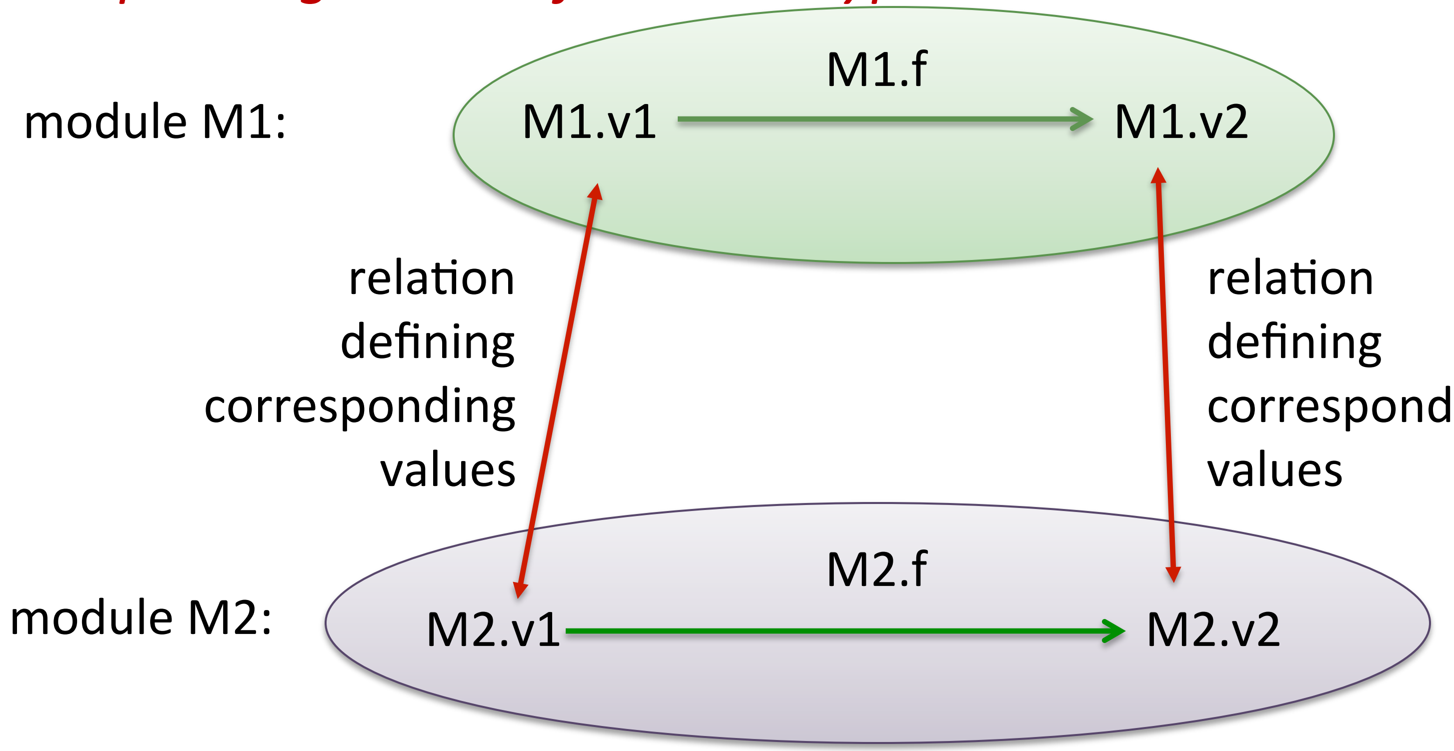
abstract then apply the abstract op == apply concrete op then abstract

