Tail Recursion

+tail.rkt
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
- fixed size, general purpose

Code

Stack
- Call frame
- Call frame
- Call frame
- arguments, variables, return address per function call

Heap
- cons cells, data structures, ...

Program Counter

Stack Pointer
Evaluation example

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

Call stacks at each step

- \text{fact 3): 3*__}
- \text{fact 2): 2*__}
- \text{fact 1): 1*__}
- \text{fact 0): 1}

Remember: \( n \rightarrow 2 \); and “rest of function” for this call.

Space: \( O(\quad) \)

Time: \( O(\quad) \)
Naturally recursive factorial

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

Base case returns base result.

Recursive case returns result so far.

Compute result so far after/from recursive call.

Compute remaining argument before/for recursive call.
Tail recursive factorial

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

(fact-tail n 1))
```

- **Base case returns full result.**
- **Recursive case returns full result.**
- **Initial accumulator provides base result.**
- **Compute result so far before/for recursive call.**
- **Accumulator parameter provides result so far.**
- **Compute remaining argument before/for recursive call.**
Common patterns of work

Natural recursion:
- Argument
- Full result

Tail recursion:
- Argument
- Base result

Reduce argument
Accumulate result so far

Base case
Base result

Deep recursive calls

Reduce argument
Accumulate result so far

Base case
Full result

Tail Recursion
Natural recursion

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

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

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

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

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

Tail Recursion
Evaluation example

Call stacks at each step

\[ (\text{fact } 3) \]
\[ (\text{ft } 3 1) \]
\[ (\text{ft } 2 3) \]
\[ (\text{ft } 1 6) \]

\[ (\text{ft } 0 6) \]

\[ (\text{ft } 3 1) : _ \]
\[ (\text{ft } 2 3) : _ \]
\[ (\text{ft } 1 6) : _ \]
\[ (\text{ft } 0 6) : 6 \]

\[ (\text{ft } 3 1) : _ \]
\[ (\text{ft } 2 3) : _ \]
\[ (\text{ft } 1 6) : _ \]
\[ (\text{ft } 0 6) : 6 \]

\[ (\text{ft } 2 3) : _ \]
\[ (\text{ft } 1 6) : _ \]
\[ (\text{ft } 0 6) : 6 \]

\[ (\text{ft } 1 6) : _ \]
\[ (\text{ft } 0 6) : 6 \]

\[ (\text{ft } 0 6) : 6 \]

\[ \text{ft} = \text{fact-tail} \]

Nothing useful remembered here.

etc.
Tail-call optimization

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

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.

*Racket, ML, most “functional” languages, but not Java, C, etc.*
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**.

A function is **tail-recursive** if it uses a recursive tail call.
Tail recursion transformation

Common pattern for transforming naturally recursive functions to tail-recursive form. Works for functions that do commutative operations (order of steps doesn't matter).

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

- **Base result becomes initial accumulator.**
- **Accumulator becomes base result.**
- **Recursive step applied to accumulator instead of recursive result.**
Practice: use the transformation

;;; Naturally recursive sum
(define (sum-natural xs)
  (if (null? xs)
      0
      (+ (car xs) (sum-natural (cdr xs)))))

;;; Tail-recursive sum
(define (sum-tail xs)
Transforming non-commutative steps

(order matters)

```
(define (reverse-natural-slow xs)
  (if (null? xs)
      null
      (append (reverse-natural-slow (cdr xs))
              (list (car xs))))
)
```

```
(define (reverse-tail-just-kidding xs)
  (define (rev xs acc)
    (if (null? xs)
        acc
        (rev (cdr xs) (append acc (list (car xs))))
      )
  )
  (rev xs null)
)
```

```
(define (reverse-tail-slow xs)
  (define (rev xs acc)
    (if (null? xs)
        acc
        (rev (cdr xs) (append (list (car xs)) acc))
      )
  )
  (rev xs null)
)
```
The transformation is not always ideal.

<table>
<thead>
<tr>
<th>(define (reverse-tail-slow xs))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (rev xs acc))</td>
</tr>
<tr>
<td>(if (null? xs)</td>
</tr>
<tr>
<td>acc</td>
</tr>
<tr>
<td>(rev (cdr xs) (append (list (car xs)) acc))))</td>
</tr>
<tr>
<td>(rev xs null))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(define (reverse-tail-good xs))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

What about map, filter?
Tail recursion ≠ accumulator pattern

; mutually tail recursive
(define (even n)
  (or (zero? n) (odd (- n 1))))
(define (odd n)
  (or (not (zero? n)) (even (- n 1))))

; tail recursive
(define (even2 n)
  (cond [(= 0 n) #t]
        [(= 1 n) #f]
        [#t (even2 (- n 2))]))

• Tail recursion and the accumulator pattern are **commonly used together**. They are **not synonyms**.
  – Natural recursion may use an accumulator.
  – Tail recursion does not necessarily involve an accumulator.
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 ________.

Python: loop iteration with mutation

\[
def \text{fib}(n):
    \text{fib}_i = 0
    \text{fib}_i\_plus\_1 = 1
    \text{for} \ i \ \text{in} \ \text{range}(n):
        \text{fib}_i\_prev = \text{fib}_i
        \text{fib}_i = \text{fib}_i\_\_plus\_1
        \text{fib}_i\_\_plus\_1 = \text{fib}_i\_prev + \text{fib}_i\_\_plus\_1
    \text{return} \ \text{fib}_i
\]

Racket: immutable natural recursion

\[
\begin{align*}
\text{(define (fib n)} & \text{ Racket: immutable natural recursion} \\
    & \text{(if (< n 2)} \\
        & \text{ n} \\
        & \text{ (+ (fib (- n 1)) (fib (- n 2)))))}
\end{align*}
\]

Racket: immutable tail recursion

\[
\begin{align*}
\text{(define (fib n)} & \text{ Racket: immutable tail recursion} \\
    & \text{(define (fib-tail n fibi fibi+1)} \\
        & \text{(if (= 0 n)} \\
            & \text{ fibi} \\
            & \text{ (fib-tail (- n 1) fibi+1 (+ fibi fibi+1)))} \\
        & \text{ (fib n 0 1))}
\end{align*}
\]

Tail Recursion 20
Identify dependences between ________.

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

```racket
(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
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
```

---

Tail Recursion 21
Fold: iterator over recursive structures
(a.k.a. reduce, inject, ...)

(fold_ combine init list)
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

Natural recursion

\((\text{foldr} \ combine \ init \ L)\)

Tail recursion

\((\text{foldl} \ combine \ init \ L)\)
Fold code: tail.rkt

- foldr implementation
- foldl implementation
- using foldr/foldl
- bonus mystery folding puzzle
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}
  – Later: argument function can use any “private” data in its environment.
  – Iterator does not have to know or help.

Tail Recursion