Tail Recursion

Topics

Recursion is an elegant and natural match for many computations and data structures.

- Natural recursion with immutable data can be space-inefficient compared to loop iteration with mutable data.
- Tail recursion eliminates the space inefficiency with a simple, general pattern.
- Recursion over immutable data expresses iteration more clearly than loop iteration with mutable state.
- More higher-order patterns: fold

Naturally recursive factorial

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

How efficient is this implementation?

Space: $O(\quad)$

Time: $O(\quad)$

CS 240-style machine model
Example

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

Tail recursive factorial

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

Common patterns of work
Natural recursion

Recursive case:
Compute result in terms of argument and accumulated recursive result.

\[
\begin{align*}
\text{(define (fact n)} & \text{)} \\
& \text{ (if } (= n 0) \\ & \quad \text{1)} \\ & \quad \text{ (* n (fact (- n 1)))})
\end{align*}
\]

Tail recursion

Recursive case:
Compute recursive argument in terms of argument and accumulator.

\[
\begin{align*}
\text{(define (fact-tail n acc)} & \text{)} \\
& \text{ (if } (= n 0) \\ & \quad \text{ acc)} \\ & \quad \text{ (* n acc))} \\ & \text{ (fact-tail (- n 1) (* n acc))})
\end{align*}
\]
Tail recursion transformation

\[
\text{(define (fact n)} \quad \text{natural recursion}
\]
\[
\begin{align*}
\text{(if} & \quad = n 0 \\
n & \quad (1) \\
& \quad (* n (fact (- n 1))) ()))
\end{align*}
\]

\[
\text{(define (fact n)} \quad \text{tail recursion}
\]
\[
\begin{align*}
\text{(define (fact-tail n acc)} & \quad \text{Accumulator becomes base result.} \\
\text{(if} & \quad = n 0 \\
\text{(fact-tail} & \quad (- n 1) (* n acc) ))) & \quad \text{Recursive step applied to accumulator instead of recursive result.} \\
\text{(fact-tail n 1)} & \quad \text{Base result becomes initial accumulator.}
\end{align*}
\]

Example

\[
\text{(define (sum xs)} \quad \text{(define (sum-tail xs acc)} \\
\text{(if} & \quad \text{(null? xs)} \\
\text{0} & \quad \text{acc} \\
& \quad \text{(sum-tail (cdr xs) (+ (car xs) acc)))} \\
\text{(sum-tail xs 0))}
\]

Practice

\[
\text{(define (rev xs)} \quad \text{(define (rev xs)} \\
\text{(if} & \quad = n 0 \\
\text{(if} & \quad (null? xs)} \\
\text{0} & \quad \text{(sum-tail (cdr xs) (+ (car xs) acc)))} \\
\text{(sum-tail xs 0))}
\]

Tail position

Recursive definition of \textit{tail position}:
- In \textit{(lambda \(x1 \ldots xn\) \(e\)}, the body \(e\) is in tail position.
- If \textit{(if \(e1 e2 e3\) is in tail position}, then \(e2\) and \(e3\) are in tail position (but \(e1\) is not).
- If \textit{(let \([- [x1 e1] \ldots [xn en]]\) \(e\) is in tail position, then \(e\) is in tail position (but the binding expressions are not).}

Note:
- If a non-lambda expression is not in tail position, then no subexpressions are.
- Critically, in a function call expression \((e1 e2)\), subexpressions \(e1\) and \(e2\) are not in tail position.

\textit{A tail call} is a function call in \textit{tail position}. 

• Naturally recursive \textit{rev} is \(O(n^2)\): each recursive call must traverse to end of list and build a fully new list.
  - \(1+2+\ldots+(n-1)\) is almost \(n^2/2\)
  - Moral: beware append, especially within outer recursion
• Tail-recursive \textit{rev} is \(O(n)\).
  - Cons is \(O(1)\), done \(n\) times.
Why tail recursion instead of loops with mutation?

1. Simpler language, but just as efficient.
2. Explicit dependences for easier reasoning.
   - Especially with HOFs like fold!

What must we inspect to identify dependences between ________.

HOF HOF

Fold: iterator over recursive structures
(a.k.a. reduce, inject, …)

What must we inspect to identify dependences between ________.

(recursive calls)

def fib(n):
    fib_i = 0
    fib_i_plus_1 = 1
    for i in range(n):
        fib_i_prev = fib_i
        fib_i = fib_i_plus_1
        fib_i_plus_1 = fib_i_prev + fib_i_plus_1
    return fib_i

Python: loop iteration with mutation

Racket: immutable natural recursion

(define (fib n) Racket: immutable natural recursion
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))

Racket: immutable tail recursion

(define (fib n) Racket: immutable tail recursion
  (define (fib-tail n fibi fibi+1)
    (if (= 0 n)
      fibi
      (fib-tail (- n 1) fibi+1 (+ fibi fibi+1))))
  (fib n 0 1))

Python: loop iteration with mutation

def fib(n):
    fib_i = 0
    fib_i_plus_1 = 1
    for i in range(n):
        fib_i_prev = fib_i
        fib_i = fib_i_plus_1
        fib_i_plus_1 = fib_i_prev + fib_i_plus_1
    return fib_i

Racket: immutable tail recursion

(define (fib n)
  (define (fib-tail n fibi fibi+1)
    (if (= 0 n)
      fibi
      (fib-tail (- n 1) fibi+1 (+ fibi fibi+1))))
  (fib n 0 1))

Racket: immutable natural recursion

-foldr f init (list 1 2 3)
computes (f 1 (f 2 (f 3 init)))

-foldl f init (list 1 2 3)
computes (f 3 (f 2 (f 1 init)))

Fold: iterator over recursive structures
accumulates result by iteratively applying
(combine element accumulator)
to each element of the list and accumulator so far
(starting from init) to produce the next accumulator.

-foldr f init (list 1 2 3)
computes (f 1 (f 2 (f 3 init)))

-foldl f init (list 1 2 3)
computes (f 3 (f 2 (f 1 init)))
**Folding geometry**

(Foldr combine init L)

(L \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \rightarrow result)

(Foldl combine init L)

**Tail recursion**

**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

- map, filter, fold + closures/lexical scope = superpower
  - Next: argument function can use any “private” data in its environment.
  - Iterator does not have to know or help.