Modules, Representation Invariants, and Equivalence

COS 326
David Walker
Princeton University

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

Last Time: Representation Invariants

A *representation invariant* inv(v) is a property that holds of all values of abstract type.

Representation invariants can be used during debugging:

- check your outputs:
  - call inv(v) on all outputs from the module of type t
- if check all outputs, then should not *need* to check inputs!
  - but you can, just in case you missed an output!

Proving representation invariants involves (roughly):

- Assuming invariants hold on inputs to functions
- Proving they hold on outputs to functions

A Higher Order Example

```
module type NAT =
  sig

   type t

   val from_int : int -> t

   val root : t -> t

   val fumble : (t -> t) -> t -> t

   val foo : (t -> t) -> t

 end
```

31-bit
natural numbers

```
module Nat31 : NAT = struct
   type t = int

   let from_int (n:int) : t =
     if n <= 0 then 0 else n

   let root n = assert n >= 0; ...

   let rec fumble f n = f (root n)

   let foo f = f (-1)

end
```

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

A Higher Order Example

```
module type NAT =
 sig

  type t

  val from_int : int -> t

  val root : t -> t

  val fumble : (t -> t) -> t -> t

  val foo : (t -> t) -> t

 end
```

```
module Nat31 : NAT = struct
   type t = int

   let from_int (n:int) : t =
     if n <= 0 then 0 else n

   let root n = assert n >= 0; ...

   let rec fumble f n = f (root n)

   let foo f = f (-1)

end
```

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

client code leading failed assert:

- foo (root)
- foo (fumble root)

A Higher Order Example

```
module type NAT =
  sig

    type

    val fr

    val ro

    val fu

    val fo

  end
```

```
module Nat31 : NAT = struct
    type t = int

    let from_int (n:int) : t =
      if n <= 0 then 0 else n

    let root n = assert n >= 0; ...

    let rec fumble f n = f (root n)

    let foo f = f (-1)

end
```

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

Why does this happen?

The module does not preserve the representation invariant!

When proving foo preserves the invariant, we must show it produces a value that is valid for type t.

We can assume f produce a value valid for type t if it is supplied a value that is valid for type t ... but f is not supplied such a value! It is supplied -1, which does not satisfy the rep inv

Score Keeping

C: -10

OCaml: 12

Java: -2

C++: depends on the version

Score Keeping

C: -10

Dave: -1

ava: -2

C++: depe

the vers

Recall from last time:

Slide snippet:

val size : t -> int

must prove nothing

val const : int

must prove nothing

We really should be proving that these are *total functions*.
ie: that they don't cause a failure on the way to producing a value.

That is quite a bit more than "nothing."

In all the other proofs we have done in the class, we've assumed we have been working with total functions so this hasn't been an issue.

However, the idea of a representation invariant is that our functions with type t -> t are only produce values when inputs v:t satisfy the invariant. In other words, they are *partial functions*.

A Higher Order Example

```
module type NAT =
  sig

    type t

    val from_int : int -> t

    val root : t -> t

    val fumble : (t -> t) -> t -> t

    val foo : (t -> t) -> t

 end
```

```
module Nat31 : NAT = struct
    type t = int

    let from_int (n:int) : t =
      if n <= 0 then 0 else n

    let root n = assert n >= 0; ...

    let rec fumble f n = f (root n)

    let foo f = f (-1)

end
```

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

bad thing isn't really due to result type

client code leading failed assert:

- foo (root)
- foo (fumble root)

A Higher Order Example

```
module type NAT =
 sig

  type t

  val from_int : int -> t

  val root : t -> t

  val fumble : (t -> t) -> t -> t

  val foo : (t -> int) -> int

 end
```

```
module Nat31 : NAT = struct
   type t = int

   let from_int (n:int) : t =
     if n <= 0 then 0 else n

   let root n = assert n >= 0; ...

   let rec fumble f n = f (root n)

   let foo f = f (-1)

end
```

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

something bad still happens

client code leading failed assert:

- foo (fun x -> root x; 0)

Moral of the Story

If a function has type t -> int, we should prove it is total

- a total function is one that *will* produce a value and won't fail

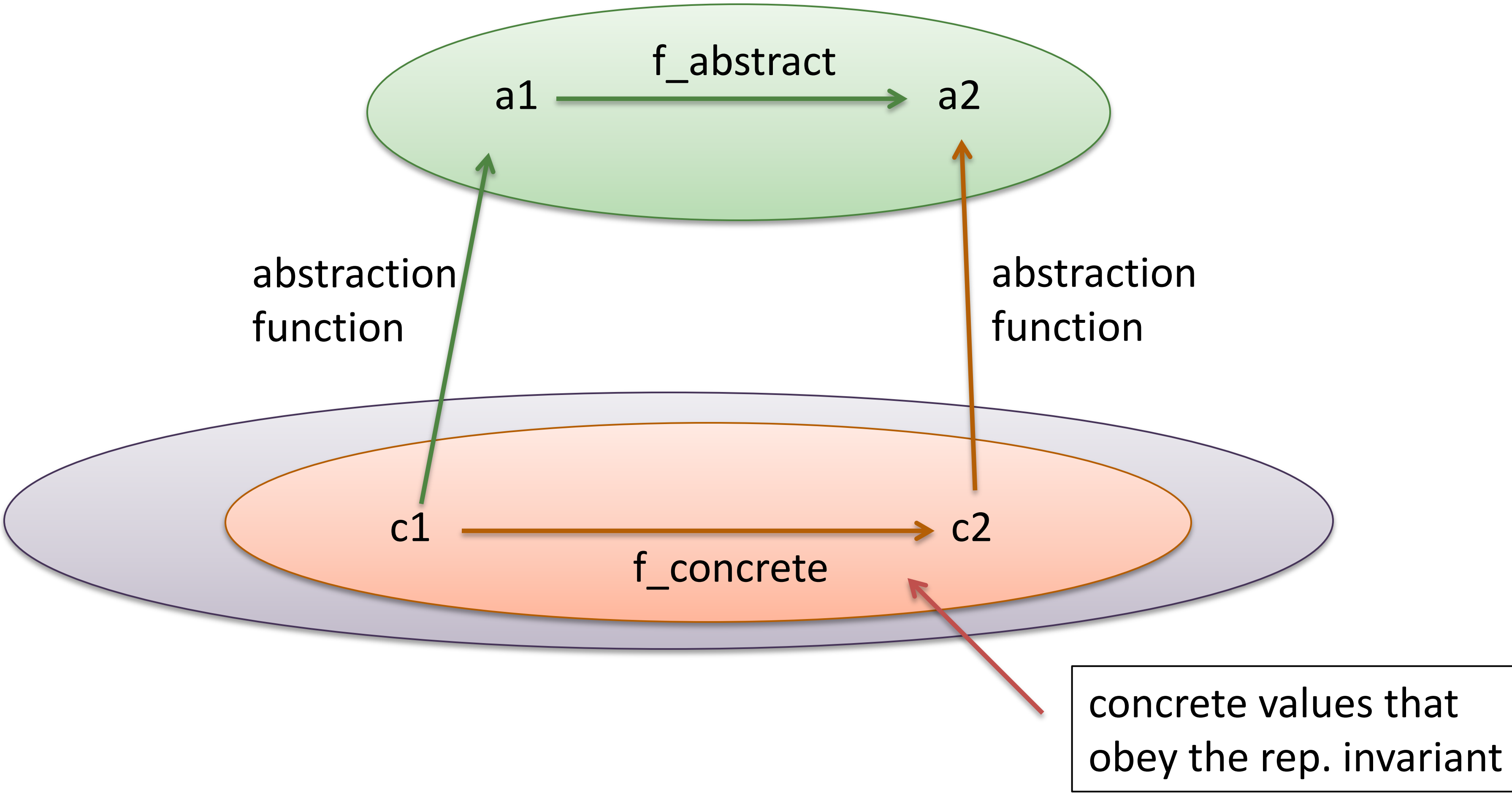
We should prove other functions, regardless of type are total too

