Reasoning About Modular Programs

Part 1: Representation Invariants

Speaker: David Walker

COS 326

Princeton University

Efficient Data Structures

In COS 226, you learned about all kinds of clever data structures:

- red-black trees, 2-3 trees
- union-find sets
- tries, ...

Not just any tree is a 2-3 tree.   Such tree satisfy *invariants*:

- eg: keys are in order in the tree
- eg: all paths from root to leaf have the same length

Efficient Data Structures

What are the invariants for?

- to bound time and space used
- to ensure results are correct
  - eg: red-black tree **lookup** depends upon the in-order invariant

Key Question: How do you arrange for the invariants to be preserved when client code is using your interface & calling your functions?

Answer: Use abstract types & representation invariants.

REPRESENTATION INVARIANTS

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
```

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

    (* size:  number of unique elements in the set *)
    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

  end
```

Back to Sets

The interesting operation:

```
(* size:  number of unique elements in the list *)
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
    ()
  else
    failwith m
```

Debugging with Representation Invariants

As a precondition on input sets:

```
(* size:  number of unique elements *)
let size (s:'a set) : int =
  check s "size:  bad set input";
  List.length s
```

As a postcondition on output sets:

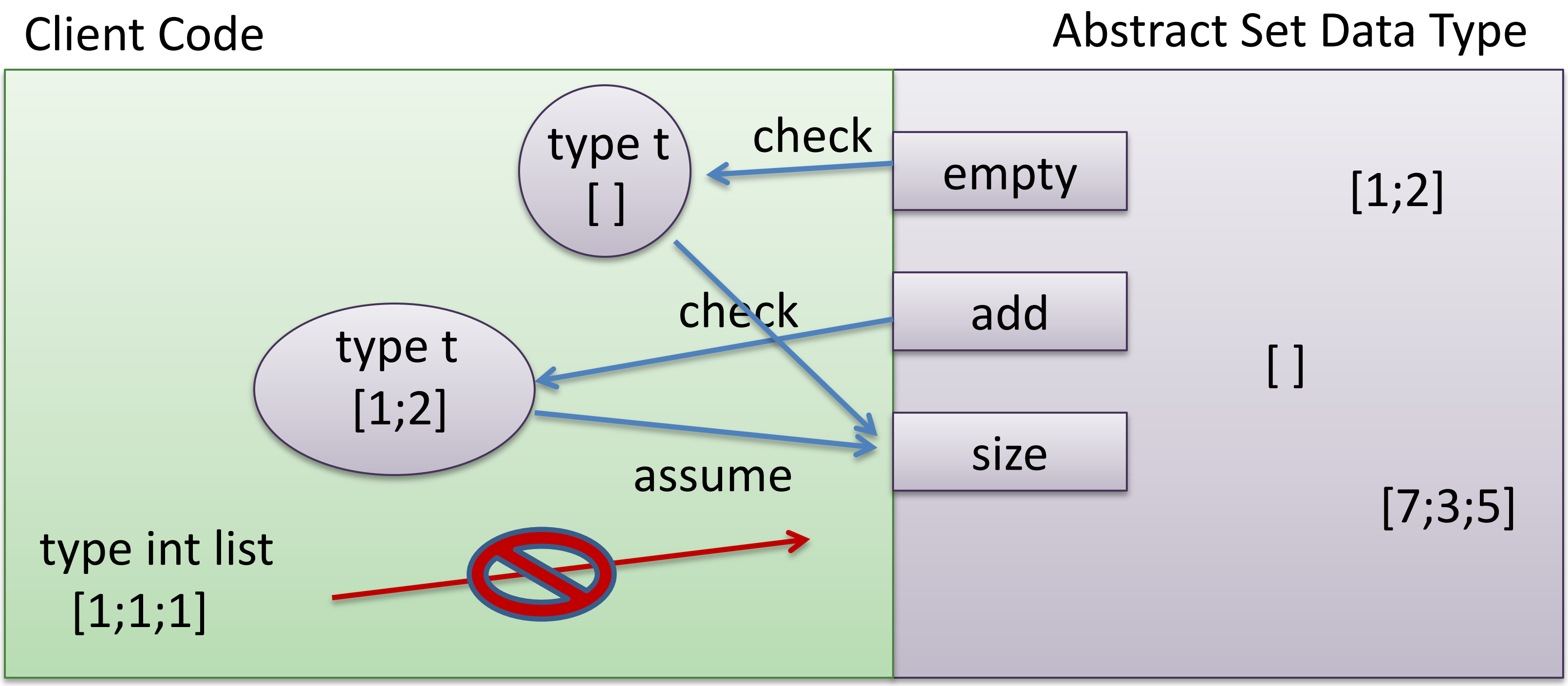
```
(* add x to set s *)
let add x s =
  let s = if mem x s then s else x::s in
  check s "add: bad set output";
  s
```

