CSCI-344
Programming Language Concepts (Section 3)

Lecture 7

Equational Reasoning and Algebraic Laws

Instructor: Hossein Hojjat

September 12, 2016

Where we are

Scheme: one of the main dialects of Lisp (List processor)

Done:

- S-expressions
- Recursive definition of lists

This session: Algebraic Laws, Equational Proofs

Recursive Functions for Recursive Data Types

- List is a recursive datatype: it mentions itself in its own definition
- A list is constructed with '() or with `cons` applied to an atom and a smaller list `(cons a as)`
- Recursive functions are used for handling recursive data structures
- Recursive functions for processing lists:
  - Base case is usually `nil`
  - Inductive case is usually the rest or `cdr` of the list

Length

```
(define length (xs)
  (if (null? xs) 0
    (+ 1 (length (cdr xs)))
  )
)
```

$$
\begin{aligned}
&\texttt{length '(a b c)} \\
=\ & 1 + \texttt{length '(b c)} \\
=\ & 1 + (\,1 + \texttt{length '(c)}\,) \\
=\ & 1 + (\,1 + (\,1 + \texttt{length '()}\,)\,) \\
=\ & 1 + (\,1 + (\,1 + 0\,)\,) \\
=\ & 3
\end{aligned}
$$

Length

```
(define length (xs)
  (if (null? xs) 0
    (+ 1 (length (cdr xs)))
  )
)
```

```
  length '(a b c)
= 1 + length '(b c)
= 1 +( 1 + length '(c) )
= 1 +( 1 +( 1 + length '() ))
= 1 +( 1 +( 1 + 0 ))
= 3
```

```
;;; tail-recursive
(define (length xs)
  (define (len xs n)
    (if (null? xs) n
      (len (cdr xs) (+ 1 n))
    )
  )
(len xs 0))
```

```
   (len '(a b c) 0)
= (len '(b c) 1)
= (len '(c) 2)
= 3
```

Algebraic Laws for Lists

- We can use **algebraic laws** to reduce and simplify complex expressions
  - Example: Identity law for addition
  - For any natural number x: $x + 0 = x$
  - This allows us to simplify the expression $x + 0$ to x

- The same way we can define algebraic laws for lists
  - Example: For all S-expressions `x` and `y`:

```
(car (cons x y)) = x
```

Algebraic Laws for Lists

Some Algebraic Laws: (there is an infinite number of laws)

```
(null?  '())          =  #t
(null?  (cons x y))   =  #f
(pair?  '())          =  #f
(pair?  (cons x y))   =  #t
(car (cons x y))      =   x
(cdr (cons x y))      =   y
```

List Operations:

- **creators/producers:** `'()` , `(cons x y)`
- **observers:** `null?` , `pair?` , `car` , `cdr`

Append

- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume $++$ is an informal math notation for "followed by"

```
xs ++ '()          =          xs
'() ++ ys          =          ys
(x ++ xs) ++ ys    =   x ++ (xs ++ ys)
xs ++ (y ++ ys)    =   (xs ++ y) ++ ys
```

Append

- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume $++$ is an informal math notation for "followed by"

```
xs ++ '()          =          xs
'() ++ ys          =          ys
(x ++ xs) ++ ys    =   x ++ (xs ++ ys)
xs ++ (y ++ ys)    =   (xs ++ y) ++ ys
```

- `snoc`: reverse of `cons`

Append

- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume $++$ is an informal math notation for "followed by"

```
xs ++ '()          =          xs
'() ++ ys          =          ys
(x ++ xs) ++ ys    =   x ++ (xs ++ ys)
xs ++ (y ++ ys)    =   (xs ++ y) ++ ys
```

- `snoc`: reverse of `cons`
  - There is no `snoc` operation

Append

- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume $++$ is an informal math notation for "followed by"

```
xs ++ '()          =          xs
'() ++ ys          =          ys
(x ++ xs) ++ ys    =   x ++ (xs ++ ys)
xs ++ (y ++ ys)    =   (xs ++ y) ++ ys
```