What I want you to know:

- functions can call other functions with a module and doing so could violate their preconditions/rep invariants
- watch for higher-order functions too
- we *should* prove functions are total, *but I won't make you actually do proofs of totality on exams*.
  - this isn't hard, but we've got other things to learn too!
- but I want you to be able to pick out examples/problems where we define functions that aren't total and hence cause failures or violate rep invs

MODULE EQUIVALENCE

Last Time: Reasoning about Abstractions



To prove an abstraction is sound (ie, a faithful description of what is going on):
abstraction function then abstract op == *concrete op then abstraction function*

An abstraction function is just one kind of *relation* between two modules.

We can use the notion of *relations* between values to reason about the *equivalence* of 2 different implementations of an interface.

As we go along, watch for a very *similar pattern* to what we saw concerning *representation invariants*.

The difference is going to be that representation invariants involve 1 module whereas module equivalence involves 2 modules.

This “pattern” is known as a *logical relation*.

Recall Expression Equivalence

Two expressions e1 and e2 are equivalent when:

- e1 -->* v1 and e2 -->* v2 and v1 = v2,
- they both diverge, or
- they both raise the same exception

(When doing our proofs, we assumed all expressions terminate normally, so our proofs focused on situations where we needed to case 1 exclusively.)

Reasoning about Module Equivalence

Two expressions e1 and e2 are equivalent when:

- e1 -->* v1 and e2 -->* v2 and v1 = v2,
- they both diverge, or
- they both raise the same exception

When are two modules equivalent?

- We can't just ask M1.f x and M2.f x to return the "*same*" value
  - the values might not even have the same type!

Reasoning about Module Equivalence

Two expressions e1 and e2 are equivalent when:

- e1 -->* v1 and e2 -->* v2 and v1 = v2
- they both diverge
- they both raise the same exception

When are two modules equivalent?

- We can't just ask M1.f x and M2.f x to return the "same" value
  - the values might not even have the same type!

```
module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
end
```

Reasoning about Module Equivalence

Two expressions e1 and e2 are equivalent when:

- e1 -->* v1 and e2 -->* v2 and v1 = v2
- they both diverge
- they both raise the same exception

When are two modules equivalent?

- We can't just ask M1.f x and M2.f x to return the "same" value
  - the values might not even have the same type!

```
module type S =
  sig
    type t
    val zero : t
    val bump : t -> t
end
```

```
module M1 : S =
  struct
    type t = int
    let bump x = x + 1
  end
```

```
module M2 : S =
  struct
    type t = Zero | S of t
    let bump x = S x
  end
```

Reasoning about Module Equivalence

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

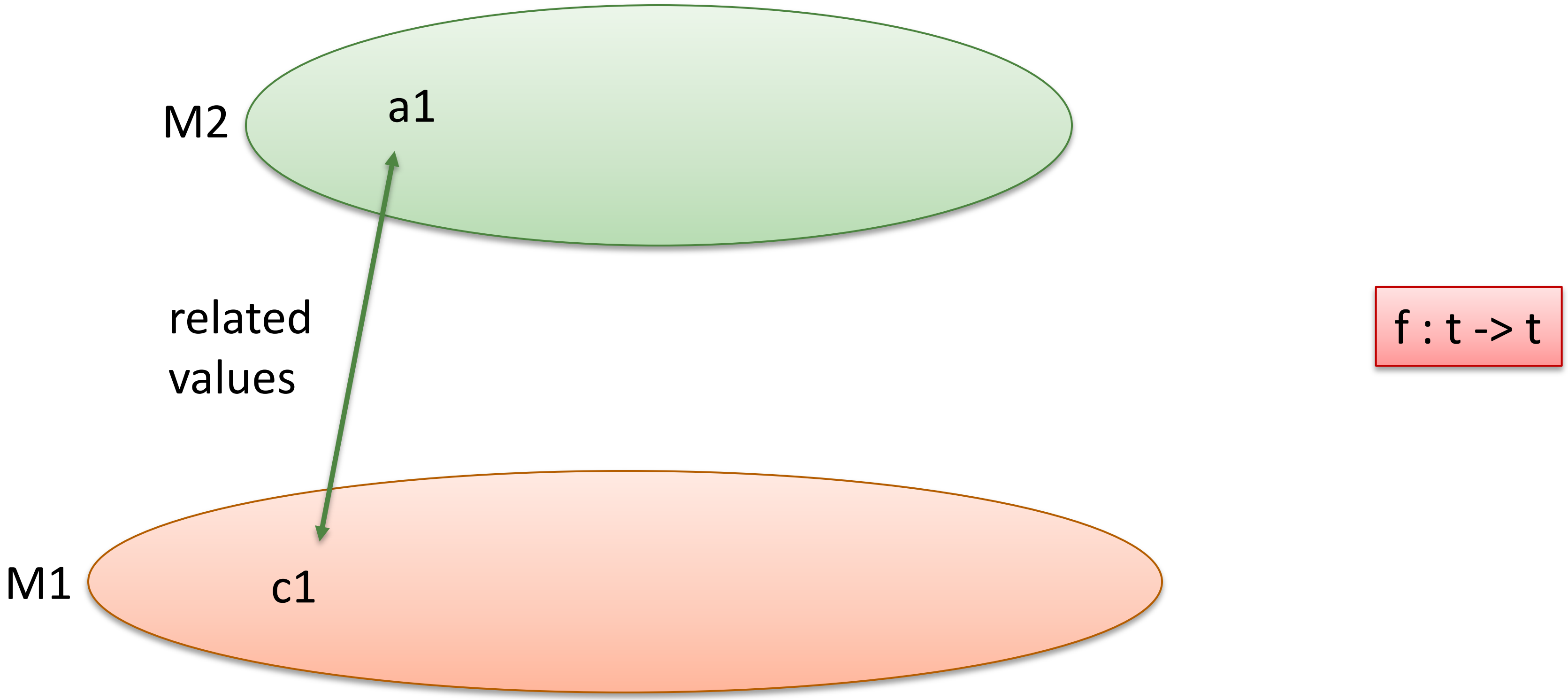
- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*

If we do indeed show the relation is "preserved" by operations of the module (an idea that depends crucially on the *types* of such operations) then *no client will ever be able to tell the difference between those two modules*!

