Modules and Representation Invariants

COS 326
David Walker
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes
In previous classes:

Reasoning about individual OCaml expressions.

Now:

Reasoning about Modules (abstract types + collections of values)
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
module Set1 : SET =

struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l = x :: l
  let rem x l = List.filter ((<>) x) l
  let rec size l =
    match l with
    | [] -> 0
    | h::t -> size t + (if mem h t then 0 else 1)
  let union l1 l2 = l1 @ l2
  let inter l1 l2 = List.filter (fun h -> mem h l2) l1
end

Very slow in many ways!
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
The interesting operation:

```ocaml
(* 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
As a precondition on input sets:

(* size: list length is number of unique elements *)

```haskell
let size (s:'a set) : int =
    ignore (check s "size: bad set input");
List.length s
```
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"
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?
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.
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!
You may

assume the invariant $\text{inv}(i)$ for module inputs $i$ with abstract type

provided you

prove the invariant $\text{inv}(o)$ for all module outputs $o$ with abstract type
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.*

How is this invariant enforced? By using abstract types. Every value of the abstract type ‘a set satisfies the invariant. Internally, the module knows that the ‘a set is ‘a list and can establish the invariant, but externally clients don’t know that and can’t mess with established invariants.
A *representation invariant* \( \text{inv}(v) \) for abstract type \( t \) is a property of all data values \( v \) with abstract data type \( t \).

### Invariants on Abstract Types

- Invariants on abstract types are *local* to the ADT because they talk about the representation. Client code doesn’t know or care what the invariant is.
- However, client code *preserves the invariant* because it can’t mess with values of abstract type directly.
A *representation invariant* \( \text{inv}(v) \) *for abstract type* \( t \) *is a property of all data values* \( v \) *with abstract data type* \( t \).

Because Clients can’t mess with the invariants on abstract types, ADT code gets to *assume the invariant for inputs with abs. type* provided it *proves the invariant for outputs with abs. type*.

These proofs are *modular*: Done in isolation in the ADT module.
Establishing Representation Invariants

E.g., when it comes to the size function:

```ml
(* signature *)
val size : 'a set -> int

(* implementation: length is # of distinct elements *)
let size l = List.length l
```

If we want to assume all arguments to size have no duplicates, then:

- we have to ensure that our client can only pass us a list with no dups
- clients get their values of type ‘a set from our module, hence we have to ensure other functions in our module only produce lists with no duplicates
  - empty, add, rem, union, intersect
- typically the proof that a function produces elements that satisfy inv depend on assumptions that function inputs satisfy inv
  - add, rem, union, intersect
PROVING THE REP INVARIANT FOR THE SET ADT
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
  match l with
  [] -> true
| hd::tail -> not (mem hd tail) && inv tail
```

Definition of empty:

```ocaml
let empty : 'a set = []
```

Proof Obligation:

```ocaml
inv (empty) == true
```

Proof:

```ocaml
inv (empty)
== inv []
== match [] with [] -> true | hd::tail -> ...
== true
```
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail
```

Checking add:

```ocaml
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

assume invariant on input
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 in to two cases:

-- one case when \(\text{mem } x \ l\) is true
-- one case where \(\text{mem } x \ l\) is false
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).

    case 1: 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))
        == true  (by (2))
**Theorem:** for all \( x: 'a \) and for all \( l: 'a \) set, if \( \text{inv}(l) \) then \( \text{inv}(\text{add}(x \cdot l)) \)

**Proof:**

(1) pick an arbitrary \( x \) and \( l \).

(2) assume \( \text{inv}(l) \).

**case 2:** assume (3) not \( \text{mem}(x \cdot l) \) == true:

\[
\begin{align*}
\text{inv}(\text{add}(x \cdot l)) \\
= & \quad \text{inv}(\text{if mem x l then l else x::l}) \quad \text{(eval)} \\
= & \quad \text{inv}(x::l) \quad \text{(by (3))} \\
= & \quad \text{not (mem x l)} \&\& \text{inv(l)} \quad \text{(by eval)} \\
= & \quad \text{true} \&\& \text{inv(l)} \quad \text{(by (3))} \\
= & \quad \text{true} \&\& \text{true} \quad \text{(by (2))} \\
= & \quad \text{true} \\
\end{align*}
\]
Representation Invariants

