CS450

Structure of Higher Level Languages

Lecture 4: Nested definitions, tail-call optimization

Tiago Cogumbreiro
A quick recap...
User data-structures

We can represent data-structures using pairs/lists. For instance, let us build a 3-D point data type.

; Constructor
(define (point x y z) (list x y z))
(define (point? x)
  (and (list? x)
       (= (length x) 3)))

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

- a default constructor with the name of the type and its fields as parameters
- one accessor per field
- function point? returns true if, and only if, the given value is a point (Exercise 3 of HW1)
Quoting exercises:

- You can serialize any code (even non-valid Racket programs) as long: (1) literals follow Racket's rules (numbers, strings, identifiers) and (2) parenthesis are well balanced
- We can write 'term rather than (quote term)
- How do we serialize term (lambda (x) x) with quote?
- How do we serialize term (+ 1 2) with quote?
- How do we serialize term (cond [(> 10 x) x] [else #f]) with quote?
- *Can we serialize a syntactically invalid Racket program?*
Quoting exercises:

- You can serialize any code (even non-valid Racket programs) as long: (1) literals follow Racket's rules (numbers, strings, identifiers) and (2) parenthesis are well balanced.
- We can write 'term rather than (quote term).
- How do we serialize term (lambda (x) x) with quote?
- How do we serialize term (+ 1 2) with quote?
- How do we serialize term (cond [(> 10 x) x] [else #f]) with quote?
- Can we serialize a syntactically invalid Racket program? No! You would not be able to serialize this expression (. Quote only accepts a S-expressions (parenthesis must be well-balanced, identifiers must be valid Racket identifiers, number literals must be valid).
- Can we serialize an invalid Racket program?
Quoting exercises:

- You can serialize any code (even non-valid Racket programs) as long: (1) literals follow Racket’s rules (numbers, strings, identifiers) and (2) parenthesis are well balanced.
- We can write 'term rather than (quote term).
- How do we serialize term (lambda (x) x) with quote?
- How do we serialize term (+ 1 2) with quote?
- How do we serialize term (cond [(> 10 x) x] [else #f]) with quote?
- Can we serialize a syntactically invalid Racket program? No! You would not be able to serialize this expression (. Quote only accepts a S-expressions (parenthesis must be well-balanced, identifiers must be valid Racket identifiers, number literals must be valid).
- Can we serialize an invalid Racket program? Yes. For instance, try to quote the term: (lambda)
#lang racket

(require rackunit)

(check-equal? 3 (quote 3)) ; Serializing a number returns the number itself
(check-equal? 'x (quote x)) ; Serializing a variable named x yields symbol 'x
(check-equal? (list '+ 1 2) (quote (+ 1 2))) ; Serialization of function as a list
(check-equal? (list 'lambda (list 'x) 'x) (quote (lambda (x) x)))
(check-equal? (list 'define (list 'x)) (quote (define (x))))
On HW1 Exercise 4

- The input format of the quoted term are precisely described in the slides of Lecture 3
- You do not need to test recursively if the terms in the body of a function declaration or definition are valid.

For instance,

```
function-def = ( lambda ( variable* ) term+)
```

- A list, with one symbol `lambda` followed by zero or more symbols, and one or more terms.
Today we will...

1. Learn about a good use of nested definitions
2. Analyse some code's performance
3. Introduce tail-call optimization

Acknowledgment: Today's lecture is inspired by Professor Dan Grossman's wonderful lecture in CSE341 from the University of Washington. (Video available)
Build a list from 1 up to n

Our goal is to build a list from 1 up to some number. Here is a template of our function and a test case for us to play with. For the sake of simplicity, we will not handle non-positive numbers.

```racket
#lang racket
(define (countup-from1 x) #f)

(require rackunit)
(check-equal? (list 1) (countup-from1 1))
(check-equal? (list 1 2) (countup-from1 2))
(check-equal? (list 1 2 3 4 5) (countup-from1 5))
```

Hint: write a helper function `count` that builds counts from n up to m.
Exercise 1: attempt #1

We write a helper function `count` that builds counts from $n$ up to $m$.

```
#lang racket
(define (countup-from1 x)  
  (count 1 x))

(define (count from to)  
  (cond  
    [(= from to) (list to)]  
    [else (cons from (count (+ 1 from) to))]))
```
Exercise 1: attempt #2

We move function `count` to be internal to function `countup-from1`, as it is a helper function and therefore it is good practice to make it `private` to `countup-from1`.

```
(define (countup-from1 x)
  ; Internally defined function, not visible from
  ; the outside
  (define (count from to)
    (cond [(equal? from to) (list to)]
           [else (cons from (count (+ 1 from) to))]))
  ; The same call as before
  (count 1 x))
```
When to nest functions

Nest functions:
- If they are unnecessary outside
- If they are under development
- If you want to hide them: Every function in the public interface of your code is something you'll have to maintain!
Intermission:
Nested definitions
Nested definition: local variables

Nested definitions bind a variable within the body of a function and are only visible within that function (these are local variables)

```
#lang racket
(define (f x)
  (define z 3)
  (+ x z))

(+ 1 z); Error: z is not visible outside function f
```
Nested definitions shadow other variables

- Nested definitions silently shadow any already defined variable

```racket
#lang racket
(define z 10)
(define (f x)
  (define x 3) ; Shadows parameter
  (define z 20) ; Shadows global
  (+ x z))
(f 1) ; Outputs 23
```
No redefined local variables