What does it mean to “preserve” the relation

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

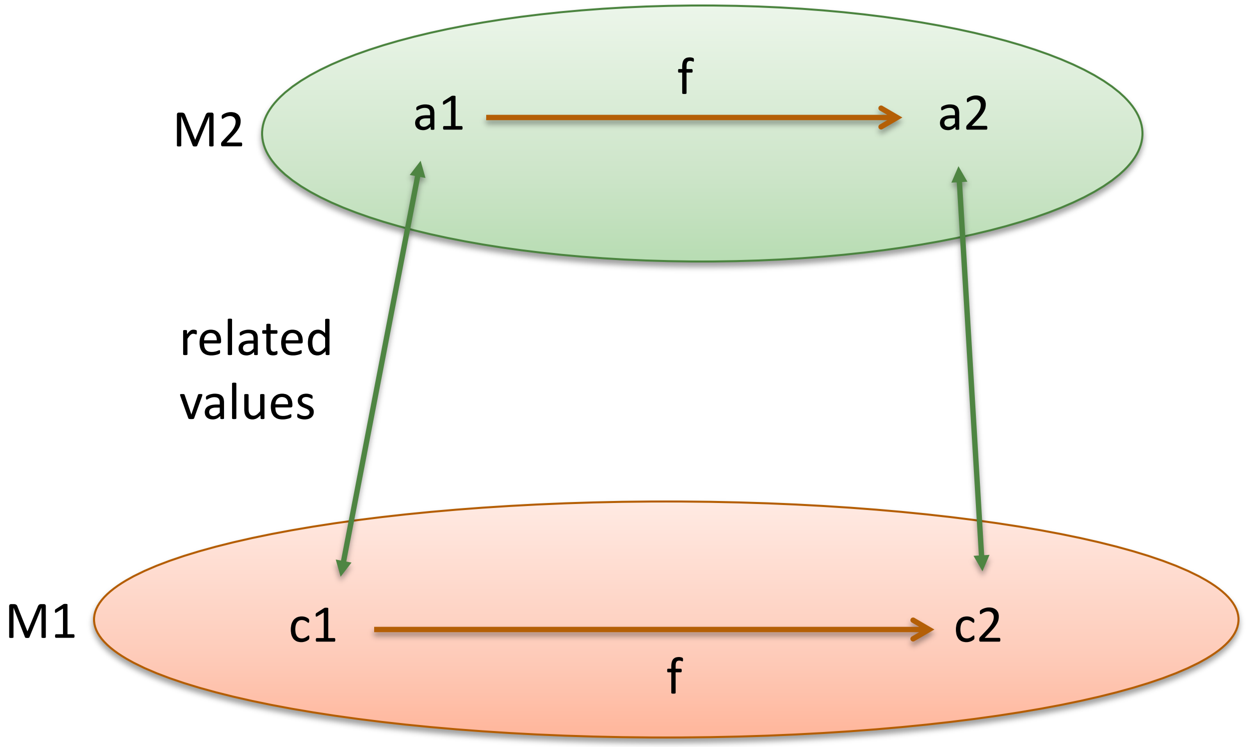
- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*