Representation Invariant for sets without duplicates:

```ocaml
let rec inv (l : 'a set) : bool =
    match l with
    [] -> true
    | hd::tail -> not (mem hd tail) && inv tail
```

Checking rem:

```ocaml
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 Invariant for sets without duplicates:

```ocaml
def let rec inv (l : 'a set) : bool =
match l with
  [] -> true
| hd::tail -> not (mem hd tail) && inv tail
```

Checking size:

```ocaml
def 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:

```ocaml
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail
```

Checking union:

```ocaml
let union (l1:'a set) (l2:'a set) : 'a set =
  ...
```

Proof obligation?

for all \( l1 : 'a \text{ set} \) and for all \( l2 : 'a \text{ set} \),

if \( \text{inv}(l1) \) and \( \text{inv}(l2) \) then \( \text{inv}(\text{union} \ l1 \ l2) \)

assume invariant on input prove invariant on output
Representation Invariants

Representation Invariant for sets without duplicates:

```ml
let rec inv (l : 'a set) : bool =
  match l with
  | [] -> true
  | hd::tail -> not (mem hd tail) && inv tail
```

Checking inter:

```ml
let inter (l1:'a set) (l2:'a set) : 'a set =
  ...
```

Proof obligation?

for all \( l1: \text{a set} \) and for all \( l2: \text{a set} \),

if \( \text{inv}(l1) \) and \( \text{inv}(l2) \) then \( \text{inv}(\text{inter } l1 \ l2) \)

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
What about more complex types?

eg: for abstract type \( t \), consider: \( \text{val op : } t \times t \rightarrow t \text{ option} \)

Basic concept: Assume arguments are “valid”; Prove results “valid”

We know what it means to be a “valid” value \( v \) for abstract type \( t \):
- \( \text{Inv}(v) \) must be true

What is a valid pair? \( v \) is valid for type \( s_1 \times s_2 \) if
- (1) \( \text{fst } v \) is valid for type \( s_1 \), and
- (2) \( \text{snd } v \) is valid for type \( s_2 \)

Equivalently: \((v_1, v_2)\) is valid for type \( s_1 \times s_2 \) if
- (1) \( v_1 \) is valid for type \( s_1 \), and
- (2) \( v_2 \) is valid for type \( s_2 \)
What is a valid pair? $v$ is valid for type $s_1 * s_2$ if

- (1) $\text{fst } v$ is valid for $s_1$, and
- (2) $\text{snd } v$ is valid for $s_2$

**Example**: for abstract type $t$, consider: $\text{val op : } t * t \rightarrow t$

```
must prove to establish rep invariant:
for all $x : t * t$,
  if $\text{Inv(fst } x)$ and $\text{Inv(snd } x)$ then
  $\text{Inv (op } x)$
```

**Equivalent Alternative:**

```
must prove to establish rep invariant:
for all $x_1 : t$, $x_2 : t$
  if $\text{Inv(x1)}$ and $\text{Inv(x2)}$ then
  $\text{Inv (op (x1, x2))}$
```
Another Example:

val v : t * (t -> t)

must prove both to satisfy the rep invariant:

1. valid (fst v) for type t:
   ie: inv (fst v)

2. valid (snd v) for type t -> t:
   ie: for all v1:t,
      if Inv(v1) then
      Inv ((snd v) v1)
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`

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

must prove to satisfy rep invariant:

for all \( x : t * t \),
if \( \text{Inv(fst } x) \) and \( \text{Inv(snd } x) \)
then
  either:
  (1) \( \text{op } x \) is `None` or
  (2) \( \text{op } x \) is `Some u` and \( \text{Inv } u \)
Representation Invariants: More Types

Suppose we are defining an abstract type \( t \).
Consider happens when the type \texttt{int} shows up in a signature.
The type \texttt{int} does not involve the abstract type \( t \) at all, in any way.

\[
\text{eg: in our set module, consider: \quad val size : t -> int}
\]

When is a value \( v \) of type \texttt{int} valid?

\[
\text{all values } v \text{ of type int are valid}
\]

\[
\begin{align*}
\text{val size : t -> int} & \quad \text{must prove nothing} \\
\text{val const : int} & \quad \text{must prove nothing} \\
\text{val create : int -> t} & \quad \text{for all } v:\texttt{int}, \\
& \hspace{1em} \text{assume nothing about } v, \\
& \hspace{2em} \text{must prove } \text{Inv (create } v) \\
\end{align*}
\]
What is a valid function? Value \( f \) is valid for type \( t_1 \rightarrow t_2 \) if

