Function evaluation in Haskell

Content is based primarily on
Tom Ellis's talk at Haskell eXchange 2016

Contents of the talk

- Rules of evaluation
- Walkthrough of the evaluation of a simple expression
- Two other examples with `foldl` and `foldl’`

Weak head normal form

- Literal
- Variable
- Constructor: fully saturated
- Let expression
- Lambda
- Function application: function itself and arguments must be variables or literals

Weak head normal form example

```haskell
-- Haskell
map f []     = []
map f (x:xs) = f x : map f xs


-- There must be a single definition
map f xs = case xs of
    []     → []
    x:xs’ → f x : map f xs’


-- We have to make it a lambda expression
map = \f xs → case xs of
    []     → []
    x:xs’ → f x : map f xs’


-- Constructors must be saturated
map = \f xs → case xs of
    []     → []
    x:xs’ → let first = f x
                 rest  = map f xs’
             in first : rest
```

Evaluation outcome

Result of an evaluation is a *value*, which is either

- a literal,
- a fully saturated constructor, or
- a lambda.

(It’s the practical version of *weak head normal form*.)

Evaluation rules

- Literals, lambdas, constructors: already evaluated.

- `let x = e in body`: create a closure for `e` on the heap and let `x` be a pointer to this closure.

- variable `x` (where `x` is a pointer to a closure): evaluate the closure and replace the closure on the heap with the resulting value. (Memoization happens automatically.)

- `f a`: evaluate `f` to a lambda, then substitute `a` in the body and evaluate the result.

- `case e of alternatives`: push the whole expression on the stack, evaluate the scrutinee `e`, check which alternative matches, and evaluate that one.

Evaluation: memory usage

- Literals, lambdas, constructors: already evaluated.

- **`let x = e in body`: create a closure for `e` on the heap and let `x` be a pointer to this closure.**

- variable `x` (where `x` is a pointer to a closure): evaluate the closure and replace it with the resulting value. (Memoization happens automatically.)

- `f a`: evaluate `f` to a lambda, then substitute `a` in the body and evaluate the result.

- **`case e of alternatives`: push the whole expression on the stack, evaluate `e`, check which alternative matches and evaluate it.**

Evaluation example

```
map f []     = []
map f (x:xs) = f x : map f xs


repeat x = xs where xs = x : xs

head (x:xs) = x
```

Evaluation example

```
map f []     = []
map f (x:xs) = f x : map f xs


repeat x = xs where xs = x : xs

head (x:xs) = x



-- Normal form


map = \f xs → case xs of
    []    → []
    x:xs’ → let first = f x
                rest  = map f xs’
            in first : rest

repeat = \x → let xs = x : xs
                in xs

head = \xs → case xs of x:xs’ → x
```

Example: evaluate an expression

```
-- Haskell
head (map (\x → x + x) (repeat (10 + 1)))
```

Example: evaluate an expression

```haskell
-- Haskell
head (map (\x → x + x) (repeat (10 + 1)))


-- Normal form
let f = \x → x + x
    t = 10 + 1
    r = repeat t
    m = map f r
in head m
```

Evaluating:

```
let f = \x → x + x
    t = 10 + 1
    r = repeat t
    m = map f r
in head m
```

Stack

Heap

```
head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
head m
```

Stack

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs'  → let first = f x
                 rest  = map f xs'
             in first : rest
```

Evaluating:

```
case m of x:xs' → x
```

Stack

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
m
```

Stack

```
case m of x:xs' → x
```

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs'  → let first = f x
                 rest  = map f xs'
             in first : rest
```

Evaluating:

```
m = map f r
```

Stack

```
case m of x:xs’ → x
```

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
m = case r of
      []     → []
      x:xs’  → let first = f x
                   rest  = map f xs’
               in first : rest
```

Stack

```
case m of x:xs’ → x
```

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
r
```

Stack

```
case m of x:xs’ → x

m = case r of
      []     → []
      x:xs’  → let first = f x
                   rest  = map f xs’
               in first : rest
```

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
r = repeat t
```

Stack

```
case m of x:xs’ → x

m = case r of
      []    → []
      x:xs’ → let first = f x
                  rest  = map f xs’
              in first : rest
```

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
r = let xs = t : xs
    in xs
```

Stack

