CSC324: Principles of Programming Languages

Lecture 6

Wednesday 27 May 2020

Last time...

… we saw how **quoting** and **quasiquoting** allows us to *defer the evaluation of an expression*.

Today, we'll continue our discussion into the practicalities of **evaluation semantics**.

Recall:

- Denotational semantics: "what does the expression evaluate to?"

- Operational semantics: "How does the expression evaluate to its value?"
    - We had previously discussed the substitution model of function application

```
; apply a squaring function to the sum of 2 and 3
((λ (x) (* x x)) (+ 2 3))
```

It's clear that we get 25, and we know that perform λ-calculus style substitution to yield 10, but what _specifically_ happens, and in what order?

Eager evaluation

```
(define (square x) (* x x))
```

```
; Eager evaluation:
(square (+ 2 3))                  ; 1. Eval the expr representing the function
((λ (x) (* x x)) (+ 2 3))         ; 2. Eval each argument's expression
((λ (x) (* x x)) 5)               ; 3. Apply the evaled arguments to the fn
(* 5 5)                           ; (4. evaluate the function expression)
25
```

Lazy evaluation

```
(define (square x) (* x x))
```

Hypothetical in Racket!!

```
; Lazy evaluation:
(square (+ 2 3))           ; 1. Eval the expr representing the function
((λ (x) (* x x)) (+ 2 3))  ; 2. Apply the unevaled arguments to the fn
(* (+ 2 3) (+ 2 3))        ; 3. Eval the function expression
(* 5 (+ 2 3))
(* 5 5)
25
```

Eval, interrupted

```
; Eager evaluation
(square (/ 1 0))
((λ (x) (* x x)) (/ 1 0))
((λ (x) (* x x)) <boom!>) ; div by zero before square is called

; Lazy evaluation
(square (/ 1 0))
((λ (x) (* x x)) (/ 1 0))
(* (/ 1 0) (/ 1 0))
(* <boom> (/ 1 0)) ; division by zero inside square
```

Consequences of eager eval in fn application

- All arguments are evaluated, even if they're not used in the body of a function

- If an argument is repeatedly used, it's only evaluated once

Consequences of lazy eval in fn application

- *Only the arguments used in the function are actually evaluated*
- *An expression repeatedly used in the function is repeatedly evaluated*
- *Warning: Laziness may get in the way of tail-call optimisation! We'll see how in a bit.*

Non-strict eval in otherwise strict-eval languages

Recall **short-circuiting boolean operators:**

```
lhs_expr && rhs_expr
```

If the left hand side
evaluates to false...

...is there any use in evaluating
the right hand side?

Non-strict eval in otherwise strict-eval languages

Recall **short-circuiting boolean operators:**

```
#lang racket

(and #f (/ 1 0))
(and #t (/ 1 0))
```

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.
#f
/: division by zero
>
```

Non-strict eval in otherwise strict-eval languages

Recall **short-circuiting boolean operators:**

The difference in how boolean operators evaluate their arguments, as compared to function application and other "normal" evaluation, suggests something special is going on.

Definition: a **syntactic form** generalises the notion of an expression.

Definition: a **special form** is a syntactic form that follows special evaluation rules.

Delaying evaluation in an eager language

Recall that the body of a function is not evaluated until the function is called.

<u>**Definition**:</u> A **thunk** is a function with zero arguments. Thunks are often defined in order to be passed into a function, that will evaluate the "expression" at some point in the future.

In other languages, *callbacks* may serve the same purpose as a thunk.

Free identifiers and Closures

```
(define x 3)
(define a-thunk (λ () (+ x 1)))
```

Here, a-thunk is a function that contains an identifier defined outside its local scope.

<u>**Definition:**</u> A **free identifier** is an identifier within a function body that:

- Is not a parameter to the function
- Is not bound in a local let-expression

Free identifiers and Closures

```
(define (add-n n)
  (λ (x) (+ n x)))
```

Here, add-n returns a function, that contains an ident defined outside its local scope.

<u>Definition:</u> A **free identifier** is an identifier within a function body that:

- Is not a parameter to the function
- Is not bound in a local let-expression

Free identifiers and Closures

