Lazy Evaluation and Folds

Ian-Woo Kim
CERN

June 26, 2014

Plan

- Lazy evaluation and folds: `foldr`, `foldl` and `foldl'`
- Space leak and profiling
- Lazy IO
- Iteratee, Coroutine and Pipe

I am planning to have short presentation on each topic in series.

Outline of today's topic

Lazy evaluation and folds: `foldr`, `foldl` and `foldl'`

- Prelude: loop vs recursion
- Tail-call optimization
- Evaluation strategy: strict vs non-strict $\rightarrow$ lazy
- Lazy evaluation in detail
- `foldl` and `foldr` : tail call vs guarded recursion
- `foldl'`: stricter foldl

Loop vs Recursion

<u>Add 10 to 1</u>

Loop:

```
     s:=0
     i:=10
1:   if (i<=0)
       then return s
       else
         s:=s+i
         i:=i-1
         goto 1
```

Recursion:

```
function sum(i):
  if(i<=0)
    then return 0
    else
      r := sum(i-1)
      return (i+r)

sum(10)
```

Loop vs Recursion

Function calls are expensive! → *prone to have stack overflow*

```
function sum(i):
  if(i<=0)
    then return 0
    else
      r := sum(i-1)
      return (i+r)
```

Address

0000C8

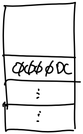
0000D0

0000D8

PC → 0000DC

function sum(i) :
  if (i ≤ 0)
    then return 0
    else
      r := sum (i-1)
      return (i+r)

STACK



put this address in stack
then jump to 0000C8
with (i-1)

Tail-Call Optimization

Rewrite function with tail-call recursion

```
function sum(i):
  return sum'(0,i)


function sum'(s,i):
  if(i<=0)
    then return s
    else
      return sum'(s+i,i-1)

sum(10)
```

Tail-Call Optimization

Compiler replaces function calls with `goto` *by tail-call optimization*

```
function sum(i):
  return sum'(0,i)

function sum'(s,i):
  if(i<=0)
    then return s
    else
      return sum'(s+i,i-1)

sum(10)
```

```
        arg s,i
1:      if (i<=0)
          then return s
          else
            s:=s+i
            i:=i-1
            goto 1
```

Evaluation Strategy

- Strict evaluation: arguments are evaluated before function call (*call-by-value*)
- Non-strict evaluation: strategy that's not strict evaluation.

Evaluation Strategy

- Strict evaluation: arguments are evaluated before function call (*call-by-value*)
- Non-strict evaluation: strategy that's not strict evaluation
  - Lazy evaluation: arguments are evaluated when needed (*call-by-need*) and the evaluated value in scope is shared (*sharing*)

Too ambiguous. Can I define lazy evaluation more precisely?

What is evaluation?

Evaluation is to eliminate REDEX.

- REDEX : REDucible EXpression. Should be directly reducible.
  e.g. `(3+4)`
- λ-expression before substituting variables are not REDEX.
  e.g. $\lambda x.(x + 3)$
- Data constructors are not REDEX.
  e.g. `Date 2014 6 26`
- Pattern match forces evaluation. *e.g.*

```
length lst = case lst of
               [] -> 0
               (x:xs) -> 1 + length xs
```

- *Evaluation strategy is to specify how to choose next REDEX elimination.*

Evaluation strategy in Haskell

Haskell performs the next REDEX elimination by reducing a eval-requested expression to Weak Head Normal Form (WHNF).

A λ-exp $(F\ E_1\ E_2\ \ldots\ E_n)$ $(n \geq 0)$ is in WHNF if and only if F is one of

1. variable
2. data constructor
3. λ-exp or built-in functions and $(F\ E_1\ E_2\ \ldots\ E_m)$ is not a REDEX for any $m \leq n$

- *To say simple, reduce an expression from LHS only up to to RHS from the definition.*
- *Values ($\neq$ data constructor yet) are stored as thunks.*

Fun with laziness examples

Infinite repetition:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

```
repeat 3
```

Fun with laziness examples

Infinite repetition:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

```
repeat 3 = (\x -> x : repeat x) 3
         = 3 : repeat 3              -- infix notation
         = (:) 3 (repeat 3)          -- prefix notation
```

Head is `(:)`, a predefined function
and `(:)`, `(:) 3`, `(:) 3 (repeat 3)` are not REDEXs.

Will not compute after this yet.