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
```

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?

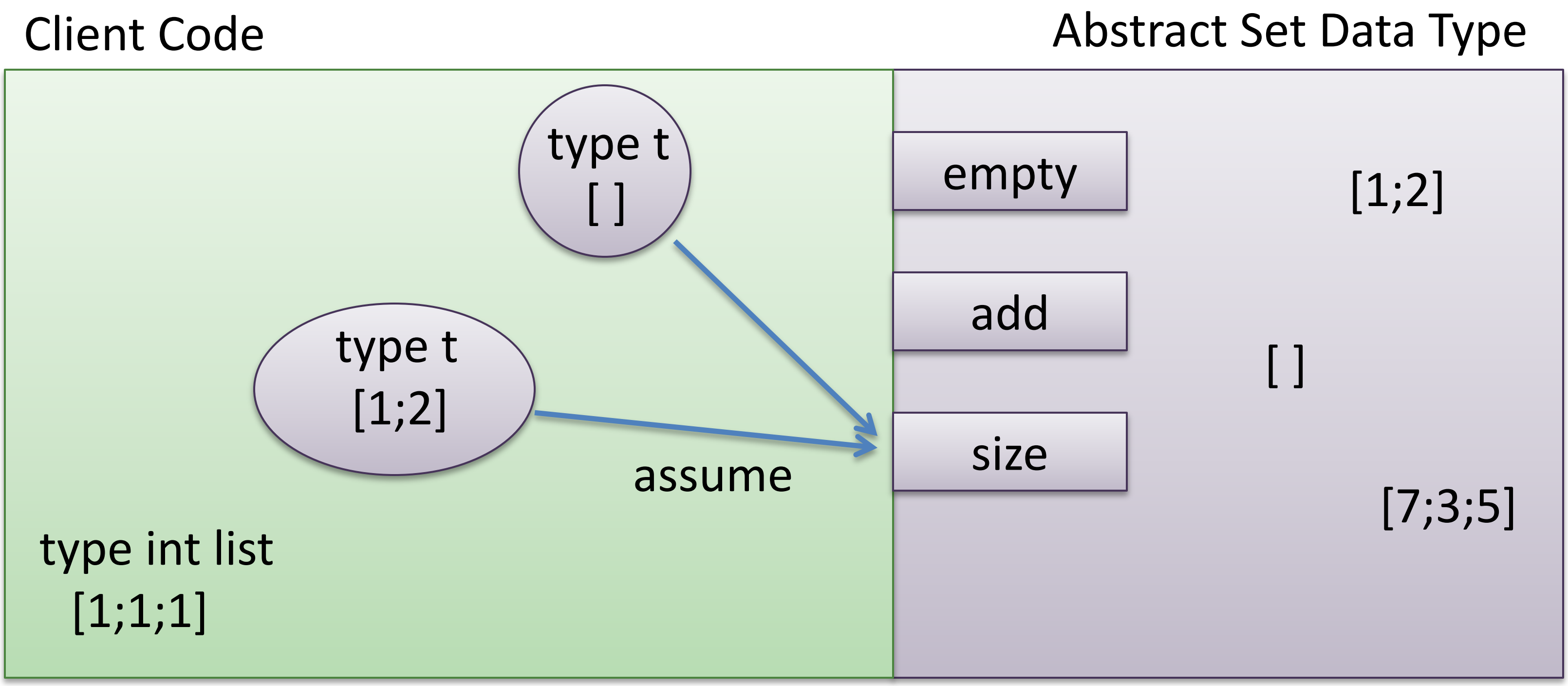
Representation Invariants Pictorially



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.

But you may want to double-check it in on entry anyway in case you made a mistake elsewhere. (In security circles, this is "defense in depth".)

Representation Invariants Pictorially



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!

Repeating myself

You may

assume the invariant inv(i) for module inputs i with abstract type

provided you

prove the invariant inv(o) for all module outputs o with abstract type

Design with Representation Invariants

A key to writing correct code is understanding your own invariants very precisely

Write down key representation invariants

- if you write them down then you can be sure you know what they are yourself!
- you may find as you write them down that they were a little fuzzier than you had thought
- easier to check, even informally, that each function and value you write satisfies the invariants once you have written them
- great documentation for others
- great debugging tool if you implement your invariant
- you'll need them to prove to yourself that your code is correct

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*