(Tail) Recursion and Efficiency

Recursion + Immutability

- Elegant and natural match for many definitions, data structures.
- Dependences between recursive levels are explicit.
  - Loop iteration dependences are not!
- Now:
  - How to reason about efficiency of recursion
  - The importance of tail recursion
  - Using an accumulator to achieve tail recursion
  - [No new language features here]

Call stacks / Environments
[naïve implementation]

While a program runs, a call stack holds a frame for each function call currently in progress.
- Call f: push frame for f onto stack.
- Return from f: pop its from stack.

Stack frames store:
- Bindings of parameters to arguments
- "What is left to do" in the function call

With recursion, stack may hold multiple frames for same function.

(not the full story, more to come today and after)

Example

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1))))
)
```

Space: O(  )
Time: O(  )

Do all work after recursive call.
(compute using result of recursive call)
Revised Example: **Tail Recursive**

```
(define (fact n)
  (define (fact-tail acc n)
    (if (= n 0)
      acc
      (fact-tail (* n acc) (- n 1))))
  (fact-tail 1 n))
```

**Accumulator parameter carries answer so far.**

Complete answer ready at base case.

Key: result of caller = result of callee.

---

Common patterns of work

**Normal recursion:**
- Break down input
- Build result
- Break down input
- Build result

**Tail recursion:**
- Break down input
- Build result
- Break down input
- Build result

---

**The call stacks**

```
(fact 3)  (fact 3):_  (fact 3):_  (fact 3):_  (fact 3):_
(fact-tail acc n)
(fact-tail (* n acc) (- n 1)))
(fact-tail 1 n)
```

```
(fact 3):_  (fact 3):_  (fact 3):_  (fact 3):_  (fact 3):_
(fact-tail acc n)
(fact-tail (* n acc) (- n 1)))
(fact-tail 1 n)
```

**Optimization under the hood**

```
(define (fact n)
  (define (fact-tail acc n)
    (if (= n 0)
      acc
      (fact-tail (* n acc) (- n 1))))
  (fact-tail 1 n))
```

**Space: \( O( ) \)**

**Time: \( O( ) \)**

Language implementation recognizes tail calls.
- Caller frame never needed again.
- Reuse same space for every recursive tail call.

*Racket, ML, most “functional” languages, but not Java, C, etc.*
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))

Does work after recursive call. (compute using result of recursive call)

(define (fact n)
  (define (fact-tail acc n)
    (if (= n 0)
      acc
      (fact-tail (* n acc) (- n 1))))
  (fact-tail 1 n))

Accumulator parameter carries answer so far.

Old base case is initial accumulator.

Complete answer ready at base case.

Do all work before recursive call.
  • pass answer so far
  • return recursive call result

Example:

(define (sum xs)
  (if (null? xs)
    0
    (+ (car xs) (sum (cdr xs)))))

(define (sum xs)
  (define (sum-tail acc xs)
    (if (null? xs)
      acc
      (sum-tail (+ (car xs) acc) (cdr xs)))))

And another

(define (rev xs)
  ...)

What about map, filter?

• Non-tail-recursive rev is O(n^2): each recursive call must traverse to end of list and build a fully new list.
  • 1+2+...+(n-1) is almost n^2/2
  • Morale: beware append, especially within outer recursion
• Tail-recursive rev is O(n).
  • Cons is O(1)(and fact), done n times.

Tail call intuition:
"nothing left for caller to do", "callee result is immediate caller result"

Precise recursive definition of tail position:
• If an expression is not in tail position, then no subexpressions are.
  • In (lambda (x1 ... xn) e), the body e is in tail position.
  • If (if e1 e2 e3) is in tail position, then e2 and e3 are in tail position (but e1 is not).
  • If (let ([x1 e1] ... [xn en]) e) is in tail position, then e is in tail position (but no binding expressions are).
  • In a function call expression (e1 e2), subexpressions e1 and e2 are not in tail position.
  • ...

A tail call is a function call in tail position.
Fold: iterator over recursive structures

(a.k.a. reduce, inject, etc.)

Accumulates answer, repeatedly applying \( f \) to answer so far:

- \((\text{foldr } f \ \text{init} \ (\text{list } 1 \ 2 \ 3))\) computes \((f \ 1 \ (f \ 2 \ (f \ 3 \ \text{init})))\)

- \((\text{foldl } f \ \text{init} \ (\text{list } 1 \ 2 \ 3))\) computes \((f \ 3 \ (f \ 2 \ (f \ 1 \ \text{init})))\)

Racket's built-in \text{foldr} and \text{foldl} also work on one or more lists (\( f \) takes 2 or more args)

See tail.rkt

Super-iterators!

- Not built into the language
  - Just a programming pattern
  - Many languages have built-in support, often allow stopping early without resorting to exceptions

- Pattern separates recursive traversal from data processing
  - Reuse same traversal, different folding functions
  - Reuse same folding functions, different data structures
  - Common vocabulary concisely communicates intent

- \text{map}, \text{filter}, \text{fold} + closures/lexical scope = superpower
  - Argument function can use any "private" data in its environment.
  - Iterator does not have to know or help.