Another Viewpoint

A specification is really just another implementation

- 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.*



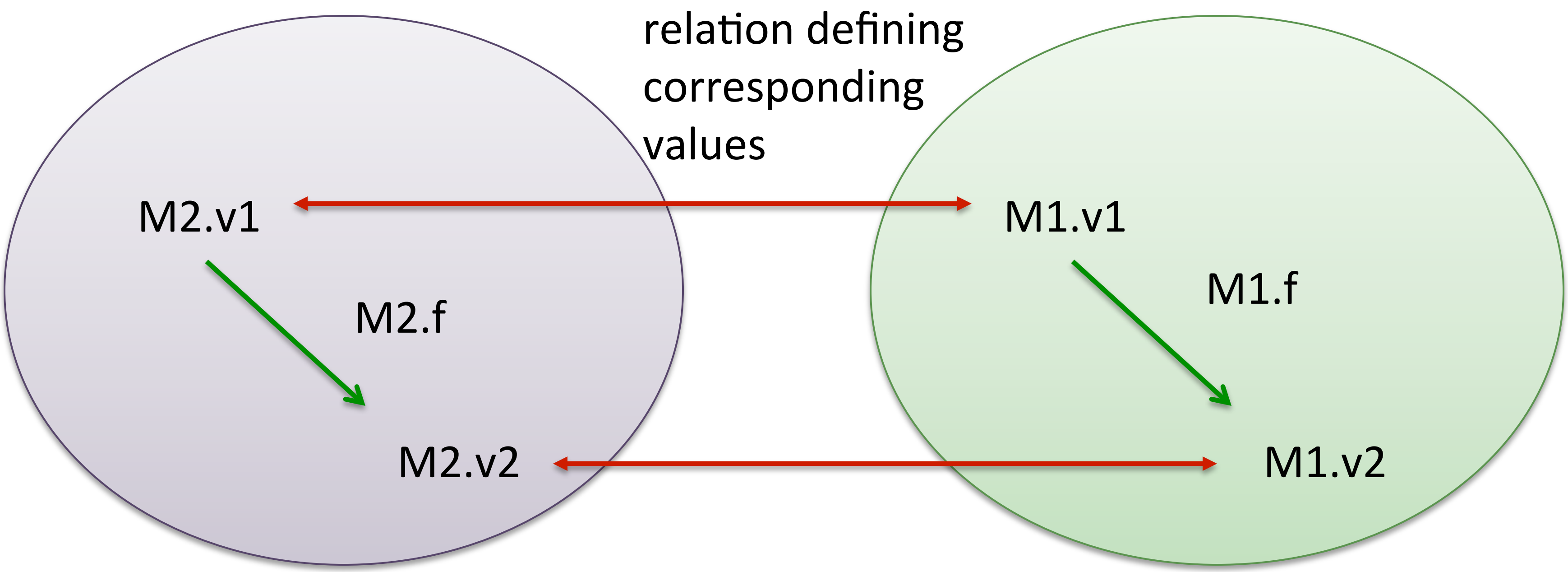
We ask: Do operations like f take related arguments to related results?

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 f1, f2 have type s1 -> s2 then we consider arg1, arg2 such that:
    - if is_related(arg1, arg2) then we prove
    - is_related(f1 arg1, f2 arg2)
  - if o1, o2 have type s option then we must prove:
    - o1 == None and o2 == None, or
    - o1 == Some u1 and o2 == 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

A SIMPLE EXAMPLE

Representing Ints

```
module type NUM =
  sig
    type t
    val create : int -> t
    val equals : t -> t -> bool
    val decr : t -> t
  end
```

```
module Num =
  struct
    type t = Zero | Pos of int | Neg of int

    let create (n:int) : t =
      if n = 0 then Zero
      else if n > 0 then Pos n
      else Neg (abs n)

    let equals (n1:t) (n2:t) : bool =
      match n1, n2 with
              Zero, Zero -> true
            | Pos n, Pos m when n = m -> true
            | Neg n, Neg m when n = m -> true
            | _ -> false

  end
```

Representing Ints

```
module type NUM =
  sig
    type t
    val create : int -> t
    val equals : t -> t -> bool
    val decr : t -> t
  end
```

```
module Num =
  struct
    type t = Zero | Pos of int | Neg of int

    let create (n:int) : t = ...

    let equals (n1:t) (n2:t) : bool = ...

    let decr (n:t) : t =
       match t with
          Zero -> Neg 1
        | Pos n when n > 1 -> Pos (n-1)
        | Pos n when n = 1 -> Zero
        | Neg n -> Neg (n+1)
  end
```

Representing Ints

```
module type NUM =
  sig
    type t
    val create : int -> t
    val equals : t -> t -> bool
    val decr : t -> t
  end
```

```
let inv (n:t) : bool =
    match n with
        Zero -> true
      | Pos n when n > 0 -> true
      | Neg n when n > 0 -> true
      | _ -> false
```

```
module Num =
  struct
    type t = Zero | Pos of int | Neg of int

    let create (n:int) : t = ...

    let equals (n1:t) (n2:t) : bool = ...

    let decr (n:t) : t =
       match t with
          Zero -> Neg 1
        | Pos n when n > 1 -> Pos (n-1)
        | Pos n when n = 1 -> Zero
        | Neg n -> Neg (n+1)
  end
```

Another Implementation

```
module type NUM =
  sig
    type t
    val create : int -> t
    val equals : t -> t -> bool
    val decr : t -> t
  end
```

```
let inv (n:t) : bool = true
```

```
module Num2 =
  struct
    type t = int

    let create (n:int) : t = n

    let equals (n1:t) (n2:t) : bool = n1 = n2

    let decr (n:t) : t = n - 1
end
```

Another Implementation

```
module type NUM =
  sig
    type t
    val create : int -> t
    val equals : t -> t -> bool
    val decr : t -> t
  end
```

```
module Num =
  struct
    type t = Zero | Pos of int | Neg of int

    let create (n:int) : t = ...

    let equals (n1:t) (n2:t) : bool = ...

    let decr (n:t) : t = ...
end
```

```
module Num2 =
  struct
    type t = int

    let create (n:int) : t = n

    let equals (n1:t) (n2:t) : bool = n1 = n2

    let decr (n:t) : t = n - 1
end
```