What does it mean to "preserve" the relation

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*

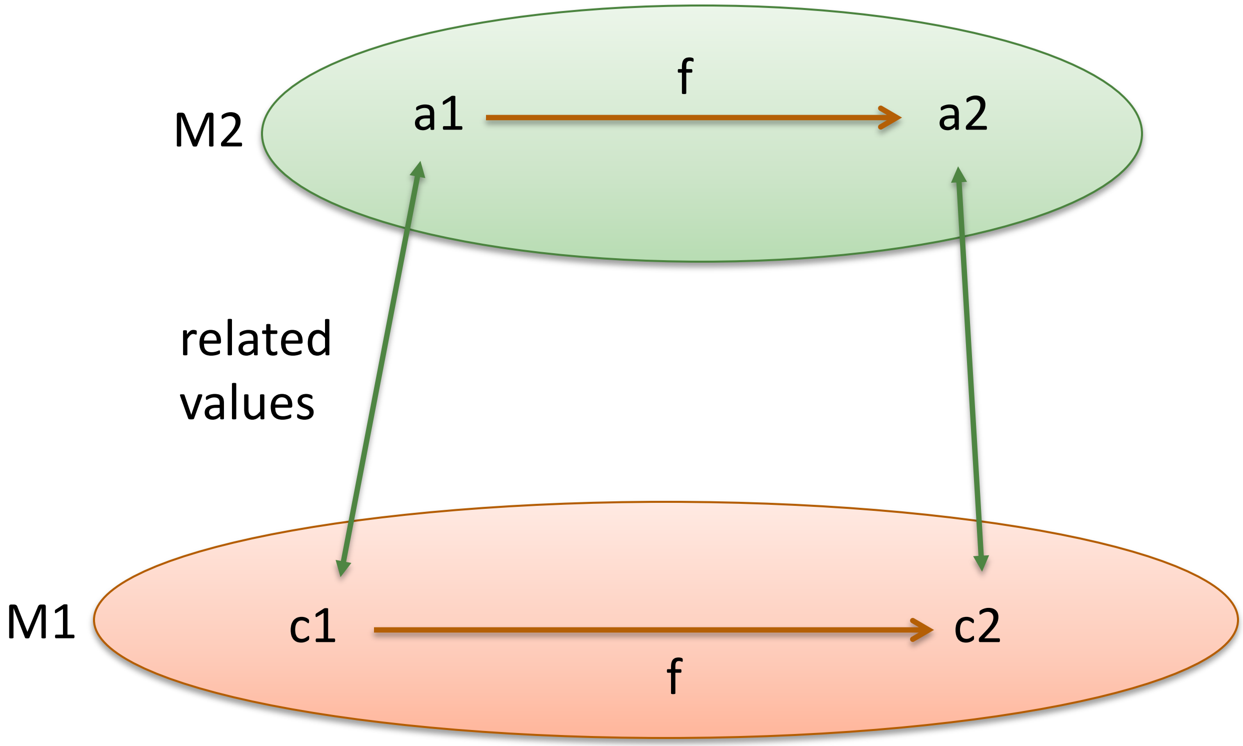


f : t -> t

What does it mean to “preserve” the relation

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*



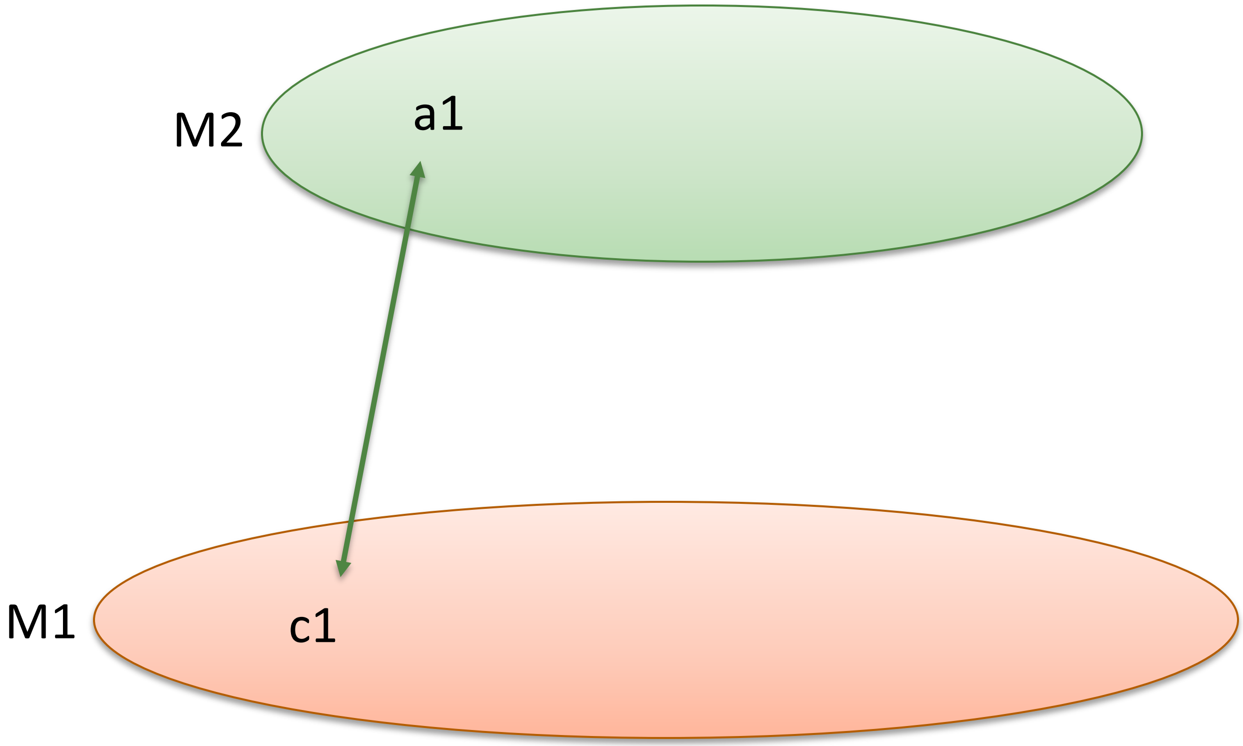
f : t -> t

if a1 and c1 are related, then M1’s version of f should produce a value a2 that is related to the value that M2’s version of f produces.