It is an error to re-define local variables

```racket
#lang racket
(define (f b)
    ; OK to shadow a parameter
    (define b (+ b 1))
    (define a 1)
    ; Not OK to re-define local variables
    ;; Error: define-values: duplicate binding name
    (define a (+ a 1))
    (+ a b))
```
Back to Exercise 1
Notice that we have some redundancy in our code. In function `count`, parameter `to` remains unchanged throughout execution.

```scheme
(define (countup-from1 x)
  ; Internally defined function, not visible from
  ; the outside
  (define (count from to)
    (cond [(equal? from to) (list to)]
          [else (cons from (count (+ 1 from) to))]))
  ; The same call as before
  (count 1 x))
```
Exercise 1: attempt #3

We removed parameter `to` from function `count` as it was constant throughout the execution. Variable `to` is captured/copied when `count` is defined.

```
(define (countup-from1 to)
    ; Internally defined function, not visible from
    ; the outside
    (define (count from)
        (cond [(equal? from to) (list to)]
            [else (cons from (count (+ 1 from)))]))
    ; The same call as before
    (count 1))
```
Example 1: summary

- Use a nested definition to hide a function that is only used internally.
- Nested definitions can refer to variables defined outside the scope of their definitions.
- The last expression of a function's body is evaluated as the function's return value.
Example 2

Maximum number from a list of integers
Example 2: attempt 1

Finding the maximum element of a list.

```racket
#lang racket
(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)]
   [>(first xs) (max (rest xs))]  ; The max of the rest is smaller than 1st
   [else (max (rest xs))]])

; A simple unit-test
(require rackunit)
(check-equal? 10 (max (list 1 2 10 4 0)))
```

We use function `error` to abort the program with an exception. We use functions `first` and `rest` as synonyms for `car` and `cdr`, as it reads better.
Example 2: attempt 1

Finding the maximum element of a list.

Let us benchmark \texttt{max} with sorted list (worst-case scenario):

- 20 elements: 18.43ms
- 21 elements: 36.63ms
- 22 elements: 75.78ms

Whenever we add an element we double the execution time. \textbf{Why?}
Example 2: attempt 1

Whenever we hit the else branch (because we can't find the maximum), we re-compute the max element.

\[
\text{(define (max xs)} \\
\text{(cond)} \\
\text{[(empty? xs) (error "max: expecting a non-empty list!")]}} \\
\text{[(empty? (rest xs)) (first xs)]} ; \text{The list only has one element (the max)} \\
\text{[ (> (first xs) (max (rest xs))) (first xs)]} ; \text{The max of the rest is smaller than 1st} \\
\text{[else (max (rest xs)))]} \quad ; \text{Otherwise, use the max of the rest}
\]
Example 2: attempt 2

We use a local variable to cache a duplicate computation.

```scheme
(define (max xs)
  (cond
    [(empty? xs) (error "max: expecting a non-empty list!")]
    [(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
Example 2 takeaways

- Use nested definitions to cache intermediate results
- Identify repeated computations and cache them in nested (local) definitions
Example 2: attempt 3

\[
\text{(define (max xs) =}
\begin{array}{l}
; \text{The maximum between two numbers}
\text{(define (max2 x y) (cond [(< x y) y] [else x]))}
\end{array}
\begin{array}{l}
; \text{Accumulate the maximum number as a parameter of recursion}
\text{(define (max-aux curr-max xs)}
\end{array}
\begin{array}{l}
; \text{Get the max between the accumulated and the first}
\text{(define new-max (max2 curr-max (first xs)))}
\end{array}
\begin{array}{l}
\begin{array}{l}
\text{(cond}
\end{array}
\begin{array}{l}
\text{[(empty? (rest xs)) new-max]} \quad ; \text{Last element is max}
\end{array}
\begin{array}{l}
\text{[else (max-aux new-max (rest xs))]} \quad ; \text{Otherwise, recurse}
\end{array}
\text{; Only test if the list is empty once}
\end{array}
\begin{array}{l}
\text{(cond}
\end{array}
\begin{array}{l}
\text{[(empty? xs) (error "max: empty list")]} \\
\text{[else (max-aux (first xs) xs)]])}
\end{array}
\end{array}
\]
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!
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 (eg, 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

\[
\text{(define (fact n)}
\text{  (cond}
\text{    [(= n 1) 1]
\text{    [else (* n (fact (- n 1)))]})
\text{)}
\]

Evaluation

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

Call-Stack

\[
\text{[n=3,return=(* 3 (fact 2))]} \rightarrow \text{[n=3,return=(* 3 (?)]},\text{[n=2,return=(* 2 (fact 1))]} \rightarrow \text{[n=3,return=(* 3 (?)]},\text{[n=2,return=(* 2 (?)]},\text{[n=1,return=1]} \rightarrow \text{[n=3,return=(* 3 (?)]},\text{[n=2,return=2]} \rightarrow \text{[n=3,return=6]}
\]

Consider executing the factorial
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.

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

Can be rewritten with:

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

In attempt #2, both calls to \texttt{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)
Guidelines to write tail-recursive code

- Create a helper function that takes an accumulator (which stores what is calculated after the call)
- The base case of the original function becomes the initial accumulator
- The base case of the new function becomes the accumulator

Caveats

- Not all recursive functions can be optimized to be tail-recursive (eg, in tree-based algorithms when the function recurses on more than one node)
- Be weary that: premature optimization is the root of all evils.