PURELY AGILE

Haskell programs: how do they run?

Demystifying lazy evaluation

Tom Ellis

6th October 2016

PURELY AGILE

A nice frame

Subtitle

- 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

I hope you will be able to work out everything else from first principles. Not everything is literally correct but it will give you the right understanding for everything except the lowest level performance hacking.

PURELY AGILE

Normal form

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

In a function application, the function itself and the arguments must be variables or literals.
Constructors must be “saturated”, i.e. no missing arguments.

Normal form

Example

```haskell
-- Haskell
let x = 1 + 1
    f y = plus y (3 * y)
      where plus = (+)
in f x

-- Normal form
let x = 1 + 1
    f = \y -> let plus = (+)
                  a1 = 3 * y
              in plus y a1
in f x
```

PURELY AGILE

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 – 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").
- constructors – already evaluated
- lambda – already evaluated
- `f a`: evaluate `f` to `\x -> e`
- `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.

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

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

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

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

```haskell
-- 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

```haskell
-- 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
...
```

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
...
```

Evaluation

Evaluation example

```
-- Evaluating
first = f t

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

PURELY AGILE

Evaluation

Evaluation example

```
-- Evaluating
first = t + t

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

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
...
```

PURELY AGILE

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
...
```