Modules, Representation Invariants, and Equivalence

COS 326
Andrew Appel
Princeton University

slides copyright 2018 David Walker and Andrew Appel
permission granted to reuse these slides for non-commercial educational purposes
ABSTRACTION FUNCTIONS
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

user’s view:

sets of integers

\{1, 2, 3\} \quad \{4, 5\}

\{\} \quad \{\} 

implementation view:

lists of integers

\[1; 1; 2; 3; 2; 3\] \quad [ ] \quad \[4, 5\] \quad [4, 5, 5]

\[1; 2; 3\] \quad \[4, 5\] \quad [5, 4]
Abstraction

user’s view:

sets of integers

\{1, 2, 3\} \quad \{4, 5\}

implementation view:

lists of integers

\[1; 1; 2; 3; 2; 3\] \quad \[4, 5\] \quad \[4, 5, 5\]

there’s a relationship here, of course!

we are trying to implement the abstraction
Abstraction

user’s view:
sets of integers
{1, 2, 3}  {4, 5}
{ }       {4, 5}

implementation
view:
lists of integers
[1; 1; 2; 3; 2; 3] [1; 2; 3] [4, 5] [4, 5, 5]

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

function called “the abstraction function”
Abstraction

User’s view:

Sets of integers:
- \{1, 2, 3\}
- \{4, 5\}
- \{\}\n
Implementation view:

Lists of integers:
- [1; 1; 2; 3; 2; 3]
- [1; 2; 3]
- [4, 5]
- [4, 5, 5]
- [5, 4]

INVARIANT cuts down the domain of the abstraction function.
Specifications

User’s view:

```
{1, 2} add 3 {1, 2, 3}
```

A specification tells us what operations on abstract values do.

Implementation view:
a specification tells us what operations on abstract values do.

user’s view:

\{1, 2\} → \text{add 3} → \{1, 2, 3\}

implementation view:

\[1; 2\]

inv(x)
A specification tells us what operations on abstract values do.

User’s view:

- \{1, 2\} → add 3 → \{1, 2, 3\}

Implementation view:

- [1; 2] → add 3 → [3; 1; 2]

inv(x)
In general: related arguments are mapped to related results.

A specification tells us what operations on abstract values do.

User’s view:
- \( \{1, 2\} \) -> add 3 -> \( \{1, 2, 3\} \)

Implementation view:
- \([1; 2]\) -> add 3 -> \([3; 1; 2]\)

inv(x)
Bug! Implementation does not correspond to the correct abstract value!
Specifications

user’s view:

{1, 2} → add 3 → {1, 2, 3}

implementation view:

[1; 2] → add 3 → [3; 1; 2]

inv(x)

specification

implementation must correspond no matter which concrete value you start with
to prove:
for all c1:t, if inv(c1) then \( f_{\text{abs}}(\text{abs} \, c1) \) == \( \text{abs} \, (f_{\text{con}} \, c1) \)

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.
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
Recall: To prove a rep. inv., assume it holds on inputs & prove it holds on outputs:
• if \( M.f \) has type \( t \rightarrow t \), we prove that:
  • if \( \text{inv}(v) \) then \( \text{inv}(M.f \ v) \)

Likewise for inter-module relations:
• if \( M1.f \) and \( M2.f \) have type \( t \rightarrow t \), we prove that:
  • if \( \text{is\_related}(v1, v2) \) then
  • \( \text{is\_related}(M1.f \ v1, M2.f \ v2) \)
Consider zero, which has abstract type t.

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)
Consider bump, which has abstract type $t \rightarrow t$.

Must prove for all $v_1 : \text{int}$, $v_2 : \text{int}$
if $\text{is}_\text{related}(v_1, v_2)$ then $\text{is}_\text{related}(\text{M1.bump } v_1, \text{M2.bump } v_2)$

Proof:
(1) Assume $\text{is}_\text{related}(v_1, v_2)$.
(2) $v_1 == v_2/2 - 1$ (by def)

Next, prove:
$(\text{M2.bump } v_2)/2 - 1 == \text{M1.bump } v_1$
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) == v2/2 - 1 (eval)
(M2.reveal v2) == v1 (by 2)
(M2.reveal v2) == M1.reveal v1 (eval, reverse)
To prove $M_1 == M_2$ relative to signature $S$,

- Start by defining a relation \texttt{is\_related}:
  - $\texttt{is\_related}(v_1, v_2)$ should hold for values with abstract type $t$ when $v_1$ comes from module $M_1$ and $v_2$ comes from module $M_2$.

