Fun With Haskell: The Benefits of Being Lazy

Nathaniel Wesley Filardo

January 11, 2012

What does it mean to be lazy?
This is your instructor being lazy.

Any questions?

What does it mean to be lazy?

- Consider this Java function call:

```
System.out.println("foo"+"bar");
```

- What happens?
- I mean, what *really* happens?

What does it mean to be lazy?

- How about this?

```
public int explode() {
throw new RuntimeException("Kablooie");
}
/* ... */
System.out.println(explode());
```

- What happens?
- I mean, what *really* happens?
- Is `System.out.println` in the stack trace?
- Important: exceptions let us see evaluation order.

What does it mean to be lazy?

- **Eager** (**strict**, **call-by-value**) languages evaluate arguments first.
  - That is, to call a function, the *caller* evaluates all the arguments (in some order) *and then* calls the function with the *results*.
- **Non-strict** languages *leave it to the callee* to evaluate their arguments, *if they need them*.
  - Which means they *might not*.

What does it mean to be lazy?

- Consider a Haskell function like

```
cube x = x * x * x
```

- Suppose I call `cube (10+3)`.
- How many additions will the system perform?

What does it mean to be lazy?

- Consider a Haskell function like

```
cube x = x * x * x
```

- Suppose I call `cube (10+3)`.
- How many additions will the system perform?
- A subtle point of distinction between **call-by-name** and **call-by-need** languages.
- Haskell is *call-by-need*:
  - *Only one* addition performed.
  - The *first need* computes the value and *overwrites* the passed-in expression (called a **thunk**).

What does it mean to be lazy?
Blowing Up in Haskell

- Haskell has (relatively fancy) exceptions.
- Because exceptions let us see evaluation order,
    - **Exceptions cannot be caught in pure code.**
- But that doesn't stop us from throwing them.
- The easiest way to blow up is

```
error :: String -> a
```

- Isn't that an odd type?

What does it mean to be lazy?
Weak Head Normal Form

- Haskell evaluates as little as possible.
- Consider the expression

```
case (error "Bang", 3) of (_,x) -> x
```

- If I evaluate this expression, what pieces got evaluated?

What does it mean to be lazy?
Weak Head Normal Form

- Haskell evaluation successively expands thunks into WHNFs, which contain pointers to other thunks.
- The full **normal form** is reached once there are no more thunks in a chunk of data.
- Consider evaluating `(1,"Hi")`:

```
[THUNK]
([THUNK], [THUNK])
(1, [THUNK])
(1, 'H':[THUNK])
(1, 'H':'i':[THUNK])
(1, 'H':'i':[])
```

Why might laziness be good?

- Lazy evaluation can do everything strict evaluation can.
- And more:
  - Improves compositionality of programs.
  - Reason about some control flow as data dependence.
  - Lets us write infinite definitions.
  - Enables (amortized) efficient persistent data structures (See Okasaki, e.g. "Purely Functional Data Structures" [2]).
  - Clever tricks for automated reasoning (e.g. LazySmallCheck).
- Longer, worked examples in Hughes' "Why Functional Programming Matters" [1].

Going with the flow

- Consider the definition of `(&&)`:

```
True  && x          =  x
False && _          =  False
```

- If the left argument is `True`, then think about the right argument.
- If the left argument is `False`, return `False`.
- This is called **Short-circuit** evaluation.

Going with the flow

- Recall `and = foldr (&&) True`
- Consider `and [True,False,error "stop"]`.
- Some equational reasoning is in order:

```
and [True,False,error "stop"]
foldr (&&) True [True,False,error "stop"]
foldr (&&) True [True,False,error "stop"]
True && (foldr (&&) True [False,error "stop"])
foldr (&&) True [False,error "stop"]
False && (foldr (&&) True [error "stop"])
False
```

- Lazy evaluation means that short-circuiting behavior persists through composition.

Dynamic Programming
Immutable Arrays

- Haskell has, among other forms, immutable non-strict arrays, available in `Data.Array` module.
- What does such a thing look like?

Dynamic Programming
Immutable Arrays

- Arrays are indexed by "`Ix`"-able types:

```
class (Ord a) => Ix a where
  range :: (a, a) -> [a]
  index :: (a, a) -> a -> Int
  {- ... -}
```

- Usual suspects: `Int`, `Integer`, `Char`,
- More interesting: pairs, triples, quads, quintuples,
- Seemingly screwy: `Ordering`, `Bool`, `()`

Dynamic Programming
Immutable Arrays

- Build one with this odd-looking function:

```
array :: Ix i => (i,i) -> [(i,e)] -> Array i e
```

  - Takes bounds and an **association list**.

- Or maybe this one:

```
listArray :: Ix i => (i,i) -> [e] -> Array i e
```

Dynamic Programming
Immutable Arrays

- Index one with the `(!)` operator:

```
(!) :: (Ix i) => Array i e -> i -> e

test = (listArray (0,4) "abcde") ! 3
```

- Update an array (inefficient):

```
(//) :: Ix i => Array i e
             -> [(i, e)] -> Array i e

test = (listArray (0,4) "abcde" // [(3,'x')])
       ! 3
```

Dynamic Programming

Dynamic Programming

- **Dynamic Programming** means
  - Recursive problem decomposition
  - Memoization of subproblems
- Consider Fibonacci numbers. Natural recursive definition:

ExpensiveFib.hs

```
fib 0 = 1
fib 1 = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
fib _ = error "negative fib"
```

- Slow to evaluate directly.