- for all inputs \( \text{arg} \) that are valid for type \( t_1 \),
- it is the case that \( f \text{arg} \) is valid for type \( t_2 \)

\[ \text{eg: for abstract type } t, \text{ consider: val op : } t \times t \rightarrow t \text{ option} \]

must prove to satisfy rep invariant:

- for all \( x : t \times t \),
- if \( \text{Inv}(\text{fst } x) \) and \( \text{Inv}(\text{fst } x) \)
- then
- either:
  - (1) \( \text{op } x \) is None or
  - (2) \( \text{op } x \) is Some \( u \) and \( \text{Inv } u \)

valid for type \( t \times t \) (the argument)

valid for type \( t \text{ option} \) (the result)
What is a valid function? Value $f$ is valid for type $t_1 \rightarrow t_2$ if

- for all inputs $\text{arg}$ that are valid for type $t_1$,
- it is the case that $f \text{arg}$ is valid for type $t_2$

**eg:** for abstract type $t$, consider: $\text{val } op : (t \rightarrow t) \rightarrow t$

must prove to satisfy rep invariant:

for all $x : t \rightarrow t$,

if

{ for all arguments $\text{arg}:t$,
  if $\text{Inv}(\text{arg})$ then $\text{Inv}(x \text{arg})$ }

then

$\text{Inv}(\text{op} x)$

valid for type $t \rightarrow t$

(the argument)

valid for type $t$

(the result)
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

Proof: By (1) and (2), inv(f x)
ANOTHER EXAMPLE
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
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

let inv n : bool =
  n >= 0
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
Verifying The Invariant

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

• Let \( t \) be the abstract type and \( \text{inv()} \) 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
    
    • \( \text{inv}(v) \)

  – \( (v_1, v_2) \) is valid for \( s_1 \times s_2 \) if
    
    • \( v_1 \) is valid for \( s_1 \), and
    • \( v_2 \) is valid for \( s_2 \)

  – \( v \) 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 \( s_1 \rightarrow s_2 \) if
    
    • for all arguments \( a \), if \( a \) is valid for \( s_1 \), then \( v \ a \) is valid for \( s_2 \)

  – \( v \) is valid for \( \text{int} \) if
    
    • \( \text{always} \)

  – \([v_1; \ldots; v_n]\) is valid for type \( s \) list if
    
    • \( v_1 \ldots v_n \) are all valid for type \( s \)
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
    ...
    let inv n : bool =
      n >= 0
  end

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

module Nat : NAT =
  struct
    type t = int
    let from_int (n:int) : t =
      if n <= 0 then 0 else n
    ...
    let inv n : bool =
      n >= 0
  end

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 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
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

module Nat : NAT =
  struct
    type t = int
    let rep map f n =
      if n = 0 then []
      else f n :: map f (n-1)
  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.

Case: n = 0
map f n == []
(Note: each value v in [ ] satisfies inv(v))
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

Case: n > 0
map f n == f n :: map f (n-1)

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 type NAT =
  sig
    type t
    val map : (t -> t) -> t -> t list
  ...
end

Must prove:
for all \( f \) valid for type \( t \to t \)
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

Case: \( n > 0 \)
map \( f \ n \) == f n :: map f (n-1)

By IH, map \( f \ (n-1) \) is valid for t list.
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

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

Case: n > 0
map f n == f n :: map f (n-1)

Proof: By induction on nat n.
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
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

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

Hooray! n is never negative so we don’t infinite loop
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*
Proving things correct, not just safe.

ABSTRACTION FUNCTIONS
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 postconditions 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:

lists of integers

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

[5, 4]
user’s view:

sets of integers

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

{}  

list of integers

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

implementation view:

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}

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

implementation view:

lists of integers

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

function called “the abstraction function”
Abstraction

**user’s view:**
- sets of integers
  - \{1, 2, 3\}
  - \{4, 5\}
  - \{\}\n
**implementation view:**
- lists of integers
  - \([1; 1; 2; 3; 2; 3]\)
  - \([1; 2; 3]\)
  - \([4, 5]\)
  - \([4, 5, 5]\)
  - \([5, 4]\)

\textit{inv(x): no duplicates}

**abstraction function**

\textit{Invariant cuts down the domain of the abstraction function}
a specification tells us what operations on abstract values do
A specification tells us what operations on abstract values do.

