CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

THE UNIVERSITY OF ARIZONA

Version: 1

February 5, 2020

This Week's Plan

THE UNIVERSITY OF ARIZONA
Computer Science

- Last week: Metatheory enables us to prove things about ALL programs in a language
- This week: Can prove algebraic laws from operational semantics
  - Higher-level of abstraction
  - Algebraic laws can help guide recursive implementations
- Context will be recursion and composition in uScheme
  - In-depth study of recursive functions
  - Two recursive data structures: the list and the S-expression
  - More powerful ways of putting functions together

Outline for today

- **Last time: Lists**
  - Are a subset of S-Expressions, what is an S-Expr that isn't a list?
  - Can be defined via a recursive equation or by inference rules
- **Algebraic Laws for writing functions**
- **The cons cost model**
- **The method of accumulating parameters**

Punchline: Algebraic Laws to Recursive Functions

- To discover recursive functions, write algebraic laws:

```
sum 0 = 0
sum n = n + sum(n-1)
```

- Which side of the equality gets smaller?

- Code:

```
(define sum (n)
    (if (= n 0) 0 (+ n (sum (- n 1)))))
```

- Another example:

```
exp x 0 = 1
exp x (n+1) = ?
```

Last time: questions about new language

- **You can ask these questions about any language**
  1. What is the abstract syntax? Syntax categories?
  2. What are the values?
  3. What environments are there? What are names mapped to?
  4. How are terms evaluated?
  5. What's in the initial basis? Primitives and otherwise, what is built in?

- **Process**
  - Understand the values: for Scheme ordinary and full s-expressions and lists
  - Do a case analysis on inputs
  - Develop, or identify, algebraic laws relevant to each case
  - Use algebraic laws to guide implementation

Forms of a List

- **A list is a sequence of elements**
  - The order of the elements in that sequence matters
  - Thus let's consider some ways of decomposing a list using the concept of sequencing (.. used to denote "followed by")
- **Decompositions of a list**
  - element .. list of values
  - list of values .. element .. list of values
  - list of values .. element
  - list of values .. list of values
- **→ For the above, what uScheme list-related functions could be used to do decomposition?**

Lists defined inductively

THE UNIVERSITY OF ARIZONA
Computer Science

$LIST(Z)$ is the **smallest** set satisfying this equation:

$$\mathrm{LIST}(\mathrm{Z}) = \{\texttt{'()}\} \cup \{\texttt{(cons}\ z\ zs\texttt{)} \mid z \in \mathrm{Z}, zs \in \mathrm{LIST}(\mathrm{Z})\}$$

Equivalently, $LIST(Z)$ is defined by these rules:

$$\frac{}{\texttt{'()} \in List(Z)}\ (\textsc{Empty})$$

$$\frac{z \in Z \qquad zs \in List(Z)}{\texttt{(cons}\ z\ zs\texttt{)} \in List(Z)}\ (\textsc{Cons})$$

Case analysis for function with two lists as input

- **Each list can be either**
  - An empty list, ‘(), e
  - Or a (cons x xs), x .. xs
- **Four cases for two input lists, when only decomposing one list at a time**

```
xs .. e
e  .. ys
(z .. zs) .. ys
xs .. (v .. vs)
```

Algebraic Law for each case

- **Each list can be either**
  - An empty list, ‘(), e
  - Or a (cons x xs), x .. xs
- **Algebraic laws relevant to append**

```
xs  ..  e             =  xs
e   ..  ys            =  ys
(z  ..  zs)  ..  ys   =  z  ..  (zs  ..ys)
xs  ..  (v  ..  vs)   =  (xs  ..  v)  ..  vs
```

- **Which one is not useful in uScheme?**
- **Which one is redundant?**

THE UNIVERSITY OF ARIZONA
Computer Science

Equations and function for append

- **Algebraic laws relevant to append**

```
e   .. ys              == ys
(z .. zs) .. ys        == z .. (zs .. ys)
```

- **Algebraic laws written using uScheme functions**

```
(append '() ys)              == ys
(append (cons z zs) ys) == (cons z (append zs ys))
```

