CS450

Structure of Higher Level Languages

Lecture 07: Tail-call optimization

Tiago Cogumbreiro
Today we will learn...

- Identifying a tail-call optimization
- Internals of the tail-call optimization
- Structures (safe and easy user-data structures)

Learning how to write tail-call optimizations is explained in future lessons. Today, we focus on what the optimization is, and on why the optimization works.

Suggested reading

SICP §1.2.1
Tail-call optimization

What is it?
max: attempt 1

(define (max xs)
  (cond
   [(empty? xs) (error "max: expecting a non-empty list!")]  ; The list only has one element (the max)
   [(empty? (rest xs)) (first xs)]  ; The max of the rest is smaller than 1st
   [(> (first xs) (max (rest xs))) (first xs)]  ; Otherwise, use the max of the rest
   [else (max (rest xs))])))
We use a local variable to cache a duplicate computation.

```
(define (max xs)
  (cond
    [(empty? xs) (error "max: expecting a non-empty list!")]  ; Attempt #1: 20 elements in 75.78ms
    [(empty? (rest xs)) (first xs)]
    [else (define rest-max (max (rest xs))) ; Cache the max of the rest
          (cond
            [>(first xs) rest-max) (first xs)]
            [else rest-max]])])
```

- Attempt #1: 20 elements in 75.78ms
- Attempt #2: 1,000,000 elements in 101.15ms

5000 × more elements for the same amount of time!
Can we do better?
(define (max xs) =
  ; 1. Abstract the maximum between two numbers
  (define (max2 x y) (cond [(< x y) y] [else x]))
  ; 2. Use parameters to store accumulated results
  (define (max-aux curr-max xs)
    ; 3. Accumulate maximum number before recursion
    (define new-max (max2 curr-max (first xs)))
    (cond
      [(empty? (rest xs)) new-max] ; Last element is max
      [else (max-aux new-max (rest xs))]) ; Otherwise, recurse
    (cond
      [(empty? xs) (error "max: empty list")]; 4. Only test if the list is empty once
      [else (max-aux (first xs) xs)]))
Comparing both attempts

<table>
<thead>
<tr>
<th>Element count</th>
<th>Execution time</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attempt #2</td>
<td>1,000,000</td>
<td>101.15ms</td>
</tr>
<tr>
<td>Attempt #3</td>
<td>1,000,000</td>
<td>20.98ms</td>
</tr>
<tr>
<td>Attempt #2</td>
<td>10,000,000</td>
<td>1410.06ms</td>
</tr>
<tr>
<td>Attempt #3</td>
<td>10,000,000</td>
<td>237.66ms</td>
</tr>
</tbody>
</table>

Why is attempt #3 so much faster?

Because attempt #3 is being target of a Tail-Call optimization!
How are both attempts different?

Attempt 2

```scheme
(define rest-max (max (rest xs)))
(cond
 [(max2 (first xs) rest-max) (first xs)]
 [else rest-max]]))
```

Attempt 3

```scheme
(define new-max (max2 curr-max (first xs)))
(cond
 [(empty? (rest xs)) new-max]
 [else (max-aux new-max (rest xs))])
```

; 1. Do recursive call
; 2. Handle accumulated result

; 1. Handle accumulated result
; 2. Do recursive call
Tail-call optimization

Why does it work?
Call stack & Activation frame

- **Call Stack**: To be able to call and return from functions, a program internally maintains a stack called the *call-stack*, each of which holds the execution state at the point of call.

- **Activation Frame**: An activation frame maintains the execution state of a running function. That is, the activation frame represents the local state of a function, it holds the state of each variable.

- **Push**: When calling a function, the caller creates an activation frame that is used by the called function (e.g., to pass arguments to the function being called).

- **Pop**: Before a function returns, it pops the call stack, freeing its local state.
Consider executing the factorial

Program

```
(define (fact n)
  (cond
    [(= n 1) 1]
    [else
     (* n (fact (- n 1)))]))
```

Evaluation

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Call-Stack

```
[n=3, return=(* 3 (fact 2))]
[n=3, return=(* 3 ?)], [n=2, return=(* 2 (fact 1))]
[n=3, return=(* 3 ?)], [n=2, return=(* 2 ?)], [n=1, return=1]
[n=3, return=(* 3 ?)], [n=2, return=2]
[n=3, return=6]
```
Call-stack and recursive functions

Recursive functions pose a problem to this execution model, as **the call-stack may grow unbounded**! Thus, most non-functional programming languages are conservative on growing the call stack.

```python
def fact(n):
    return 1 if n <= 1 else n * fact(n - 1)
fact(1000)
```

**Outputs**

File "<stdin>", line 1, in fact
RuntimeError: maximum recursion depth exceeded
Factorial: attempt #2

Program

```
(define (fact n)
 (define (fact-iter n acc)
   (cond
     [(= n 0) acc]
     [else
      (fact-iter (- n 1) (* acc n))]))
 (fact-iter n 1))
(fact 3)
```

Evaluation

```
(fact 3)
(fact-iter 3 1)
(fact-iter 2 3)
(fact-iter 1 6)
6
```
Factorial: attempt #2

Call stack

\[
\begin{align*}
[n=3, \text{return}=(\text{fact-iter } 3 \ 1)] \\
[n=3, \text{return}=?], [n=3, \text{acc}=1, \text{return}=(\text{fact-iter } 2 \ 3)] \\
[n=3, \text{return}=?], [n=3, \text{acc}=1, \text{return}=?], [n=2, \text{acc}=3, \text{return}=(\text{fact-iter } 1 \ 6)] \\
[n=3, \text{return}=?], [n=3, \text{acc}=1, \text{return}=?], [n=2, \text{acc}=3, \text{return}=?], [n=1, \text{acc}=6, \text{return}=6] \\
[n=3, \text{return}=?], [n=3, \text{acc}=1, \text{return}=?], [n=2, \text{acc}=3, \text{return}=6] \\
[n=3, \text{return}=?], [n=3, \text{acc}=1, \text{return}=6] \\
[n=3, \text{return}=6]
\end{align*}
\]
The **tail position** of a sequence of expressions is the last expression of that sequence.