- Extend \texttt{is\_related} to types other than just abstract $t$. For example:
  - if $v_1, v_2$ have type $\texttt{int}$, then they must be exactly the same
    - ie, we must prove: $v_1 == v_2$
  - if $v_1, v_2$ have type $\texttt{s1 -> s2}$ then we consider $\text{arg}_1, \text{arg}_2$ such that:
    - if $\texttt{is\_related}(\text{arg}_1, \text{arg}_2)$ then we prove
    - $\texttt{is\_related}(v_1 \ \text{arg}_1, v_2 \ \text{arg}_2)$
  - if $v_1, v_2$ have type $\texttt{s option}$ then we must prove:
    - $v_1 == \text{None}$ and $v_2 == \text{None}$, or
    - $v_1 == \text{Some} \ u_1$ and $v_2 == \text{Some} \ u_2$ and $\texttt{is\_related}(u_1, u_2)$ at type $s$

- For each $\texttt{val} \ v:s$ in $S$, prove $\texttt{is\_related}(M_1.v, M_2.v)$ at type $s$.  

MODULE EQUIVALENCE
To prove an abstraction is sound (ie, a faithful description of what is going on):

\[
\text{abstraction function then abstract op } \equiv \text{ concrete op then abstraction function}
\]
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:

- \( e1 \rightarrow^* v1 \) and \( e2 \rightarrow^* v2 \) 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 $e_1$ and $e_2$ are equivalent when:

- $e_1 \rightarrow v_1$ and $e_2 \rightarrow v_2$ and $v_1 = v_2$,
- they both diverge, or
- they both raise the same exception

When are two modules equivalent?

- We can’t just ask $M_1.f \ x$ and $M_2.f \ x$ to return the “same” value
  - the values might not even have the same type!
Reasoning about Module Equivalence

Two expressions $e_1$ and $e_2$ are equivalent when:

- $e_1 \rightarrow^\ast v_1$ and $e_2 \rightarrow^\ast v_2$ and $v_1 = v_2$
- they both diverge
- they both raise the same exception

When are two modules equivalent?

- We can’t just ask $M_1.f \ x$ and $M_2.f \ x$ to return the “same” value — the values might not even have the same type!

```ocaml
module type S = sig
  type t
  val zero : t
  val bump : t -> t
end
```
Reasoning about Module Equivalence

Two expressions $e_1$ and $e_2$ are equivalent when:

- $e_1 \rightarrow^* v_1$ and $e_2 \rightarrow^* v_2$ and $v_1 = v_2$
- they both diverge
- they both raise the same exception

When are two modules equivalent?