- **Implementation in uScheme**

```
;; inputs are xs and ys
;; z = ?, zs = ?, check for empty list vs cons?
<put code here>
```

Homework 3: scheme

- Prove things about sets of values (Exercise 1)
- Using operational semantics to prove algebraic rules
  - Problem A in homework.
  - Use approach in HW2.
- Prove algebraic laws with other algebraic laws
  - 3 (or Exercise 31 on page 203)
- Use and develop algebraic laws to guide implementation
  - 2 (or Exercise 10 on page 195), B, C, D

Let's prove something about a set of values

- **Cost model for the append function**
  - Main cost is cons because it corresponds to allocation
  - How many cons cells are allocated?
- **Induction Principle for List(Z)**
  - If we can prove
    - The induction hypothesis for the empty set, IH('())
    - Whenever z in Z and also IH(zs), then IH((cons z zs))
  - Then forall zs in List(Z), IH(zs)
- **The cost of append**
  - Claim: IH(xs,ys) defined as (cost of append xs ys) = (length xs)
  - Proof: by induction on the structure of xs

Structural Induction on List values, List(Z)

The University of Arizona
Computer Science

- **The cost of append**
  - Claim: IH(xs,ys) defined as (cost of append xs ys) = (length xs)
  - Proof: by induction on the structure of xs
- **Base Case: xs = ‘()**
  - How can we show that (cost of append ‘() ys) = (length ‘())?
  - → Step through on whiteboard
- **Inductive case: xs = (cons z zs)**
  - What is the inductive hypothesis for what we are trying to prove?
- **Cost of (append xs ys) is O(n), where n=(length xs)**

List Reversal

- **Algebraic Laws for list reversal**

```
(reverse '())              == '()
(reverse (cons x xs))      == (append ? ?)
```

- **Straight-forward list reversal**

```
(define reverse (xs)
    (if (null? xs)
        '()
        (append ? ?)))
```

- **How many cons cells allocated? Let n=|xs|**
  - Calls to append? Need to know number of calls to reverse?
  - Length of lists passed to append?
  - List1 call cost?
  - Study question: how would we prove O(N^2) cost using induction?

Method of accumulating parameters

THE UNIVERSITY OF ARIZONA
Computer Science

- O(N^2) is usually too expensive
- Instead write a helper function revapp that takes two arguments that satisfies these algebraic rules

```
(revapp xs ys)            == (append (reverse xs) ys)
(reverse xs)              == (revapp xs '())
```

- ➔ Write the
  - (1) two cases and
  - (2) use algebraic rules that do NOT involve append to
  - (3) implement code for revapp

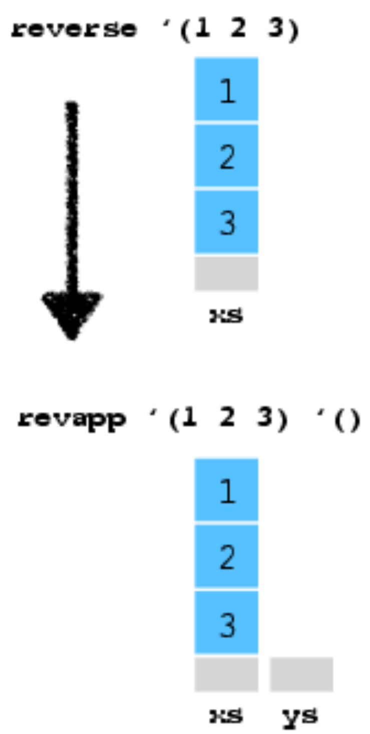
```
revapp xs     ys           == (rev xs) .. ys
revapp e      ys           == ?
revapp(z .. zs)    ys      == ?
```

