Modules
and Representation Invariants

COS 326
Andrew W. Appel
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
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

```ocaml
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:

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

```ocaml
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.
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:

```ocaml
(* 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 precondition on input sets:

```haskell
(* 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:

```haskell
(* 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*
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 \( \text{inv}(l) \) then \( \text{inv}(\text{add} \ x \ l) \)

Proof:

1. pick an arbitrary \( x \) and \( l \).
2. assume \( \text{inv}(l) \).

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 \( x : \texttt{a} \) and for all \( l : \texttt{a set} \), if \( \text{inv}(l) \) then \( \text{inv}(\text{add } x \ l) \)

Proof:

(1) pick an arbitrary \( x \) and \( l \).  
(2) assume \( \text{inv}(l) \).

\[
\text{case 1: assume (3): mem } x \ l = \text{ true:}
\]

\[
\begin{align*}
\text{inv } (\text{add } x \ l) \\
= \text{ inv } (\text{if } \text{mem } x \ l \text{ then } l \text{ else } x::l) \quad \text{(eval)} \\
= \text{ inv } (l) \quad \text{(by (3))} \\
= \text{ true} \quad \text{(by (2))}
\end{align*}
\]
Representation Invariants

Theorem: for all \(x: \texttt{a}\) and for all \(l: \texttt{a set}\), if \(\text{inv}(l)\) then \(\text{inv}(\text{add } x \ l)\)

Proof:

(1) pick an arbitrary \(x\) and \(l\).  
(2) assume \(\text{inv}(l)\).

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

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

Representation Invariant for sets without duplicates:

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

Checking rem:

let rem (x:'a) (l:'a set) : 'a set =
  List.filter ((<> x) 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 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 size:

```ocaml
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:

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

Checking union:

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

Proof obligation?

for all l1:'a set and for all l2:'a set,

if inv(l1) and inv(l2) then inv (union l1 l2)

assume invariant on input

prove invariant on output
Representation Invariant for sets without duplicates:

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

Checking inter:

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

Proof obligation?

for all l1:'a set and for all l2:'a set,
if inv(l1) and inv(l2) then inv (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
```

证明:
- 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: \( \text{val } \text{op} : t \times t \rightarrow t \text{ option} \)

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 \( s_2 \)

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

must prove to establish rep invariant:

for all \( x : t * t \),

if \( \text{Inv(} \text{fst } x \text{)} \) and \( \text{Inv(} \text{snd } x \text{)} \) then

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

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

\( \text{Inv (} \text{op (} x_1, x_2 \text{)} \)
Another Example:

\[
\text{val } v : t \times (t \rightarrow t)
\]

must prove both to satisfy the rep invariant:

1. valid (fst v) for type \( t \):
   \[
   \text{ie: } \text{inv} \ (\text{fst} \ v)
   \]

2. valid (snd v) for type \( t \rightarrow t \):
   \[
   \text{ie: for all } v_1 : t, \\
   \text{if } \text{Inv}(v_1) \text{ then } \\
   \text{Inv} ((\text{snd} \ v) \ v_1)
   \]
What is a valid option? \( v \) is valid for type \( s_1 \text{ option} \) if

- (1) \( v \) is \textit{None}, or
- (2) \( v \) is \textit{Some} \( u \), and \( u \) is valid for type \( s_1 \)

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

must prove to satisfy rep invariant:

for all \( x : t \times t \),

if \( \text{Inv}(\text{fst } x) \) and \( \text{Inv}(\text{snd } x) \)

then

either:

- (1) \( \text{op} x \) is \textit{None} or
- (2) \( \text{op} x \) is \textit{Some} \( u \) and \( \text{Inv} u \)
Suppose we are defining an abstract type $t$.
Consider what happens when the type `int` shows up in a signature.
The type `int` does not involve the abstract type $t$ at all, in any way.