```
case m of x:xs’ → x

m = case r of
     []    → []
     x:xs’ → let first = f x
                 rest  = map f xs’
             in first : rest
```

Heap

```
f = \x → x + x
t = 10 + 1
r = repeat t
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
r = xs
```

Heap

```
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Stack

```
case m of x:xs’ → x

m = case r of
     []    → []
     x:xs’ → let first = f x
                 rest  = map f xs’
             in first : rest
```

Evaluating:

```
r = t : xs
```

Stack

```
case m of x:xs’ → x

m = case r of
     []     → []
     x:xs’  → let first = f x
                  rest  = map f xs’
              in first : rest
```

Heap

```
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []     → []
    x:xs'  → let first = f x
                 rest  = map f xs'
             in first : rest
```

Evaluating:

```
m = case r of
      []     → []
      x:xs’  → let first = f x
                   rest  = map f xs’
               in first : rest
```

Stack

```
case m of x:xs’ → x
```

Heap

```
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
m = let first = f t
        rest  = map f xs
    in first : rest
```

Stack

```
case m of x:xs' → x
```

Heap

```
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
m = first : rest
```

Stack

```
case m of x:xs’ → x
```

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = map f r

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
m = first : rest
-- This is a fully evaluated
—   constructor
```

Stack

```
case m of x:xs’ → x
```

Heap

```
rest   = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to  xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
first = f t
```

Stack

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
first = f t
-- Substitute t into the
--  body of f
```

Stack

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
first = t + t
```

Stack

Heap

```
rest   = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to  xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Primitive addition

```
-- (+) is strict in both arguments,
--   and calls the primitive addition op.
(+) = \x y → case x of
    x' → case y of
        y' → primitive_plus x' y'
```

Primitive addition

```
-- (+) is strict in both arguments,
--   and calls the primitive addition op.
(+) = \x y → case x of
    x' → case y of
        y' → primitive_plus x' y'


— or, in more detail, showing the boxing/unboxing
(+) = \x y → case x of
    Int# x' → case y of
        Int# y' → case +# x' y' of
            v → Int# v
```

Evaluating:

```
t
```

Stack

```
first = t + t
```

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
t = 10 + 1
```

Stack

```
first = t + t
```

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 10 + 1
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
t = 11
```

Stack

```
first = t + t
```

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 11
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []    → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluating:

```
first = t + t
```

Stack

Heap

```
rest  = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 11
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                 rest  = map f xs'
             in first : rest
```

Evaluating:

```
first = 11 + 11
-- Again: although t was used
--    twice, we evaluated it
--    only once.
```

Stack

Heap

```
rest   = map f xs
first = f t
xs = t : xs
f = \x → x + x
t = 11
r = pointer to  xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
               in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                 rest  = map f xs'
             in first : rest
```

Evaluating:

```
22
```

Stack

```
Evaluation stack is empty

 ⇒ We're finished evaluating!
```

Heap

```
rest   = map f xs
first = 22
xs = t : xs
f = \x → x + x
t = 11
r = pointer to xs
m = first : rest

head = \xs → case xs of x:xs' → x

repeat = \x → let xs = x : xs
              in xs

map = \f xs → case xs of
    []     → []
    x:xs' → let first = f x
                rest  = map f xs'
            in first : rest
```

Evaluation: foldl

```haskell
-- This is not exactly the one in the standard Haskell library, but not too different from it.
foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'

-- Evaluate this:
foldl (+) 0 [1..100]
```

Evaluation: foldl

```
-- This is not exactly the one in the standard Haskell library, but not too
different from it.
foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'

-- Evaluate this:
foldl (+) 0 [1..100]
```

HaskellWiki says:

```
foldl (+) 0 [1,2,3]
⟹ foldl (+) (0 + 1) [2,3]
⟹ foldl (+) ((0 + 1) + 2) [3]
⟹ foldl (+) (((0 + 1) + 2) + 3) []
⟹ ((0 + 1) + 2) + 3
⟹ (1 + 2) + 3
⟹ 3 + 3
⟹ 6
```

But what does it mean in practice?

