PURELY AGILE

Haskell programs: how do they run?

Demystifying lazy evaluation

Tom Ellis

6th October 2016

PURELY AGILE

Mysterious things

that you may have heard about lazy evaluation

- graph reduction
- redex
- weak head normal form (WHNF)
- constant applicative form (CAF)
- “delays the evaluation of an expression until its value is needed”
- call by need

Normal form

- literals
- variables
- constructors
- let
- lambda
- function application
- case

- function application: function itself and arguments must be variables or literals
- constructors: must be “saturated”, i.e. no missing arguments.

It’s a language with a direct operational (or imperative!) reading.

PURELY AGILE

Normal form

Example

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

-- Normal form
map = \f xs -> case xs of
  []    -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest
```

Evaluation

What do we evaluate to?

The result of an evaluation is a "value". A value is

- a (fully saturated) constructor (including primitive types), or
- a lambda

Evaluation

How do we evaluate?

- 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, i.e. all mentions of `x` scoped by this binding point to that closure.
- variables `x`: `x` is a pointer to a closure. Evaluate that closure to a value and overwrite its memory location with the value (“memoization”).
- `f a`: evaluate `f` to `\x -> e`, substitute `a` and evaluate the result
- `case e of alts`: evaluate `e`, check which alternative matches and evaluate it

Evaluation

Heap and stack

- The only thing that allocates on the heap is `let`.
- The only thing that consumes stack (that we care about) is `case` whilst it is evaluating its scrutinee.

```
case scrutinee of ... -> ...
```

Evaluation

Evaluation example

```
repeat x = xs where xs = x : xs

repeat = \x -> let xs = x : xs
               in xs

head (x:xs) = x

head = \xs -> case xs of x:xs’ -> x
```

Evaluation

Evaluation example

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

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

Evaluation

Evaluation example

```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
```

PURELY AGILE

Evaluation

Evaluation example

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

-- Heap
map = ...
repeat = ...
head = ...
```

Evaluation

Evaluation example

```
-- Evaluating
head m

-- Heap
map = ...
repeat = ...
head = \xs -> case xs of x:xs' -> x
f = \x -> x + x
t = 10 + 1
r = repeat t
m = map f r
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x

-- Heap
map = ...
repeat = ...
head = ...
f = \x -> x + x
t = 10 + 1
r = repeat t
m = map f r
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m

-- Heap
map = ...
repeat = ...
head = ...
f = \x -> x + x
t = 10 + 1
r = repeat t
m = map f r
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = map f r

-- Heap
map = ...
repeat = ...
head = ...
f = \x -> x + x
t = 10 + 1
r = repeat t
m = map f r
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = case r of
  []     -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest

-- Heap
r = repeat t
...
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = case r of
  []     -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest
|
r

-- Heap
r = repeat t
...
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = case r of
  []     -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest
|
r = repeat t

-- Heap
r = repeat t
...
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = case r of
  []     -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest
|
r = let xs = t : xs
    in xs

-- Heap
r = repeat t
...
```

PURELY AGILE

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = case r of
  []     -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest
|
r = xs

-- Heap
xs = t : xs
r = repeat t
...
```

PURELY AGILE

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = case r of
  []     -> []
  x:xs' -> let first = f x
               rest  = map f xs'
           in first : rest
|
r = xs

-- Heap
xs = t : xs
r ---^
...
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = let first = f t
        rest  = map f xs
    in first : rest

-- Heap
xs = t : xs
r ---^
...
```

Evaluation

Evaluation example

```
-- Evaluating
case m of x:xs' -> x
|
m = first : rest

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
m = first : rest
...
```

Evaluation

Evaluation example

```
-- Evaluating
first

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
t = 10 + 1
f = \x -> x + x
...
```

PURELY AGILE

Evaluation

Evaluation example

```
-- Evaluating
first = f t

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
t = 10 + 1
f = \x -> x + x
...
```

PURELY AGILE

Evaluation

Evaluation example

```
-- Evaluating
first = t + t

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
t = 10 + 1
f = \x -> x + x
...
```

