CS 3110

Proofs about Programs, part II

Nate Foster
Spring 2020

Today's scene: One Ring Donuts

Review

Previously in 3110:

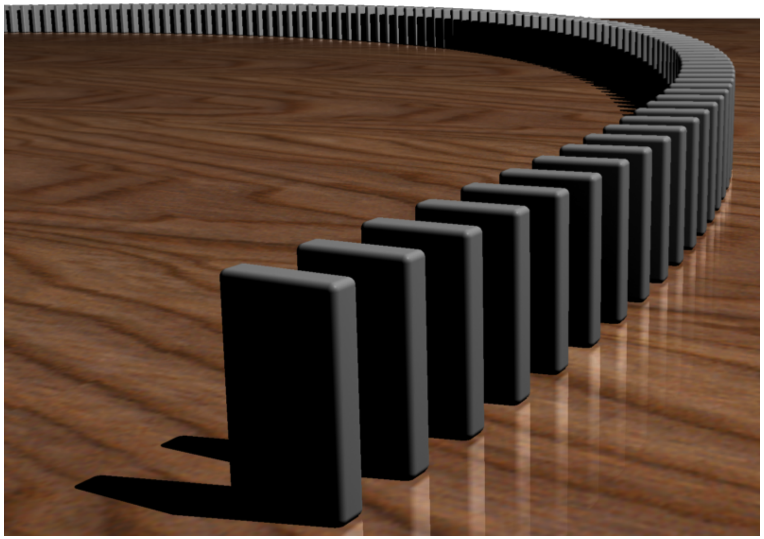
- Proofs about programs
- Equational reasoning

Today:

- Induction on natural numbers, lists, trees
- Proofs about recursive functions on those types
- Algebraic specifications of data structures

Induction principle on naturals

forall properties P,
  if P(0)
  and (forall k, P(k) implies P(k+1))
  then (forall n, P(n))



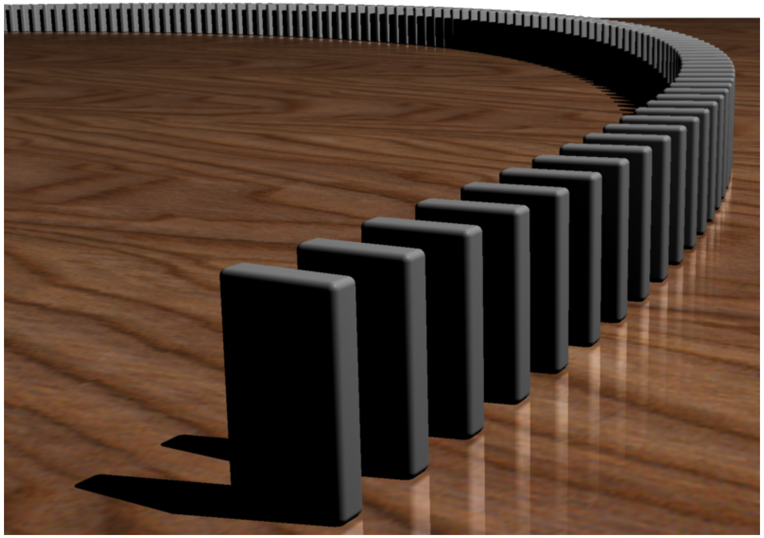
Induction principle on lists

forall properties P,

 if P([])

 and (forall h t, P t implies P (h :: t))

 then (forall lst, P lst)



Induction on lists

Theorem: forall lst, P(lst).

Proof: by induction on lst

Base case: lst = []
Show: P([])

Inductive case: lst = h :: t
IH: P(t)
Show: P(h :: t)

QED

Example: map

Theorem: forall f g,

(map f) << (map g) = map (f << g)

```
let rec map f = function
  | [] -> []
  | h :: t -> f h :: map f t

let compose f g x = f (g x)
let (<<) = compose
```

Binary trees

```
type 'a tree =
  | Leaf
  | Node of 'a tree * 'a * 'a tree
```

Induction principle on trees

forall properties P,
  if P(Leaf)
  and (forall v l r,
    (P(l) and P(r)) implies P(Node (l, v, r)))
  then forall t, P(t)

Induction on trees

Theorem: forall t, P(t).

Proof: by induction on t

Case: n = Leaf
Show: P(Leaf)

Case: n = Node (l, v, r)
IH1: P(l)
IH2: P(r)
Show: P(Node (l, v, r))

QED

Example: leaves and nodes

```
let rec nodes = function
   | Leaf -> 0
   | Node (l, _, r) ->
     1 + nodes l + nodes r


let rec leaves = function
   | Leaf -> 1
   | Node (l, _, r) ->
     leaves l + leaves r
```

Theorem: forall t, leaves t = 1 + nodes t

What induction principles tell us about

INDUCTION VS. RECURSION

Induction vs. recursion

Inductive proofs are like recursive programs

	Proofs	Programs
Per constructor...	One proof case	One pattern-matching branch
On smaller value...	Use IH	Make recursive call

PART II: ALGEBRAIC SPECIFICATIONS

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
```

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

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

is_empty

mem

remove

x

empty

add

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

- [Friday]: Project MS2 due on CMS
- [Monday/Tuesday]: Project MS2 demos

This is verified.

THIS IS 3110