Proving Properties of Recursive Functions and Data Structures

CS 550 Programming Languages
Jeremy Johnson
Objective

- To implement and verify recursive functions for processing recursive data structures.
- To use structural induction to prove properties of recursive functions and recursive data structures.
- To study a non-trivial example involving arithmetic expressions
Outline

- **Binary Trees**
  - Definition
  - Height and IPL
  - Full binary trees
  - Call trees and number of internal and leaf nodes

- **Arithmetic Expressions**
  - Evaluation
  - Differentiation
  - Simplification
Binary Trees

- **Recursive Definition**
  - Either empty or consists of value, and recursively a left and a right binary tree
  - BinTree := null | (Any BinTree BinTree)

- **Constructor**
  - (define (BinTree value left right)
    (list value left right))
  - (BinTree 1 (BinTree 2 null null)
    (BinTree 3 (BinTree 4 null null)
      (BinTree 5 null null)))
  - '(1 (2 null null) (3 (4 null null) (5 null null)))
Height of Binary Trees

Height of tree T, H(T), is the max length of the paths from the root all of the leaves

; Input: T a binary tree
; Output: a non-negative integer = height of T
(define (Height BT)
  (if (null? BT)
      -1
      (+ (max (Height (left BT))
              (Height (right BT)))
         1))))
Correctness of Height

- Show (Height BT) = length of the longest path in BT
- Prove by structural induction
  - Base case. -1 for null is needed to obtain the correct height of a tree with a single node.
  - Assume (Height (left BT)) and (Height (right BT)) compute the lengths of the longest paths in the left and right subtrees
  - The longest path in BT is one more than the longest path in either the left or right subtrees which by induction is one more than the maximum of (Height (left BT)) and (Height (right BT))
Internal Path Length (IPL)

- The sum of the length of the paths from the root to every node in the tree

- Recursive definition

; Input: T a binary tree
; Output: a non-negative integer = internal path length of T
(define (IPL (BT))
  (if (null? null)
      0
      (+ (IPL (left BT))
         (IPL (right BT))
         (- (Nnodes BT) 1))))
IPL Verification

(check= (IPL nil) 0)
(check= (let ((BT (consBinTree 1 (consBinTree 2 nil nil)
                          (consBinTree 3 (consBinTree 4 nil nil)
                          (consBinTree 5 nil nil))))) (IPL BT)) 6)

IPL = 6 = 0 + 2 + 4

Nnodes = 5
IPL((left BT)) = 0
IPL((right BT)) = 2
Correctness of IPL

- Show $(ipl \ BT) = \text{sum of the lengths of the paths from the root to all nodes in the tree}$.

- Prove by structural induction
  - Base case. 0 since there are no paths in empty tree
  - Assume $(ipl \ (\text{left } BT))$ and $(ipl \ (\text{right } BT))$ compute the sum of the lengths of paths in the left and right subtrees
  - The length of each path in $BT$ is one more than the lengths of the paths in the left and right subtrees. Thus one has to be added for each node in the left and right subtrees and since there are $(- (N\text{nodes } BT) \ 1)$ such nodes this must be added to the sum of $(ipl \ (\text{left } BT))$ and $(ipl \ (\text{right } BT))$
Call Tree

- Nodes for each instance of a function call
- Parent corresponds to function that made the call
- Children correspond to the calls that instance makes
- Leaf nodes correspond to base cases

```
(define (fib n)
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fib (- n 1)) (fib (- n 2)))])
)
```

- Number of leaf nodes $L = (fib\ n)$
- Number of internal nodes $= L - 1$
- Exponential number of nodes for $(fib\ n)$
Full Binary Trees

- Binary tree where every node has either 0 or 2 children.
- FBTREE := (All null null) | (All FBTREE FBTREE)

L = l + 1
The number of leaf nodes is one more than the number of internal nodes.

- Proof by structural induction on $T$
- Base case. $T$ has one node.
- Induction. Assume the property for all smaller trees, and

Remove two leaf nodes

$L = 3$
$L = 2$
The number of leaf nodes is one more than the number of internal nodes.

- Proof by structural induction on $T$
- Base case. $T$ has one node.
- Induction. Assume the property for all smaller trees, and

Remove two leaf nodes

\[
L' = L-2+1 = L-1 = 2
\]
\[
l' = l-1 = 1
\]
Arithmetic Expressions

- AExp := Constant | Variable | (+ AExp AExp) | (* AExp AExp)

- E.G. \(x \cdot (x + 5)\)
Predicate

; Input: Any
; Output: (boolean (arith-expr? expr) #t if expr is an arithmetic expression

(define (arith-expr? expr)
  (cond
    [ (constant? expr) #t ]
    [ (variable? expr) #t ]
    [ (plus? expr) (and (arith-expr? (op1 expr))
                         (arith-expr? (op2 expr))) ]
    [ (mult? expr) (and (arith-expr? (op1 expr))
                         (arith-expr? (op2 expr))) ]
    [else #f ]
  ))
Expression Tree Size

; Inputs:  $E$ is an arithmetic expression
; Output: a non-negative integer equal to the number of constructors in $E$.

