Principles of Programming Languages IV

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.cs.toronto.edu/~wael/324.html
Today

• Administrative
  – Weekly Survey
  – Assignment 2

• Roadmap
  – Continuations
  – High-order Functions
  – Examples
  – Algebraic Proofs
A higher order function used to apply another function to every element of a list:

\[(\text{map} \; \langle \text{func} \rangle \; \langle \text{arg-list} \rangle)\]

- Arguments: a function and arguments lists
- \(\langle \text{func} \rangle\) must be a function taking as many arguments as there are in \(\langle \text{arg-list} \rangle\) and returning a single value
- The dynamic order in which \(\langle \text{func} \rangle\) is applied to the elements of the \(\langle \text{arg-list} \rangle\) is unspecified.

Examples:

\[\Rightarrow\ (\text{map} \; + \; '(\ 1 \ 2 \ 3) \ ' (\ 4 \ 5 \ 6 ))\]
(5 7 9)

\[\Rightarrow\ (\text{map} \ (\lambda (x) \ (+ \ 1 \ x)) \ '(\ 1 \ 2 \ 3))\]
(2 3 4)

\[\Rightarrow\ (\text{map} \ \text{abs} \ '(-1 \ 2 \ -3 \ -4))\]
(1 2 3 4)
Another high order function *(map)*

**Syntax:**

\[(for\text{-}each \ func \ list)\]

– Guaranteed to apply function *func* to each element of a list.
– Value returned by for-each is unspecified

**Implementation:**

\[
(define \ (for\text{-}each \ proc \ list)
    (cond
        ((null? \ (cdr \ list)) \ (proc \ (car \ list)))) ; \ one-element \ list
        (else \ (proc \ (car \ list)) \ (for\text{-}each \ proc \ (cdr \ list))))
\]

**Example:**

\[
]⇒ \ (for\text{-}each \ display \ '(a \ b \ c \ (d \ e \ f )))
abc(d \ e \ f)
\]
Scheme: higher order func - reduce

• A high order function (we have already seen \textit{map})

• Syntax:

\[(reduce \text{ func initial-list})\]
– Reduce a list of values to a single value by repeatedly applying a binary function to the list of values

• Implementation:

\[
\text{(define (reduce func list id)}
\text{(if (null? list)}
\text{id}
\text{(func (car list) (reduce func (cdr list) id ) ) ) ) )}
\]

• Example:

\[
\text{=} \Rightarrow (reduce + '(1 2 3 4 5) 0) \quad ;15
\text{=} \Rightarrow (reduce * '(1 2 4 6 8 10) 1) \quad ;3840
\text{=} \Rightarrow (reduce append '((1 2 3) (4 5 6) (7 8)) ( ) ) \quad ;(1 2 3 4 5 6 7 8)
\text{=} \Rightarrow (reduce expt '(2 2 2 2) 1) \quad ;65536
\]
Another compound structure (*lists, paris*)
- Similar to lists, it can hold heterogeneous elements

What’s the problem with lists?
- Access time... Vectors are accessed in constant time.

Example:
```scheme
=> (make-vector 5 0) ; creates a vector of length 5 initialized to 0
=> #(0 0 0 0 0) ; prefix notation # to do the same thing
=> (vector-length (make-vector 150)) ; returns 150
=> (vector-ref #(a b c) 2) ; returns the third element c
=> (let ((v (vector 'a 'b 'c 'd 'e))
           (vector-set! v 2 'x)) ; #(a b x d e)
  )
=> (vector-fill! v y) ; replace each element of v with y
=> (vector->list #(a b c)) ; return list (a b c)
=> (list->vector '(a b c)) ; return vector #(a b c)
=> (let ((v (make-vector 5)))
   (for-each (lambda (i)
               (vector-set! v i (* i i))
             )
             '(0 1 2 3 4))
```

...
What is a continuation?

- The current continuation at any point in the execution of a program is an abstraction of the rest of the program.

- A continuation of the evaluation of an expression E in a surrounding context C represents the entire future of the computation, which waits for the value of E

- E.g.