Evaluating:

```
foldl (+) 0 [1..100]
```

Stack

Heap

```
foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
case [1..100] of
    []     → 0
    x:xs'  → let z' = (+) 0 x
             in foldl (+) z' xs'
```

Stack

Heap

```
foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
let z' = (+) 0 1
  in foldl (+) z' [2..100]
```

Stack

Heap

```
foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
foldl (+) z’1 [2..100]
```

Stack

Heap

```
z’1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
case [2..100] of
    []     → 2
    x:xs'  → let z' = (+) z'1 x
             in foldl (+) z' xs'
```

Stack

Heap

```
z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
let z' = (+) z'1 2
  in foldl (+) z' [3..100]
```

Stack

Heap

```
z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
foldl (+) z'2 [3..100]
```

Stack

Heap

```
z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
foldl (+) z’3 [4..100]
```

Stack

Heap

```
z'3 = (+) z’2 3

z'2 = (+) z’1 2

z’1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
foldl (+) z’4 [5..100]
```

Stack

Heap

```
z'4 = (+) z’3 4

z'3 = (+) z’2 3

z'2 = (+) z’1 2

z’1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

100 steps later…

Evaluating:

```
foldl (+) z’99 [100]
```

Stack

Heap

```
z'99 = (+) z’98 99

...

z'4 = (+) z’3 4

z'3 = (+) z’2 3

z'2 = (+) z’1 2

z’1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
foldl (+) z'100 []
```

Stack

Heap

```
-- foldl builds up a long chain of thunks

z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
case [] of
    []    → z’100
    x:xs' → let z' = f z’100 x
            in foldl f z' xs'
```

Stack

Heap

```
z'100 = (+) z’99 100

z'99 = (+) z’98 99

...

z'4 = (+) z’3 4

z'3 = (+) z’2 3

z'2 = (+) z’1 2

z’1 = (+) 0 1

foldl = \f z xs → case xs of
    []    → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
z'100
```

Stack

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
(+) z'99 100
```

Stack

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
z'99
```

Stack

```
(+) z'99 100
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
(+) z'98 99
```

Stack

```
(+) z'99 100
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
z'98
```

Stack

```
(+) z'99 100

(+) z'98 99
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

100 steps later…

Evaluating:

```
z'2
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
(+) z'1 2
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
z'1
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3

(+) z'1 2
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []    → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
(+) 0 1
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3

(+) z'1 2
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = (+) 0 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
1
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3

(+) z'1 2
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
(+) z'1 2
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
            in foldl f z' xs'
```

Evaluating:

```
(+) 1 2
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = (+) z'1 2

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
3
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = 3

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
3
```

Stack

```
(+) z'99 100

(+) z'98 99

...

(+) z'2 3
```

Heap

```
z'100 = (+) z'99 100

z'99 = (+) z'98 99

...

z'4 = (+) z'3 4

z'3 = (+) z'2 3

z'2 = 3

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs' → let z' = f z x
              in foldl f z' xs'
```

100 steps later…

Evaluating:

```
z'99 = 4950
```

Stack

```
(+) z'99 100
```

Heap

```
z'100 = (+) z'99 100

z'99 = 4950

...

z'4 = 10

z'3 = 6

z'2 = 3

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
(+) z'99 100
```

Stack

Heap

```
z'100 = (+) z'99 100

z'99 = 4950

...

z'4 = 10

z'3 = 6

z'2 = 3

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
(+) 4950 100
```

Stack

Heap

```
z'100 = (+) z'99 100

z'99 = 4950

...

z'4 = 10

z'3 = 6

z'2 = 3

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluating:

```
5050
-- Evaluation done, and it only needed 300
steps and 200 units of memory. Yay!
```

Stack

Heap

```
z'100 = (+) z'99 100

z'99 = 4950

...

z'4 = 10

z'3 = 6

z'2 = 3

z'1 = 1

foldl = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in foldl f z' xs'
```

Evaluation with foldl’

```
-- Evaluate this:
foldl' (+) 0 [1..100]


-- With the strict version of foldl
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluation with foldl’

```haskell
-- Evaluate this:
foldl' (+) 0 [1..100]


-- With the strict version of foldl
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'


-- Could have written it with `seq`:
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → let z' = f z x
             in z' `seq` foldl' f z' xs'
```

Evaluating:

```
foldl' (+) 0 [1..100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
case [1..100] of
    []     → 0
    x:xs'  → case (+) 0 x of
        z'  → foldl' (+) z’ xs'
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z'  → foldl' f z' xs'
```