```
(define (add-n n)
  (λ (x) (+ n x)))
```

Here, a-thunk is a function that contains an identifier defined outside its local scope.

<u>Definition:</u> A function with a free identifier is said to **close over** that identifier, and we say that the function in question is a **closure**.

Haskell: lazy by design

In contrast to Racket, Haskell **uses non-strict evaluation semantics** for:

- arguments to functions
- name bindings

Haskell: evaluating function arguments lazily

```
Prelude> f x y = x
Prelude> f 3 (error "Second eval argument")
3
Prelude>
Prelude> my_and x y = if x then y else False
Prelude> my_and False (error "uh oh")
False
Prelude> my_and True (error "uh oh")
*** Exception: uh oh
CallStack (from HasCallStack):
  error, called at <interactive>:25:14 in interactive:Ghci15
```

Haskell: evaluating name bindings lazily

Imagine implementing a function that behaves like Python's `range()`, which lazily constructs the list [0,1,2,..(n-1)]:

```
➜  ~ python3
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
[Clang 11.0.0 (clang-1100.0.33.16)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>>
```

Haskell: evaluating name bindings lazily

```haskell
my_range x =
    let
        iter n
            | x == n     = []
            | otherwise = [n] ++ iter (n+1)
    in
        iter 0
```

```
my_range 0
-> []
```

Haskell: evaluating name bindings lazily

```haskell
my_range x =
    let
        iter n
            | x == n     = []
            | otherwise = [n] ++ iter (n+1)
    in
        iter 0
```

```
my_range 1
-> [0] ++ (iter (0+1))
```

Haskell: evaluating name bindings lazily

```haskell
my_range x =
    let
        iter n
            | x == n     = []
            | otherwise = [n] ++ iter (n+1)
    in
        iter 0
```

```
my_range 1
->[0] ++ (iter (0+1))
```

Not evaluated until someone reads an element from the list

Haskell: evaluating name bindings lazily

```haskell
my_range x =
    let
        iter n
            | x == n     = []
            | otherwise = [n] ++ iter (n+1)
    in
        iter 0
```

```
my_range 1
-> [0] ++ (iter (0+1))
```

Not evaluated until someone reads **the second element** from the list

Haskell: evaluating name bindings lazily

```
*Main> my_take 3 (my_range 10)
[0,1,2]
*Main>
```

The thunk that would have produced the list [3,4,...8,9] was left unevaluated!

Tail-call optimisation and lazy evaluation

Recall our right fold from the previous class. The size of the expression grows linearly with the size of the input.

```
; a right fold consumes a binary operator f,
; an initial value, and a list, and produces the
; result of applying the binary operator to all
; elements of the list, ending with the fold.
;
; (a -> b -> b) -> b -> list a -> b
(define (foldr f acc l)
  (match* (l)
    [('()) acc]
    [((cons x xs)) (f x (foldr f acc xs))]))

(foldr + 0 (list 1 2 3))
(+ 1 (foldr + 0 (list 2 3)))
(+ 1 (+ 2 (foldr 0 (list 3))))
(+ 1 (+ 2 (+ 3 (foldr 0 '()))))
(+ 1 (+ 2 (+ 3 0)))
(+ 1 (+ 2 3))
(+ 1 5)
6
```

Tail-call optimisation and lazy evaluation

We saw how this problem was avoided in Racket by using a left fold instead.

The fold evaluates the reduction function with the current list element and the accumulator for every recursive call.

```
; foldl consists of a binary function to apply,
; an accumulator, and a list of a, and applies
; the binary operator in turn to each of a, with the
; accumulator passed to the function.
; (a -> b -> b) -> b -> list a -> b
(define (foldl f acc l)
  (match* (l)
    [('()) acc]
    [((cons x xs)) (foldl f (f x acc) xs)]))

(foldl + 0 '(1 2 3))
(foldl + (+ 1 0) '(2 3))
(foldl + 1 '(2 3))
(foldl + (+ 2 1) '(3))
(foldl + 3 '(3))
(foldl + (+ 3 3) '())
(foldl + 6 '())
6
```

Tail-call optimisation and lazy evaluation

However, in Haskell, both fold variations run out of memory with a large list.