PURELY AGILE

Evaluation

Primitive addition

```
-- (+) evaluates its arguments and
--  calls a primitive operation
(+) = \x y -> case x of
     x' -> case y of
          y' -> primitive_plus x' y'
```

PURELY AGILE

Evaluation

Evaluation example

```
-- Evaluating
first = t + t
|
t = 10 + 1

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
t = 10 + 1
...
```

Evaluation

Evaluation example

```
-- Evaluating
first = t + t
|
t = 11

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
t = 11
...
```

Evaluation

Evaluation example

```
-- Evaluating
first = 11 + 11

-- Heap
xs = t : xs
r ---^
first = f t
rest  = map f xs
t = 11
...
```

Evaluation

Evaluation example

```
-- Evaluating
first = 22

-- Heap
xs = t : xs
r ---^
first = 22
rest  = map f xs
t = 11
...
```

Evaluation

Evaluation example

```
-- Finished evaluating!
22

-- Heap
xs = t : xs
r ---^
first = 22
rest  = map f xs
t = 11
...
```

PURELY AGILE

Sharing

Eta reduction

These evaluate the same way.

```
plus1 x = (+) 1 x

plus1'  = (+) 1
```

PURELY AGILE

Sharing

These don't evaluate the same way.

```
enum1 = zip ns
    where ns = [1..]
```

```
enum2 xs = zip ns xs
    where ns = [1..]
```

PURELY AGILE

Sharing

These don't evaluate the same way.

```
enum1 = zip ns
    where ns = [1..]

let enum1 = let ns = [1..]
            in zip ns
```

```
enum2 xs = zip ns xs
    where ns = [1..]

let enum2 = \xs -> let ns = [1..]
                   in zip ns xs
```

Sharing

These don't evaluate the same way.

```
enum1 = zip ns
    where ns = [1..]

let enum1 = let ns = [1..]
            in zip ns
-- ns is shared by all invocations of enum1

enum2 xs = zip ns xs
    where ns = [1..]

let enum2 = \xs -> let ns = [1..]
                   in zip ns xs
-- ns is created afresh by each invocations of enum2
```

foldl

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


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

foldl

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

PURELY AGILE

foldl

```
-- Evaluating
foldl (+) z'1 xs'1

-- Heap
xs'1 = [2..100]
z'1 = (+) 0 1
```

PURELY AGILE

foldl

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

-- Heap
xs'1 = [2..100]
z'1 = (+) 0 1
```

foldl

```
-- Evaluating
foldl (+) z'2 xs'2

-- Heap
xs'2 = [3..100]
z'1 = (+) 0 1
z'2 = (+) z'1 2
```

foldl

```
-- Evaluating
foldl (+) z'3 xs'3

-- Heap
xs'3 = [4..100]
z'1 = (+) 0 1
z'2 = (+) z'1 2
z'3 = (+) z'2 3
```

foldl

```
-- Evaluating
foldl (+) z'4 xs'4

-- Heap
xs'4 = [5..100]
z'1 = (+) 0 1
z'2 = (+) z'1 2
z'3 = (+) z'2 3
z'4 = (+) z'3 4
```

foldl

```
-- Evaluating
foldl (+) z'100 xs'100

-- Heap
xs'100 = []
z'1 = (+) 0 1
z'2 = (+) z'1 2
z'3 = (+) z'2 3
z'4 = (+) z'3 4
...
z'100 = (+) z'99 100
```

foldl

```
-- Evaluating
z'100

-- Heap
xs'100 = []
z'1 = (+) 0 1
z'2 = (+) z'1 2
z'3 = (+) z'2 3
z'4 = (+) z'3 4
...
z'100 = (+) z'99 100
-- ``Building up a long chain of thunks''
```

foldl

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

-- Heap
xs'100 = []
z'1 = (+) 0 1
z'2 = (+) z'1 2
z'3 = (+) z'2 3
z'4 = (+) z'3 4
...
z'100 = (+) z'99 100
-- ''Building up a long chain of thunks''
```

