Proving the Equivalence of Two Modules

COS 326

Speaker: David Walker
Princeton University

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 should be defined in terms of those abstract values
- *How are these abstract values connected to the implementation?*

Abstraction

user's view:

sets of integers

{1, 2, 3}

{4, 5}

{ }

implementation view:

[1; 1; 2; 3; 2; 3]

[ ]

[4, 5]

[4, 5, 5]

[1; 2; 3]

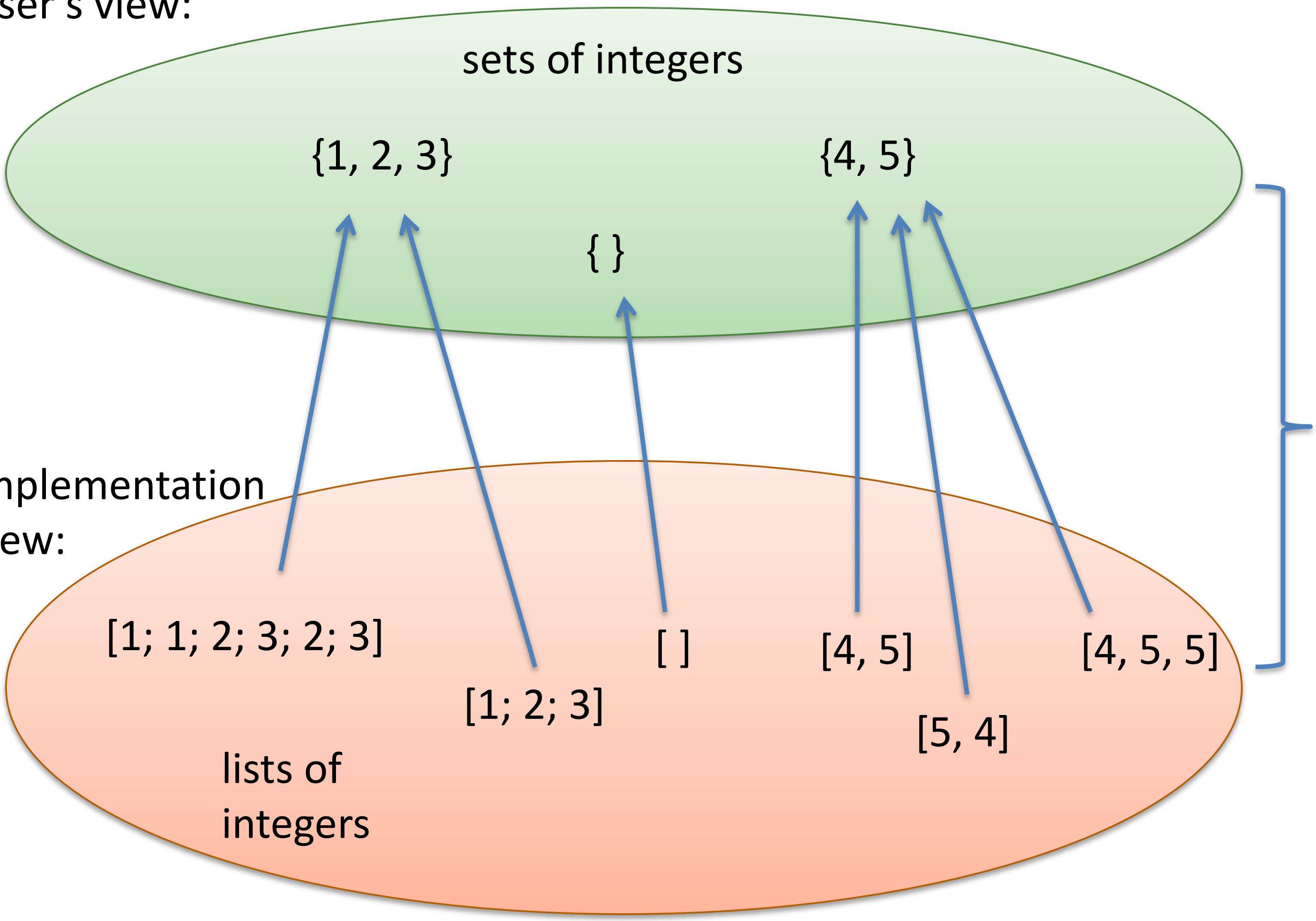
[5, 4]

lists of integers

Abstraction

user's view:

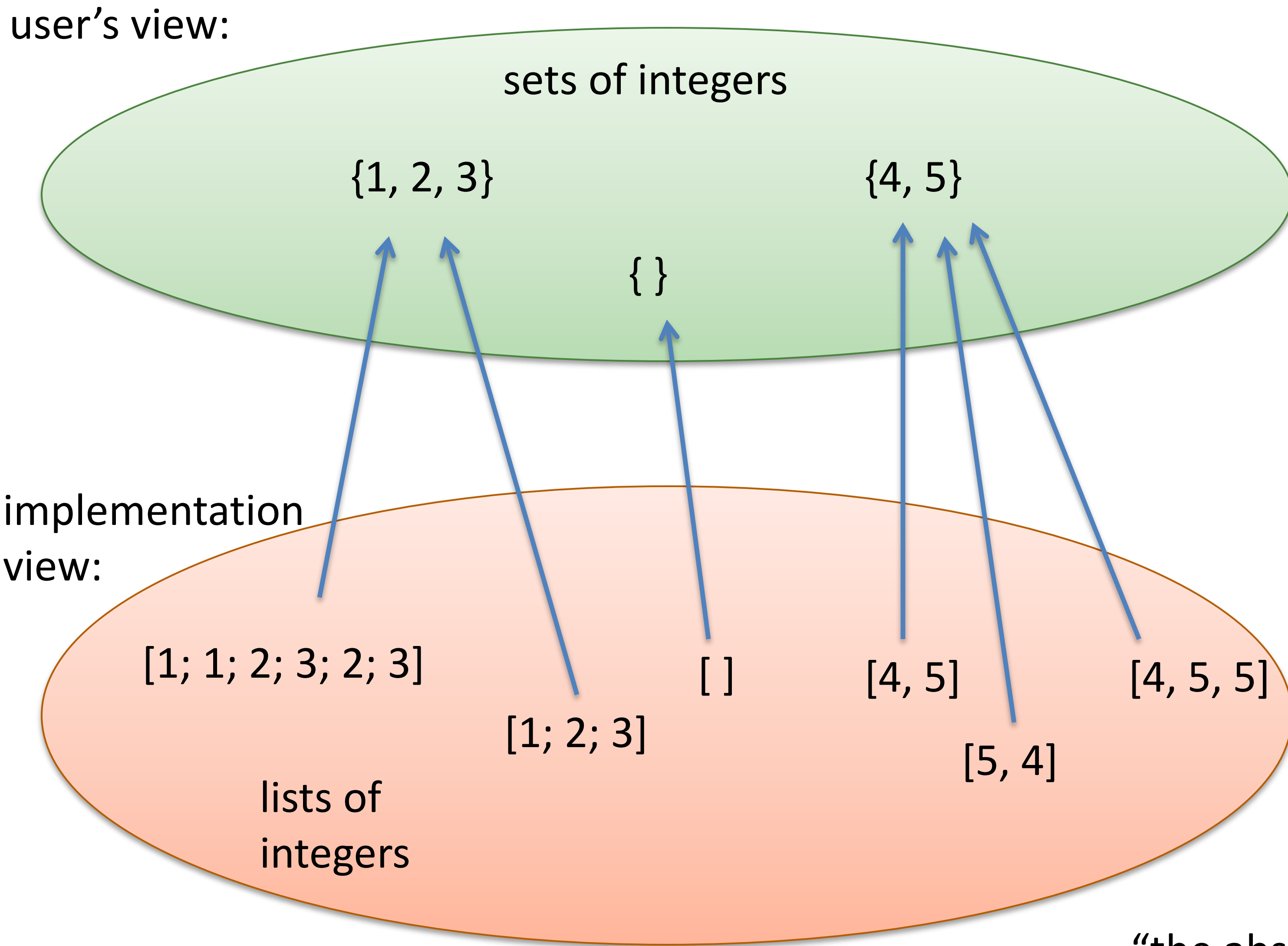
implementation view:



there's a relationship here, of course!

we are trying to *implement* the *abstraction*

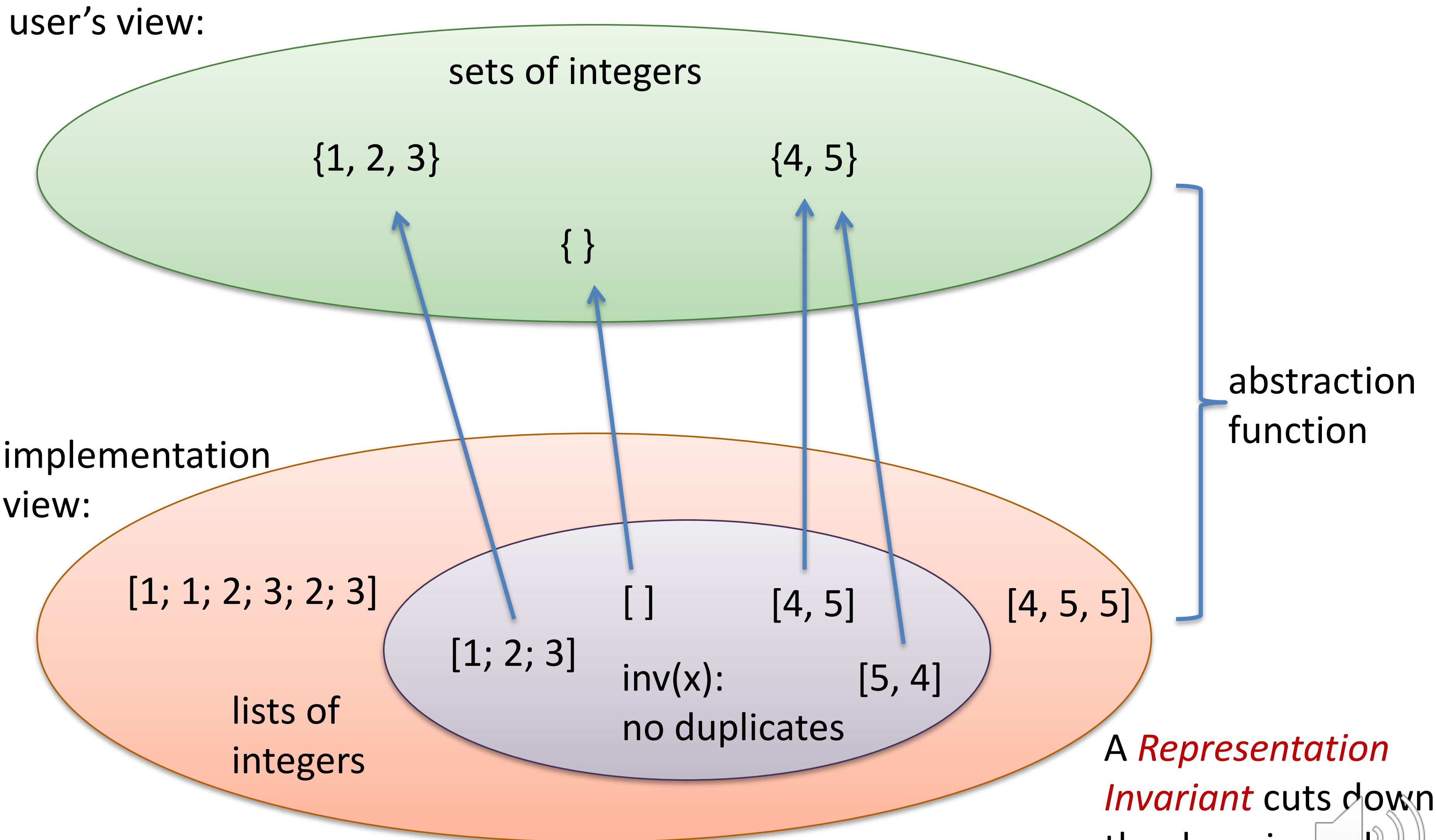
Abstraction



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

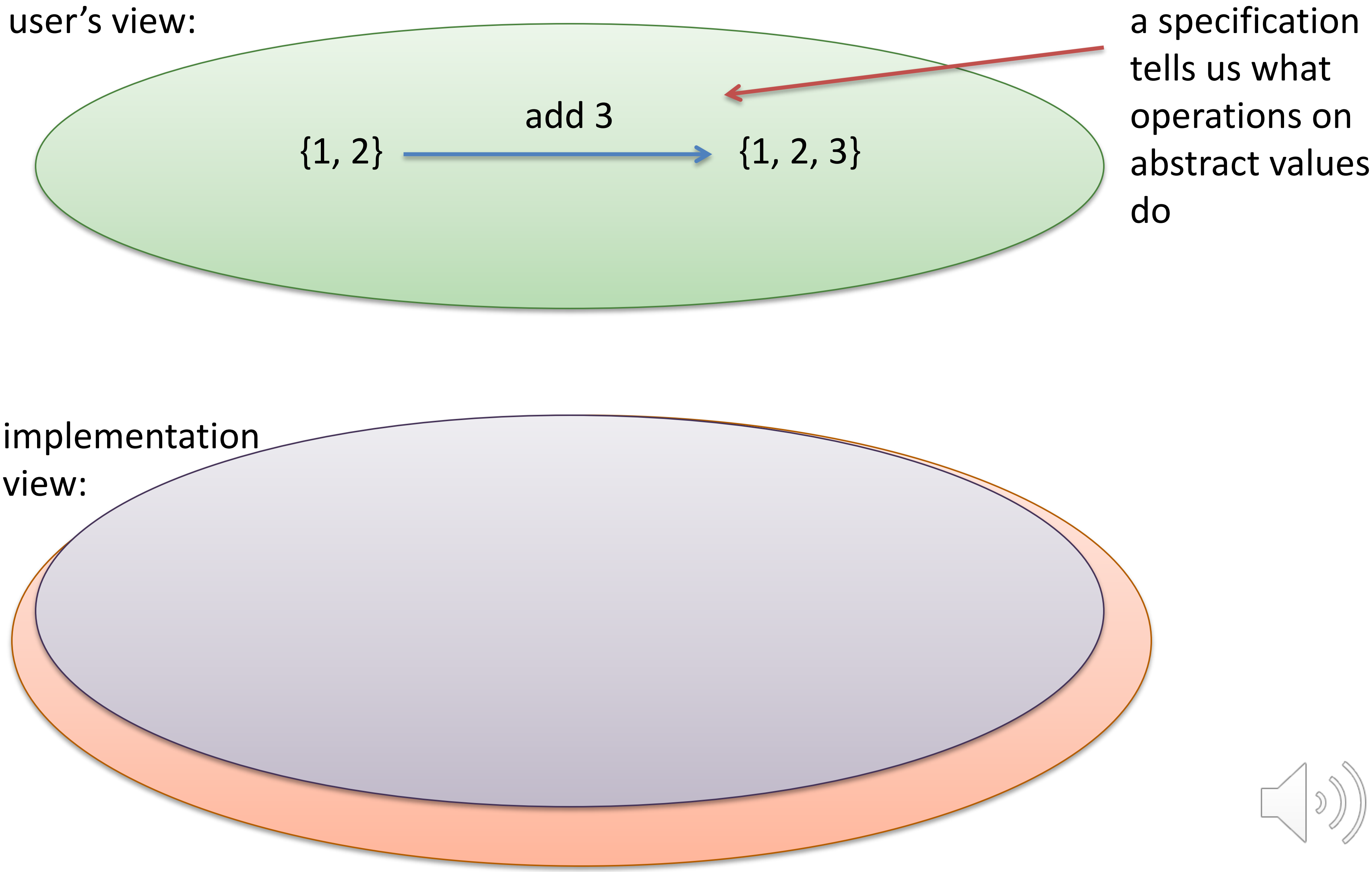
function called “the abstraction function”