```
Prelude> foldr (+) 0 [0..100000000000]
*** Exception: stack overflow
Prelude> foldl (+) 0 [0..100000000000]
*** Exception: stack overflow
```

Why? Isn't foldl tail-recursive?

```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs
```

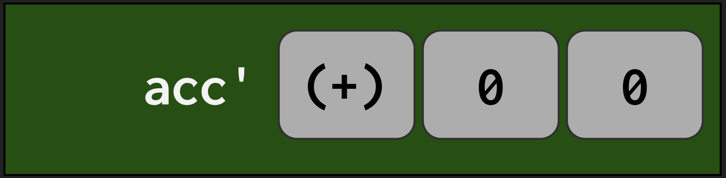
Heap

Function Applications

```
my_foldl (+) acc            [0..1000000]
```

```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs
```

Heap



Function Applications

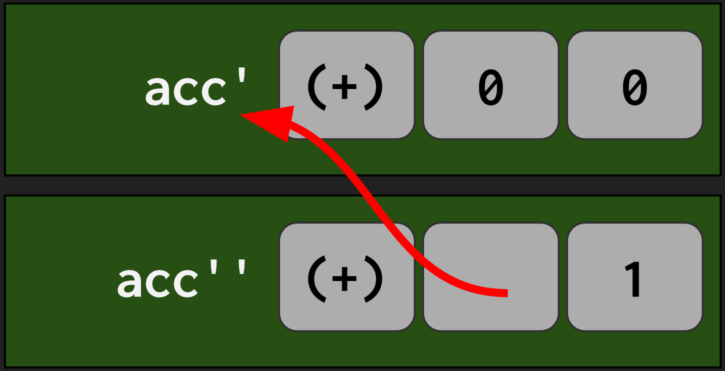
```
my_foldl (+) acc           [0..1000000]
my_foldl (+) (acc + 1)     [1..1000000]
```

```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs
```

Function Applications

```
my_foldl (+) acc          [0..1000000]
my_foldl (+) (acc + 1)    [1..1000000]
my_foldl (+) (acc' + 1)   [2..1000000]
```

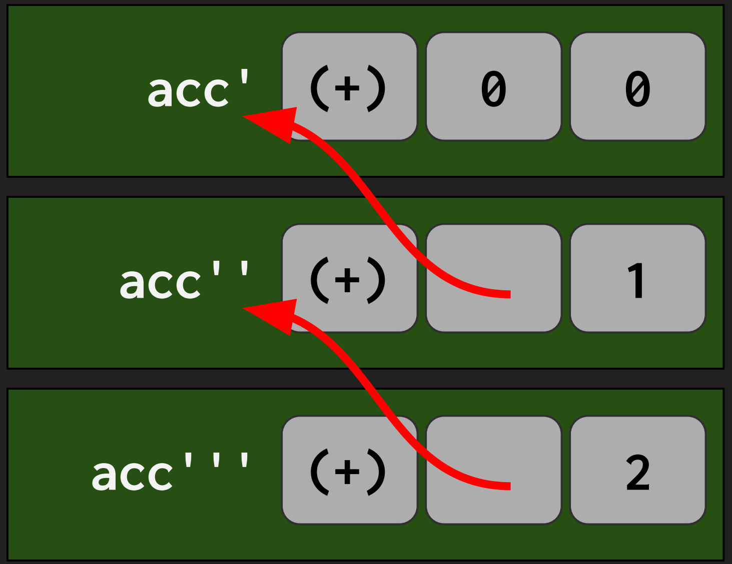
Heap