Linear reverse, graphically

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

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

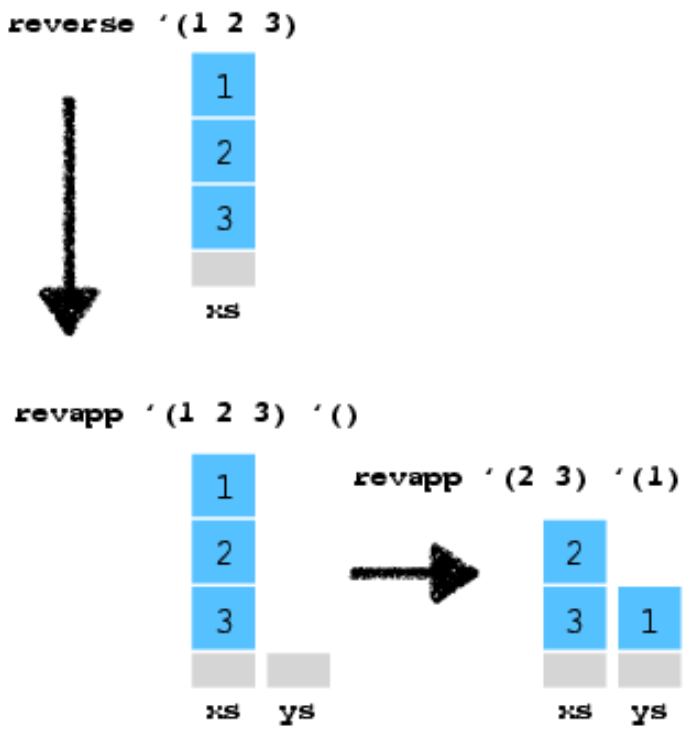
Reverse calls revapp with ys=‘()