foldl

```
-- Evaluating
(+) z'99 100
|
z'99
```

foldl

```
-- Evaluating
(+) z'99 100
|
(+) z'98 99
```

PURELY AGILE

foldl

```
-- Evaluating
(+) z'99 100
|
(+) z'98 99
|
z'98
```

PURELY AGILE

foldl

```
-- Evaluating
(+) z'99 100
|
(+) z'98 99
|
(+) z'97 98
|
...
(+) z'1 2
|
(+) 0 1
```

foldl

```
-- Evaluating
(+) z'99 100
|
(+) z'98 99
|
(+) z'97 98
|
...
(+) z'1 2
|
1
```

foldl

```
-- Evaluating
(+) z'99 100
|
(+) z'98 99
|
(+) z'97 98
|
...
3
-- Finally we unwind the stack
```

foldl'

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


foldl' = \f z xs -> case xs of
  []     -> z
  -- In Haskell we would use seq instead of case
  x:xs' -> case f z x of
      z' -> foldl' f z' xs'


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

foldl’

```
-- Evaluating
case [1..100] of
  []     -> 0
  x:xs' -> case (+) 0 x of
      z' -> foldl' (+) z' xs'
```

foldl’

```
case (+) 0 1 of
     z’ -> foldl’ (+) z’ xs’1

-- Heap
xs’1 = [2..100]
```

foldl’

```
case (+) 0 1 of
      z’ -> foldl’ (+) z’ xs’1
|
(+) 0 1

-- Heap
xs’1 = [2..100]
```

foldl’

```
case (+) 0 1 of
      z’ -> foldl’ (+) z’ xs’1
|
1

-- Heap
xs’1 = [2..100]
```

PURELY AGILE

foldl’

```
foldl' (+) 1 xs'1

-- Heap
xs'1 = [2..100]
```

PURELY AGILE

foldl’

```
foldl' (+) 3 xs'2

-- Heap
xs'2 = [3..100]
```

foldl’

```
foldl’ (+) 6 xs’3
-- Evaluation will proceed in constant space

-- Heap
xs’3 = [4..100]
```

foldr

```haskell
foldr = \f z xs -> case xs of
  []     -> z
  x:xs' -> let rest = foldr (+) 0 xs'
              in f x rest

-- Evaluate
foldr (+) 0 [1..100]
```

foldr

```haskell
-- Evaluate
case [1..100] of
  []     -> z
  x:xs' -> let rest = foldr (+) 0 xs'
           in (+) x rest
```

foldr

```
-- Evaluate
(+) 1 rest1

-- Heap
rest1 = foldr (+) 0 xs'1
xs'1 = [2..100]
```

foldr

```
-- Evaluate
(+) 1 rest1
|
rest1

-- Heap
rest1 = foldr (+) 0 xs'1
xs'1 = [2..100]
```

foldr

```
-- Evaluate
(+) 1 rest1
|
rest1 = foldr (+) 0 xs'1

-- Heap
rest1 = foldr (+) 0 xs'1
xs'1 = [2..100]
```

foldr

```
-- Evaluate
(+) 1 rest1
|
rest1 = case xs'1 of
  []    -> 0
  x:xs' -> let rest = foldr (+) 0 xs'
           in (+) x rest

-- Heap
rest1 = foldr (+) 0 xs'1
xs'1 = [2..100]
```

foldr

```
-- Evaluate
(+) 1 rest1
|
rest1 = (+) 2 rest2

-- Heap
rest1 = foldr (+) 0 xs'1
rest2 = foldr (+) 0 xs'2
xs'2 = [3..100]
```

foldr

```
-- Evaluate
(+) 1 rest1
|
rest1 = (+) 2 rest2
|
rest2 = (+) 3 rest3
|
rest3 = (+) 4 rest4
|
...
|
rest99 = (+) 100 rest100
|
rest100 = 0

-- Heap -- so big it won't fit on the slide
...
```

Conclusion

All Haskell programs can be translated straightforwardly to a simple normal form which has a simple imperative-style interpretation.

By following through the execution of the program we can understand how it uses memory resources.