```
val size : t -> int
```

When is a value $v$ of type `int` valid?

- All values $v$ of type `int` are valid.
- `val size : t -> int` must prove nothing.
- `val const : int` must prove nothing.
- `val create : int -> t` must prove nothing.
- For all $v$ : `int`, assume nothing about $v$, must prove $\text{Inv}(\text{create } v)$.
What is a valid function? Value \( f \) is valid for type \( t1 \rightarrow t2 \) if:

- for all inputs \( \text{arg} \) that are valid for type \( t1 \),
- it is the case that \( f \text{arg} \) is valid for type \( t2 \)

**Example:** for abstract type \( t \), consider:

\[
\text{val op : } t \times t \rightarrow t \text{ option}
\]

must prove to satisfy rep invariant:

for all \( x : t \times t \),

\begin{align*}
\text{if Inv(fst x) and Inv(fst x) then } & \text{ either: } \\
&(1) \text{op x is None or } \\
&(2) \text{op x is Some u and Inv u}
\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 \( t_2 \)

Example: for abstract type \( t \), consider:

\[
\text{val op : } (t \rightarrow t) \rightarrow t
\]

must prove to satisfy rep invariant:

\[
\text{for all } x : t \rightarrow t, \quad \text{if} \quad \{ \text{for all arguments } \text{arg}:t, \quad \text{if } \text{Inv(\text{arg}) then } \text{Inv}(x \text{arg}) \} \quad \text{then} \quad \text{Inv}(\text{op } x)
\]

valid for type \( t \rightarrow t \)

(\text{the argument})

valid for type \( t \)

(\text{the result})
Representation Invariants: More Types

sig
  type t
  val create : int -> t
  val incr : t -> t
  val apply : t * (t -> t) -> t
  val check_t : t -> t
end

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

struct
  type t = int
  let create n = abs n
  let incr n = if n<maxint then n + 1
                else raise Overflow
  let apply (x, f) = f x
  let check_t x = assert (x >= 0); x
end

function apply, must prove:
  for all x:t,
    for all f:t -> t
      if (1) inv(x)
        and (2) for all y:t, if inv(y) then inv(f y)
      then inv(f x)

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
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

since function result has type t, must prove the output satisfies \( \text{inv}(\cdot) \)

since function input has type t, assume the output satisfies \( \text{inv}(\cdot) \)

for \( \text{map } f \ x \), assume:
(1) \( \text{inv}(x) \), and
(2) \( f \)'s results satisfy \( \text{inv}(\cdot) \) when it's inputs satisfy \( \text{inv}(\cdot) \).

then prove that all elements of the output list satisfy \( \text{inv}(\cdot) \)
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 \( s_1 * s_2 \) 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 \( s1 \rightarrow s2 \) if
    
    • for all arguments \( a \), if \( a \) is valid for \( s1 \), then \( v \ a \) is valid for \( s2 \)
  
  – \( v \) is valid for \( \text{int} \) if
    
    • always
  
  – \([v_1; ...; v_n]\) is valid for type \( s \) list if
    
    • \( v_1 \) ... \( v_n \) 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
    ...
  end

let inv n : bool =
  n >= 0

Must prove:
  for all n,
    inv (from_int n) == true

Proof strategy: Split into 2 cases.
  (1) n > 0, and (2) n <= 0
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 =
  if n <= 0 then 0 else n

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
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 0
== true
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)
    let inv n : bool = 
      n >= 0
  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 n.
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  == []
(Note: each value v in [ ] satisfies inv(v))

Proof: By induction on nat 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 == f n :: map f (n-1)
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  == 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

Must prove:
for all f valid for type t -> t
for all n valid for type t
map f n is valid for type t list

Proof: By induction on nat n.

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

let inv n : bool = n >= 0

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

By IH, map f (n-1) is valid for t list.
Since f valid for t -> t and n valid for t
f n::map f (n-1) is valid for t list
module type NAT =
  sig
    type t
    val map : (t -> t) -> t -> t list
  end

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

End result: We have proved a strong property \((n \geq 0)\) 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 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} {4, 5}
{} {}
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]
  - [5, 4]
  - [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\} \rightarrow \{\}\rightarrow \{4, 5\}

implementation view:

lists of integers

[1; 1; 2; 3; 2; 3] \rightarrow [1; 2; 3] \rightarrow \{4, 5\}\rightarrow [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}

{ }

implementation view:

lists of integers

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

inv(x): no duplicates

Representation Invariant cuts down the domain of the abstraction function
a specification tells us what operations on abstract values do

user’s view:

{1, 2} \rightarrow \text{add 3} \rightarrow \{1, 2, 3\}

implementation view:
A specification tells us what operations on abstract values do.

User's view:

\{1, 2\} \rightarrow \text{add 3} \rightarrow \{1, 2, 3\}

Implementation view:

\[1; 2\] \rightarrow \text{inv}(x)
Specifications

user’s view:

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

add 3

add 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\} \rightarrow add 3 \rightarrow \{1, 2, 3\}

implementation view:
- [1; 2] \rightarrow add 3 \rightarrow [3; 1; 2]

in\(v(x)\)

a specification tells us what operations on abstract values do

In general: related arguments are mapped to related results
Specifications

User's view:

\[ \{1, 2\} \xrightarrow{\text{add 3}} \{1, 2, 3\} \not\equiv \{3; 1\} \]

Implementation view:

\[ [1; 2] \xrightarrow{\text{add 3}} [3; 1; 3] \]

Bug! Implementation does not correspond to the correct abstract value!
Specifications

user’s view:

implementation view:

add 3

\{1, 2\} \longrightarrow \text{add 3} \longrightarrow \{1, 2, 3\}

\text{implementation must correspond no matter which concrete value you start with}

\text{specification}

inv(x)

\{1, 2\} \rightarrow \text{add 3} \rightarrow \{3; 1; 2\}

\{2; 1\} \rightarrow \text{add 3} \rightarrow \{3; 2; 1\}
A more general view

Abstract operation with type $t \rightarrow t$

Abstraction function

Abstract operation

Concrete operation

To prove:

for all $c_1 : t$, if $\text{inv}(c_1)$ then $f_{\text{abs}}(\text{abs } c_1) = \text{abs } (f_{\text{con}} c_1)$

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.
Consider a client that might use the module:

```ocaml
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?*
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 \( \text{inv}(M.v) \) 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
One Signature, Two Implementations

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

module M1 : S =
  struct
    type t = int
    let zero = 0
    let bump n = n + 1
    let reveal n = n
  end

module M2 : S =
  struct
    type t = int
    let zero = 2
    let bump n = n + 2
    let reveal n = n/2 - 1
  end

Recall: To prove a rep. inv., assume it holds on inputs & prove it holds on outputs:
- if M.f has type t -> t, we prove that:
  - if inv(v) then inv(M.f v)

Likewise for inter-module relations:
- if M1.f and M2.f have type t -> t, we prove that:
  - if is_related(v1, v2) then
  - is_related(M1.f v1, M2.f v2)

related functions
produce related results
from related arguments
Consider zero, which has abstract type \( t \).

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

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

Proof:

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

\[
\text{is\_related} \ (x1, x2) = x1 == x2/2 - 1
\]
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

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

== (v2 + 2)/2 – 1
== (v2/2 – 1) + 1
== v1 + 1
== M1.bump v1

(eval)
(math)
(by 2)
(eval, reverse)
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) = (M2.reveal v2)
== v2/2 - 1
== v1
== M1.reveal v1

(eval)
(by 2)
(eval, reverse)
To prove $M_1 == M_2$ relative to signature $S$,

- Start by defining a relation "is_related":
  - $\text{is\_related}(v_1, v_2)$ 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 $\text{int}$, then they must be exactly the same
    - ie, we must prove: $v_1 == v_2$
  - if $f_1, f_2$ have type $s_1 \to s_2$ then we consider $\text{arg}_1, \text{arg}_2$ such that:
    - if $\text{is\_related}(\text{arg}_1, \text{arg}_2)$ then we prove
      - $\text{is\_related}(f_1 \text{arg}_1, f_2 \text{arg}_2)$
  - if $o_1, o_2$ have type $s \text{ option}$ then we must prove:
    - $o_1 == \text{None}$ and $o_2 == \text{None}$, or
    - $o_1 == \text{Some } u_1$ and $o_2 == \text{Some } u_2$ and $\text{is\_related}(u_1, u_2)$ at type $s$

- For each $\text{val } v:s$ in $S$, prove $\text{is\_related}(M_1.v, M_2.v)$ 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: Debugging with Rep. Invariants

- 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 inv(x) for values of type t
   – Prove each value of type s provided by M is 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 type t then prove inv(v)
   – if s is a base type like int then v is always valid
   – if s is \( s_1 * s_2 \) then prove:
     • \( \text{fst} \ v \) is valid for type \( s_1 \)
     • \( \text{snd} \ v \) is valid for type \( s_2 \)
   – if s is \( s_1 \text{ option} \) then prove:
     • v is None, or
     • v is Some u and u is valid for type \( s_1 \)
   – if s is \( s_1 \rightarrow s_2 \) then prove:
     • for all x:s1, if x is valid for type s1 then \( v \ x \) is valid for type s2

Aside: This kind of proof is known as a proof using logical relations. It lifts a property on a basic type like inv( ) to a property on higher types like \( t_1 * t_2 \) and \( t_1 \rightarrow t_2 \)
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