When a function call is in the tail position we named it the **tail call**.
Tail call and the call stack

A tail call does not need to push a new activation frame! Instead, the called function can "reuse" the frame of the current function. For instance, in $(\text{fact } 3)$, the call $(\text{fact-iter } 3 \ 1)$ is a tail call.

Can be rewritten with:

$$
\begin{align*}
&[n=3, \text{return}=(\text{fact-iter } 3 \ 1)] \\
&[n=3, \text{return}=?], [n=3, \text{acc}=1, \text{return}=(\text{fact-iter } 2 \ 3)]
\end{align*}
$$

In attempt #2, both calls to fact-iter are tail calls.
Tail-Call Optimization

- Eschews the need to allocate a new activation frame
- In a recursive tail call, the compiler can convert the recursive call into a loop, which is more efficient to run (recall our $5 \times$ speedup)
Revisiting user data structures
User data structures

Recall the 3D point from Lecture 3

; Constructor
(define (point x y z) (list x y z))

; Accessors
(define (point-x pt) (first pt))
(define (point-y pt) (second pt))
(define (point-z pt) (third pt))

And the name data structure

; Constructor
(define (name f m l) (list f m l))

; Accessor
(define (name-first n) (first n))
(define (name-middle n) (second n))
(define (name-last n) (third n))

How do we prevent such errors?

(define p (point 1 2 3))
(name-first p) ; This should be an error, and instead it happily prints 1
#lang racket

(require rackunit)

(struct point (x y z) #:transparent)

(define pt (point 1 2 3))

(check-equal? 1 (point-x pt)); the accessor point-x is automatically defined
(check-equal? 2 (point-y pt)); the accessor point-y is automatically defined

(struct name (first middle last))

(define n (name "John" "M" "Smith"))

(check-equal? "John" (name-first n))

(check-true (name? n)); We have predicates that test the type of the value
(check-false (point? n)); A name is not a point
(check-false (list? n)); A name is not a list

;(point-x n);; Throws an exception
;point-x: contract violation
;  expected: point?
;  given: #<name>
Benefits of using structs

- Reduce boilerplate code
- Ensure type-safety
Implementing Racket's AST

Grammar

```
expression = value | variable | apply | define
value = number | void | lambda
apply = ( expression+ )
lambda = ( lambda ( variable* ) term+)
```
Implementing values

\[
\text{value} = \text{number} \mid \text{void} \mid \text{lambda}
\]

\[
\text{lambda} = ( \lambda ( \text{variable}^* ) \text{term}^+)\]
Implementing values

value = number | void | lambda

lambda = (lambda (variable* ) term+)

(define (r:value? v)
  (or (r:number? v)
    (r:void? v)
    (r:lambda? v)))

(struct r:void () #:transparent)
(struct r:number (value) #:transparent)
(struct r:lambda (params body) #:transparent)

- We are using a prefix r: because we do not want to redefined standard-library definitions.
Implementing expressions

expression = value | variable | apply
apply = ( expression+ )
Implementing expressions

expression = value | variable | apply
apply = ( expression+)

(define (r:expression? e)
  (or (r:value? e)
       (r:variable? e)
       (r:apply? e)))
(struct r:variable (name) #:transparent)
(struct r:apply (func args) #:transparent)

In r:apply we distinguish between the expression that represents the function func, and the (possibly empty) list of arguments args.
Implementing terms

term = define | expression
define = ( define identifier expression ) | ( define ( variable+ ) term+)

CS450 ☾ Tail-call optimization ☾ Lecture 07 ☾ Tiago Cogumbreiro

26 / 27
Implementing terms

\[
\text{term} = \text{define} \mid \text{expression}
\]

\[
\text{define} = (\text{define} \ \text{identifier} \ \text{expression}) \mid (\text{define} \ (\text{variable}+) \ \text{term}+)
\]

\[
\begin{align*}
\text{(define} \ (r:\text{term}? \ t) \\
\quad \text{(or} \ (r:\text{define}? \ t) \\
\quad \quad (r:\text{expression}? \ t))) \\
\text{(struct} \ r:\text{define} \ (\text{var} \ \text{body}) \ #:\text{transparent})
\end{align*}
\]

For our purposes of defining the semantics in terms of implementing an interpreter, we do not want to distinguish between a basic definition and a function definition, as this would unnecessarily complicate our code. We, therefore, represent a definition with a single structure, which pairs a variable and an expression (e.g., a lambda). In our setting, the distinction between a basic and a function definition is syntactic (not semantic).
Summary of `struct`

```lisp
(struct point (x y z) #:transparent)
```

Simplifies the definition of data structures:

- Creates selectors automatically, eg, `point-x`
- Creates type query, eg, `point?`
- Ensures that functions of a given struct can only be used on values of that struct. *Because, not everything is a list.*

What is `#:transparent`? A transparent struct prints its contents when rendered as a string.