What does it mean to "preserve" the relation

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*

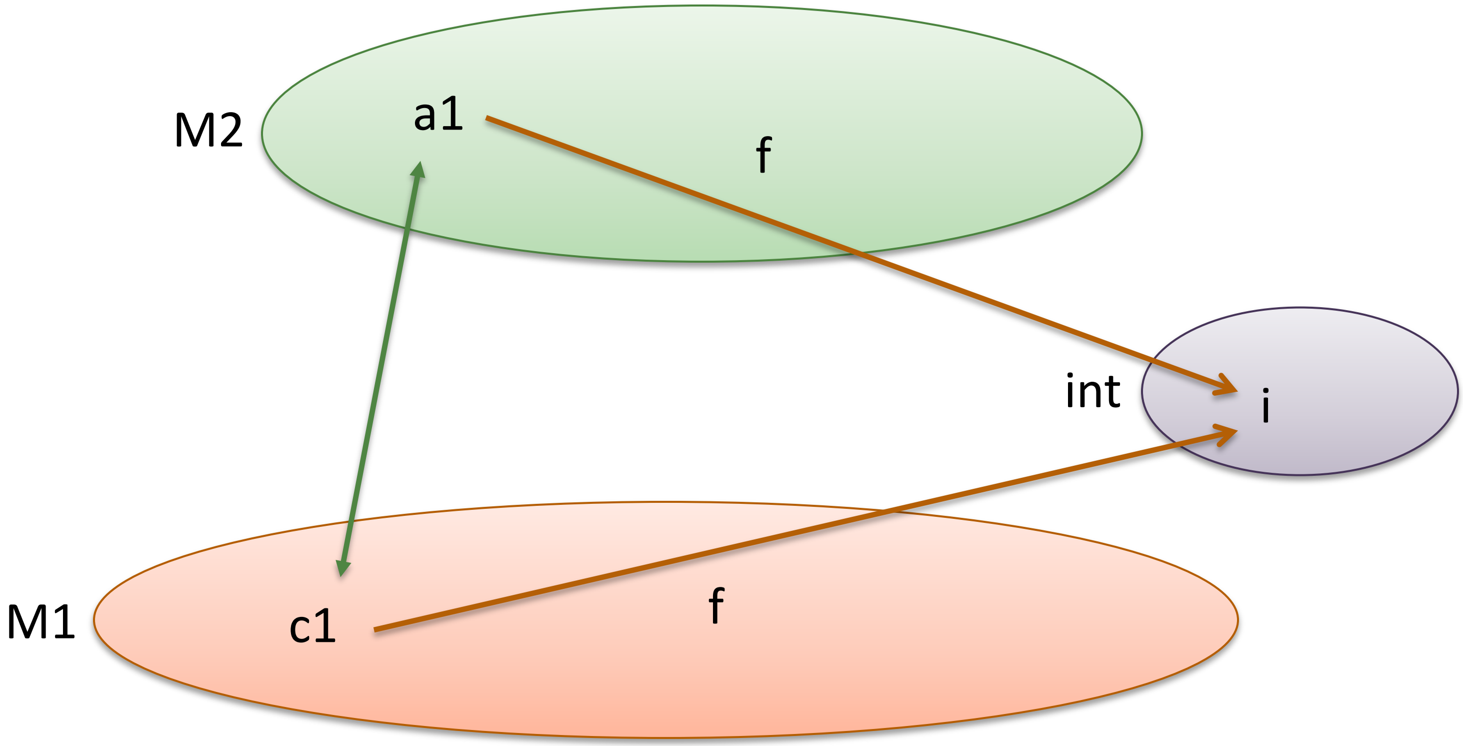


f : t -> int

What does it mean to "preserve" the relation

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*

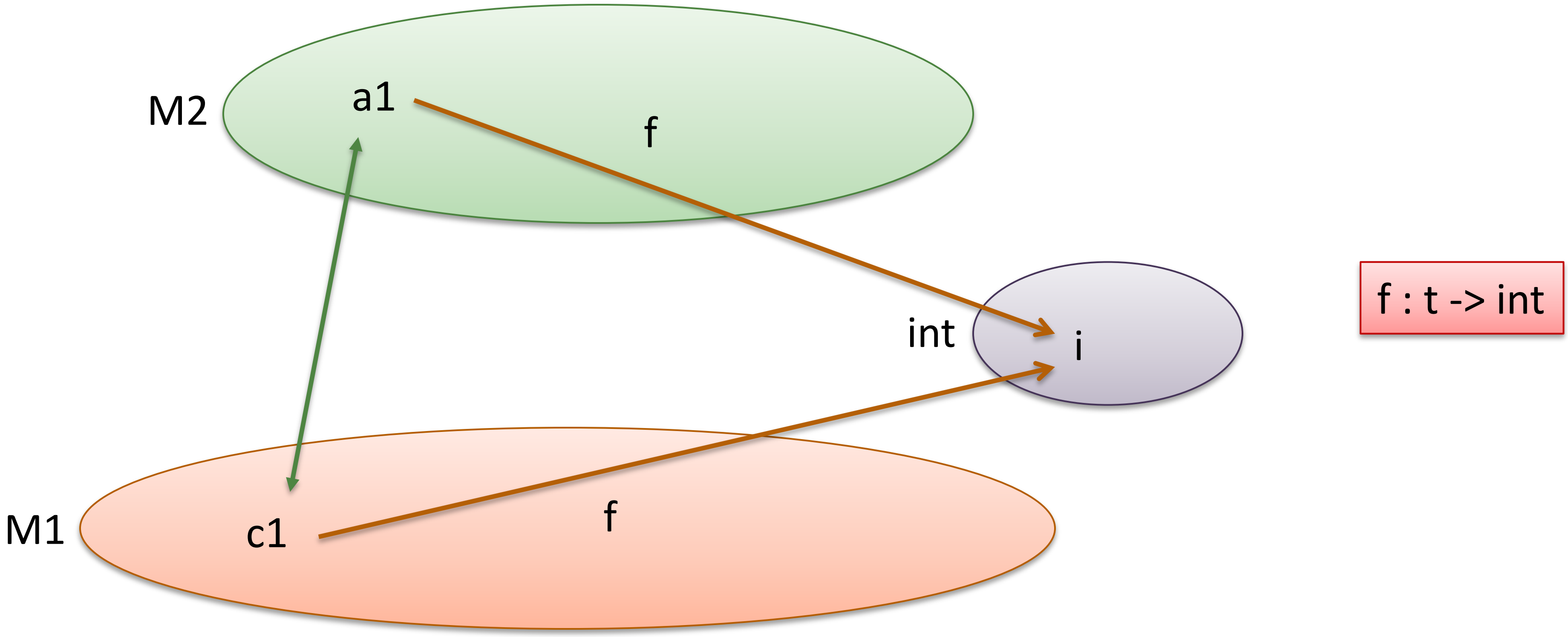


f : t -> int

What does it mean to “preserve” the relation

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*

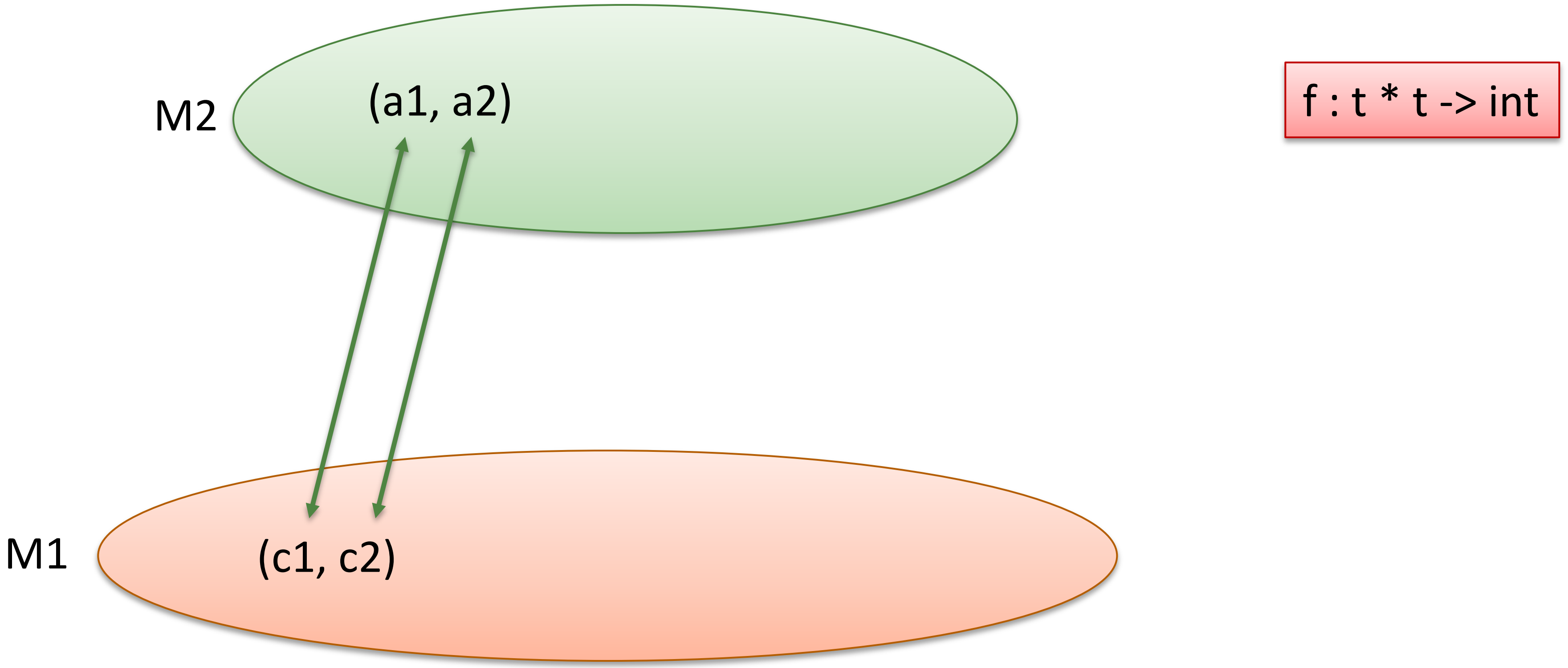


if a1 and c1 are related, then M1’s version of f should produce a value i that is *identical* to the value that M2 produces.