Fun with laziness examples

Infinite repetition combined with take function:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

```
take :: Int -> [a] -> [a]
take _ [] = []
take 0 _ = []
take n (x:xs) = x:(take (n-1) xs)
```

```
take 5 (repeat 3)
```

Fun with laziness examples

Infinite repetition combined with `take` function:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

```
take :: Int -> [a] -> [a]
take _ [] = []
take 0 _ = []
take n (x:xs) = x:(take (n-1) xs)
```

```
take 5 (repeat 3)
  = case repeat 3 of ...   -- pattern matching
  = case (3 : repeat 3) of ...
  = 3 : (take (5-1) (repeat 3))
```

reached WHNF. No more computations yet.

Fun with laziness examples

Infinite repetition combined with take function and print:

```
repeat :: a -> [a]
repeat x = x : repeat x
```

```
take :: Int -> [a] -> [a]
take _ [] = []
take 0 _ = []
take n (x:xs) = x:(take (n-1) xs)
```

```
print (take 5 (repeat 3))
  = print (3 : (take (5-1) (repeat 3)))
  -> print 3          -- side effect
print (take (5-1) (repeat 3))
  = print ( case (5-1) of .., case (repeat 3) of ..)
  = print ( take 4 (3 : repeat 3) )
  = print ( 3 : take (4-1) (repeat 3) )
  -> print 3         -- side effect
...
print ( take 0 (repeat 3) )
  = print []
```

Lesson from Laziness

- Call-by-need evaluation does not put program counter on stack!
- Evaluation is done by call graph reduction.
  $\rightarrow$ *STG machine in GHC implementation!*

Tail calls in haskell

- Tail call position in functional programming language: outer-most function call in the body of λ-exp.

  ```
  sum' s 0 = s
  sum' s i = sum' (s+i) (i-1)
  ```

- Most haskell implementations do TCO.

Tail calls in haskell

- Tail call position in functional programming language: top-most function call in the body of λ-exp.

```
sum' s 0 = s
sum' s i = sum' (s+i) (i-1)
```

- Most haskell implementations do TCO.
- How about non-tail-call functions?

```
sum 0 = 0
sum i = i + sum (i-1)
```

Tail calls in haskell

- Tail call position in functional programming language: top-most function call in the body of λ-exp.

```
sum' s 0 = s
sum' s i = sum' (s+i) (i-1)
```

- Most haskell implementations do TCO.
- How about non-tail-call functions?

```
sum 0 = 0
sum i = i + sum (i-1)
```

```
sum 10 = 10 + sum (10-1)
       = 10 + (9 + sum (9-1))
       = 10 + (9 + (8 + sum (8-1))
       = ...
```

- It does not fill $\mathcal{O}(n)$ call stack thanks to laziness, but it makes $\mathcal{O}(n)$ thunks. *Can it be useful?*

Guarded recursion

`(+)` makes non-tail-call `sum` useless.

```
sum 0 = 0
sum i = i + sum (i-1)
```

```
(+) :: Int -> Int -> Int
!x + !y = ...
```

- `(+)` is meaningful only when both arguments are already evaluated.
- *Are there some functions meaningful without knowing the value of arguments, then?*

Guarded recursion

(+) makes non-tail-call `sum` useless.

```
sum 0 = 0
sum i = i + sum (i-1)
```

```
(+) :: Int -> Int -> Int
!x + !y = ...
```

- (+) is meaningful only when both arguments are already evaluated.
- *Are there some functions meaningful without knowing the value of arguments, then?*
  - *Yes, there are! Data constructors!*
    e.g. `x:(y:ys)`

Guarded recursion

Replace `(+)` by `(:)` in `sum` (rename to `mklist`):

```
mklist :: Int -> [Int]
mklist 0 = []
mklist i = i : mklist (i-1)
```

```
mklist 10 = 10 : mklist (10-1)
```

- 10 is ready for use without further evaluating `mklist (10-1)`
- It's a generator for a data structure.
- *Recursion inside Data Constructor is called guarded recursion*.

