CSC 347 - Concepts of Programming Languages

Tail Recursion

Instructor: Stefan Mitsch

Learning Objectives

❓ How to get recursive iteration without stack penalty?

- Understand tail recursion

Call Stack

- Contains *activation records* (AR) for active calls, also known as *stack frames*
- Changes to call stack
  - AR pushed when a function/method call is made
  - AR popped when a function/method returns
- Runtime environments limit size of call stacks?
- Can cause problems with deep recursion
  - Java: `StackOverflowError`
  - C: stack limits set by operating system
  - Scheme: depends on interpreter
  - Scala: see Java

Recursion and Stack Limitations

```
def countDown (x:Int) : Int = if x == 0 then 0 else 1 + countDown (x - 1)
```

- Each `(1 + ...)` represents a new AR

```
countDown (5)
--> 1 + countDown (4)
--> 1 + (1 + countDown (3))
--> 1 + (1 + (1 + countDown (2)))
--> 1 + (1 + (1 + (1 + countDown (1))))
--> 1 + (1 + (1 + (1 + (1 + countDown (0)))))
--> 1 + (1 + (1 + (1 + (1 + 0))))
```

- Summing up left to after the last recursive call returns!

Tail Recursive Calls

```
// *tail-recursive functions* because all recursive calls are tail-recursive
def countDownAux (x:Int,result:Int) : Int =
  if x == 0 then result
  else countDownAux(x-1,1+result) // *tail-recursive call*

def countDown (int x) = countDownAux(x,0)
```

```
countDownAux (5,0)
--> countDownAux (4,1)
--> countDownAux (3,2)
--> countDownAux (2,3)
--> countDownAux (1,4)
--> countDownAux (0,5)
```

- Result sum computed before recursive call is made, no work left!

Tail Call Optimization

- Many compilers implement *tail-call optimization*
  - overwrite existing activation record instead of creating new
- Recursive calls must be *tail-recursive*
- Includes *mutual recursion*
  - `f` calls to `g`, which calls back to `f`

Tail Recursive Call: C

- Refactor work and accumulate result

```
typedef struct node node;
struct node { int item; node *next; };

int sum_aux (node *data, int result) {
  if (!data) {
    return result;
  } else {
    return sum_aux (data->next, result + data->item);
  }
}

int sum (node *data) {
  return sum_aux (data, 0);
}
```

Optimize to Loop

```
$ gcc -std=c99 -O2 -S tail-recursion2.c
```

```
sum_aux:
        testq   %rdi, %rdi
        movl    %esi, %eax
        je      .L7
.L9:
        addl    (%rdi), %eax
        movq    8(%rdi), %rdi
        testq   %rdi, %rdi
        jne     .L9
.L7:
        rep
        ret
```

Tail Recursion: Scheme

- Implementations perform *tail-call optimization*
- For *tail-recursive* functions
- More generally, for *tail-calls*

Exponential Growth: Scheme

- Want very long linked lists: avoid stack overflow!
- Is it tail recursive?
  - `sublist` necessary?

```scheme
(define (long-list n)
  (if (= n 0)
      '(1)
      (let ([sublist (long-list (- n 1))])
        (append sublist sublist))))
```

```
$ csi

#;1> (long-list 0)
(1)
#;2> (long-list 1)
(1 1)
#;3> (long-list 2)
(1 1 1 1)
#;4> (long-list 4)
(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1)
#;6> (length (long-list 20))
1048576
```

Tail Recursion: Scheme

- Tail recursive, then recursive summation

```
(define (sum-tail-recursive xs)
  (let loop ([xs xs] [result 0])
    (if (eq? xs '())
        result
        (loop (cdr xs) (+ (car xs) result)))))

(define (sum xs)
  (if (eq? xs '())
      0
      (+ (car xs) (sum (cdr xs)))))
```

```
$ csi

#;8> (sum-tail-recursive (long-list 20))
1048576

#;10> (sum (long-list 20))
Segmentation fault
```

Exponential Growth: Scala

- As in Scheme

```
def longList (n:Int) : List[Int] =
  if n == 0 then
    List (1)
  else
    val sublist = longList (n - 1)
    sublist ::: sublist
```

Tail Recursion: Scala

- `tailrec` annotation

```
import scala.annotation.tailrec

def sumTailRecursive (xs:List[Int]) : Int =
  @tailrec
  def aux (xs:List[Int], result:Int) : Int =
    xs match
      case Nil   => result
      case y::ys => aux (ys, y + result)
  aux (xs, 0)
```

```
scala> longList (20).length
res0: Int = 1048576

scala> sumTailRecursive (longList (20))
res1: Int = 1048576
```

Tail Recursion: Scala

- `tailrec` annotation fails if not optimized

```scala
import scala.annotation.tailrec

def sumTailRecursive (xs:List[Int]) : Int =
  @tailrec
  def aux (xs:List[Int], result:Int) : Int =
    xs match
      case Nil    => result
      case y::ys => 1 + aux (ys, y + result)   // bogus "1 + ..."
  aux (xs, 0)
```

- Scala compiler rejects the code

```
error: could not optimize @tailrec annotated method aux:
  it contains a recursive call not in tail position
```

Exercise: Recursive vs. Tail-Recursive Fibonacci

```
def fib(n:Int) : Long =
  if n <= 1 then n
  else fib(n-1) + fib(n-2)
```

- Time complexity
  - $O(2^n)$
- ❓ How to improve?
-

fib(0)	fib(1)	fib(2)	fib(3
0	1	1	2

- Represent sliding window in result (not tail-recursive!)

```
def fib(n:Int) : (Long, Long) =
  if n <= 1 then (0, n)
  else
    val (a, b) = fib(n-1)
    (b, a+b)
```

- Represent sliding window in arguments (tail-recursive)

```
def fib(n:Int, a:Long=0, b:Long=1) : Long =
  if n == 0 then a
  else if n == 1 then b
  else fib(n-1, b, a+b)
```

Summary

- Tail-call optimization
  - avoids the performance penalty of creating activation records
  - overwrites an existing activation record
  - all recursive calls must be in *tail position* (last operation)