CS 3110

Proofs about Programs, part 3

Prof. Clarkson
Fall 2019

Today's music: U Plus Me by Mary J. Blige

CLICKER QUESTION 1

Review

Previously in 3110:

- Equational reasoning
- Induction on natural numbers, lists, trees
- Proofs about recursive functions on those types

Today: Algebraic specifications of data structures

STACKS

Stack

```
module type Stack = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val peek     : 'a t -> 'a
  val push     : 'a -> 'a t -> 'a t
  val pop      : 'a t -> 'a t
end
```

Specification comment

```
(** [push x s] is the stack [s]
    with [x] pushed on the top *)
val push : 'a -> 'a stack -> 'a stack
```

Not suitable for verification: no equational proof suggested by spec

Equational specification

aka *algebraic specification*

```
1.is_empty empty = true
2.is_empty (push x s) = false
3.peek (push x s) = x
4.pop (push x s) = s
```

Every equation shows how to simplify an expression

Simplification

peek (pop (push 1 (push 2 empty)))

= { simplify pop/push with eq 4 }

peek (push 2 empty)

= { simplify peek/push with eq 3 }

2

Algebraic specification

```
(a + b) + c = a + (b + c)
a + b = b + a
a + 0 = a
a + (-a) = 0
(a * b) * c = a * (b * c)
a * b = b * a
a * 1 = a
a * 0 = 0
a * (b + c) = a * b + a * c
```

Stack implementation, as list

```
module Stack = struct
  type 'a t = 'a list
  let empty = []
  let is_empty s = (s = [])
  let peek = List.hd
  let push = List.cons
  let pop = List.tl
end
```

All of our equations hold simply “by evaluation” for this impl.

Example proof: eq 4

```
   pop (push x s)
=     { eval push and pop }
   tl (x :: s)
=     { eval tl }
   s
```

QUEUES

Queue

```
module type Queue = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val front    : 'a t -> 'a
  val enq      : 'a -> 'a t -> 'a t
  val deq      : 'a t -> 'a t
end
```

Stack

```
module type Stack = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val peek     : 'a t -> 'a
  val push     : 'a -> 'a t -> 'a t
  val pop      : 'a t -> 'a t
end
```

Queue specification

- `is_empty empty = true`
- `is_empty (enq x q) = false`
- `front (enq x q)`
```
= x           if is_empty q = true
= front q     if is_empty q = false
```
- `deq (enq x q)`
```
= empty                if is_empty q = true
= enq x (deq q)        if is_empty q = false
```

Queue implementation, as list

```
module ListQueue : Queue = struct
  type 'a t = 'a list
  let empty = []
  let is_empty s = (s = [])
  let front = List.hd
  let enq x s = s @ [x]
  let deq = List.tl
end
```

All of our equations hold simply “by evaluation” for this impl.

Queue implementation, as two lists

```
module TwoListQueue : Queue = struct
  (* AF: (f, b) represents the queue f @ (List.rev b).
     RI: given (f, b), if f is empty then b is empty. *)
  type 'a t = 'a list * 'a list
  let empty = [], []
  let is_empty (f, _) = f = []
  let enq x (f, b) =
    if f = [] then [x], []
    else f, x :: b
  let front (f, _) = List.hd f
  let deq (f, b) =
    match List.tl f with
    | [] -> List.rev b, []
    | t -> t, b
end
```

Proofs in notes...

AF and RI in proofs

RI is a precondition for every operation.

E.g., for enqueue, if f is empty, then b must also be:

```
let enq x (f, b) =
  if f = [] then [x], []
  else f, x :: b
```

AF and RI in proofs

AF specifies when two concrete values should be treated as equal:

if AF(e) = AF(e') then e = e'

Extends our notion of equality

In proof we end up needing that

(rev f, [x]) = (rev (x :: f), [])

e.g.,

AF(rev [1;2;3], [4]) = (rev [1;2;3]) @ (rev [4]) = [3;2;1;4]

AF(rev (4 :: [1;2;3]), []) = (rev (4 :: [1;2;3])) @ (rev []) = [3;2;1;4]

CLICKER QUESTION 2

DESIGNING EQUATIONS

Canonical form

canonical: conforming to some rule

Only build up structure

- Not canonical: pop (push 1 (push 2 empty))
- Canonical: push 2 empty

Every value of data structure can be created solely with operations that create canonical forms

Categories of operations

- **Generator:** create canonical form
- **Manipulator:** create non-canonical form
- **Query:** create value of different type

Stack

```
module type Stack = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val peek     : 'a t -> 'a
  val push     : 'a -> 'a t -> 'a t
  val pop      : 'a t -> 'a t
end
```

generator

query

generator

manipulator

Queue

```
module type Queue = sig
  type 'a t
  val empty    : 'a t
  val is_empty : 'a t -> bool
  val front    : 'a t -> 'a
  val enq      : 'a -> 'a t -> 'a t
  val deq      : 'a t -> 'a t
end
```

generator

query

generator

manipulator

Designing equations

query

manipulator

X

generator

```
is_empty empty = true
is_empty (push x s) = false
peek (push x s) = x
pop (push x s) = s
```

Note what's missing: peek empty, pop empty

SETS

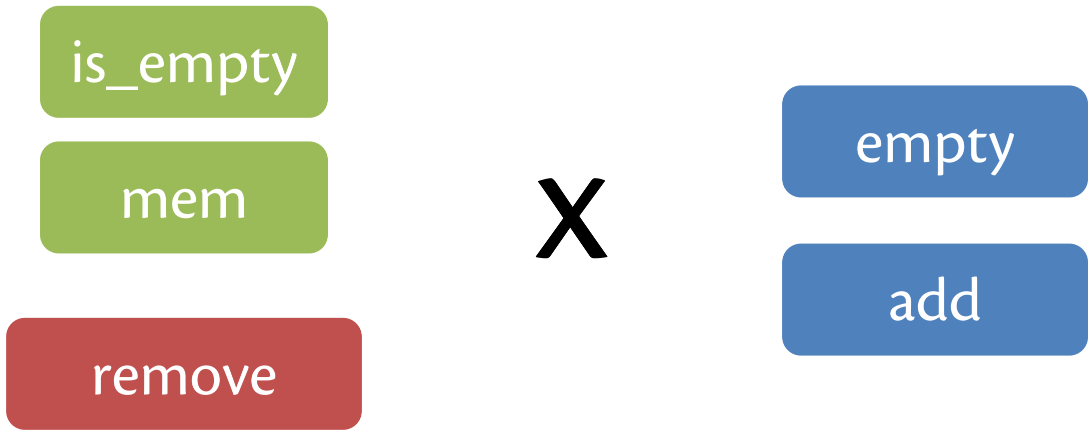
Sets

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val mem : 'a -> 'a t -> bool
  val remove : 'a -> 'a t -> 'a t
end
```

Sets

```
module type Set = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val add : 'a -> 'a t -> 'a t
  val mem : 'a -> 'a t -> bool
  val remove : 'a -> 'a t -> 'a t
end
```

Designing equations



Equational specification

- is_empty empty = true
- is_empty (add x s) = false
- mem x empty = false
- mem y (add x s) = true if x = y
- mem y (add x s) = mem y s if x <> y
- remove x empty = empty
- remove y (add x s) = remove y s if x = y
- remove y (add x s) = add x (remove y s) if x <> y

RHS of eqn applies non-generator to smaller input than LHS

Upcoming events

- [Tue/Wed]: MS2 demos
- [Thur]: MS2 due in CMS, no late submissions

This is verified.

THIS IS 3110