Evaluating:

```
case (+) 0 1 of
    z' → foldl' (+) z' [2..100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
(+) 0 1
```

Stack

```
case (+) 0 1 of
    z' → foldl' (+) z' [2..100]
```

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
1
```

Stack

```
case (+) 0 1 of
    z' → foldl' (+) z' [2..100]
```

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
case 1 of
    z' → foldl' (+) z' [2..100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
foldl' (+) 1 [2..100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
case [2..100] of
    []     → 1
    x:xs' → case (+) 1 x of
        z' → foldl' (+) z' xs'
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
case (+) 1 2 of
    z' → foldl' (+) z' [3‥100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
(+) 1 2
```

Stack

```
case (+) 1 2 of
    z' → foldl' (+) z' [3..100]
```

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
case 3 of
    z' → foldl' (+) z' [3..100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
foldl' (+) 3 [3..100]
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

100 steps later…

Evaluating:

```
foldl' (+) 5050 []
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
case [] of
    []     → 5050
    x:xs'  → case (+) 5050 x of
        z'  → foldl' (+) z' xs'
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z'  → foldl' f z' xs'
```

Evaluating:

```
5050
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs' → case f z x of
        z' → foldl' f z' xs'
```

Evaluating:

```
5050
-- And we're done!
```

Stack

Heap

```
foldl' = \f z xs → case xs of
    []     → z
    x:xs'  → case f z x of
        z' → foldl' f z' xs'
```

Homework

- Simulate `foldr`, together with the stack and the heap

- Play around with Quchen's STG implementation, which turns Haskell code into STG language. https://github.com/quchen/stgi

```
7. Function application
  - Inspect value xs
  - Unused local variables discarded: acc (0x04), f (0x01)
--------------------------------------------------------------------------------
Code: Enter 0x06
Stack
    2. Ret Alts:   Nil -> acc;
                   Cons y ys -> case f acc y of
                       acc' -> foldl' f acc' ys;
                   badList -> Error_foldl' badList
           Locals: acc -> 0x04
                   f -> 0x01
                   xs -> 0x06
    1. Upd 0x03
Heap (7 entries)
    0x00 -> Fun \x y -> case x of
                Int# x' -> case y of
                    Int# y' -> case +# x' y' of
                        v -> Int# v;
                    err -> Error_add_1 err;
                err -> Error_add_2 err
    0x01 -> Fun \x y -> y
    0x02 -> Fun \f acc xs -> case xs of
                Nil -> acc;
                Cons y ys -> case f acc y of
                    acc' -> foldl' f acc' ys;
                badList -> Error_foldl' badList
    0x03 -> Blackhole (from step 1)
    0x04 -> Con \ -> Int# 0#
    0x05 -> Fun \f xs ys -> case xs of
                Nil -> Nil;
                Cons x xs' -> case ys of
                    Nil -> Nil;
                    Cons y ys' ->
                        let fxy = \(f x y) => f x y;
                            rest = \(f xs' ys') => zipWith f xs' ys'
                        in Cons fxy rest;
                    badList -> Error_zipWith badList;
                badList -> Error_zipWith badList
    0x06 -> Thunk \ =>
                letrec fib0 = \(fib1) -> Cons zero fib1;
                       fib1 = \(fib2) =>
                           let one = \ -> Int# 1#
                           in Cons one fib2;
                       fib2 = \(fib0 fib1) => zipWith add fib0 fib1
                in fib0
Globals
    add -> 0x00
    flipConst -> 0x01
    foldl' -> 0x02
    main -> 0x03
    zero -> 0x04
    zipWith -> 0x05
Step: 7
```

Resources

- Tom Ellis: *Haskell programs: How do they run?* https://skillsmatter.com/skillscasts/8726-haskell-programs-how-do-they-run
- David Luposchainsky: Functional and low-level: watching the STG execute https://skillsmatter.com/skillscasts/8800-functional-and-low-level-watching-the-stg-execute#video
- HaskellWiki: Let vs. Where. https://wiki.haskell.org/Let_vs._Where
- HaskellWiki: Thunks. https://wiki.haskell.org/Thunk
- WHNF vs. NF. https://stackoverflow.com/a/6889335/8424390