Question: Is Num2

Representing Ints

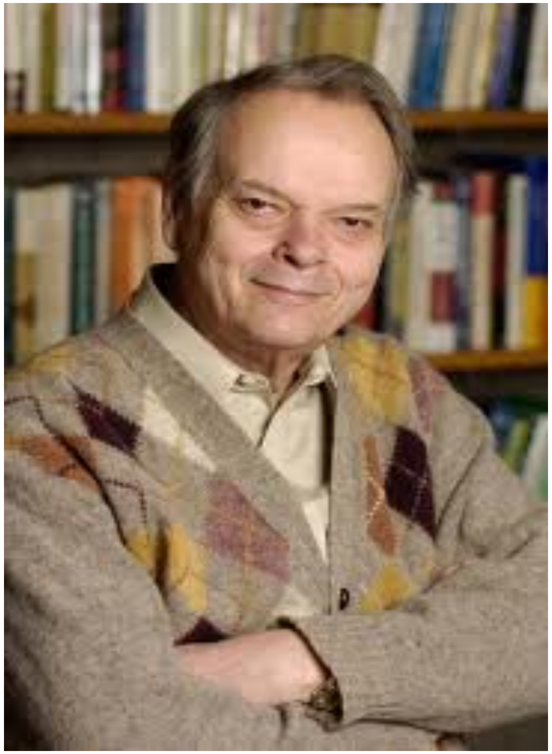
```
module type NUM =
  sig
    type t
    val create : int -> t
    val equals : t -> t -> bool
    val decr : t -> t
  end
```

```
module Num =
  struct
    type t = Zero | Pos of int | Neg of int

    let create (n:int) : t = ...

    let equals (n1:t) (n2:t) : bool = ...

    let decr (n:t) : t =
      match t with
        Zero -> Neg 1
      | Pos n when n > 1 -> Pos (n-1)
      | Pos n when n = 1 -> Zero
      | Neg n -> Neg (n+1)
```

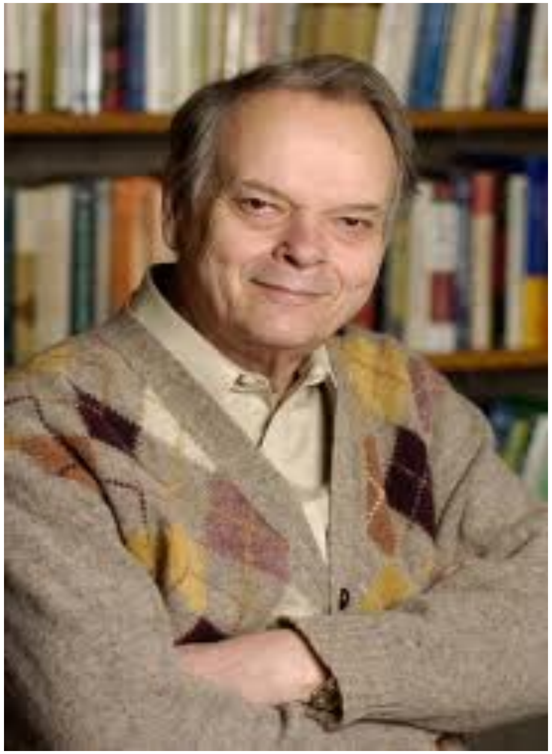
```
let inv (n:t) : bool =
   match n with
      Zero -> true
    | Pos n when n > 0 -> true
    | Neg n when n > 0 -> true
    | _ -> false
```

```
let abs(n:t) : int =
  match t with
    Zero -> 0
  | Pos n -> n
  | Neg n -> abs n
```

Serial Killer or PL Researcher?



Serial Killer or PL Researcher?



John Reynolds: super nice guy. Discovered the polymorphic lambda calculus. (OCaml with just functions)

Developed Relational Parametricity: A technique for proving the equivalence of modules.

Luis Alfredo Garavito: super evil guy. In the 1990s killed between 139-400+ children in Columbia. According to wikipedia, killed more individuals than any other serial killer. Due to Columbian law, only imprisoned for 30 years; decreased to 22.

Final Summary

Representation invariants define the valid implementations of an abstract data type

- Assume the invariant on inputs; prove it on outputs
- To debug, implement the invariant function
  - apply it on abstract inputs and outputs to find violations

Abstraction functions define the relationship between a concrete implementation and the abstract view of the client

- We should prove concrete operations implement abstract ones

We prove any two modules are equivalent by

- Defining a relation between values of the modules with abstract type
- We get to assume the relation holds on inputs; prove it on outputs

Rep invs and "is_related" predicates are called "logical relations"

END