(define (ExpTreeSize E)
  (cond
   [(constant? E) 1]
   [(variable? E) 1]
   [(plus? E) (+ 1 (ExpTreeSize (op1 E)) (ExpTreeSize (op2 E)))]
   [(mult? E) (+ 1 (ExpTreeSize (op1 E)) (ExpTreeSize (op2 E)))]
   [else #f])))
Environments

❖ Need to look up the values of all variables occurring in the expression

❖ Environment contains a list of bindings where a binding is a pair (name value) that binds a value to the name of the variable

  • E.G. env = ((x 2) (z 5))

❖ lookup returns the value associated with a given variable in an environment

  • (lookup ‘x env) → 2
Evaluation

; Input: (and (arith-expr? expr) (environment? env))
; all variables in expr are bound in env
; Output: a number = to the evaluation of expr with values of vars in env

(define (arith-eval expr env)
  (cond
   [ (constant? expr) expr ]
   [ (variable? expr) (lookup expr env) ]
   [ (plus? expr) (+ (arith-eval (op1 expr) env)
                      (arith-eval (op2 expr) env)) ]
   [ (mult? expr) (* (arith-eval (op1 expr) env)
                      (arith-eval (op2 expr) env)) ]
  )
)
Properties of Differentiation

- **Definition**

\[
\frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}
\]

- **Derivative of \( x \) and \( x^2 \)**

\[
\frac{d}{dx} x = \lim_{\Delta x \to 0} \frac{(x + \Delta x) - x}{\Delta x} = 1
\]

\[
\frac{d}{dx} x^2 = \lim_{\Delta x \to 0} \frac{(x + \Delta x)^2 - x^2}{\Delta x} = 2x
\]
Properties of Differentiation

- Linearity of derivative

\[ \frac{d}{dx} (f(x) + g(x)) = \frac{d}{dx} f(x) + \frac{d}{dx} g(x) \]

- Product rule

\[ \frac{d}{dx} (f(x)g(x)) = f(x) \frac{d}{dx} g(x) + g(x) \frac{d}{dx} f(x) \]
Properties of Differentiation

- **Linearity of derivative**
  - Proof follows from definition

- **Product rule**
  - Proof follows from definition using
  - $f(x+\Delta x) = f(x) + f'(x)\Delta x + O(\Delta x^2)$
  - $g(x+\Delta x) = g(x) + g'(x)\Delta x + O(\Delta x^2)$
Properties of Differentiation

- Derivative of powers
  \[ \frac{d}{dx} x^n = nx^{n-1} \]

- Proof by induction on \( n \)
  
  1. **Base case** \( n=1 \)
  
  2. **Assume true for smaller powers**

\[
\frac{d}{dx} x^n = \frac{d}{dx} x \cdot x^{n-1} = x \frac{d}{dx} x^{n-1} + x^{n-1} \cdot \frac{d}{dx} x
\]

\[
= (n-1)x^{n-1} + x^{n-1} = nx^{n-1}
\]
Differentiation

; Input: (and (arith-expr? expr) (variable? var))
; Output: (arith-expr? (diff expr var)) equal to derivative of expr wrt var

(define (diff expr var)
  (cond
   [(constant? expr) 0]
   [(variable? expr) (if (equal? expr var) 1 0)]
   [(plus? expr) (make-plus (diff (op1 expr) var)
                              (diff (op2 expr) var))]
   [(mult? expr) (make-plus
                    (make-mult (op1 expr) (diff (op2 expr) var))
                    (make-mult (diff (op1 expr) var) (op2 expr)))])
)

Differentiation Example

\[ \frac{d}{dx} x^*(x + 5) = 1^* (x+5) + x^*(1 + 0) \]
Simplification of Plus

; Input: (and (arith-expr? expr1) (arith-expr? expr2))
; Output: (arith-expr? (plus-simp expr1 expr2))
; which is equivalent to expr

(define (plus-simp expr1 expr2)
  (cond
   [ (and (constant? expr1) (constant? expr2)) (+ expr1 expr2) ]
   [ (equal? expr1 0) expr2 ]
   [ (equal? expr2 0) expr1 ]
   [ (make-plus expr1 expr2) ]
  )
)
)
Simplification of Mult

