Modules and Representation Invariants

COS 326
David Walker
Princeton University

LAST TIME

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 you check all ouputs, then you should not *need* to check your inputs! (but you can, just in case you missed an output you should have checked!

Proving representation invariants involves (roughly):

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

Last Time: Module Equivalence

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

- one can *define a relation* between values of type t in the two modules, and
- one can show that *the relation is preserved by all operations*

As with representation invariants, one *assumes* the inputs to a function are related, and one *proves* that the outputs to the function are related.

Example for modules M1 and M2 with signature S:

- define is_related(v1,v2) for v1 from M1 and v2 from M2
- for all functions f : t -> t in S, prove:
  - if is_related(v1,v2) then is_related(M1.f v1, M2.f v2)

ASIDE:
WHAT DOES CHECKING YOUR "OUTPUTS" MEAN?

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

    let create (n:int) (b:bool) : t =
        if b then
          Either n
        else
          Or (n*2)

    let item : t = Either 0

end
```

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

    let create (n:int) (b:bool) : t =
      if b then
        Either n
      else
        Or (n*2)

    let item : t = Either 0

end
```

"outputs"

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

    let inv (v:t) : t  = ...

    let check(v:t) (m:string) : t =
      if inv(v) then v else failwith m

    let create (n:int) (b:bool) : t =
        check (
          if b then
            Either n
         else
            Or ((abs n)*2)
        )

    let item : t = check (Either 0)
end
```

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    val process : t -> t
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

    let inv (v:t) : t  = ...

    let check(v:t) (m:string) : t =
      if inv(v) then v else failwith m

    let process (v:t) : t =
      if not (inv v) then
        blow_up_the_world()  (* undesirable! *)
      else
        ...
end
```

client:

```
let x = create 3 true in
process x
```

we want to be sure the world doesn't blow up.
clients can obtain outputs and then pass them back as inputs to the module
so we need to check the outputs.

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    val process : t -> t
    val baz : (t -> unit) -> int
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

    let baz (f:t -> unit) : int =
      let x = Or (-3) in
      f x;
      17
end
```

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    val process : t -> t
    val baz : (t -> unit) -> int
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

   let process (v:t) : t =
       if not (inv v) then
         blow_up_the_world()
       else
         ...

    let baz (f:t -> unit) : int =
       let x = Or (-3) in
       f x;
       17
end
```

client:

```
let f x =  let _ = process x in () in
baz f
```

A simple program

```
module type FOO =
  sig
    type t
    val create : int -> bool -> t
    val item : t
    val process : t -> t
    val baz : (t -> unit) -> int
    ...
end
```

```
module FOO =
  struct
    type t = Either of int | Or of int

   let process (v:t) : t =
       if not (inv v) then
         blow_up_the_world()
       else
         ...

    let baz (f:t -> unit) : int =
       let x = Or (-3) in
       f (check x);
       17
end
```

client:

```
let f x =  let _ = process x in () in
baz f
```

need to check inputs
to functions passed in as
arguments!

A neat thing about types

t1 -> **t2**

(**t3** -> t4) -> **t2**

((t5 -> **t6**) -> t4) -> **t2**

(((**t7** -> t8) -> **t6**) -> t4) -> **t2**

positive positions in types
the positions you have to check!

the arrow -> acts like an operator
that flips the "sign" from positive
to negative or negative to positive

A SIMPLE EXAMPLE

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 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
```

```
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 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 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
```

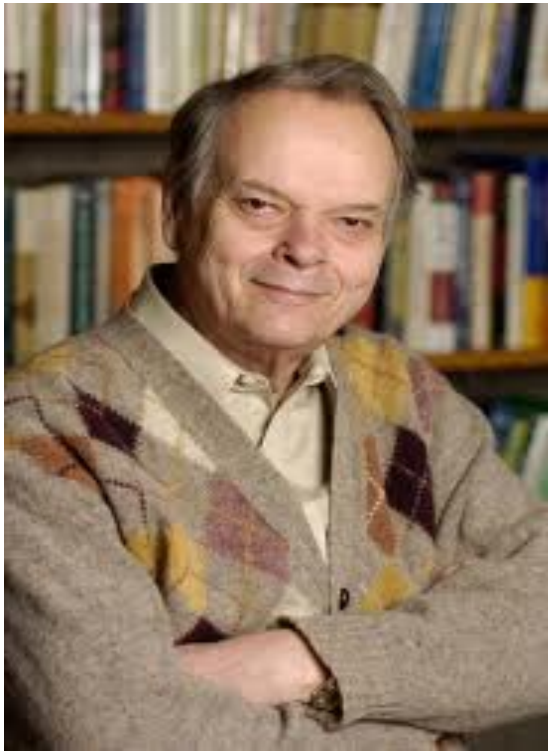
```
module Num2 =
  struct
    type t = int

    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
```

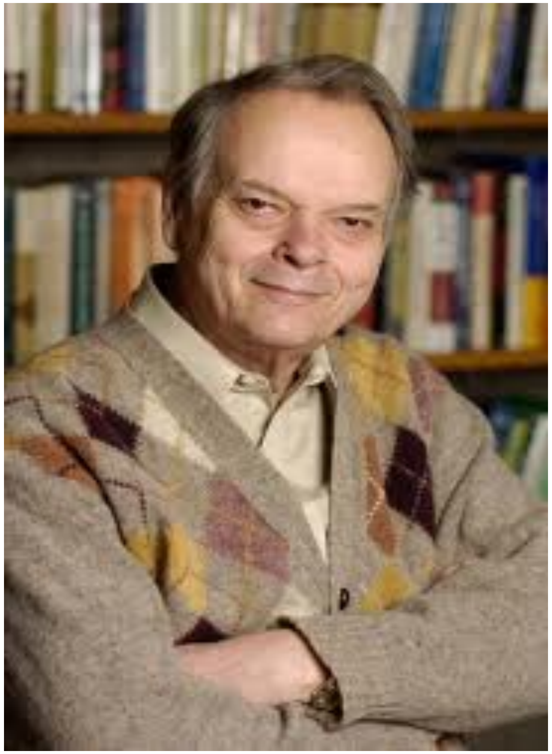
Next, prove the modules establish the relation.

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

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

```
struct
  type t = int



  let create n = abs n
  let incr n = n + 1
  let apply (x, f) = f x
end
```

```
struct
  type t = int
  let inv x : bool = x >= 0
  let check_t x = assert (inv x); x

  let create n = check(abs n)
  let incr n = check ((check n) + 1)
  let apply (x, f) = check (f (check x))
end
```

check output produced

check input assumption

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

Summary: Rep. Invariants Proof Summary

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 s1 * s2 then prove:
  - fst v is valid for type s1
  - snd v is valid for type s2
- if s is s1 option then prove:
  - v is None, or
  - v is Some u and u is valid for type s1
- if s is s1 -> s2 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 t1 * t2 and t1 -> t2

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

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”