Accumulators and Tail Calls

Corky Cartwright
Vivek Sarkar
Department of Computer Science
Rice University

Plan for today

Provide a more precise technical motivation for accumulator concepts

- Recognizing the need for accumulators
  - Avoiding non-tail recursive calls
  - Avoiding unnecessary traversals of a list (which often correlates with non-tail recursive calls)
- Note: tail calls and tail recursion are not discussed in the book. Tail-recursion is *extremely* important because it corresponds to iteration (loops).

Last class: Accumulators

Looked at three examples

- Computing the partial sums for a sequence
- Reversing a list
- Flattening a general list

about the Complexity of Recursive Functions (Naïve version of Partial Sums)

```
(define (sums alon)
  (cond [(empty? alon) empty]
          [else
            (cons (first alon)
               (map (lambda (x) (+ x (first alon)))
                        (sums (rest alon))))]))
```

- Let T(n) = number of additions performed when function sums is invoked on a list, alon, of size n
- $T(0) = 0$
- $T(n) = T(n-1) + (n-1)$

➔ $T(n) = n*(n-1)/2$

- Recurrences may be hard to solve sometimes, but are usually easy to verify

Recursive Functions (Accumulator version of Partial Sums)

```
(define (sums-help alon sum)
  (cond
    [(empty? alon) empty]
    [else
      (local [(define new-sum (+ sum (first alon)))]
         (cons new-sum (sums-help (rest alon) new-sum)))]))
```

- Let $T(n)$ = number of additions performed when function sums-help is invoked on a list, alon, of size n
- $T(0) = 0$
- $T(n) = T(n-1) + 1$

➔ $T(n) = n$

Accumulators

Three key things to remember:

- What should the initial value for the accumulator be?
- How will we modify the accumulator in each recursive call? (What will we "accumulate"?)
- How will we use the accumulator to produce the final result?

Two versions of Factorial: Which is faster?

-
  ```
  (define (fact n)
     (cond [(= n 0) 1]
               [else (* n (fact (sub1 n)))]))
  ```

-
  ```
  (define (fact-help n ans)
     (cond [(= n 0) ans]
               [else (fact-help (sub 1) (* n ans))]))
  ```

-
  ```
  (define (fast-fact n) (fact-help n 1))
  ```

- The recursive call on fact-help is in *tail position* in the function body, meaning that no subsequent processing is performed on the result of the call in evaluating the body.

Factorial

- Even factorial could benefit from an accumulator!
- Why? The recursive call on fact is embedded inside another call (on *) which means the computer must maintain a calling stack to manage the recursion! Call-stack maintenance is NOT free.

Tail Position

- Consider our law for evaluating Scheme expressions. A call to function *g* embedded in a definition for function *f* is said to be in tail position if, whenever the call to *g* is invoked, it is the last form to be reduced in evaluating the call to *f*
  - If *g* = *f*, the call to *g* is referred to as a "tail-recursive call"

- Note that a function definition may have several different embedded calls that are in tail position. How?

Examples

- Which function calls are in tail position?

```
(define (fact n)
  (cond [(= n 0) 1]  ;; no function call in result
        [else (* n (fact (sub1 n)))]))
```

- (define (fact-help n ans)

```
  (cond [(= n 0) ans]  ;; no function call in result
[else (fact-help (sub 1) (* n ans))]))
```

- (define (fast-fact n) (fact-help n 1))

Reduction pattern (factorial)

- (fact 3)
- = (* 3 (fact 2))
- = (* 3 (* 2 (fact 1)))
- = (* 3 (* 2 (* 1 (fact 0))))
- = (* 3 (* 2 (* 1 1)))
- = (* 3 (* 2 1))
- = (* 3 2)
- = 6

Reduction pattern (fast-fact)

- (fast-fact 3)
- = (fact-help 3 1)
- = (fact-help 2 (* 3 1))
- = (fact-help 2 3)
- = (fact-help 1 (* 2 3))
- = (fact-help 1 6)
- = (fact-help 0 (* 1 6))
- = (fact-acc 0 6)
  = 6

Why are tail calls more expensive than non-tail calls?

- Non-tail calls compile (translate) to expensive "jump to subroutine" instructions (which are covered in both Elec 220 and Comp 221).
- A "jump to subroutine" instruction allocates space in the control stack (a chunk of memory reserved by the OS for each computation) to save information about the machine state at the point of the call, saves critical information and then jumps to the subroutine. The saved information enables a subsequent "subroutine return" instruction to restore the machine state at the the point of call (with the returned value in a designated register).
- In contrast, a tail call *never needs to return* to the point of call. Tail-calls compile to jumps (branches). Tail recursive calls compile to loop (backwards) branches. Branches are only slightly more expensive than ordinary instructions. In fact they can actually be cheaper with the right hardware support.

Conversion to tail form

- Always possible? Always profitable?
  - Always possible, but requires passing closures (functions) as arguments, which requires heap allocation. Called conversion to continuation-passing style (CPS). May or may not be profitable.
  - What about our examples? They did not use closures. Tail calls are very profitable when they substitute a cheap operation like cons for an expensive one like append. They can potentially produce a huge space improvement if no closure allocation is required and significant time improvement if the cost of corresponding operations is unchanged.

Naive Reverse

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

The subcomputation in red must be performed after the call on rev completes

Tail-recursive Reverse

```
(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))
```

- The recursive call on rev-help can be implemented by branching back to the code for the enclosing cond operation reusing the stack frame for the calling invocation of rev-help . The initial call on rev-help (in reverse) can be "inlined". So the optimized machine translation of this call looks just like it does for corresponding loop code.

Design Recipe for Accumulators Revisited

When do we need an accumulator?

We pointed out some tell-tail(!) signs:

- When we need some context (or history) information
- When the result of the recursive call is processed by another function i.e., when the recursive call is not a non-tail call
- . . .

For Next Week

- Midterm to be distributed next Friday (Feb 19th)