Reasoning About Modular Programs

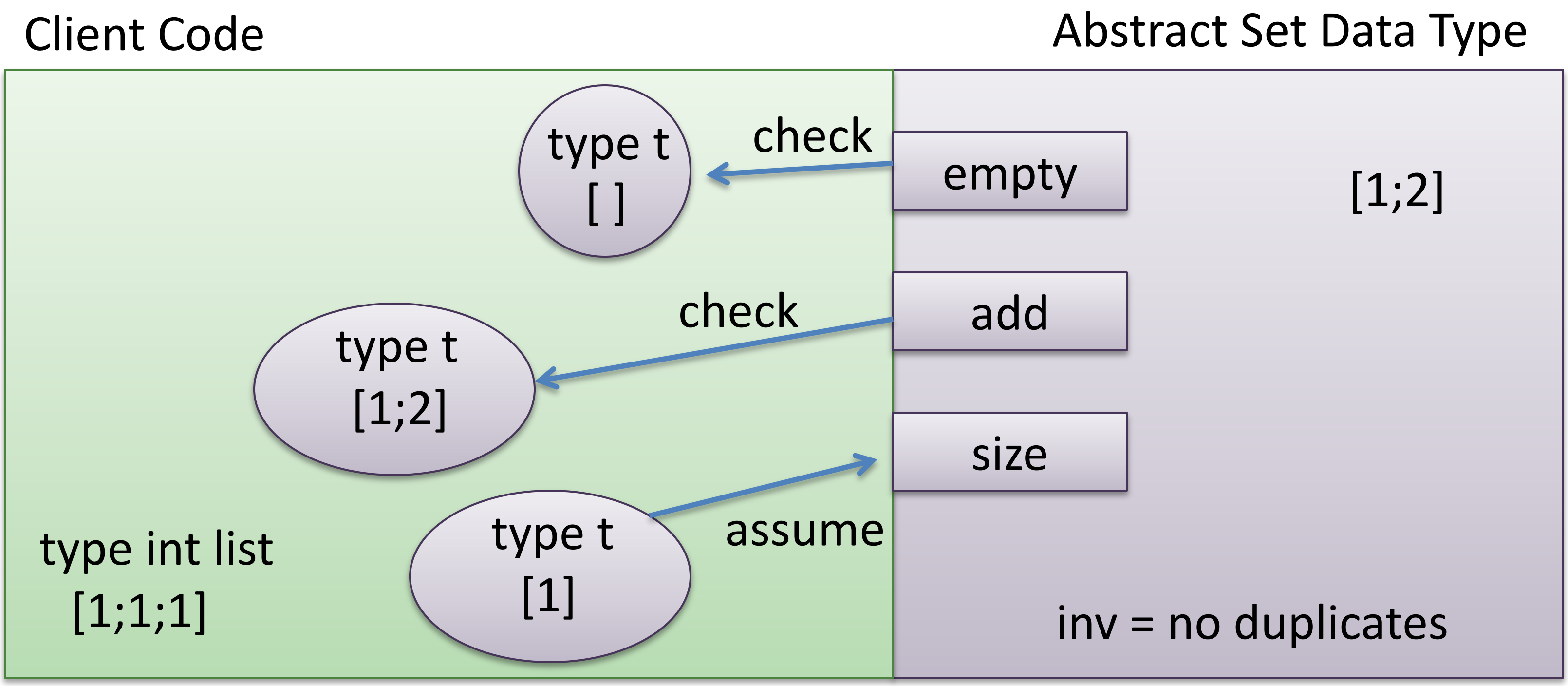
Part 2: Proving Representation Invariants

Speaker: David Walker

COS 326

Princeton University

Representation Invariants



All values of abstract type must satisfy the representation invariant.
Assume the invariant on entry to the module; prove it on leaving the module

A Signature for Sets

```
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
```

Proving 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
```

Definition of empty:

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

Proof Obligation:

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

Proof:

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

Proving 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 add:

```
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

Aside: Universal Theorems

Lots of theorems have the form:

forall x:t. P(x)

To prove such theorems, we often pick an arbitrary representative r of the type t and then prove P(r) is true.

(Often times we just use “x” as the name of the representative. This just helps prevent a proliferation of names.)

If we can’t do the proof by picking an arbitrary representative, we may want to split values of type t into cases or use induction

Aside: Conditional Theorems

Lots of theorems have the form:

if P(x) then Q(y)

To prove such theorems, we typically assume P(x) is true and then under that assumption, prove Q(y) is true.

Such conditionals are equivalent to logical implications:

P(x) ==> Q(y)

Aside: Conditional Theorems

Putting ideas together, proving:

for all x:t,y:t’, if P(x) then Q(y)

will involve:

(1) picking arbitrary x:t, y:t’
(2) assuming P(x) is true and then using that assumption to
(3) prove Q(y) is true.

Representation Invariants

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

```
let add (x:'a) (l:'a set) : 'a set =
  if mem x l then l else x::l
```

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 into two cases:

-- one case when mem x l is true

-- one case where mem x l is false

Representation Invariants

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

```
let add (x:'a) (l:'a set) : 'a set =
  if mem x l then l else x::l
```

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).

<u>case 1:</u> 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), eval)
== true (by (2))

Representation Invariants

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

```
let add (x:'a) (l:'a set) : 'a set =
  if mem x l then l else x::l
```

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 2: assume (3) not (mem x l) == true:

inv (add x l)	
== inv (if mem x l then l else x::l)	(eval)
== inv (x::l)	(by (3))
== not (mem x l) && inv (l)	(by eval)
== true && inv(l)	(by (3))
== true && true	(by (2))
== true	(eval)

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) 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 (→ inv (rem x l))

assume invariant on input (→ inv(l))

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 size:

```
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 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 inter:

```
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

Summary: Representation Invariants So Far

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
```

val value : t — prove: Inv (value)

val constructor : int -> t — prove: for all x:int, Inv (constructor x)

val transform : int -> t -> t — prove:
  for all x:int,
    for all v:t,
      if Inv(v)
      then Inv (transform x v)

val destructor : t -> int — assume Inv(t))