What does it mean to "preserve" the relation

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

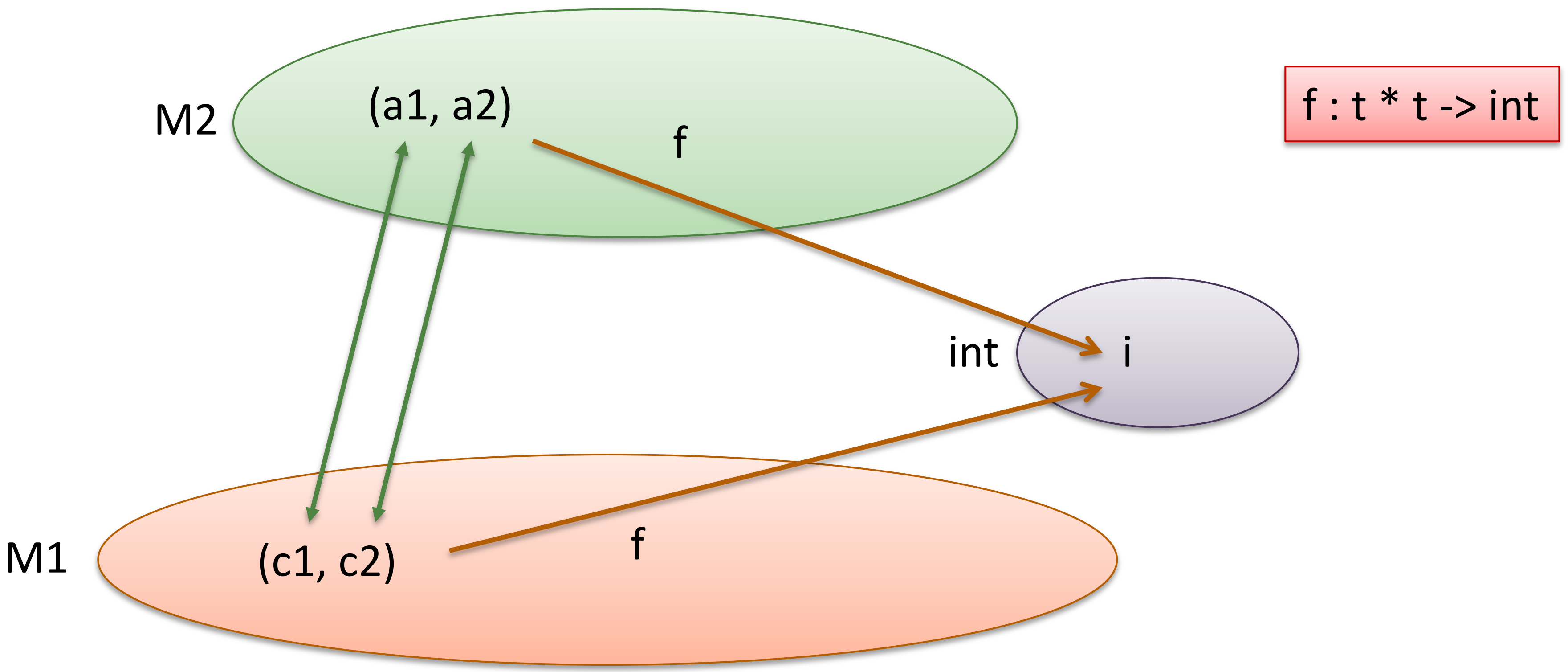
- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*



What does it mean to "preserve" the relation

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

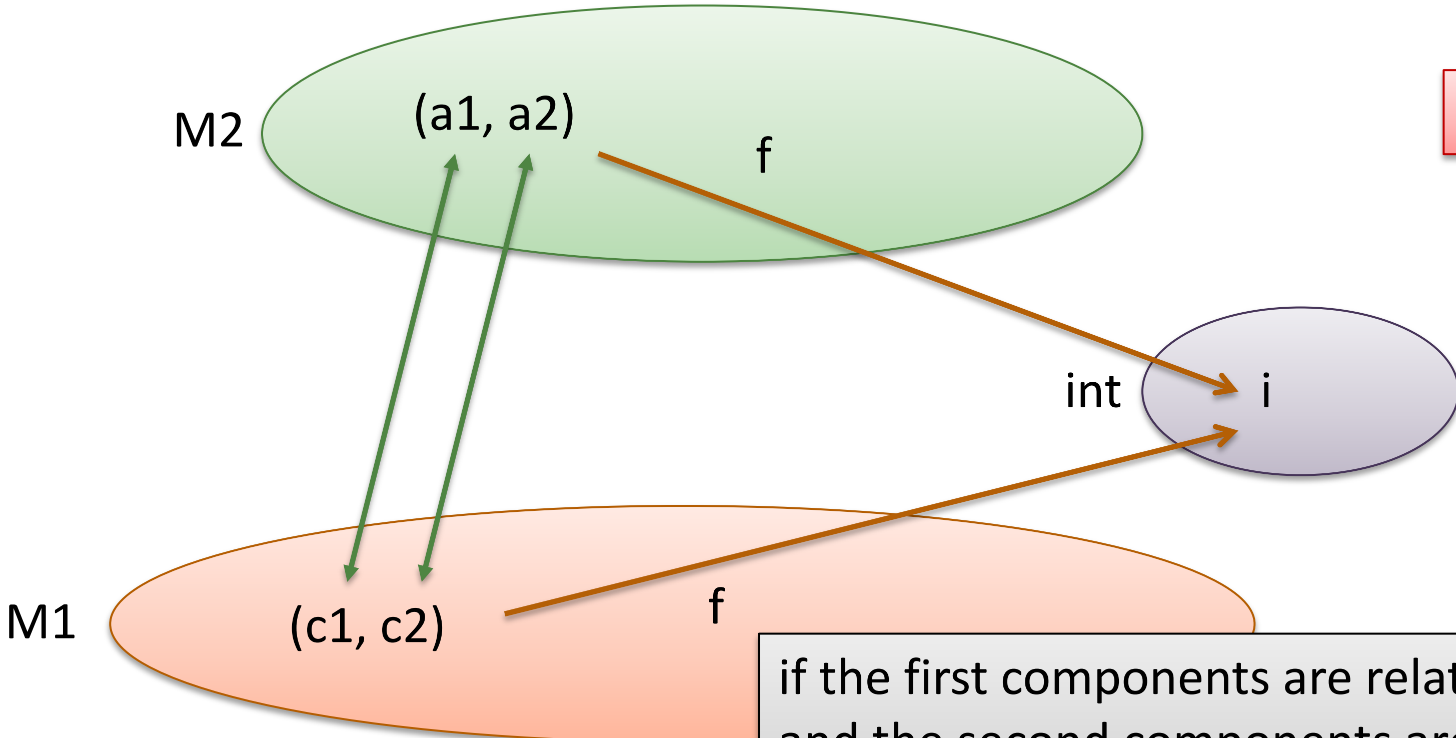
- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*



What does it mean to "preserve" the relation

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*



f : t * t -> int

if the first components are related,
and the second components are related,
then M1's version of f should produce a value i
that is identical to the value that M2 produces.