- `snoc`: reverse of `cons`
  - There is no `snoc` operation

Append

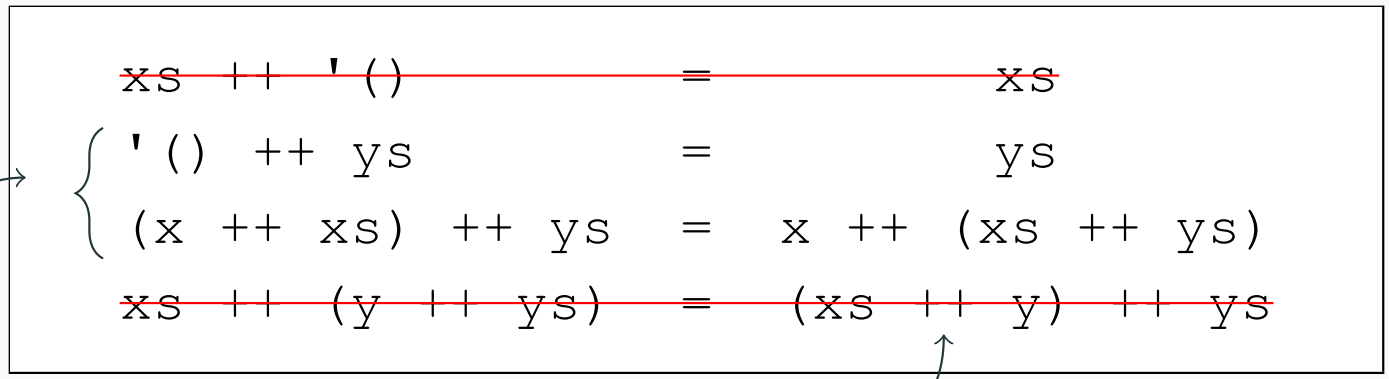
- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume $++$ is an informal math notation for "followed by"

```
  xs ++ '()          =          xs
  '() ++ ys          =          ys
  (x ++ xs) ++ ys    =   x ++ (xs ++ ys)
  xs ++ (y ++ ys)    =   (xs ++ y) ++ ys
```

- `snoc`: reverse of `cons`
  - There is no `snoc` operation
- Complete case analysis on the first argument

Append

- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume $++$ is an informal math notation for "followed by"



- `snoc`: reverse of `cons`
  - There is no `snoc` operation
- Complete case analysis on the first argument

Append

- We can often transform algebraic laws for an operation to an actual Scheme program
- Consider the algebraic laws we want `append` to satisfy
- Assume ++ is an informal math notation for "followed by"

```
xs ++ '()          =          xs
'() ++ ys          =          ys
(x ++ xs) ++ ys    =   x ++ (xs ++ ys)
xs ++ (y ++ ys)    =   (xs ++ y) ++ ys
```

```
(define (append xs ys)
  (if (null? xs) ys
    (cons (car xs) (append (cdr xs) ys))))
```

Cost Model

```
(define (append xs ys)
  (if (null? xs) ys
    (cons (car xs) (append (cdr xs) ys))))
```

- Major cost is `cons` because it corresponds to allocation
- How many pairs are allocated by the operation `(append xs ys)`?
- **Claim:** Cost of append is linear in length of the first argument
- **Proof:** Structural Induction.

snoc

```
(define (append xs ys)
  (if (null? xs) ys
    (cons (car xs) (append (cdr xs) ys))))
```

- We can define `snoc` in terms of append:

  ```
  (define (snoc xs x) (append xs (list1 x)))
  ```

- The `list1` function maps an atom `x` to the singleton list containing `x`

  ```
  (define (list1 x) (cons x '()))
  ```

List Reversal

Algebraic laws for list reversal:

```
reverse '()          =          '()
reverse (x ++ xs)    =   (reverse xs) ++ x
```

Scheme implementation corresponding the algebraic laws:

```
(define reverse (xs)
   (if (null? xs) '()
       (append (reverse (cdr xs))
               (list1 (car xs)))))
```

List Reversal

Algebraic laws for list reversal:

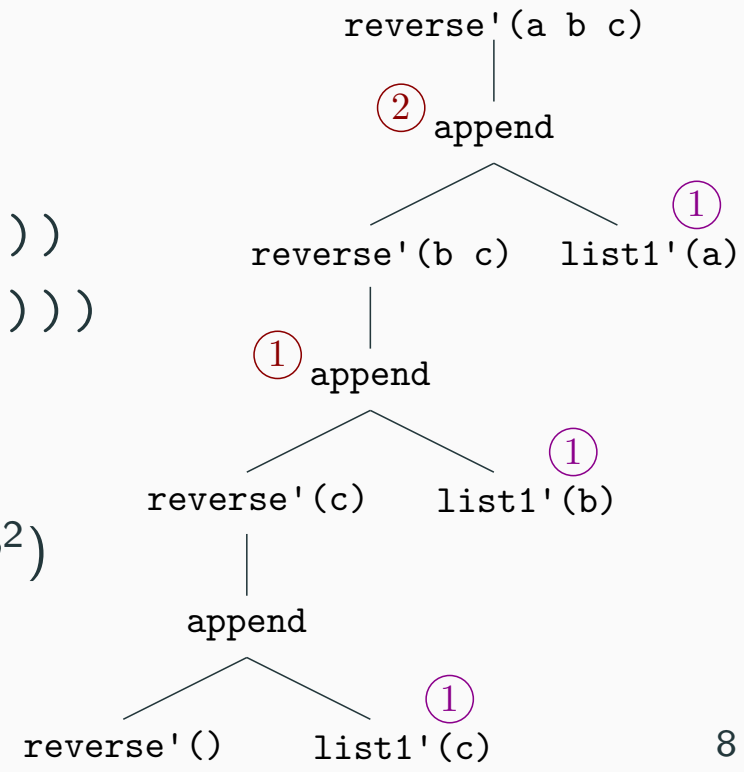
```
reverse '()          =          '()
reverse (x ++ xs)    =   (reverse xs) ++ x
```

Scheme implementation corresponding the algebraic laws:

```
(define reverse (xs)
   (if (null? xs) '()
       (append (reverse (cdr xs))
               (list1 (car xs)))))
```

How many pairs are constructed?

$$\Big((n-1)+(n-2)+\cdots+1\Big)+n(1)=O(n^2)$$



List Reversal (Accumulating Parameter)

Consider a different set of algebraic laws for list reversal:

```
(reverse (x ++ xs)) ++ zs  =  (reverse xs) ++ x ++ zs
reverse '() ++ zs          =               zs
```

```
(define revapp (xs zs)
   (if (null? xs) zs
       (revapp (cdr xs)
               (cons (car xs) zs))))

(define reverse (xs) (revapp xs '()))
```

List Reversal (Accumulating Parameter)

Consider a different set of algebraic laws for list reversal:

```
(reverse (x ++ xs)) ++ zs  =  (reverse xs) ++ x ++ zs
reverse '() ++ zs          =              zs
```

```
(define revapp (xs zs)
   (if (null? xs) zs
       (revapp (cdr xs)
               (cons (car xs) zs))))

(define reverse (xs) (revapp xs '()))
```

- The cost of this version is linear in the length of the list being reversed
- Parameter `zs` is the accumulating parameter (powerful and general technique)

Equational Reasoning

- How can we prove different properties of programs?
- Take a set of laws as definition for primitive operations
  - e.g. `car`, `cons`, `cdr`
- Use "Equational Reasoning": substituting equals by equals for the rest of operations
  - e.g. `length`, `append`, `reverse`
- Equational reasoning expands (or contract) definitions of functions

$$
\begin{aligned}
& term_1 \\
&= \{\text{Justification that } term_1 = term_2\} \\
& term_2
\end{aligned}
$$

Equational Reasoning: Example Proof

Theorem: `(append '() xs) = xs`

Proof:

```
(append '() xs)
```

Equational Reasoning: Example Proof

Theorem: `(append '() xs) = xs`

Proof:

```
(append '() xs)
```

= {substitute actual parameters in definition of `append`}

```
(if (null?  '())
  xs
  (cons (car '()) (append (cdr '()) xs)))
```

Equational Reasoning: Example Proof

Theorem: `(append '() xs) = xs`

Proof:

```
(append '() xs)
```

= {substitute actual parameters in definition of `append`}

```
(if (null?  '())
  xs
  (cons (car '()) (append (cdr '()) xs)))
```

= {`null?`-empty law}

```
(if #t
  xs
  (cons (car '()) (append (cdr '()) xs)))
```

Equational Reasoning: Example Proof

Theorem: `(append '() xs) = xs`

Proof:

```
(append '() xs)
```

= {substitute actual parameters in definition of `append`}

```
(if (null?  '())
  xs
  (cons (car '()) (append (cdr '()) xs)))
```

= {`null?`-empty law}

```
(if #t
  xs
  (cons (car '()) (append (cdr '()) xs)))
```

= {`if`-#t law}

```
xs
```

Equational Reasoning: Examples

- We can prove the following properties using equational reasoning

1. `append`-empty-left law:

```
(append '() xs) = xs
```

2. `append-cons`-left law:

```
(append (cons x xs) ys) = (cons x (append xs ys))
```

3. `length`-empty law:

```
(length '()) = 0
```

4. `length-cons` law:

```
(length (cons x xs)) = (+ 1 (length xs))
```

Structural Induction

- How can we prove

`(length (append xs ys)) = (+ (length xs) (length ys))`?