Abstraction

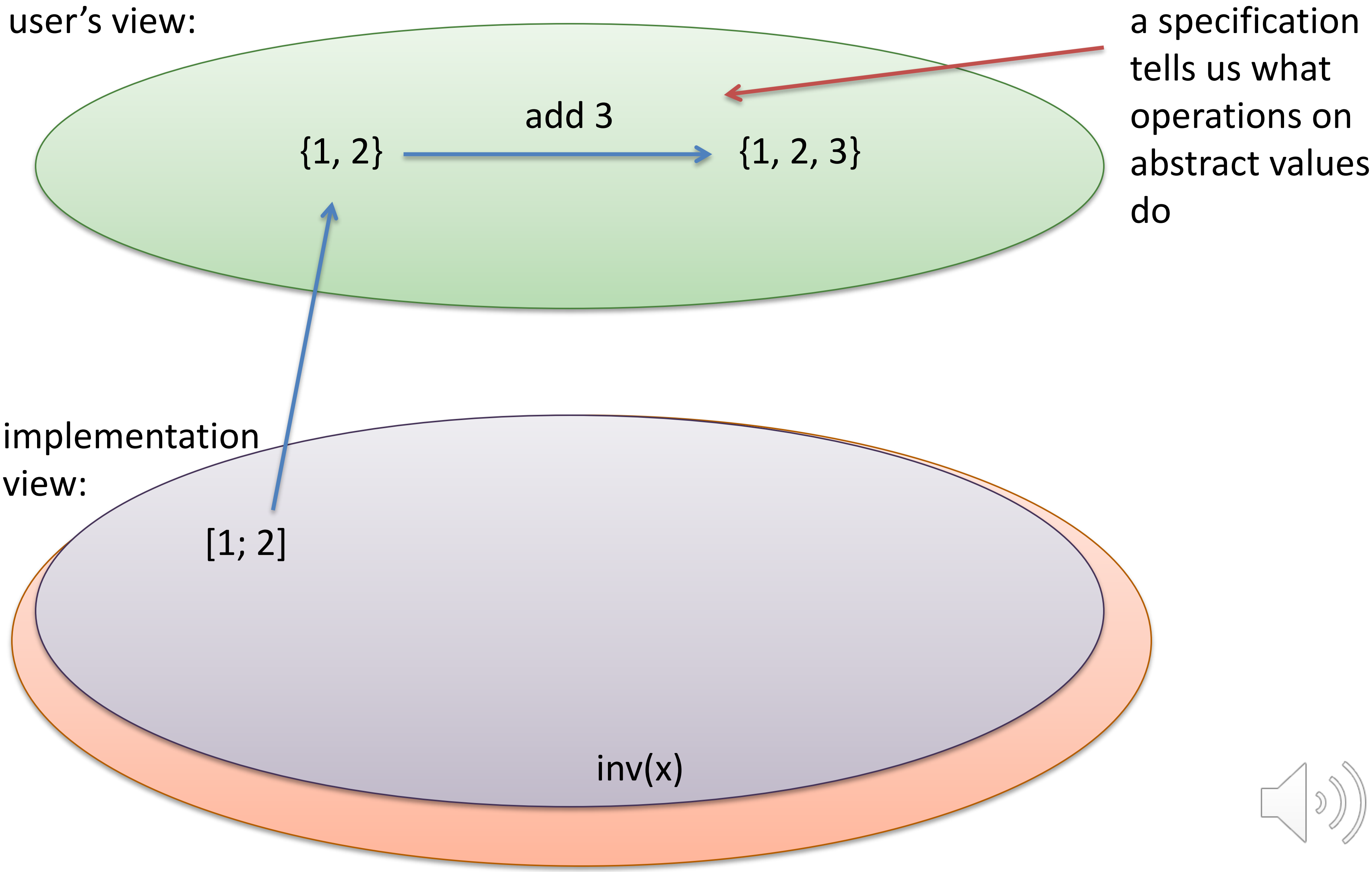


A *Representation Invariant* cuts down the domain of the abstraction function

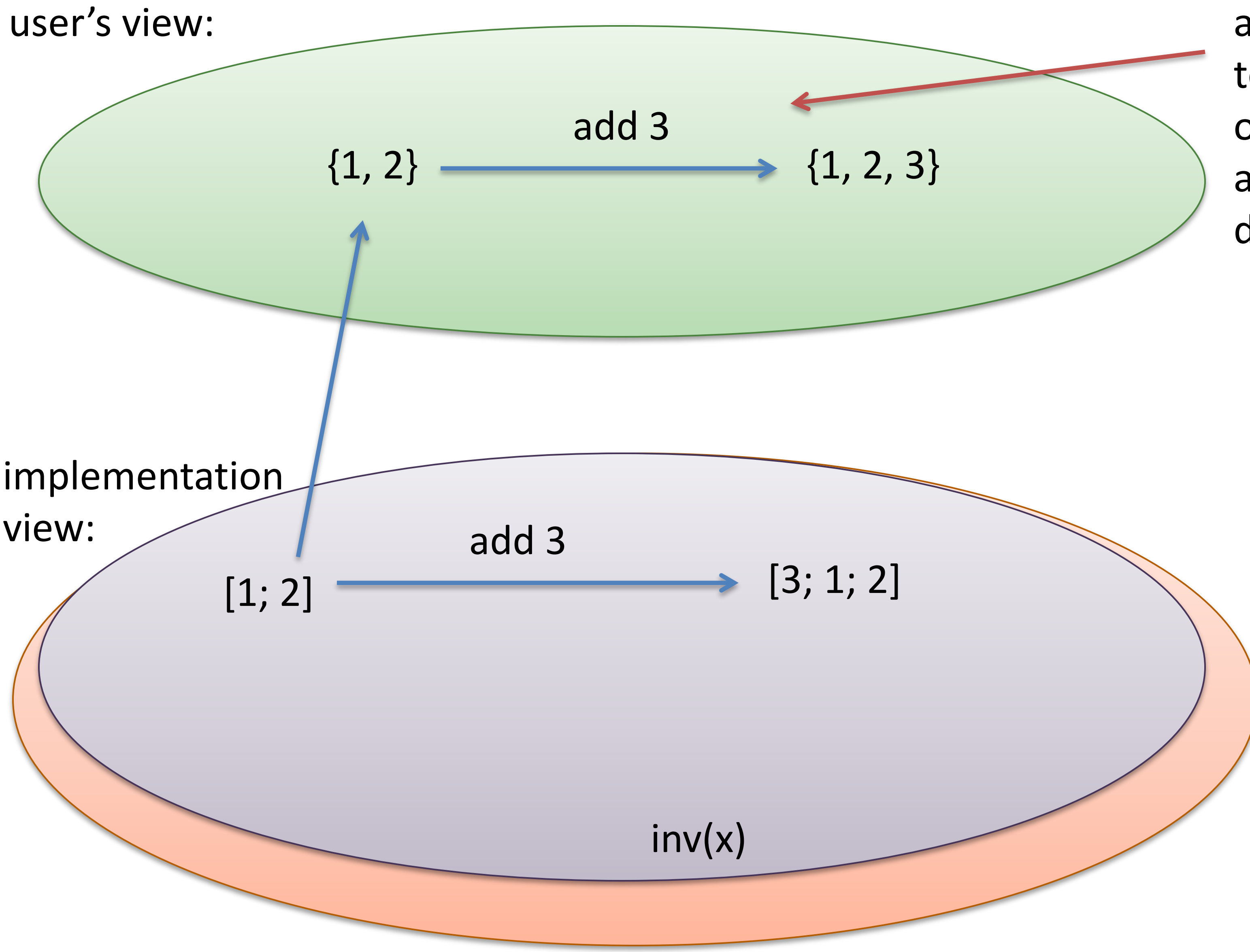