What does it mean to “preserve” the relation

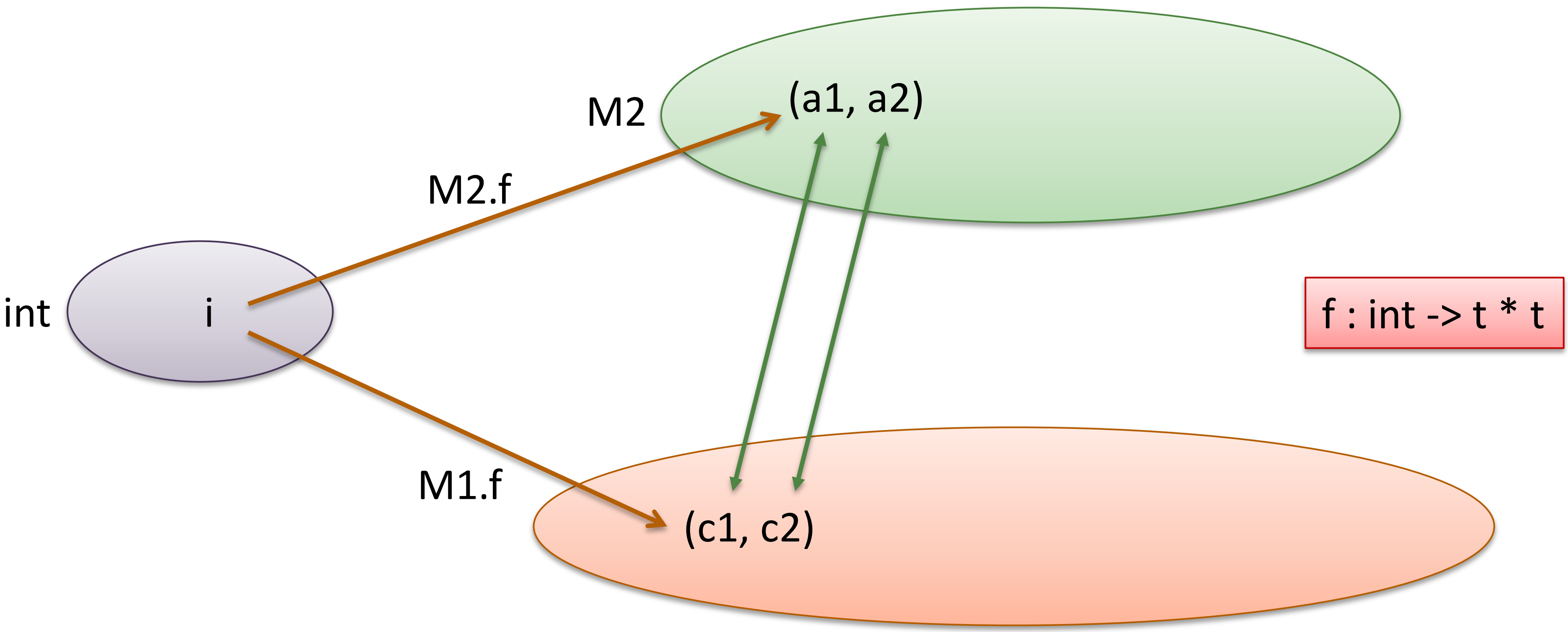
Two modules with abstract type t will be declared equivalent if:

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

What does it mean to “preserve” the relation

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

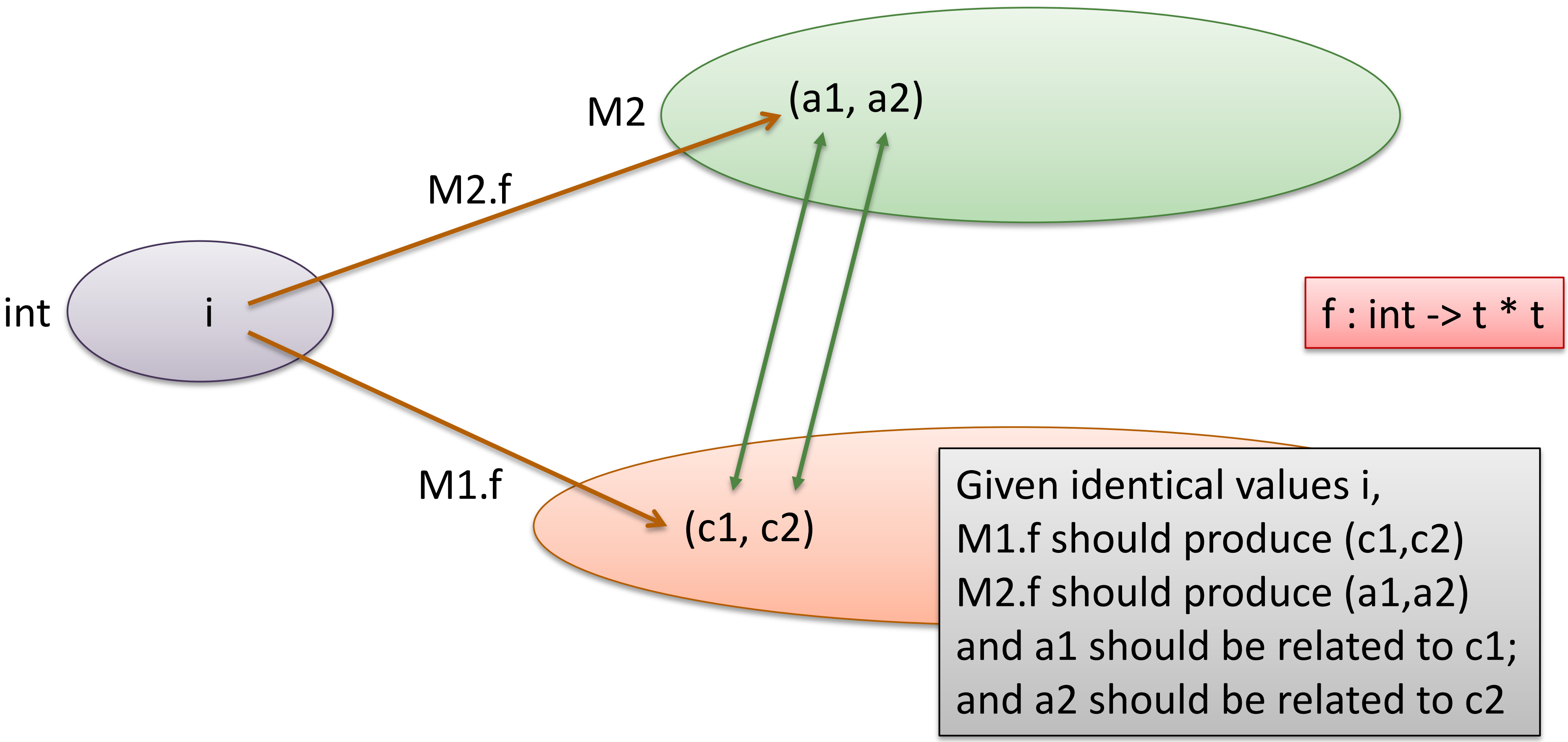
- one can *define a relation between corresponding values of type t*
- one can show that *the relation is preserved by all operations*