Structural Induction

- How can we prove

```
(length (append xs ys)) = (+ (length xs) (length ys))?
```

- Equational Reasoning (replacing equals for equals) never stops expanding
- We need a more powerful technique: structural induction

Base Case: Prove the theorem is true for empty list `'()`
Inductive Step: Assume the theorem is true for the list `xs`, prove it is also true for `(cons x xs)`

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Base Case: `xs = '()`

`(length (append '() ys))`

`(append '() xs) = xs` (1)

`(append (cons x xs) ys) = (cons x (append xs ys))` (2)

`(length '()) = 0` (3)

`(length (cons x xs)) = (+ 1 (length xs))` (4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Base Case: `xs = '()`

```
(length (append '() ys))
= {(1)}
(length ys)
```

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Base Case: `xs = '()`

`(length (append '() ys))`

$=$ {(1)}

`(length ys)`

$=$ {additive Identity}

`(+ 0 (length ys)`

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Base Case: `xs = '()`

```
(length (append '() ys))
= {(1)}
(length ys)
= {additive Identity}
(+ 0 (length ys)
= {(3)}
(+ (length '()) (length ys))
```

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Base Case: `xs = '()`

```
(length (append '() ys))
```

= {(1)}

```
(length ys)
```

= {additive Identity}

```
(+ 0 (length ys)
```

= {(3)}

```
(+ (length '()) (length ys))
```

= {substitute `xs`}

```
(+ (length xs) (length ys))
```

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Inductive Step: Suppose theorem holds for `xs` prove it holds for `(cons x xs)`

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Inductive Step: Suppose theorem holds for `xs` prove it holds for `(cons x xs)`

`(length (append (cons x xs) ys))`

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Inductive Step: Suppose theorem holds for `xs` prove it holds for `(cons x xs)`

```
(length (append (cons x xs) ys))
```

$= \{(2)\}$

```
(length (cons x (append xs ys)))
```

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Inductive Step: Suppose theorem holds for `xs` prove it holds for `(cons x xs)`

```
(length (append (cons x xs) ys))
```

$= \{(2)\}$

```
(length (cons x (append xs ys)))
```

$= \{(4)\}$

```
(+ 1 (length (append xs ys)))
```

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Inductive Step: Suppose theorem holds for `xs` prove it holds for `(cons x xs)`

```
(length (append (cons x xs) ys))
```

$= \{(2)\}$

```
(length (cons x (append xs ys)))
```

$= \{(4)\}$

```
(+ 1 (length (append xs ys)))
```

$=$ {by inductive hypothesis}

```
(+ 1 (length xs) (length ys))
```

`(append '() xs) = xs`	(1)	`(length '()) = 0`	(3)
`(append (cons x xs) ys) = (cons x (append xs ys))`	(2)	`(length (cons x xs)) = (+ 1 (length xs))`	(4)

Structural Induction

Theorem: `(length (append xs ys)) = (+ (length xs) (length ys))`

Inductive Step: Suppose theorem holds for `xs` prove it holds for `(cons x xs)`

```
(length (append (cons x xs) ys))
```
$= \{(2)\}$
```
(length (cons x (append xs ys)))
```
$= \{(4)\}$
```
(+ 1 (length (append xs ys)))
```
$=$ {by inductive hypothesis}
```
(+ 1 (length xs) (length ys))
```
$= \{(4)\}$
```
(+ (length (cons x xs) (length ys))
```

```
(append '() xs) = xs                    (1)
```

```
(append (cons x xs) ys) =               (2)
(cons x (append xs ys))
```

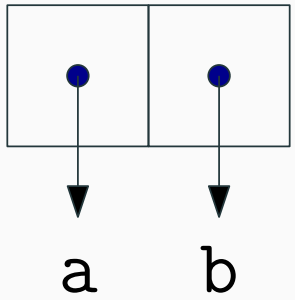
```
(length '()) = 0                        (3)
```

```
(length (cons x xs)) =                  (4)
(+ 1 (length xs))
```

Proper List

- Lists that we considered in this lecture are more precisely known as **Proper Lists**
- A proper list is either the empty list,
  or a pair whose `cdr` is a proper list
- `cons` is not for just building list: it can pair any two values
- `(cons 'a 'b)` is not a proper list
- The pair that is not a proper list is represented by dot notation:

`(a . b)`



Summary

This Lecture

- Recursive functions for processing lists
- We discuss equational reasoning and structural induction

Next Lecture

- Functions as first-class citizens