```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs

```

Heap



Function Applications

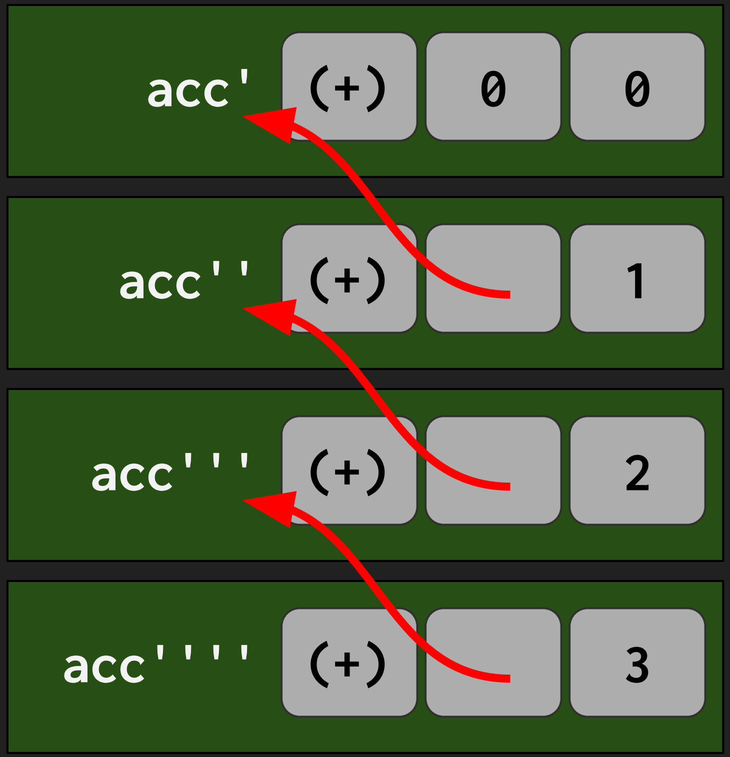
```
my_foldl (+) acc          [0..1000000]
my_foldl (+) (acc + 1)    [1..1000000]
my_foldl (+) (acc' + 1)   [2..1000000]
my_foldl (+) (acc'' + 1)  [3..1000000]
```

```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs
```

Function Applications

```
my_foldl (+) acc              [0..1000000]
my_foldl (+) (acc + 1)        [1..1000000]
my_foldl (+) (acc' + 1)       [2..1000000]
my_foldl (+) (acc'' + 1)      [3..1000000]
my_foldl (+) (acc''' + 1)     [4..1000000]
```

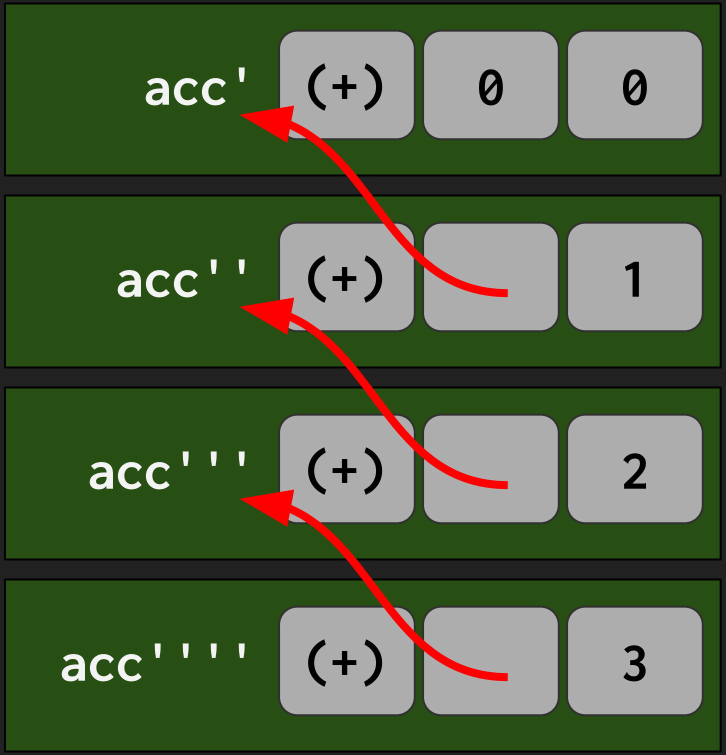
Heap



foldl: Where's the stack overflow?

All the thunks were being allocated on the heap, and foldl is in tail-position, so how can we blow our stack?

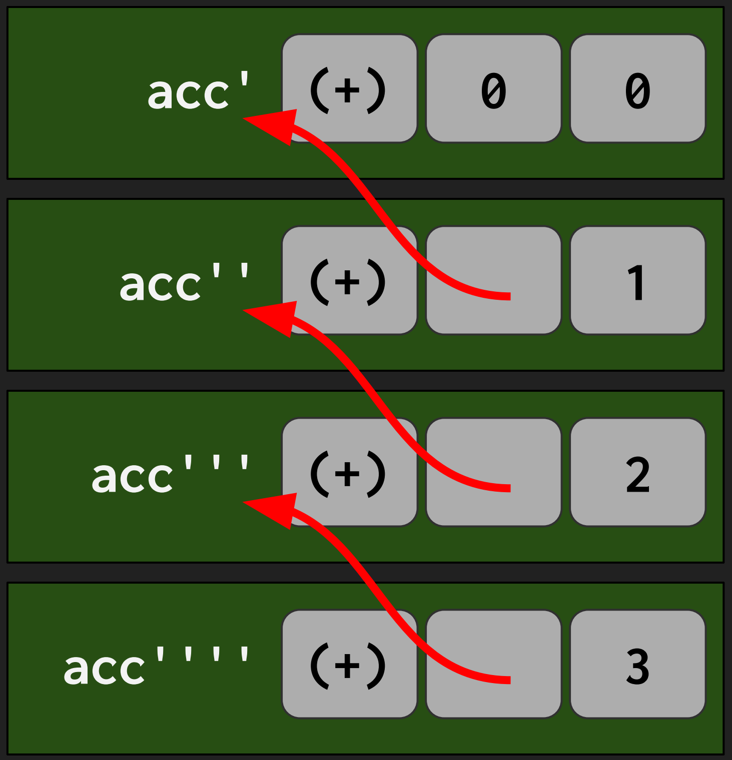
The stack overflow happens when we *evaluate the thunks, which are not in tail-position,* after reaching the base case.



