Complexity and Accumulators

Corky Cartwright
Department of Computer Science
Rice University

Today’s goals

- Overview of accounting for cost of computation (complexity)
- Intuitively, accumulators can capture “history”

Accumulators can be used to

- Improve performance
- Avoid non-termination (uncommon)
- Improve expressivity (simplify code)
- How do we recognize when they are needed?

Cost accounting

- Measure computation cost in reduction steps using our reduction semantics. Models actual cost reasonably well.
- Consider three algorithms
  - $\text{Cost-A}(n) = 2*n^3 + n^2 + 50$
  - $\text{Cost-B}(n) = 3*n^2 + 100$
  - $\text{Cost-C}(n) = 2^n$
- Which algorithm is best?
- Which algorithm works best for large n?
- Can we formalize this notion?

Order of Complexity

- We'll say that Cost-X is "order f(n))", or simply "$O(f$(n))" (read "Big-O of f(n))") if
  - Cost-X(n) < factor * f(n) for sufficiently large n
- Examples:
  - Cost-A(n) = $2*n^3 + n^2 + 1$ Cost-A is $O(n^3)$
  - Cost-B(n) = $3*n^2 + 10$ Cost-B is $O(n^2)$
  - Cost-C(n) = 2^n Cost-C is $O(2^n)$

Famous "Complexity Classes"

$O(1)$	constant-time	(head, tail)
$O(\log n)$	logarithmic	(binary search)
$O(n)$	linear	(vector multiplication)
$O(n * \log n)$	"n log n"	(sorting)
$O(n^2)$	quadratic	(matrix addition)
$O(n^3)$	cubic	(matrix multiplication)
$n^{O(1)}$	polynomial	(…many! …)
$2^{O(n)}$	exponential	(guess password)

Improving Performance

- Consider the sequence accumulation function
  - Takes '(1 1 2 3 -1) and produces '(1 2 4 7 6)
- How do we write this function using the list template?
- We can do much better!
- What information do we need to do better?
  - This is basically the “lost history” in the recursive call

Partial Sums Program

```
;; sums: (listOf number) -> (listOf number)
;; (sums alon) replaces each number n in alon by the sum
;; of the numbers preceding (and including) n.
;; (sums '(1 2 3)) = '(1 3 6)
(define (sums alon)
  (cond [(empty? alon) empty]
          [else
            (cons (first alon)
              (map (lambda (x) (+ x (first alon)))
                   (sums (rest alon))))]))
```

Accumulator version of same program

- Idea: as the list is successively decomposed into first and rest, the sums function can accumulate the sum of the numbers to the left of rest.
- Template Instantiation:

```
(define (sums-help lon sum)
  (cond [(empty? lon) … ]
        [else … (first lon) … sum …
              (sums-help (rest lon) ..) ]))
```

Accumulator version of same program

```
;; sums-help: (listOf number) number -> (listOf number)
(define (sums-help alon sum)
   (cond
     [(empty? alon) empty]
     [else
        (local [(define new-sum (+ sum (first l)))]
            (cons new-sum (sums-help (rest l) new-sum)))]))
;; sums: (listOf number) -> (listOf number)
(define (sums alon) (sums-help alon 0))
```

Formulating an Accumulator

- If we decide to use an accumulator, we need to answer three questions:
  - How will we use the accumulator to produce the final result?
  - How will we modify the accumulator in each recursive call? (What will we "accumulate"?)
  - What should the initial value for the accumulator be?

Another Example

- ;; (flatten: (genListOf symbol) -> (listOf symbol)
  ;; (flatten agl) returns a list of the symbols in order of appearance
  ;; (flatten '((a b) c ((d))) = '(a b c d)
- ```
  (define (flatten agl)
    (cond [(empty? agl) empty]
              [else (local [(define head (first agl))
                                    (define tail (flatten (rest agl)))]
                                (cond [(empty? head)  tail]
                                          [(cons? head)  (append (flatten head) tail)]
                                          [else  (cons head tail)]))]))
  ```
- Note: we wrote this function so that the symbol type can be replaced by any non-list type.
  ```

Accumulator version

```
;; flatten-help: (genListOf symbol) (listOf symbol) -> (listOf symbol)
;; (flatten agl los) returns a list of the symbols in agl appended to los
;; (flatten '((a b) c ((d)) '(e)) = '(a b c d e)
;;
;; Template Instantiation:
  (define (flatten-help agl los)
    (cond [(empty? agl) … ]
          [else …. (first agl) … los … (flatten-help agl ..) …]))
```

```
(define (flatten-help agl los)
  (cond [(empty? agl) los]
        [else (local [(define head (first agl))
                      (define tail (flatten-help (rest agl) los))]
                (cond [(empty? head)  tail]
                      [(cons? head)  (flatten-help head tail)]
                      [else  (cons head tail)]))]))
```

Other Examples

- Graph searching: avoid repetition/cycles by accumulating set of nodes already seen and testing membership in this set. In most cases, mutation (marking) is better in practice.

Added Expressivity

- Code simplication using accumulators
- Consider the list reverse function
  - Takes '(1 2 3 4 5) and produces '(5 4 3 2 1)
- How do we write this function using the list template? Use append. Ugh.
- What information do we need to do better?
  - This is basically the “lost history” of the recursive call
- Is this list reversal example really different from the list accumlation example?

Naive reverse

```
(define (rev l)
  (cond [(empty? l) empty]
        [else (append (rev (rest l))
                      (list (first l)))]))
```

Reverse using an accumulator

```
(define (rev-help l ans)
  (cond [(empty? l) ans]
        [else (rev-help (rest l) (cons (first l) ans))]))

(define (fast-rev l)  (rev-help l empty))
```

For Next Class

- Bonus lecture this afternoon at 2 in DH 1042
- Homework due Monday
- Midterm:
  - Take home exam distributed Friday February 10; due Friday, February 17.
  - Covers Scheme Material (Chs. 1- 32 of HTDP except 28, 29.3)
- Reading:
  - Chs 29 .1, 29.2, 30-32