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Intuitive continuation of E in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 5 (* 4 3))</td>
<td>The adding of 5 to the value of E</td>
</tr>
<tr>
<td>(cons 1 (cons 2 (cons 3 '())))</td>
<td>The consing of 3, 2 and 1 to the value of E</td>
</tr>
<tr>
<td>(define x 5)</td>
<td></td>
</tr>
<tr>
<td>(if (= 0 x)</td>
<td>The multiplication of E by x - 1 followed by a division by x</td>
</tr>
<tr>
<td>'undefined</td>
<td></td>
</tr>
<tr>
<td>(remainder (* (+ x 1) (- x 1)) x))</td>
<td></td>
</tr>
</tbody>
</table>
What is a continuation?

- A continuation of the evaluation of an expression E in a surrounding context C represents the entire future of the computation, which waits for the value of E

- A more precise notation of the continuation of E

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Continuation of E in C</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 5 (* 4 3))</td>
<td>(lambda (e) (+ 5 e))</td>
</tr>
<tr>
<td>(cons 1 (cons 2 (cons 3 '())))</td>
<td>(lambda (e) (cons 1 (cons 2 (cons 3 e))))</td>
</tr>
<tr>
<td>(define x 5)</td>
<td></td>
</tr>
<tr>
<td>(if (= 0 x)</td>
<td></td>
</tr>
<tr>
<td>'undefined</td>
<td></td>
</tr>
<tr>
<td>(remainder (* (+ x 1) (- x 1)) x))</td>
<td>(lambda (e) (remainder (* e (- x 1)) x)) c</td>
</tr>
</tbody>
</table>
**Scheme: continuations**

- Scheme provides a primitive that captures a continuation of an expression $E$ in a context $C$.

- **Syntax**

  ```scheme
call-with-current-continuation( func (param) )  call/cc( func (param) )
  ```

  - `call/cc` takes a parameter, which is a receiver function of one parameter.
  - `call/cc` captures its own continuation and passes it to the receiver, i.e., `call/cc` applies the receiver to its own continuation.

<table>
<thead>
<tr>
<th>Context $C$ and expression $E$</th>
<th>Context $C$ and the capturing</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 5 (* 4 3))</td>
<td>(+ 5 (call/cc (lambda (e) (* 4 3) )) )</td>
</tr>
<tr>
<td>(cons 1 (cons 2 (cons 3 '())))</td>
<td>(cons 1 (cons 2 (cons 3 (call/cc (lambda (e) '())) )))</td>
</tr>
<tr>
<td>(define x 5)</td>
<td>(define x 5)</td>
</tr>
<tr>
<td>(if (= 0 x)</td>
<td>(if (= 0 x)</td>
</tr>
<tr>
<td>'undefined</td>
<td>'undefined</td>
</tr>
<tr>
<td>(remainder (* (+ x 1) (- x 1)) x))</td>
<td>(remainder (* (call/cc (lambda (e) (+ x 1)) ) (- x 1)) x))</td>
</tr>
</tbody>
</table>
**Scheme: continuations**

- **Syntax**  
  \[ \text{call/cc}( \text{func} (\text{param}) ) \]
  - call/cc takes a parameter, which is a receiver function of one parameter.
  - call/cc captures its own continuation and passes it to the receiver, i.e., `call/cc' applies the receiver to its own continuation.

<table>
<thead>
<tr>
<th>Context C and expression E</th>
<th>Value of C</th>
<th>Application of call/cc</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+ 5 (call/cc (lambda (e) (set! cont-remember e) (* 4 3))))</td>
<td>17</td>
<td>(cont-remember 3)</td>
<td>8</td>
</tr>
<tr>
<td>(cons 1 (cons 2 (cons 3 (call/cc (lambda (e) (set! cont-remember e) '())))))</td>
<td>(1 2 3)</td>
<td>(cont-remember '(7 8))</td>
<td>(1 2 3 7 8)</td>
</tr>
<tr>
<td>(define x 5) (if (= 0 x) 'undefined (remainder (* (call/cc (lambda (e) (set! cont-remember e) (+ x 1)) (- x 1)) x))</td>
<td>4</td>
<td>(cont-remember 3)</td>
<td>2</td>
</tr>
</tbody>
</table>
Scheme: continuations

• **How to use call/cc to change continuation?**
  Consider (func n) in (+ (func n) 3)
  1. We make (func n) the body of a function
  2. Let us have one argument passed to that function
  3. Let us call this argument “return“
     (lambda (return) (func n))
  4. Use call/cc
     (call/cc (lambda (return) (func n)))
  The above step will bind the current continuation to return and executes (func n)

• **Primary usage:**
  – Exception handling
  – Backtracking
Scheme: continuations

• E.g.

```
]=> (define aFuture '())

]=> (display (+ 2 (call/cc (lambda (cont)
                        (set! aFuture cont) 8))))
10

]=> (display aFuture)
#[continuation 1]

]=> (display aFuture)
#[continuation 1]

]=> (display (+ 2 (call/cc (lambda (cont)
                            (set! aFuture cont) 8))))
10

]=> (display aFuture)
#[continuation 2]

]=> (define error '())

]=> (call/cc (lambda (cont)
               (set! error cont) (even? 9)))
() 

]=> (display error)
#[continuation 3]```
E.g.