What does it mean to “preserve” the relation

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*



More Generally

To prove M1 == M2 relative to signature S,

- Start by defining a relation "is_related" for the abstract type t:
  - 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) for type s1 then we prove
    - is_related(v1 arg1, v2 arg2) for 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

Logical Relations

is_related (v1, v2) *at type t* *-- for module equivalence*

valid (v) *at type t* *-- for establishing rep invariants*

are both *logical relations*. They lift properties at abstract type t to properties at higher types (like t -> t) in a .... logical way.

AN EXAMPLE MODULE EQUIVALENCE

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)
in M1.reveal x1
```

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

What is the relationship?

```
let 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
```

To prove module equivalence, we have to consider all elements of the signature S separately. ie: zero, bump and reveal

For each such operation, we need to show is_related(v1,v2) at type s when v1 is from M1 and v2 is from M2 and s is the type of that element in the signature.

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.

Must prove: is_related (M1.zero, M2.zero)

Equivalent to proving: M1.zero == M2.zero/2 – 1

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

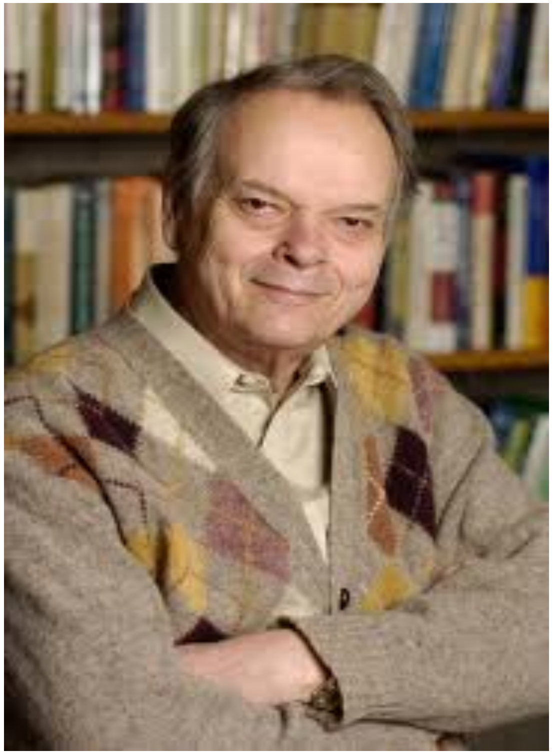
To prove M1 == M2 relative to signature S,

- Start by defining a relation "is_related" on abstract type t:
  - is_related (v1, v2) should hold for values with abstract type t when v1 comes from module M1 and v2 comes from module M2

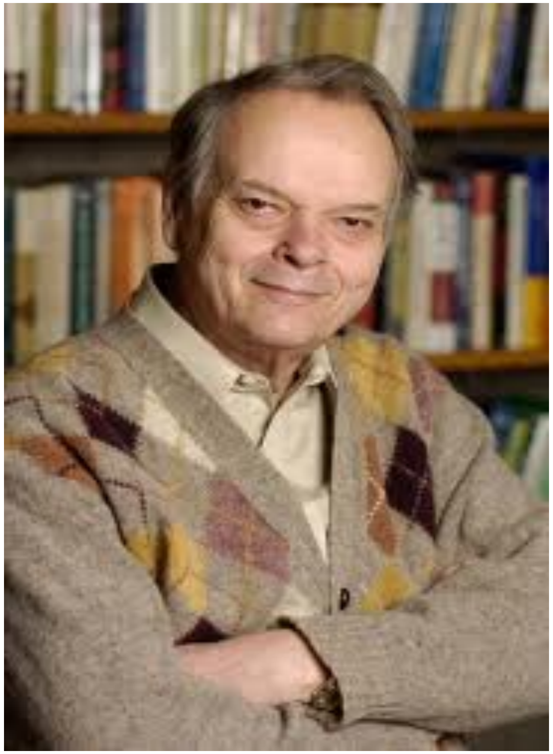
- Extend "is_related" to types other than just abstract t. For example:
  - if v1, v2 have type int, then they must be exactly the same
    - ie, we must prove: v1 == v2
  - if v1, v2 have type s1 -> s2 then we consider arg1, arg2 such that:
    - if is_related(arg1, arg2) then we prove
    - is_related(v1 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

Serial Killer or PL Researcher?



Serial Killer or PL Researcher?



John Reynolds: super nice guy, 1935-2013

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

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 Colombia. According to wikipedia, killed more individuals than any other serial killer. Due to Colombian law, only imprisoned for 30 years; decreased to 22.

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 invs and “is_related” predicates are called logical relations

COMBINING REP INVS AND MODULE EQUIVALENCE

(NOT COVERED IN LECTURE, BUT TAKE A LOOK)

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
```

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
```

To prove inv is a good rep invariant, prove that:

(1) for all x:int, inv(create x)

(2) nothing for equals

(3) for all v1:t, if inv(v1) then inv(decr v1)

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
```

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

once we have proven the rep inv, we can use it.
eg, if we add abs to the module (and prove it doesn't violate the rep inv) then we can use inv to show that abs always returns a non-negative number.

Another Implementation

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

```
let inv2 (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: can client programs tell Num, Num2 apart?

Another Implementation

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

```
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
```

First, find relation between valid representations of the type t.

Another Implementation

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

```
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
```

First, find relation between valid representations of the type t.

```
let rel(x:t, y:int) : bool =
  match x with
    Zero -> y = 0
| Pos n -> y = n
| Neg n -> -y = n
```

Another Implementation

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

```
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
```

Next, prove the modules establish the relation.

Another Implementation

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

```
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
```

Next, prove the modules establish the relation.

```
for all x:int,
 rel (Num.create x) (Num2.create x)
```

Another Implementation

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

```
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
```

Next, prove the modules establish the relation.

```
for all x1,x2:t, y1,y2:int
 if inv(x1), inv(x2), inv2(y1), inv2(y2) and
    rel(x1,y1) and rel(x2,y2)
then
    (Num.equals x1 x2)  = (Num2.equals y1 y2)
```

Another Implementation

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

```
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
```

Next, prove the modules establish the relation.

```
for all x1:t, y1:int
 if inv(x1) and inv2(y1) and
   rel(x1,y1)
then
  rel (Num.decr x1) (Num2.decr y1)
```