- We can’t just ask $M_1.f \ x$ and $M_2.f \ x$ to return the “same” value
  – the values might not even have the same type!

```ocaml
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!*
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*
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*
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 \( a_1 \) and \( c_1 \) are related, then M1’s version of \( f \) should produce a value \( a_2 \) that is related to the value that M2’s version of \( f \) produces.
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
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

\[ M_2 \overset{a_1}{\rightarrow} f \overset{M_1}{\leftarrow} c_1 \]

\[ f : t \rightarrow \text{int} \]
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?

If $a_1$ and $c_1$ are related, then M1’s version of $f$ should produce a value $i$ that is identical to the value that M2 produces.
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 $(a_1, a_2)$?

Diagram:

- $M_1$ with $(c_1, c_2)$
- $M_2$ with $(a_1, a_2)$
- Function $f : t \times t \rightarrow \text{int}$
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 

\[
(f: t \times t \rightarrow \text{int})
\]

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.
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
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?

Consider the function \( M_2.f : \text{int} \rightarrow t \times t \) and \( M_1.f : \text{int} \rightarrow (c_1, c_2) \). The relation is preserved by all operations if, for any inputs \( i \), the following holds:

\[
(a_1, a_2) = f(c_1, c_2)
\]
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?

Given identical values \( i \), \( M1.f \) should produce \( (c1, c2) \) and \( M2.f \) should produce \( (a1, a2) \) and \( a1 \) should be related to \( c1 \); and \( a2 \) should be related to \( c2 \).
To prove $M_1 == M_2$ relative to signature $S$,

- Start by defining a relation “\textit{is\_related}” for the abstract type $t$:
  - $\textit{is\_related}(v_1, v_2)$ should hold for values with abstract type $t$ when $v_1$ comes from module $M_1$ and $v_2$ comes from module $M_2$

- Extend “\textit{is\_related}” to types other than just abstract $t$. For example:
  - if $v_1$, $v_2$ have type \texttt{int}, then they must be exactly the same
    - ie, we must prove: $v_1 == v_2$
  - if $v_1$, $v_2$ have type \texttt{s1 \rightarrow s2} then we consider arg1, arg2 such that:
    - if $\textit{is\_related}(\text{arg1, arg2})$ for type $s_1$ then we prove
    - $\textit{is\_related}(v_1 \text{ arg1, v2 arg2})$ for type $s_2$
  - if $v_1$, $v_2$ have type \texttt{s option} then we must prove:
    - $v_1 == \text{None}$ and $v_2 == \text{None}$, or
    - $v_1 == \text{Some u1}$ and $v_2 == \text{Some u2}$ and $\textit{is\_related}(u1, u2)$ at type $s$

- For each $\texttt{val v:s}$ in $S$, prove $\textit{is\_related}(M1.v, M2.v)$ at type $s$
is_related (v1, v2) \textit{at type t} \quad -- \textit{for module equivalence}

valid (v) \textit{at type t} \quad -- \textit{for establishing rep invariants}

are both \textit{logical relations}. They lift properties at abstract type \( t \) to properties at higher types (like \( t \rightarrow t \)) in a .... logical way.
AN EXAMPLE 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 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
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.
Consider zero, which has abstract type \( t \).

Must prove:  \( \text{is\_related} (\text{M1.zero}, \text{M2.zero}) \)

Equivalent to proving:  \( \text{M1.zero} == \text{M2.zero}/2 - 1 \)

Proof:

\[
\begin{align*}
\text{M1.zero} \\
== 0 & \quad \text{(substitution)} \\
== 2/2 - 1 & \quad \text{(math)} \\
== \text{M2.zero}/2 - 1 & \quad \text{(substitution)}
\end{align*}
\]
One Signature, Two Implementations

Consider `bump`, which has abstract type \( t \rightarrow t \).

Must prove for all \( v_1 : \text{int} \), \( v_2 : \text{int} \)
if \( \text{is}_\text{related}(v_1, v_2) \) then \( \text{is}_\text{related}(\text{M1}.\text{bump} \ v_1, \text{M2}.\text{bump} \ v_2) \)

Proof:
(1) Assume \( \text{is}_\text{related}(v_1, v_2) \).
(2) \( v_1 = v_2/2 - 1 \) (by def)

Next, prove:
(\( \text{M2}.\text{bump} \ v_2 \))/2 \(-\) 1 == \( \text{M1}.\text{bump} \ v_1 \)
One Signature, Two Implementations

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

\[ \text{is_related} (x1, x2) = x1 == x2/2 - 1 \]
Summary of Proof Technique

To prove $M_1 == M_2$ relative to signature $S$,

- Start by defining a relation “is_related” on abstract type $t$:
  - $is\_related\ (v_1, v_2)$ should hold for values with abstract type $t$ when $v_1$ comes from module $M_1$ and $v_2$ comes from module $M_2$

- Extend “is_related” to types other than just abstract $t$. For example:
  - if $v_1, v_2$ have type $int$, then they must be exactly the same
    - ie, we must prove: $v_1 == v_2$
  - if $v_1, v_2$ have type $s_1 -> s_2$ then we consider $\text{arg1}, \text{arg2}$ such that:
    - if $is\_related(\text{arg1}, \text{arg2})$ then we prove
    - $is\_related(v_1 \ \text{arg1}, v_2 \ \text{arg2})$
  - if $v_1, v_2$ have type $s\ \text{option}$ then we must prove:
    - $v_1 == \text{None}$ and $v_2 == \text{None}$, or
    - $v_1 == \text{Some u1}$ and $v_2 == \text{Some u2}$ and $is\_related(u_1, u_2)$ at type $s$

- For each $\text{val v:s in S}$, prove $is\_related(M1.v, M2.v)$ at type $s$
John Reynolds, 1935-2013
Discovered the polymorphic lambda calculus (first polymorphic type system).
Developed *Relational Parametricity*: A technique for proving the equivalence of modules.
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)
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
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
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
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

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

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

```ocaml
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

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 nonnegative 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
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

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 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

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

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
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

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 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

Next, prove the modules establish the relation.

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

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 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

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)