```scheme
]=> (define some-flag #t)

]=> (define (my-abortable-proc escape-proc)
    (display " in my-abortable-proc ")
    (if some-flag
        (escape-proc "ABORTED")
        (display " still in my-abortable-proc ")
        " NOT ABORTED ")

]=> (my-abortable-proc display)
in my-abortable-proc ABORTED still in my-abortable-proc NOT ABORTED

]=> (define (my-resumable-proc)
    (display " in my-resumable-proc 1 ")
    (display (call-with-current-continuation my-abortable-proc))
    (display " in my-resumable-proc 2 ")

]=> (my-resumable-proc)
in my-resumable-proc 1 in my-abortable-proc ABORTED in my-resumable-proc 2

]=> (set! some-flag #f)
```

```java
try
{
    // something
}
catch(Exception exp)
{
    // handle exception
}
```
• Another control structure *(we have already seen cond and if)*

• Syntax:

  \[(\textbf{begin} \; \text{exp}_0 \; \text{exp}_1 \; \ldots \; \text{exp}_n)\]
  
  – expressions are evaluated from left to right
  
  – The value of the rightmost expression is the value of the entire expression. Other expressions are for side-effects only

• Example:

  \[\Rightarrow (\text{define} \; \text{x} \; 3)\]
  \[\Rightarrow (\text{begin}\]
  \[\hspace{1em} (\text{set!} \; \text{x} \; (+ \; \text{x} \; 1))\]
  \[\hspace{1em} (+ \; \text{x} \; \text{x}) \; \] ; returns 8
Scheme: assoc function

- **assoc** does lookup in a list

- **Eg:**

  ```scheme
  ]=> (define NAMES '(  (Smith Pat Q)
                      (Jones Chris J)
                      (Walker Kelly T)
                      (Thompson Shelly P)))

  ]=> (assoc 'Smith NAMES)
  (Smith Pat Q)

  ]=> (assoc 'Walker NAMES)
  (Walker Kelly T)

- **assoc** returns the first sublist if more than one sublist with the same key exist.
Scheme: I/O

• Input/output
  – (read ...) ; reads and returns an expression
  – (read-char ...) ; reads and returns a character
  – (peek-char ...) ; returns next available character w/o updating
  – (char-ready? ...) ; returns #t if char has been entered
  – (write-char ...) ; outputs a single character
  – (write OBJECT ...) ; outputs the object (strings are in quotes,..)
  – (display OBJECT...) ; outputs the object in a more readable form
  – (newline) ; outputs end-of-line

• Reading/writing files
  – (open-input-file)
  – (open-output-file)

• Example:
  > (display “hello world”)
  > (define r (read))
  > (display r)
• Reading an input file

(define (echo filename)
  (let ((p (open-input-file filename)))
    (let loop ((obj (read p)))
      (if (eof-object? obj) #t
       (begin
         (write obj)
         (newline)
         (loop (read p)))))
    ))
Scheme: tree example

- **Constructing a tree:**
  ```scheme
  => (define TREE
      ' (dog
          (bird (aardvark () ()) (cat () ()) )
          (possum (frog () ()) (wolf () ()) )))
  ```

- **Simple tree functions:** *(empty?, root, left-subtree, right-subtree)*
  ```scheme
  => (define empty?
      (lambda (tree)
        (null? tree)))

  => (define root
      (lambda (tree)
        (if (empty? tree)
            'Error
            (car tree)))))

  => (define left-subtree
      (lambda (tree)
        (if (empty? tree)
            'Error
            (cadr tree)))) ; (car (cdr tree))

  => (define right-subtree
      (lambda (tree)
        (if (empty? tree)
            'Error
            (caddr tree)))) ; (car (cdr (cdr tree)))
  ```
• **Tree searching:**

```
(define contains?
    (lambda (tree sym)
        (cond ((empty? tree) #f)
            ((equal? (root tree) sym) #t)
            (else (or (contains? (left-subtree tree))
                       (contains? (right-subtree tree)))))))
```

