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

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

https://cs.wellesley.edu/~cs251/f19/
Example

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

<table>
<thead>
<tr>
<th>n</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3*2</td>
</tr>
<tr>
<td>4</td>
<td>4*3</td>
</tr>
</tbody>
</table>

Remember: $n \mapsto 2$; and “rest of function” for this call.

Space: $O( )$

Time: $O( )$

Tail Recursion

Naturally recursive factorial

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

Base case returns base result.

Recursive case returns result so far.

Compute remaining argument before/for recursive call.

Compute result so far after/from recursive call.

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

Base case returns base result.

Recursive case returns result so far.

Compute remaining argument before/for recursive call.

Compute result so far after/from recursive call.

Tail recursive factorial

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

Accumulator parameter provides result so far.

Base case returns full result.

Recursive case returns full result.

Initial accumulator provides base result.

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

Base case returns full result.

Recursive case returns full result.

Common patterns of work

Natural recursion:

- Argument
- Full result

Tail recursion:

- Argument
- Base result

Reduce argument

Accumulate result so far

Accumulate result so far

Base case

Base result

Full result
### Natural recursion

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

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

### Tail recursion

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

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

### The call stacks

Nothing useful remembered here.

- `(fact 3)`: 24
- `(fact-tail 3, 24)`: 24
- `(fact-tail 4, 24)`: 24

### Optimization under the hood

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.

- `(fact 0 6)`: 6
- `(fact-tail 0, 24): 24`
- `(fact-tail 3, 2): 24`
- `(fact-tail 2, 3): 24`
- `(fact-tail 1, 24): 24`
- `(fact-tail 4, 24): 24`

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

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

Natural recursion

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

Tail recursion

Example

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

```scheme
(define (sum xs)
  (define (sum-tail xs acc)
    (if (null? xs)
        acc
        (sum-tail (cdr xs) (+ (car xs) acc)))
  )
  (sum-tail xs 0))
```

Practice

```scheme
(define (rev xs)
)
```

```scheme
(define (rev xs)
  
  What about map, filter?
)
```

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 ________.

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

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

\[L \rightarrow v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_{n-1} \rightarrow v_n\]

\[(\text{foldl} \ \text{combine} \ \text{init} \ L)\]

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

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