Specifications



Specifications

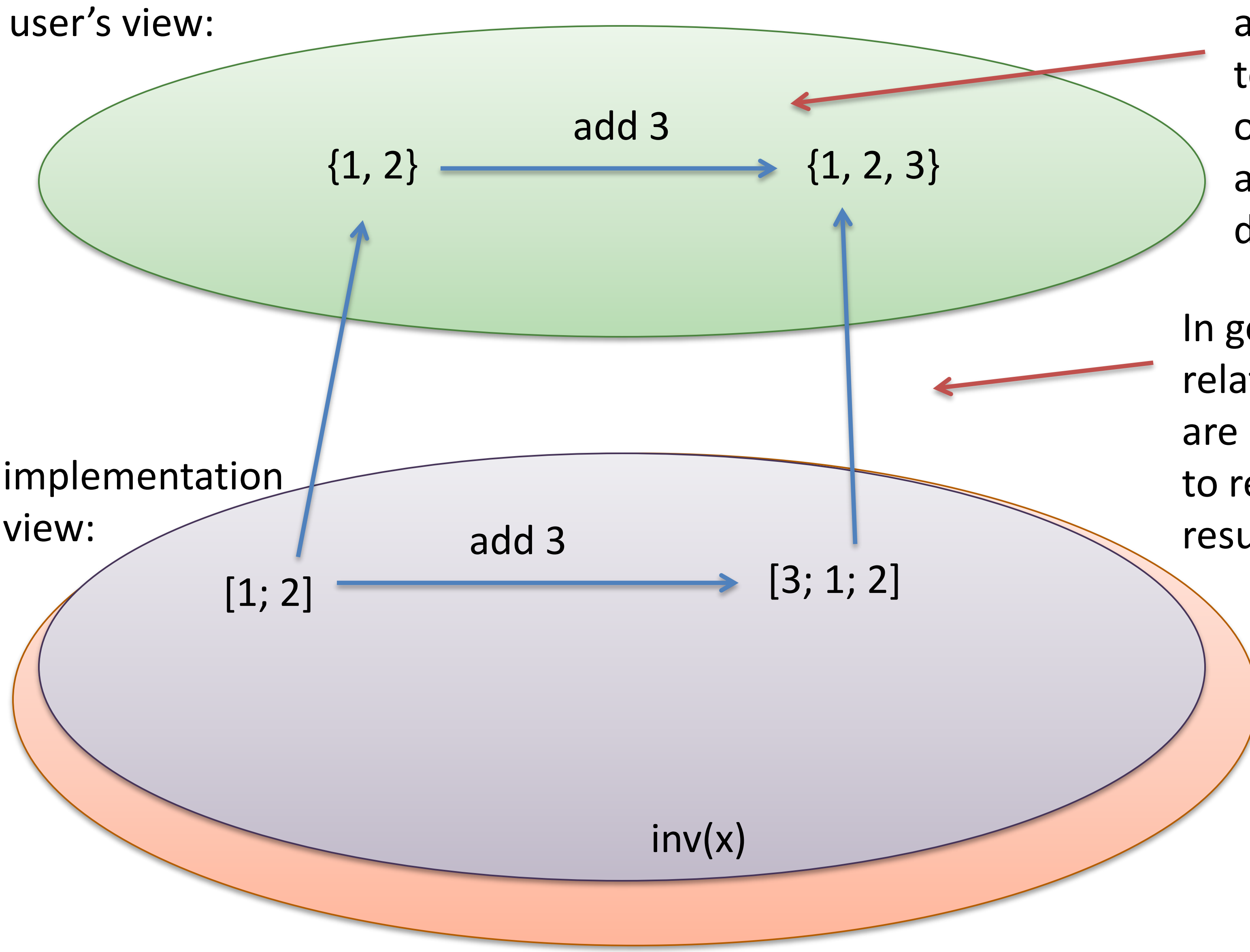


Specifications



a specification tells us what operations on abstract values do