• **Alternate tree searching:**

```
(define bst-contains?
    (lambda (bstree sym)
        (cond ((empty? bstree) #f)
            ((= (root bstree) sym) #t)
            ((> (root bstree) sym) (bst-contains? (left-subtree bstree) sym))
            (else (bst-contains? (right-subtree bstree) sym)))))
```
Scheme: tree example - cont’d

- Tree traversal:

  ```scheme
  => (define pre-order
       (lambda (tree)
         (if (null? tree) '()
          (append
            (list (car tree))
            (pre-order (cadr tree))
            (pre-order (caddr tree))))))
  (dog bird aardvark cat possum frog wolf)
  ```

  ```scheme
  => (define in-order
       (lambda (tree)
         (if (null? tree) '()
          (append
            (in-order (cadr tree))
            (list (car tree))
            (in-order (caddr tree))))))
  (aardvark bird cat dog frog possum wolf)
  ```

  ```scheme
  => (define post-order
       (lambda (tree)
         (if (null? tree) '()
          (append
            (post-order (cadr tree))
            (list (car tree))
            (post-order (caddr tree)))))
  (aardvark cat bird frog wolf possum dog)
  ```
Scheme: tree example - cont’d

- **Backtracking:**
  - It is an approach to find a target path in a tree, e.g. just a path, all paths, the shortest path

- E.g. find the path that sums to 94
  - 27, 15, 4 (46 → then go 1 step up. Visit remaining children else go up)
  - 27, 15, 22 (64 → then go 1 step up. Visit remaining children else go up)
  - 27, 33, 32 (92 → then go 1 step up. Visit remaining children else go up)
  - 27, 33, 34 (94)
Scheme: quasi object oriented

- Stack implementation:

  ```scheme
  (define (stack)
      (let ((s '()))
          (lambda (op . args)
              (cond
                  (equal? op 'push!)
                    (set! s (cons args s))
                    (car s))
                (equal? op 'pop!)
                  (if (null? s) #f
                      (let ((top (car s)))
                          (set! s (cdr s))
                          top))
                (equal? op 'empty?)
                  (null? s))
          (else #f)))
  )

  (define stk (stack))
  (stk 'push! 1) ; 1 , s = (1)
  (stk 'push! 3) ; 3 , s = (3 1)
  (stk 'push! 'x) ; x , s = (x 3 1)
  (stk 'pop!) ; x , s = (3 1)
  (stk 'empty?) ; #f, s = (3 1)
  (stk 'dump) ; #f, s = (3 1)
  ```
Software Verification

- **Software crisis**
  - Quality of software?

- **Enforce roles/responsibilities**
  - Architects $\leftrightarrow$ Programmers $\leftrightarrow$ QA
  - QA test the program on many different inputs
  - However, subtle bugs may remain undiscovered, only to appear at random, inconvenient or dangerous moments.

- **Implement quality processes** (e.g. CMM, ISO...)
  - However, bugs may remain undiscovered

- **Use Algebraic proofs**
  - Especially useful for safety-critical software (e.g. Air control,...)
Software Verification: proving properties of programs

- Example:

  (define (append X Y)
    (if (null? X) Y
        (cons (car X) (append (cdr X) Y ))))

  (define (length X)
    (if (null? X) 0
        (+ 1 (length (cdr X)))))

- Prove the following:

  Theorem: (length (append X Y)) = (length X) + (length Y)
  for all lists X and Y
Software Verification: prove outline

• **Use mathematical induction on the length of X**
  - A two-part method of proving a theorem involving an integral parameter.
    • First the theorem is verified for the smallest admissible value of the integer.
    • Then it is proven that if the theorem is true for any value of the integer, it is true for the next greater value. The final proof contains the two parts.

• **First, prove that the theorem is true for lists of length 0 [Basis]**

• **Then, prove that if the theorem is true for lists of length N, then it is also true for lists of length N+1 [Inductive Step]**

• **This implies that the theorem is true for lists of any length (i.e. for any list)**
Software Verification: structural induction

• Actually, we will use a variation of induction that emphasizes the **structure** of lists, not their length.