Guarded recursion

Replace `(+)` by `(:)` in `sum` (rename to `mklist`):

```
mklist :: Int -> [Int]
mklist 0 = []
mklist n = i : mklist (i-1)
```

```
mklist 10 = 10 : mklist (10-1)
```

- 10 is ready for use without further evaluating `mklist (10-1)`
- It's a generator for a data structure.
- *Recursion inside Data Constructor is called guarded recursion.*

We are haskellers! Are you ready for abstraction?

foldl and foldr:

```
foldl :: (b -> a -> a) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

foldr :: (a -> b -> a) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

- `foldl`: `foldl` is at the outer-most position, *i.e.* tail-call poisition.
- `foldr`: `f` is at the outer-most position. non-tail-call function

Sum in terms of `foldl`

```
foldl :: (b -> a -> a) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

sum :: [Int] -> Int
sum = foldl (+) 0
```

```
sum [1..10]
```

Sum in terms of `foldl`

```
foldl :: (b -> a -> a) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs

sum :: [Int] -> Int
sum = foldl (+) 0
```

```
sum [1..10] = foldl (+) 0 [1..10]
            = foldl (+) (0+1) [2..10]   -- pattern match
            = foldl (+) ((0+1)+2) [3..10]
            = foldl (+) (((0+1)+2)+3) [4..10]
            ...
```

- In fact, `((((0+1)+2)+3)+...)` builds up $\mathcal{O}(n)$ thunks.
- $(0+1)$ should be evaluated before `foldl` finds the next WHNF reduction.

foldl'

```
foldl' :: (b -> a -> a) -> b -> [a] -> b
foldl' f !acc [] = acc
foldl' f !acc (x:xs) = foldl' f (f acc x) xs

sum :: [Int] -> Int
sum = foldl' (+) 0
```

- Strictness annotation : evaluate arguments with bang on the LHS up to WHNF before proceeding the RHS

foldl’

```
foldl’ :: (b -> a -> a) -> b -> [a] -> b
foldl’ f !acc [] = acc
foldl’ f !acc (x:xs) = foldl’ f (f acc x) xs

sum :: [Int] -> Int
sum = foldl’ (+) 0
```

```
sum [1..10] = foldl’ (+) 0 [1..10]
            = foldl’ (+) (0+1) [2..10]
            = foldl’ (+) 1 [2..10]
            = foldl’ (+) (1+2) [3..10]
            = foldl’ (+) 3 [3..10]
            ...
```

Map as `foldr`

```
foldr :: (a -> b -> a) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)

map :: (a -> b) -> [a] -> [b]
map f = foldr (\x acc -> f x : acc) []
```

```
map (+1) [1..10]
```

Map as `foldr`

```
foldr :: (a -> b -> a) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)

map :: (a -> b) -> [a] -> [b]
map f = foldr (\x acc -> f x : acc) []
```

```
map (+1) [1..10]
  = foldr (\x acc -> (x+1) : acc) [] [1..10]
  = (1+1) : foldr (\x acc -> (x+1) : acc) [] [2..10]
```

- Evaluation ends here for the above exp, but 1+1 is ready for use.
- Next elements are evaluated as needed if this exp is consumed by other function.
  e.g. `take 3 (map (+1) [1..10])`

Exercise

```
print (take 3 (map (+1) [1..10]))
```

Exercise

```
print (take 3 (map (+1) [1..10]))
  = print (take 3 (foldr (\x a->(x+1):a) [] [1..10]))
    -- expanded to pattern-match
  = print (take 3 ((1+1):foldr (\x a->(x+1):a) [] [2..10])
  = print (1+1):(take 2 (foldr (\x a->(x+1):a) [] [2..10]))
-> print 2
print (take 2 (foldr (\x a->(x+1):a) [] [2..10]))
  = print (take 3 ((2+1):foldr (\x a->(x+1):a) [] [3..10])
  = print (2+1):(take 2 (foldr (\x a->(x+1):a) [] [3..10])
-> print 3
...
```

- Mapped values are delivered lazily to `print` one by one as needed.

Exercise

```
print (take 3 (map (+1) [1..10]))
```

- Can we implement this in terms of `foldl` or `foldl'`? Why not possible?

Exercise

```
print (take 3 (map (+1) [1..10]))
```

- Can we implement this in terms of `foldl` or `foldl'`? Why not possible?
- *We cannot take off `foldl` before finishing the iteration!*

```
foldl' :: (b -> a -> a) -> b -> [a] -> b
foldl' f !acc [] = acc
foldl' f !acc (x:xs) = foldl' f (f acc x) xs

foldr :: (a -> b -> a) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

- Guarded recursion is only for `foldr`.

Conclusion

- Lesson: `foldl'` is a tail call recursion, and `foldr` is a guarded recursion combined with data constructor.
- `foldl'` for collapsing a structure to a value.
- `foldr` for a constructing structure.
- This is in general correspondent to Map-Reduce pattern.
- Often, a problem is reduced to `(foldl'.foldr)`. It is called *hylomorphism* : build-fold pattern

Time for break!
Thank you!