```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs
```

Function Applications

Heap




```haskell
my_foldl (+) ((((acc + 1) + 1) + 1) + 1) [4..1000000]
```

Enforcing strictness in a lazy world

Our problem is that we would like to evaluate the function call on line 4 eagerly.

```
1 my_foldl f acc [] = acc
2 my_foldl f acc (x:xs) =
3     let
4         acc' = f acc x
5     in
6         my_foldl f acc' xs
7
```

Function Applications

```
my_foldl (+) 0 [1..1000000]
my_foldl (+) 1 [2..1000000]
my_foldl (+) 3 [3..1000000]
my_foldl (+) 6 [4..1000000]
my_foldl (+) 10 [5..1000000]
...
```

foldl': an eager implementation of foldl

```haskell
my_foldl f acc [] = acc
my_foldl f acc (x:xs) =
    let
        acc' = f acc x
    in
        my_foldl f acc' xs
```

```haskell
my_foldl' f acc [] = acc
my_foldl' f acc (x:xs) =
    let
        acc' = f acc x
    in
        seq acc' (my_foldl f acc' xs)
```

Introducing strictness with seq

seq's behaviour is equivalent to the definition

```
seq x y = y
```

but with the additional effect that it forces whatever expression is bound by x to be evaluated.

```haskell
my_foldl' f acc [] = acc
my_foldl' f acc (x:xs) =
    let
        acc' = f acc x
    in
        seq acc' (my_foldl f acc' xs)

```

Gotchas with seq (1)

In this example, the intention was to eagerly evaluate (f acc x), but this implementation doesn't bind it to a name, so there's no relationship, as far as seq is concerned, between the two (f acc x) calls.

A compiler's ability to detect that both (f acc x) calls are equivalent is called **common subexpression elimination (CSE)**, but the Haskell compiler can't do it here.

```
my_borked_foldl' f acc [] = acc
my_borked_foldl' f acc (x:xs) =
        seq (f acc x)
            (my_foldl f (f acc x) xs)
```

Gotchas with seq (2)

When is seq evaluated here? When someFunc is lazily called, but that might not happen for some time!

```haskell

uhoh x y = someFunc (seq x y)


```

Aside: semantics of division by zero (TODO: figure out when to talk about total vs partial functions, mvoe this there)

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (/ 1 0)
/: division by zero
>
```

```
Prelude> 1/0
Infinity
```

We said previously that every expression produces a value, but in Racket, division by zero performs some action akin to "raising an exception", whereas Haskell defines a special "infinity" value to produce.