• First, prove that the theorem is true for $X = ' ()$ i.e. when $X$ has length 0 [**Basis**]

• Then, prove that if the theorem is true for $X = L$, then it is also true for $X = (\text{cons} \ E \ L)$ [**Inductive Step**]
  – Note: if $L$ has length $N$, then $(\text{cons} \ E \ L)$ has length $N + 1$
Software Verification: preliminaries

• Before using induction (or any other technique) to prove a complex property of a program, write down the basis properties that can be trivially verified by inspecting the program code

• The inductive proof should only use these properties of the code.

• The code itself plays no other role in a proof of correctness and can be henceforth ignored.

• Note: If the basic properties are wrong, then the entire proof is wrong. So be sure to get them right!!
Software Verification: basic properties of append

• Code:
  (define (append X Y)
    (if (null? X) Y
        (cons (car X) (append (cdr X) Y))))

• Using $X = ' ()$
  (append ' () Y ) = Y  \[1\]

• Using $X = (\text{cons } E L)$
  (append (cons E L) Y ) = (cons E (append L Y))  \[2\]

• Note:
  – if $X = (\text{cons } E L)$, then (car X) = E, (cdr X) = L
Software Verification: basic properties of length

- **Code:**
  
  ```scheme
  (define (length X)
      (if (null? X) 0)
      (+ 1 (length (cdr X))))
  ```

- **Using** $X = ' ()$
  
  $$\text{(length ' () ) = 0}$$

  [3]

- **Using** $X = (\text{cons} E L)$
  
  $$\text{(length (cons E L) ) = 1+ (length L )}$$

  [4]

- **Note:**
  
  - if $X = (\text{cons} E L)$, then $(\text{cdr} X) = L$
Software Verification: summary of basic properties

• \((\text{append } ' () Y) = Y\)  \[1\]

• \((\text{append } \text{ (cons } E L) Y) = \text{ (cons } E \text{ (append } L Y))\)  \[2\]

• \((\text{length } ' () ) = 0\)  \[3\]

• \((\text{length } \text{ (cons } E L) ) = 1+ (\text{length } L )\)  \[4\]

• Using these basic properties, we shall prove (by induction) a more complex property:
  
  Theorem: \((\text{length } \text{ (append } X Y))) = (\text{length } X) + (\text{length } Y)\)
Software Verification: basis proof

- Using induction on the structure of X

- **Basis:** when \( X = ' () \)

\[
\begin{align*}
\text{(length (append X Y))} \\
= \text{(length (append '() Y))} \\
= \text{(length Y)} & \quad \text{[by 1]} \\
= 0 + \text{(length Y)} \\
= \text{(length '()) + (length Y)} & \quad \text{[by 3]} \\
= \text{(length X) + (length Y)}
\end{align*}
\]

Therefore, the theorem is true when \( X = ' () \)
Software Verification: inductive step

• **Suppose** that the theorem holds for $X = L$, i.e. suppose that 
  
  $(\text{length} \ (\text{append} \ L \ Y)) = (\text{length} \ L) + (\text{length} \ Y)$

  \{Inductive hypothesis\}

• Now, **prove** that the theorem holds for $X = (\text{cons} \ E \ L)$
Software Verification: proof of inductive step

• If $X = \text{cons } E \ L$ then

\[
\begin{align*}
\text{(length (append } X \ Y)) & \\
= & \text{(length (append (cons } E \ L) \ Y)) \\
= & \text{(length (cons } E \ (append } L \ Y))) & \text{[by 2]} \\
= & 1 + \text{(length (append } L \ Y)) & \text{[by 4]} \\
= & 1 + \text{(length } L) + \text{(length } Y) & \text{[by inductive hypothesis]} \\
= & \text{(length (cons } E \ L)) + \text{(length } Y) & \text{[by 4]} \\
= & \text{(length } X) + \text{(length } Y)
\end{align*}
\]
Software Verification: summary of proof of theorem

• For **any** List, Y,

• **Basis:** the theorem holds for $X = '()$

• **Inductive Step:** if the theorem holds for $X = L$, then it holds for $X = (\text{cons } E\ L)$

• **By the principle of structural induction, the theorem holds for **any** lists $X$ and $Y$

• **Theorem:**

  $(\text{length } (\text{append } X\ Y)) = (\text{length } X) + (\text{length } Y)$
Example 2:

```
(define (member A X)
  (cond ((null? X) #f)
        ((equal? A (car X)) #t)
        (else (member A (cdr X)))))
```