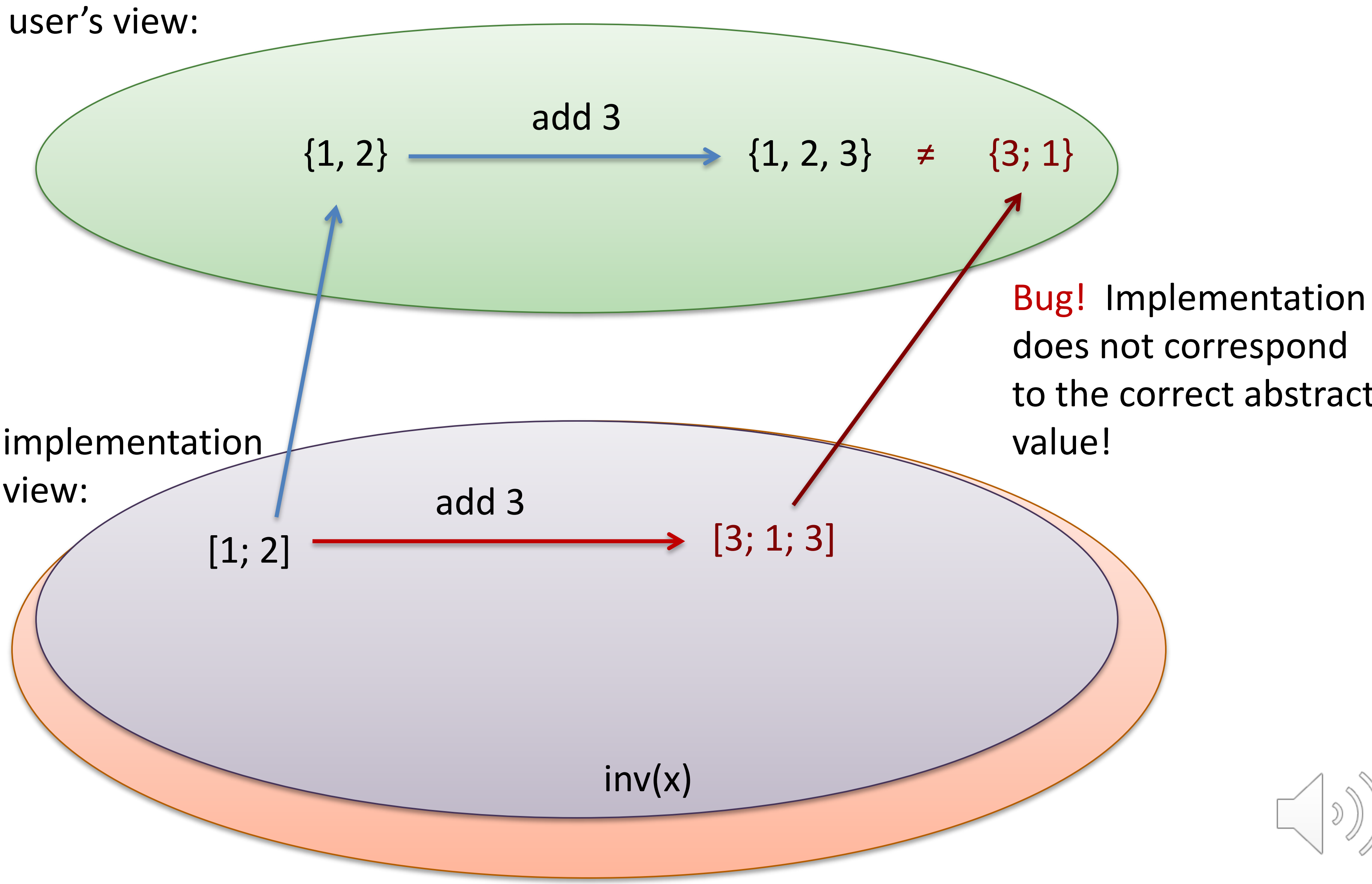
Specifications



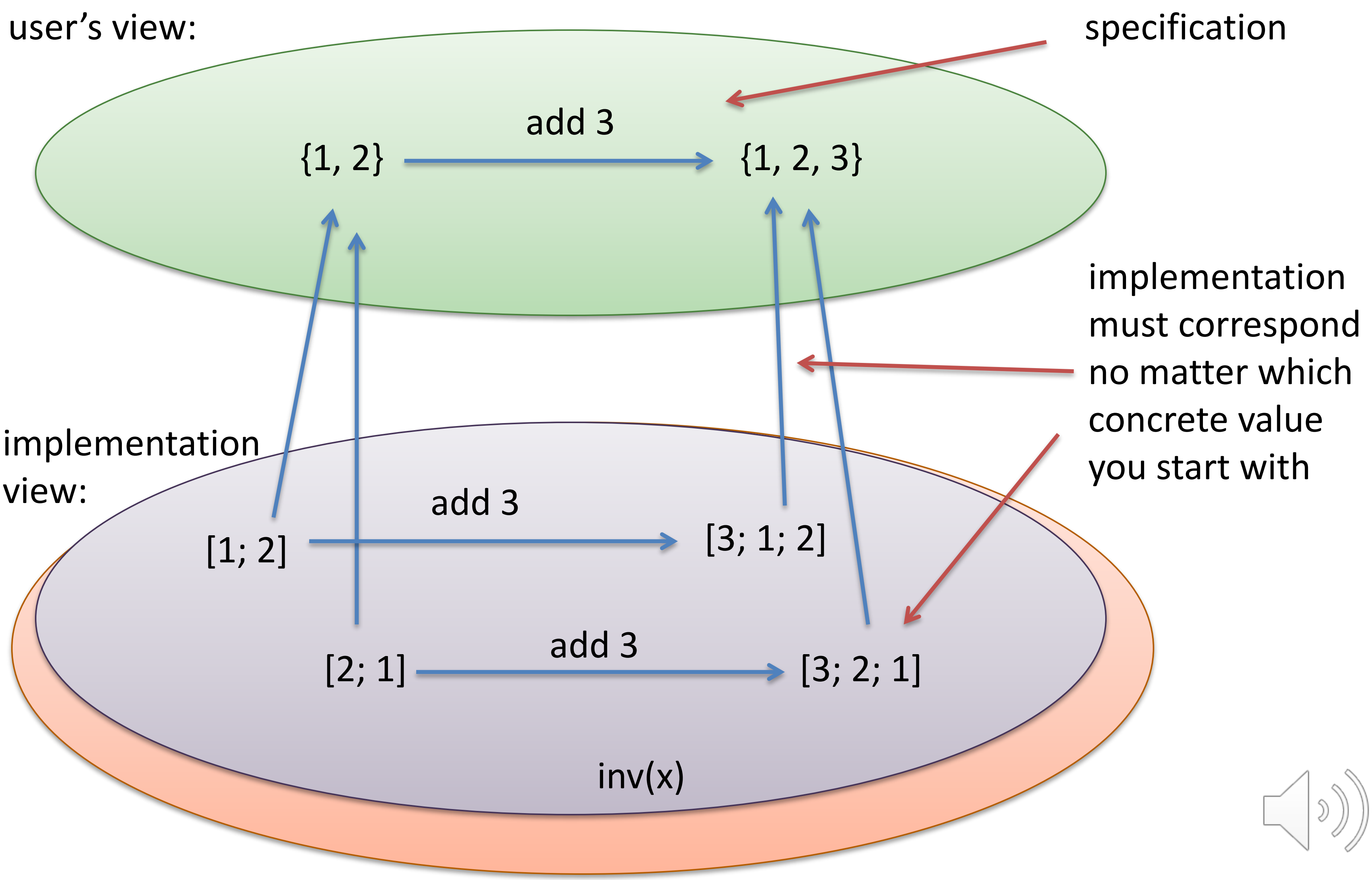
a specification tells us what operations on abstract values do