; Input: (and (arith-expr? expr1) (arith-expr? expr2))
; Output: (arith-expr? (mult-simp expr1 expr2))
; which is equivalent to expr

(define (mult-simp expr1 expr2)
  (cond
   [ (and (constant? expr1) (constant? expr2)) (* expr1 expr2) ]
   [ (equal? expr1 0) 0 ]
   [ (equal? expr2 0) 0 ]
   [ (equal? expr1 1) expr2 ]
   [ (equal? expr2 1) expr1 ]
   [ else (make-mult expr1 expr2) ]
  )
)
Recursive Simplification

; Input: (arith-expr? expr)
; Output: (arith-expr? (arith-simp expr)) equivalent to expr

(define (arith-simp expr)
  (cond
   [ (constant? expr) expr ]
   [ (variable? expr) expr ]
   [ (plus? expr) (let ([simpexpr1 (arith-simp (op1 expr))]
                         [simpexpr2 (arith-simp (op2 expr))])
                 (plus-simp simpexpr1 simpexpr2)) ]
   [ (mult? expr) (let ([simpexpr1 (arith-simp (op1 expr))]
                         [simpexpr2 (arith-simp (op2 expr))])
                 (mult-simp simpexpr1 simpexpr2)) ]
   )
  )
)
\[ 1 \cdot (x+5) + x \cdot (1 + 0) = (x+5) + x \]
Properties

1. \((\text{diff expr var}) = \frac{d}{d\text{var}} \text{expr}\)

2. \((\text{arith-eval (arith-simp expr) env}) = (\text{arith-eval expr env})\)

3. \((\text{is-simplified (arith-simp expr)})\)
Is-Simplified?

; Input:  (arith-expr expr)
; Output: (boolean? (is-simplified? expr).
; #t if expr is simplified
(define (is-simplified? expr)
  (if (constant? expr)
      #t
      (and (noconstant-arith? expr)
           (nozeros? expr)
           (nomult1? expr))))
Check for Nozeros

; Input: (arith-expr expr)
; Output: (boolean? (nozeros? expr)). #t if expr contains any zeros.
(define (nozeros? expr)
  (cond
   [ (constant? expr) (not (equal? expr 0)) ]
   [ (variable? expr) #t ]
   [ (plus? expr) (and (nozeros? (op1 expr))
                       (nozeros? (op2 expr))) ]
   [ (mult? expr) (and (nozeros? (op1 expr))
                       (nozeros? (op2 expr))) ]
  )
)
)
Check for No Multiplications by 1

; Input: (arith-expr expr)
; Output: (boolean? (nomult1? expr)). #t if expr contains no multis by 1.
(define (nomult1? expr)
  (cond
   [ (constant? expr) #t ]
   [ (variable? expr) #t ]
   [ (plus? expr) (and (nomult1? (op1 expr))
                      (nomult1? (op2 expr))) ]
   [ (mult? expr) (if (or (equal? (op1 expr) 1) (equal? (op2 expr) 1))
                   #f
                   (and (nomult1? (op1 expr)) (nomult1? (op2 expr)))) ]
  )
)
No Constant Arithmetic

; Input: (arith-expr expr)
; Output: (boolean? (constant-arith? expr).
; #t if expr contains + or * with both operands constants
(define (noconstant-arith? expr)
  (cond
   [ (constant? expr) #f ]
   [ (variable? expr) #f ]
   [ (plus? expr) (if (not (and (constant? (op1 expr)) (constant? (op2 expr))))
                         #t
                         (and (noconstant-arith? (op1 expr)) (noconstant-arith? (op2 expr)))]
   [ (mult? expr) (if (not (and (constant? (op1 expr)) (constant? (op2 expr))))
                         #t
                         (and (noconstant-arith? (op1 expr)) (noconstant-arith? (op2 expr)))]
   [else #f ]
  ))
Proof of Property 1

- By induction on expr (structural induction)

- Base cases
  1. $expr = \text{constant}$
     \[(\text{diff } expr \ var) = 0 = \frac{d}{d\text{var}} expr\]
  2. $expr = \text{a variable}$
     \[(\text{diff } expr \ var) = 1 \text{ if } expr = \text{var}\]
Proof of Property 1

\[ \text{Expr} = (+ E_1 E_2) \]

- Assume \((\text{diff } E_i \text{ var}) = d E_i/d\text{var} \) [IH]

- \((\text{diff } (+ E1 E2) \text{ var}) = (+ (\text{diff } E1 \text{ var}) (\text{diff } E2 \text{ var})) \) [By def of diff]

\[ = d E_1/d\text{var} + d E_2/d\text{var} \] [by IH]

\[ = d \text{expr}/d\text{var} \] [by linearity of the derivative]
Proof of Property 1

