Reasoning About Modular Programs

Part 3: More Representation Invariants

Speaker: David Walker
COS 326
Princeton University

Last Time: Proving Simple Representation Invariants

```
sig
  type t

  val value : t

  val constructor : int -> t

  val transform : int -> t -> t

  val destructor : t -> int

end
```

- value: prove: Inv (value)
- constructor: prove: for all x:int, Inv (constructor x)
- transform: prove:
  for all x:int,
    for all v:t,
      if Inv(v)
      then Inv (transform x v)
- destructor: assume Inv(t))

Representation Invariants: More Types

What about more complex types?

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

Basic concept:

- Assume arguments are "valid" and prove results "valid"
- What it means to be "valid" depends on the *type* of the value

Given a definition of what it means to be valid for *the abstract type* t, we will explain how to *lift* that definition to *any complex type s:*

- s * s
- *s option*
- *s list*
- s -> s

“valid for type t”

What is a valid pair? v is valid for type s1 * s2 if

- (1) fst v is valid for type s1, and
- (2) snd v is valid for type s2

Equivalently: (v1, v2) is valid for type s1 * s2 if

- (1) v1 is valid for type s1, and
- (2) v2 is valid for type s2

Representation Invariants: More Types

What is a valid pair? v is valid for type s1 * s2 if

(1) fst v is valid for s1, and

(2) snd v is valid for s2

eg: for abstract type t, consider: val op : t * t -> t

must prove to establish rep invariant:
  for all x : t * t,
    if Inv(fst x) and Inv(snd x) then
    Inv (op x)

Equivalent Alternative:

must prove to establish rep invariant:
  for all x1:t, x2:t
    if Inv(x1) and Inv(x2) then
    Inv (op (x1, x2))

Representation Invariants: More Types

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

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

```
must prove to satisfy rep invariant:
 for all x : t * t,
   if Inv(fst x) and Inv(snd x)
   then
     either:
       (1) op x is None or
       (2) op x is Some u and Inv u
```

Representation Invariants: More Types

Suppose we are defining an abstract type t.

Consider 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.

eg: in our set module, consider: 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 ← for all v:int,
assume nothing about v,
must prove Inv (create v)

Representation Invariants: More Types

What is a valid function? Value f is valid for type t1 -> t2 if

- for all inputs arg that are valid for type t1,
- it is the case that f arg is valid for type t2

Note: We've been using this idea all along for all operations!

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

```
must prove to satisfy rep invariant:
 for all x : t * t,
    if Inv(fst x) and Inv(snd x)
    then
       either:
          (1) op x == None or
          (2) op x == Some u and Inv u
```

valid for type t * t (the argument)

valid for type t option (the result)

Representation Invariants: More Types

Consider: val op : (t -> t) -> t

must prove to satisfy rep invariant:
for all x : t -> t,
  if
    {for all arguments arg:t,
      if Inv(arg) then Inv(x arg) }
  then
    Inv (op x)

valid for type t -> t
(the argument)

valid for type t
(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
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 apply (x,f) valid for t

```
struct
  type t = int
  let create n = abs n
  let incr n = if n < maxint then n + 1
                    else n (* overflow .. *)
  let apply (x, f) = f x
  let check x = assert (x >= 0)
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(apply (x,f))

Proof: apply (x,f) == f x (by eval).
Hence, we must show: inv(f x)
By (1) and (2), inv(f x)

ANOTHER EXAMPLE

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
```

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
```

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
```

```
let inv n : bool =
  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
```

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)  (eval)
== inv n                          (by n > 0, eval)
== true                           (by 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)  (eval from_int)
== inv 0                          (by n <= 0, eval)
== true                           (eval inv)
```

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

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
```

```
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 nat n.

Natural Numbers

```
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 == []

(Note: each value v in [ ] satisfies inv(v))

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
```

```
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 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
```

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.

Natural Numbers

```
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

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
```

Hooray! n is never negative so we don't infinite loop

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

One More example

```
module type NAT =
  sig

    type t

    val from_int : int -> t

    val to_int : t -> int

    val map : (t -> t) -> t -> t list

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

  end
```

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

```
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)

    let foo f = f (-1)

end
```

One More Example

```
module type NAT =
  sig

    type t

    ...

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

  end
```

```
module Nat : NAT =
  struct
    ...

    let foo f = f (-1)

  end
```

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

Must prove:

for all f valid for type t -> t
foo f is valid for type t

Proof?

Consider any f valid for type t -> t
for all arguments v, if inv (v) then inv (f v).
What can we prove about f (-1) ?

Nothing!

Exercise

```
module type NAT =
  sig

    type t

    val from_int : int -> t

    val to_int : t -> int

    val map : (t -> t) -> t -> t list

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

  end
```

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

create a program that loops forever

```
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)

    let foo f = f (-1)

  end
```

Summary of Proof Obligations

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

- Let t be the abstract type and 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
    - inv(v)
  - (v1, v2) is valid for s1 * s2 if
    - v1 is valid for s1, and
    - v2 is valid for s2
  - 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 -> s2 if
    - for all arguments a, if a is valid for s1, then v a is valid for s2
  - v is valid for int if
    - always
  - [v1; ...; vn] is valid for type s list if
    - v1 ... vn are all valid for type s