In general: related arguments are mapped to related results

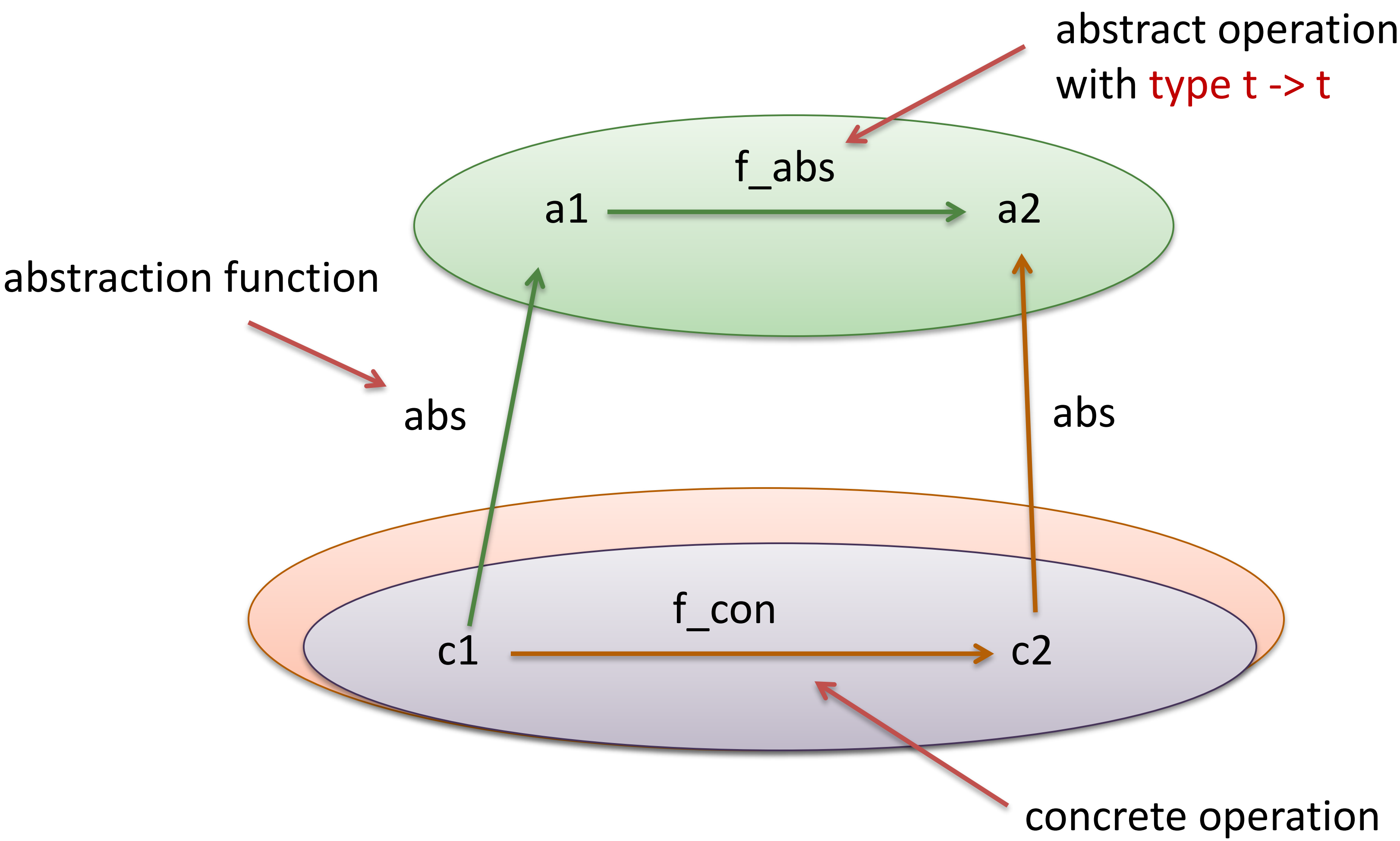
Specifications



Specifications



A more general view



to prove:

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

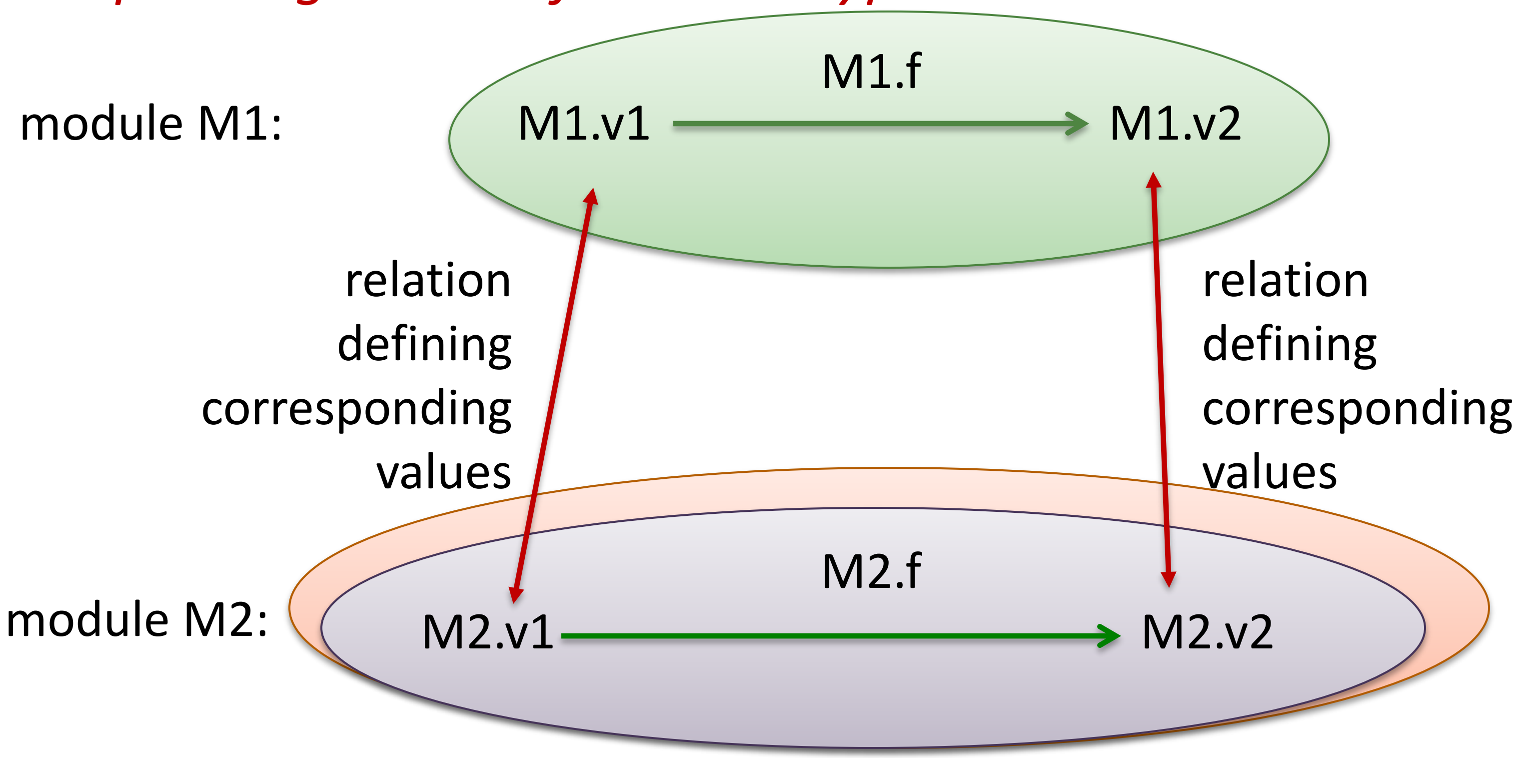
abstract then apply the abstract op == apply concrete op then abstract

Another Viewpoint

A specification is really just another implementation (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.*



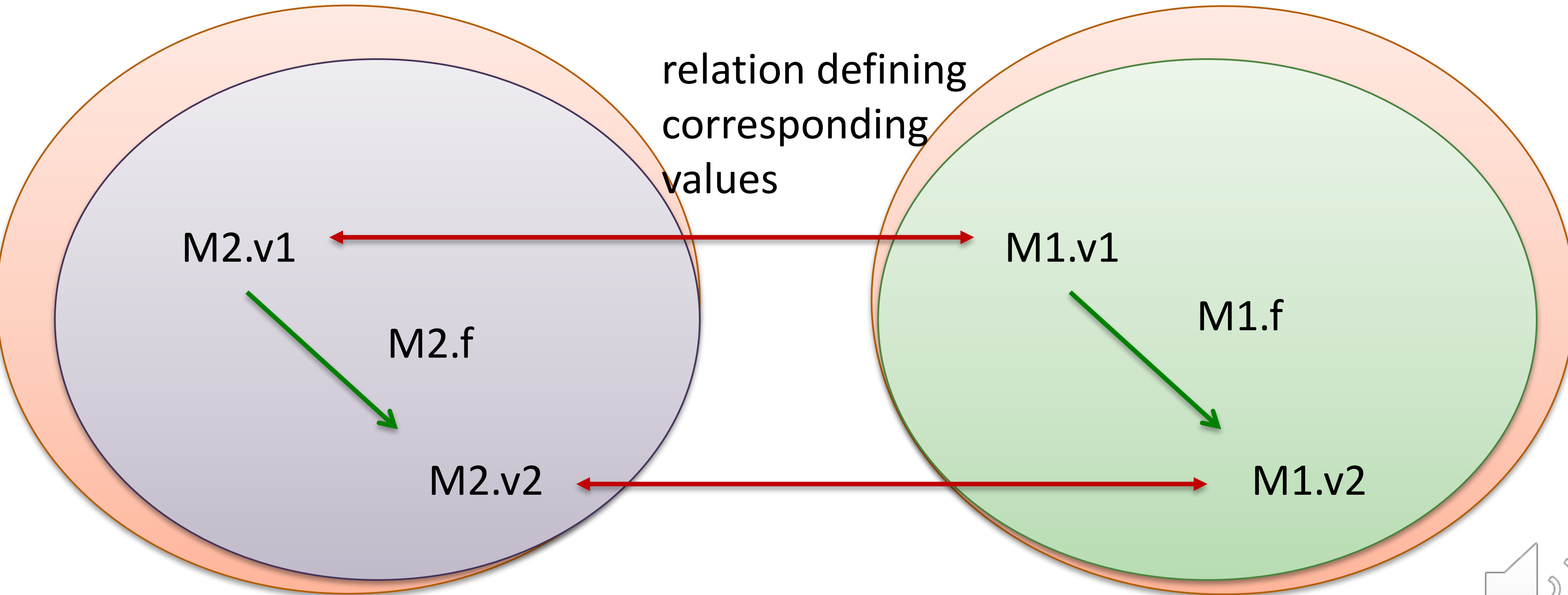
We ask: Do operations like f take related arguments to related results?