Dynamic Programming

Dynamic Programming

- Slow to evaluate directly:

```
fib 5
fib 4 + fib 3
(fib 3 + fib 2) + (fib 2 + fib 1)
((fib 2 + fib 1) + (fib 1 + fib 0))
  + ((fib 1 + fib 0) + fib 1)
(((fib 1 + fib 0) + fib 1) + (fib 1 + fib 0))
  + ((fib 1 + fib 0) + fib 1)
```

Dynamic Programming

Dynamic Programming

- Better: for all `n`, compute `fib n` *only once*!
- How?

Dynamic Programming

Dynamic Programming

- Better: for all `n`, compute `fib n` *only once*!
- How?
- One possible way would be

FibsArray.hs

```
import Data.Array

fib n | n >= 0 = let
        fibsN = listArray (0,n)
                   (1:1:map fibDefn [2..n])
        fibDefn n = (fibsN ! (n - 1))
                  + (fibsN ! (n - 2))
  in fibsN ! n
```

Dynamic Programming
Dynamic Programming

- One possible way would be

FibsArray.hs

```
import Data.Array

fib n | n >= 0 = let
        fibsN = listArray (0,n)
                    (1:1:map fibDefn [2..n])
        fibDefn n = (fibsN ! (n - 1))
                   + (fibsN ! (n - 2))
  in fibsN ! n
```

- Builds an array of thunks which all have references to the array.

To Infinity...
Eating Your Own Tail

- Let's define an infinitely long list of zeros:

```
Prelude> let zeros = 0 : zeros
```

- What would a strict language do?
- What does Haskell do?
  - Think about WHNF!

To Infinity...
Eating Your Own Tail

- Let's define an infinitely long list of zeros:

```
Prelude> let zeros = 0 : zeros
```

- What would a strict language do?
- What does Haskell do?
  - Think about WHNF!
- Function in the standard library: `zeros = repeat 0`

To Infinity...
Eating Your Own Tail

- Let's define an infinitely long list of zeros:

```
Prelude> let zeros = 0 : zeros
```

- OK, that's nice; can I *see* the list?

To Infinity... Eating Your Own Tail

- Let's define an infinitely long list of zeros:

```
Prelude> let zeros = 0 : zeros
```

- OK, that's nice; can I *see* the list?
- Well you could ask GHCi to show it...

To Infinity...
Eating Your Own Tail

- Let's define an infinitely long list of zeros:

```
Prelude> let zeros = 0 : zeros
```

- OK, that's nice; can I *see* the list?
- Well you could ask GHCi to show it...
  - If you just did that, hit Control-C.

To Infinity... *Eating Your Own Tail*

- Let's define an infinitely long list of zeros:

```
Prelude> let zeros = 0 : zeros
```

- OK, that's nice; can I *see* the list?
- Well you could ask GHCi to show it...
  - If you just did that, hit Control-C.
- Try this: `take 10 zeros`.

To Infinity...
Eating Your Own Tail

- List of all natural numbers?

To Infinity…
Eating Your Own Tail

- List of all natural numbers?
- `nats = 0 : map (+1) nats`

To Infinity...
Eating Your Own Tail

- List of all natural numbers?
- `nats = 0 : map (+1) nats`
- Again, standard library function

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

nats = iterate (+1) 0
```

- In fact, since Ints are `Enumable`, new syntax: `nats = [0..]`.

To Infinity...
Frame Shifts Aren't Always Bad

- A very useful function:

```
zip :: [a] -> [b] -> [(a,b)]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zip = zipWith (\a b -> (a,b))
    = zipWith (,)
```

- Some examples:

```
Prelude> zip [0..4] [1..5]
[(0,1),(1,2),(2,3),(3,4),(4,5)]
Prelude> zipWith (+) [0..4] [1..5]
[1,3,5,7,9]
```

To Infinity...
Frame Shifts Aren't Always Bad

- Even works on infinite lists:

```
Prelude> take 5 $ zipWith (+) [0..] [0..]
[0,2,4,6,8]
```

To Infinity...
Frame Shifts Aren't Always Bad

- Even works on infinite lists:

```
Prelude> take 5 $ zipWith (+) [0..] [0..]
[0,2,4,6,8]
```

- Another definition of fibs:

```
fibs = 1 : 1 : (zipWith (+) fibs (tail fibs))
fib n = fibs !! n
```

- Draw the thunks!

Next time

- Effects in a lazy system, or "How I Learned to Stop Worrying and Love Monads":
  - I/O
  - Mutation
  - Exceptional control flow

Bib

J. Hughes.
Why Functional Programming Matters.
Computer Journal, 32(2):98–107, 1989.

Chris Okasaki.
Purely Functional Data Structures.
Cambridge University Press, 1998.