- **Prove the following:**

  **Theorem:** If (member A X) then
  
  (member A (append X Y))

  for any A, and any lists X and Y.
Software Verification: outline of proof

• As before, we will use structural induction on X

• First, prove that the theorem is true for $X='()$ [basis]

• Then, prove that if the theorem is true for $X=L$, then it is true for $X=(\text{cons } E \ L)$ [inductive step]

• However, this time proof will be more complex, because the program can terminate its recursion in two ways.
Software Verification: basic properties of member

• Code:

(define (member A X)
  (cond ((null? X) #f)
        ((equal? A (car X)) #t)
        (else (member A (cdr X)))))

• Using $X = ' ()$

  (member A ' ()) = #f \[5\]

• Using $X = (\text{cons} A L)$

  (member A (cons A L)) = #t \[6\]

• Note:
  – (car X) = A
Software Verification: basic properties of member – cont’d

• Code:

(define (member A X)
  (cond ((null? X) #f)
        ((equal? A (car X)) #t)
        (else (member A (cdr X))))

• Using $X = (\text{cons } E \ L)$ where $E \neq A$

  $(\text{member } A (\text{cons } E \ L)) = (\text{member } A \ L)$  \[7\]

• Note:
  – $(\text{car } X) = E$, $(\text{cdr } X) = L$
Software Verification: summary of basic properties

• **member Code:**
  
  – (member A ' ()) = #f \[5\]

  – (member A (cons A L)) = #t \[6\]

  – If E!=A then
    
    (member A (cons E L)) = (member A L) \[7\]

• **append Code:**
  
  – (append ' () Y ) = Y \[1\]

  – (append (cons E L) Y ) = (cons E (append L Y)) \[2\]
Software Verification: proof of theorem

- **Theorem:** If (member A X) then
  
  (member A (append X Y))

- **Proof:** by induction on X

- **Basis:** if X = ' () then

  
  (member A X)  
  = (member A ' ())  
  = #f  
  [by 5]

Therefore, the premise of the theorem is false.
Therefore, the theorem is trivially true (for X = ' ())
Software Verification: inductive step

• **Suppose** the theorem is true for X=L, i.e.
  if (member A L) then
    (member A (append L Y))
  {Inductive hypothesis}

• **Now, prove** the theorem is true for X = (cons E L)
Software Verification: proof of inductive step

- Let \( X = (\text{cons } E L) \), there are two cases

- Case 1: \( E = A \)

\[
\text{Therefore, } X = (\text{cons } A L)
\]

\[
\text{Therefore, } (\text{member } A (\text{append } X Y))
\]

\[
= (\text{member } A (\text{append } (\text{cons } A L) Y))
\]

\[
= (\text{member } A (\text{cons } A (\text{append } L Y))) \quad [\text{by 2}]
\]

\[
= \#t \quad [\text{by 6}]
\]

Therefore, the conclusion of the theorem is true

Therefore, the theorem itself is trivially true for \( X = (\text{cons } A L) \)
Software Verification: proof of inductive step – cont’d

• Let $X = (\text{cons } E \ L)$, there are two cases

• Case 2: $X = (\text{cons } E \ L)$ where $E \neq A$

if\( (\text{member } A \ X) = \#t \) then

\[
(\text{member } A \ (\text{cons } E \ L))
\]

\[
= ??
\]

\[
= (\text{member } A \ L)
\]

\[
= (\text{member } A \ (\text{append } L \ Y)) \quad \text{[by inductive hypothesis]}
\]

\[
= (\text{member } A \ (\text{cons } E \ (\text{append } L \ Y))) \quad \text{[by 7]}
\]

\[
= (\text{member } A \ (\text{append } (\text{cons } E \ L) \ Y)) \quad \text{[by 2]}
\]

\[
= (\text{member } A \ (\text{append } X \ Y))
\]

\[
Therefore, \ the \ theorem \ is \ true \ when \ X = (\text{cons } E \ L) \ and \ E \neq A
\]
Software Verification: summary of proof of theorem

- For any A, and any list Y

- Basis
  
  The theorem holds for X = ' ()

- Inductive Step:
  
  If the theorem holds for X = L, then it holds for X = (cons E L)
  
  Therefore, by the principle of structural induction, the theorem holds for any A and any lists X and Y.

- Theorem:  
  
  If (member A X) then
  
  (member A (append X Y))
Software Verification

• Interested to learn more…