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!

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!

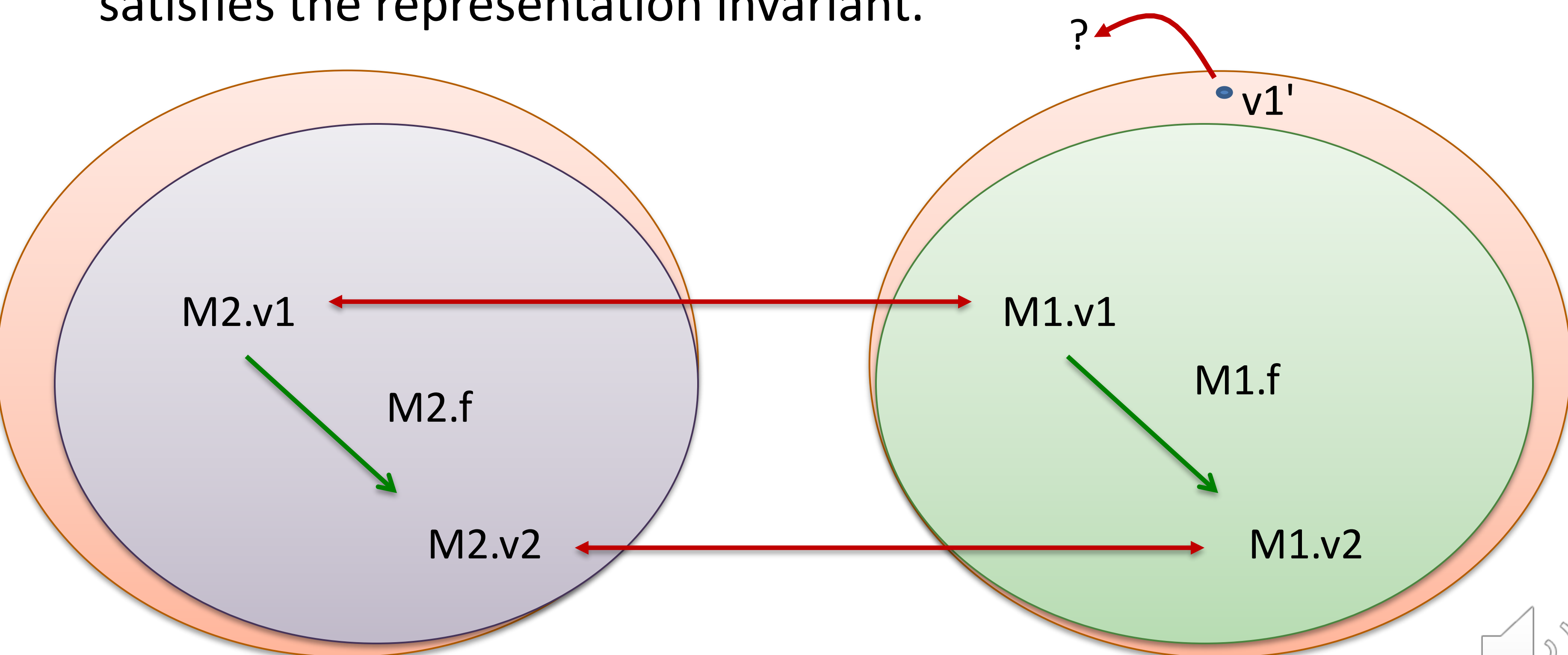
- if M1.v and M2.v have abstract type t,
  - we want is_related(M1.v, M2.v) to be true

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

Relations may imply the Rep Inv

When defining our relation, we will often do so in a way that implies the representation invariant.

ie: a value in M1 will not be related to any value in M2 unless it satisfies the representation invariant.



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) *&& x1 >= 0 && even x2*

is_related (x1, x2) implies *x1 >= 0*

rep inv for M1

is_related (x1, x2) implies *even x2 && x2 > 0*

rep inv for M2

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
```

But For Now:

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

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        (substitution)
```

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) at type s1 then we prove
    - is_related(v1 arg1, v2 arg2) at type s2
  - 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

MODULES WITH DIFFERENT IMPLEMENTATION TYPES

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
```

Reasoning about Module Equivalence

```
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 x = x + 1
    let reveal x = x
  end
```

```
module M2 : S =
  struct
    type t = Zero | S of t
    let zero = Zero
    let bump x = S x
    let rec reveal x =
      match x with
      | Zero -> 0
      | S x -> 1 + reveal x
  end
```

The Same Principle Applies!

Two modules with abstract type t will be declared equivalent if:

- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*

If we do indeed show the relation is "preserved" by operations of the module (an idea that depends crucially on the *signature* of the module) then *no client will ever be able to tell the difference between the two modules even though their data structures are implemented by completely different types*!

Reasoning about Module Equivalence

```
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 x = x + 1
    let reveal x = x
 end
```

```
module M2 : S =
  struct
    type t = Zero | S of t
    let zero = Zero
    let bump x = S x
    let rec reveal x =
       match x with
       | Zero -> 0
       | S x -> 1 + reveal x
  end
```

```
is_related (x1, x2) =
  x1   ==   M2.reveal x2
```

Module Abstraction

John Reynolds, 1935-2013

Discovered the polymorphic lambda calculus (first polymorphic type system).

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

Summary: Abstraction and Equivalence

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

- We should prove concrete operations implement abstract ones described to our customers/clients

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 invariants and "is_related" predicates are called logical relations