User’s view:

- **{1, 2}**
- **add 3**
- **{1, 2, 3}**

Implementation view:

- **[1; 2]**
- **inv(x)**
Specifications

user’s view:

{1, 2}  

add 3  

{1, 2, 3}

implementation view:

[1; 2]  

add 3  

[3; 1; 2]

inv(x)

a specification tells us what operations on abstract values do
Specifications

user’s view:

\{1, 2\} → \text{add 3} → \{1, 2, 3\}

implementation view:

\[1; 2\] → \text{add 3} → \[3; 1; 2\]

a specification tells us what operations on abstract values do

In general: related arguments are mapped to related results

inv(x)
The diagram illustrates the difference between the user's view and the implementation view.

User's view: 
- \{1, 2\} → \{1, 2, 3\} → \{3; 1\}

Implementation view: 
- \[1; 2\] → \[3; 1; 3\]

The implementation does not correspond to the correct abstract value, indicating a bug.
Specifications

user’s view:

implementation view:

specification

implementation must correspond no matter which concrete value you start with

inv(x)
A more general view

abstract operation with type $t \rightarrow t$

abstraction function

abstract operation

concrete operation

abs

abs

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

to prove:
for all $c1:t$, if $\text{inv}(c1)$ then $f_{\text{abs}}(\text{abs } c1) == \text{abs } (f_{\text{con}} c1)$
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?
Recall: A representation invariant is a property that holds for all values of abs. type:
- if M.v has abstract type t,
  - we want \( \text{inv}(M.v) \) to be true

Inter-module relations are a lot like representation invariants!
- if M1.v and M2.v have abstract type t,
  - we want \( \text{is\_related}(M1.v, M2.v) \) 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)
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: \texttt{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)

\texttt{is\_related} (x1, x2) = x1 == x2/2 - 1
Consider bump, which has abstract type $t \rightarrow t$.

Must prove for all $v1:int$, $v2:int$
if $\text{is\_related}(v1,v2)$ then $\text{is\_related}(\text{M1.bump } v1, \text{M2.bump } v2)$

Proof:
(1) Assume $\text{is\_related}(v1, v2)$.
(2) $v1 == v2/2 - 1$ (by def)

Next, prove:
$(\text{M2.bump } v2)/2 - 1 == \text{M1.bump } v1$
One Signature, Two Implementations

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`

```plaintext
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
```
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
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

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

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 = 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 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: are Num, Num2 related?
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.
• It’s good practice to implement your representation invariants
• Use them to check your assumptions about inputs
  – find bugs in other functions
• Use them to check your outputs
  – find bugs in your function
If a module M defines an abstract type t
   – Think of a representation invariant \( \text{inv}(x) \) for values of type t
   – Prove each value of type s provided by M is \textit{valid for type s} relative to the representation invariant

If \( v : s \) then prove \( v \) is valid for type \( s \) as follows:
   – if \( s \) is the abstract \textit{type t} then prove \( \text{inv}(v) \)
   – if \( s \) is a base type like \texttt{int} then \( v \) is always valid
   – if \( s \) is \( s_1 \ast s_2 \) then prove:
     • \( \text{fst} \ v \) is valid for type \( s_1 \)
     • \( \text{snd} \ v \) is valid for type \( s_2 \)
   – if \( s \) is \texttt{s1 option} then prove:
     • \( v \) is \texttt{None}, or
     • \( v \) is \texttt{Some} \( u \) and \( u \) is valid for type \( s_1 \)
   – if \( s \) is \( s_1 \to s_2 \) then prove:
     • for all \( x:s_1 \), if \( x \) is valid for type \( s_1 \) then \( v \ x \) is valid for type \( s_2 \)

\textbf{Aside:} This kind of proof is known as a proof using \textit{logical relations}. It lifts a property on a basic type like \( \text{inv}(\ ) \) to a property on higher types like \( t_1 \ast t_2 \) and \( t_1 \to t_2 \)
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”
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”

Amal Ahmed
Assistant Professor at Northeastern University
PhD Princeton 2004
Leading theorist of logical relations for data abstraction & compiler correctness