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

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



Take 1 off front of xs and put on front of ys

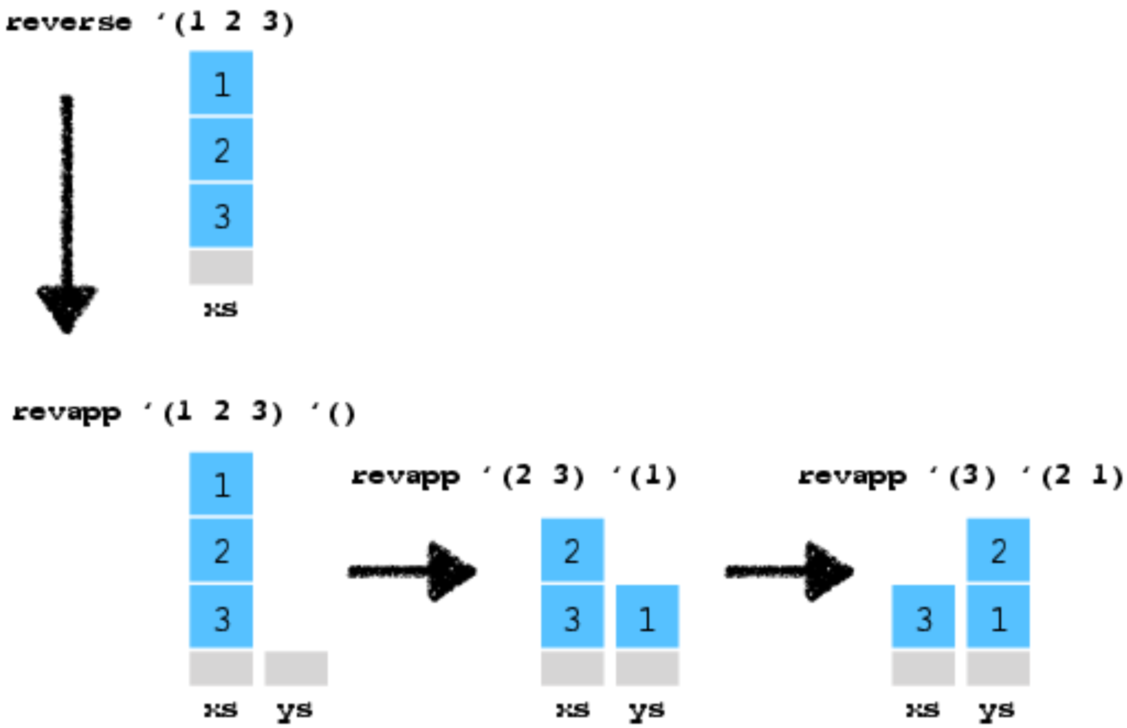

```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

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



Take 2 off front of xs and put on front of ys

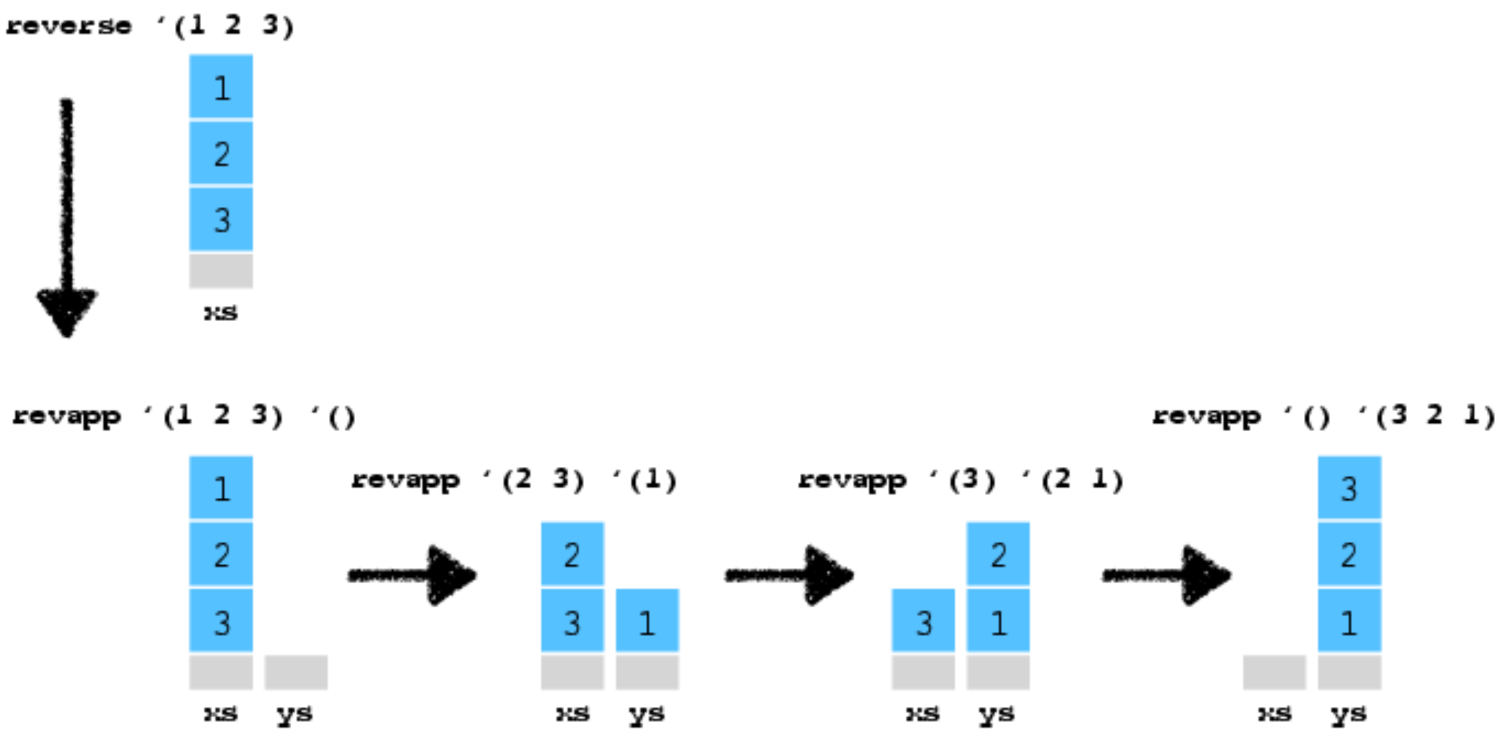
THE UNIVERSITY OF ARIZONA
Computer Science

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

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



Take 3 off front of xs and put on front of ys

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

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



Finish helper recursion, which returns ys

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

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