Tail Recursion

• 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

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

How efficient is this implementation?

- Space: \(O(\quad)\)
- Time: \(O(\quad)\)

CS 240-style machine model

Registers | Code | Stack
---|---|---
fixed-size, general purpose | | Call frame
Call frame | | Call frame
Call frame | | arguments, variables, return address per function call
cons cells, data structures, ...

Program Counter | Stack Pointer

https://cs.wellesley.edu/~cs251/s20/
Evaluation example

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

Call stacks at each step

(fact 3)  (fact 3): 3*
(fact 2)  (fact 2): 2*
(fact 1)  (fact 1): 1*
(fact 0)  (fact 0)

Remember: n ↦ 2; and "rest of function" for this call.

Space: O(      )
Time: O(      )

Tail recursive factorial

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

(fact-tail n 1)

Accumulator parameter provides result so far.
Base case returns full result.
Recursive case returns full result.

Initial accumulator provides base result.

Common patterns of work

Natural recursion:
Reduce argument
Accumulate result so far
Full result

Tail recursion:
Reduce argument
Accumulate result so far
Base case
Base result
Full result
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.

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

Evaluation example

Call stacks at each step

<table>
<thead>
<tr>
<th>(fact 3)</th>
<th>(fact 3)</th>
<th>(fact 3)</th>
<th>(fact 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ft 3 1)</td>
<td>(ft 3 1)</td>
<td>(ft 2 3)</td>
<td>(ft 1 6)</td>
</tr>
<tr>
<td>ft = fact-tail</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tail-call optimization

Space: O(  )
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.

Racket, ML, most “functional” languages, but not Java, C, etc.
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 e3)`, subexpressions `e1` and `e2` are **not** in tail position.

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

A recursive function is **tail-recursive** if and only if all of its recursive calls are tail calls.

Practice: use the transformation

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

;; Tail-recursive sum
(define (sum-tail xs)
  (define (sum-onto xs acc)
    (if (null? xs)
        acc
        (sum-onto (cdr xs) (+ (car xs) acc)))
  )
  (sum-onto xs 0)
)
```

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))))
)
```

Tail recursion

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

Transforming non-commutative steps

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

```
(define (rev xs acc)
  (if (null? xs)
      acc
      (rev (cdr xs) (append acc (list (car xs))))
  )
)
```

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

```
(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 acc (list (car xs))))
    )
  )
  (rev xs null)
)
```

(order matters)
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) acc</td>
</tr>
<tr>
<td>(rev (cdr xs) (append (list (car xs)) acc)))</td>
</tr>
<tr>
<td>(rev xs null))</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
</tbody>
</table>

\[ O(n^2) \]

- Append-based recursive reverse 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 recursion
- Tail-recursive reverse can avoid append in \( O(n) \).
  - Cons is \( O(1) \), done \( n \) times.

Tail recursion ≠ accumulator pattern

<table>
<thead>
<tr>
<th>define (even n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cond ((= 0 n) #t)</td>
</tr>
<tr>
<td>((= 1 n) #f)</td>
</tr>
<tr>
<td>(#t (even (- n 1))))</td>
</tr>
<tr>
<td>----------------</td>
</tr>
</tbody>
</table>

- 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 ________.

<table>
<thead>
<tr>
<th>define (fib n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(if (&lt; n 2) n</td>
</tr>
<tr>
<td>(+ (fib (- n 1)) (fib (- n 2))))</td>
</tr>
<tr>
<td>----------------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>define (fib n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(define (fib-tail n fibi fibi+1)</td>
</tr>
<tr>
<td>(if (= 0 n) fibi</td>
</tr>
<tr>
<td>(fib-tail (- n 1) fibi+1 (+ fibi fibi+1))))</td>
</tr>
<tr>
<td>(fib n 0 1))</td>
</tr>
<tr>
<td>----------------</td>
</tr>
</tbody>
</table>

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
Identify dependences between recursive calls.

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

(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))
```

```
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, ...)

```
(foldr 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

```
(foldr combine init L)
```

```
L ⟷ v_1 ⟷ v_2 ⟷ ... ⟷ v_{n-1} ⟷ v_n
```

```
(result ← combine ← combine ← ... ← combine ← combine ← Init)
```

```
L ─> Init ─> combine ─> combine ─> ... ─> combine ─> combine ─> result
```

Fold code: tail-practice.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

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