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: $O(\_\_)$

Time: $O(\_\_)$

CS 240-style machine model

Registers

Stack

Code

Heap

- Call frame
- Call frame
- Call frame
- Arguments, variables, return address per function call
- Cons cells, data structures, ...
Example

\[
\text{(define (fact } n) \\
  \text{  (if } (= n 0) \\
  1 \\
  (\ast n (\text{fact } (- n 1)))))))
\]

Space: \(O( )\)
Time: \(O( )\)

Tail recursive factorial

\[
\text{(define (fact } n) \\
  \text{  (define (fact-tail } n \ acc) \\
  \text{    (if } (= n 0) \\
  \text{      \ acc \\
  \text{        (fact-tail } (- n 1) (* n \ acc))))})
\]

Common patterns of work

Natural recursion:
- Argument
- Full result
- Reduce argument
- Accumulate result so far
- Deeper recursive calls
- Base case
- Base result

Tail recursion:
- Argument
- Base result
- Reduce argument
- Accumulate result so far
- Deeper recursive calls
- Base case
- Full result
Natural recursion

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

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

Tail recursion

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

\[
\text{(define (fact n)} \\
\quad \text{(define (fact-tail n acc)} \\
\quad \quad \text{(if (= n 0)}} \\
\quad \quad \quad \text{acc)} \\
\quad \quad \quad \text{(fact-tail (- n 1) (* n acc))}) \\
\quad \text{(fact-tail n 1))}
\]

The call stacks

Nothing useful remembered here.

\[
\begin{align*}
\text{(fact 3)} & \quad \text{(fact 3): _} & \quad \text{(fact 3): _} & \quad \text{(fact 3): _} & \quad \text{(fact 3): _} \\
\quad \text{(ft 3 1)} & \quad \text{(ft 3 1): _} & \quad \text{(ft 3 1): _} & \quad \text{(ft 3 1): _} & \quad \text{(ft 3 1): _} \\
\quad & \quad \quad \text{(ft 2 3): _} & \quad \text{(ft 2 3): _} & \quad \text{(ft 2 3): _} & \quad \text{(ft 2 3): _} \\
\quad & \quad \quad \quad \text{(ft 1 6): _} & \quad \text{(ft 1 6): _} & \quad \text{(ft 1 6): _} & \quad \text{(ft 1 6): _} \\
\quad & \quad \quad \quad \quad \text{(ft 0 6): _} & \quad \text{(ft 0 6): _} & \quad \text{(ft 0 6): _} & \quad \text{(ft 0 6): _}
\end{align*}
\]

Optimization under the hood

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

\[\text{Space: } O(\_ ) \quad \text{Time: } O(\_ )\]

Language implementation recognizes tail calls.

- Caller frame never needed again.
- Reuse same space for every recursive tail call.
- Low-level: acts just like a loop.

\textit{Racket, ML, most “functional” languages, but not Java, C, etc.}
Tail recursion transformation

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

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

(fact-tail n 1)
```

---

Example

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

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

(sum-tail xs 0)
```

---

Practice

```
(define (rev xs)

  (if (null? xs)
      ()
      (cons (car xs) (rev (cdr xs)))
  )
)
```

```
(define (rev xs)

  (define (rev-tail xs acc)
    (if (null? xs)
        acc
        (rev-tail (cdr xs) (cons (car xs) acc))
    ))

  (rev-tail xs 0)
)
```

- Naturally recursive `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 `rev` is $O(n)$.
  - Cons is $O(1)$, done n times.

---

Tail position

Recursive definition of tail position:
- 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 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.

A tail call is a function call in tail position.
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!

Identify dependences between _______.

Racket: immutable natural recursion

```scheme
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Racket: immutable tail recursion

```scheme
(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))
```

Python: loop iteration with mutation

```python
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
```

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

```scheme
(fold_ combine init list)
```

accumulates result by iteratively applying

```scheme
(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 v_2 \rightarrow \ldots \rightarrow v_{n-1} \rightarrow v_n)

\[
(foldl \ combine \ init \ L)
\]

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