- Expr = (* E₁ E₂)
  - Assume (diff Eᵢ var) = d Eᵢ/dvar [IH]

- (diff (* E₁ E₂) var)
  = (+ (* E₁ (diff E₂ var))
    (* (diff E₁ var) E₂))       [By def of diff]
  = E₁d E₂/dvar + d E₁/dvar E₂  [by IH ]
  = d expr/dvar               [by product rule]
Proof of Property 2

- **Lemma 1.** \((\text{arith-eval} \ (\text{plus-simp} \ E1 \ E2) \ env) = (\text{arith-eval} \ (+ \ E1 \ E2) \ env)\)

- **Proof by case analysis**
  1. \(E1 \text{ and } E2 \text{ constants}\)
  2. \(E1 = 0\)
  3. \(E2 = 0\)
  4. \(\text{General case}\)
Lemma 1. \((\text{arith-eval} \ (\text{plus-simp} \ E1 \ E2) \ env) = (\text{arith-eval} \ (+ \ E1 \ E2) \ env)\)

Proof by case analysis

1. \(E1\) and \(E2\) constants
\[(\text{arith-eval} \ (\text{plus-simp} \ E1 \ E2) \ env) = (\text{arith-eval} \ (+ \ E1 \ E2) \ env)\) [by def of plus-simp]
\[= (+ (\text{arith-eval} \ E1 \ env) (\text{arith-eval} \ E2 \ env))\) [by def of arith-eval applied to constants]
\[= (\text{arith-eval} \ ‘(+ \ E1 \ E2) \ env)\) [by def of arith-eval]
Proof of Property 2

- **Lemma 1.** \( (\text{arith-eval} (\text{plus-simp} E_1 E_2) \ \text{env}) = (\text{arith-eval} (+ E_1 E_2) \ \text{env}) \)

- **Proof by case analysis**
  1. \( E_1 = 0 \)
     \[
     (\text{arith-eval} (\text{plus-simp} E_1 E_2) \ \text{env}) \\
     = (\text{arith-eval} E_2 \ \text{env}) \ [\text{by def of plus-simp}] \\
     = (+ (\text{arith-eval} E_1 \ \text{env}) (\text{arith-eval} E_2 \ \text{env})) \ [\text{by def of arith-eval and arithmetic since } E_1 = 0] \\
     = (\text{arith-eval} '(+ E_1 E_2) \ \text{env}) \ [\text{by def of arith-eval}]
     \]
Proof of Property 2

- By induction on expr (structural induction)

- Base cases
  1. expr = constant
     
     \[ (\text{arith-eval} \ (\text{arith-simp} \ expr) \ \text{env}) = (\text{arith-eval} \ expr \ \text{env}) \]  
     
     [by def of arith-simp]
  2. expr a variable
     
     \[ (\text{arith-eval} \ (\text{arith-simp} \ expr) \ \text{env}) = (\text{arith-eval} \ expr \ \text{env}) \]  
     
     [by def of arith-simp]
Proof of Property 2

- Expr = (+ E1 E2)
  - Assume (arith-eval (arith-simp Ei) env) = (arith-eval Ei env)
  - (arith-eval (arith-simp (+ E1 E2)) env)

  = (arith-eval (plus-simp (arith-simp E1) (arith-simp E2)) env) [by def of arith-simp]

  = (arith-eval (+ (arith-simp E1) (arith-simp E2)) env) [by Lemma 1]
Proof of Property 2

$\sqrt{Expr = (+ E1 E2)}$

$= (+ (arith-eval (arith-simp E1) env) (arith-eval (arith-simp E2) env))$ [by def of arith-eval]

$= (+ (arith-eval E1 env) (arith-eval E2 env))$ [by IH]

$= (arith-eval (+ E1 E2) env)$ [by def of arith-eval]
Proof of Property 2

Lemma 2. \((\text{arith-eval (mult-simp } E1 \ E2) \ \text{env}) = (\text{arith-eval } (* \ E1 \ E2) \ \text{env})\)

Proof by case analysis

1. \(E1\) and \(E2\) constants
2. \(E1 = 0\)
3. \(E2 = 0\)
4. \(E1 = 1\)
5. \(E2 = 1\)
6. General case
Exercise

- Prove Lemma 2 using case analysis and equational reasoning.

- Prove $(\text{arith-eval} \ (\text{arith-simp } \textit{expr}) \ \textit{env})$
  \[= (\text{arith-eval } \textit{expr} \ \textit{env})\] when $\textit{expr} = (\ast \ E1 \ E2)$

- Prove property 3