Keep Your Laziness In Check

Kenny Foner and **Hengchu Zhang**

November 20, 2017

University of Pennsylvania

Being lazy is fun and useful, but. . .

. . . sometimes it leads to unintended consequences.

Wouldn’t it be nice to QuickCheck laziness?

Traditional property-based testing (such as QuickCheck):

- ✓ Great for testing functional correctness
- ✓ Write a specification, fuzz inputs to functions to automatically test against that specification
- ✗ Can’t observe (or even specify) properties beyond functional correctness

If we were able to **specify** and **observe** laziness, we could treat it *just like* functional correctness.

StrictCheck

"We actually can do that thing."

Observing strictness (part I)

```
instrumentListWithRef :: IORef Int -> [a] -> [a]
instrumentListWithRef _     []       = []
instrumentListWithRef count (a : as) =
  unsafePerformIO $ do
    modifyIORef' count (\x -> x + 1)
    return (a : instrumentListWithRef count as)
```

Strictness doesn’t exist in a vacuum

```
type Context a = a -> ()
```

Strictness doesn't exist in a vacuum

```
type Context a = a -> ()
-----------------------------------------------
lazy, whnf, spineStrict :: Context [a]
lazy        = \xs -> const () xs
```

Strictness doesn't exist in a vacuum

```
type Context a = a -> ()
--------------------------------------------
lazy, whnf, spineStrict :: Context [a]
lazy         = \xs -> const () xs
whnf         = \xs -> case xs of
                        []    -> ()
                        (_:_) -> ()
```

Strictness doesn't exist in a vacuum

```haskell
type Context a = a -> ()
-----------------------------------------------
lazy, whnf, spineStrict :: Context [a]
lazy        = \xs -> const () xs
whnf        = \xs -> case xs of
                       []    -> ()
                       (_:_) -> ()
spineStrict =
  \xs -> whnf (foldl' (\ys y -> y : ys)) [] xs)
```

Demanding an answer, lazily

```haskell
evaluate :: () -> IO ()
evaluate () = return ()
```

Demanding an answer, lazily

```haskell
evaluate :: () -> IO ()
evaluate () = return ()

demandCount :: Context b -> ([a] -> b) -> [a] -> Int
demandCount c f as =
  unsafePerformIO $ do
    count <- newIORef 0
    let observableList =
          instrumentListWithRef count as
    evaluate ((c . f) observableList)
    readIORef count
```

Examples of `demandCount`

```
ghci> let f = take 6
```

Examples of `demandCount`

```
ghci> let f = take 6
ghci> demandCount lazy f [1..]
0
```

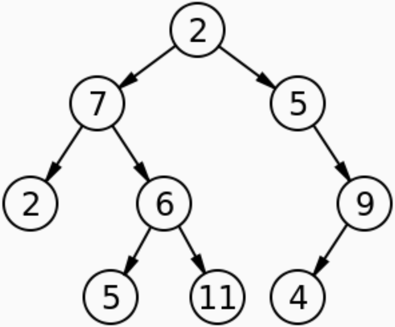
Examples of `demandCount`

```
ghci> let f = take 6
ghci> demandCount lazy f [1..]
0
ghci> demandCount whnf f [1..]
1
```

Examples of `demandCount`

```
ghci> let f = take 6
ghci> demandCount lazy f [1..]
0
ghci> demandCount whnf f [1..]
1
ghci> demandCount spineStrict f [1..]
6
```

Beyond lists and numbers



?

ListDemand

```
data List a =
    Cons a (List a)
  | Nil

data Thunk a = T | E a

data ListDemand d =
    ConsD (Thunk d)
          (Thunk (ListDemand d))
  | NilD

data IntDemand = IntD
```

Examples

```
data ListDemand d =
    ConsD (Thunk d)
          (Thunk (ListDemand d))
  | NilD

data IntDemand = IntD
---------------------------------------------------
ConsD T (ConsD (E IntD) (ConsD T T))

ConsD (E IntD) (ConstD T (ConsD (E IntD) (E NilD)))
```

TreeDemand

```haskell
data Tree a =
    Node (Tree a) a (Tree a)
  | Leaf

data TreeDemand d =
    NodeD (Thunk (TreeDemand d))
          (Thunk d)
          (Thunk (TreeDemand d))
  | LeafD
```

ListDemand vs TreeDemand

```
data ListDemand d =
    ConsD (Thunk d)
          (Thunk (ListDemand d))
  | NilD

data TreeDemand d =
    NodeD (Thunk (TreeDemand d))
          (Thunk d)
          (Thunk (TreeDemand d))
  | LeafD
```

Computing demand, generically

```
demandList :: Context b -> ([Int] -> b)
           -> [Int]
           -> (b, Thunk (ListDemand IntDemand))

demandTree :: Context b -> (Tree Int -> b)
           -> Tree Int
           -> (b, Thunk (TreeDemand IntDemand))
```

Computing demand, generically

```
demandList :: Context b -> ([Int] -> b)
           -> [Int]
           -> (b, Thunk (ListDemand IntDemand))

demandTree :: Context b -> (Tree Int -> b)
           -> Tree Int
           -> (b, Thunk (TreeDemand IntDemand))

------------------------------------------------
demand     :: Context b -> (a -> b) -> a
           -> (b, Thunk (Demand a))
```

Generic Demand calculation

```
type family Demand (x :: *) :: * where
 Demand (a -> b)  = FuncDemand
 Demand (a :+: b) = Demand a :+: Demand b
 Demand (a :*: b) = Thunk (Demand a) :*: Thunk (Demand b)
```

First-order specifications

```haskell
-- uncurried take
take     :: (Int, [a]) -> [a]

takeSpec :: Int -> [a]
         -> Demand [a]
         -> (Thunk (Demand Int), Thunk (Demand [a]))
```

Connecting specification with observation

PLACEHOLDER FOR PICTURE

PLACEHOLDER FOR
PICTURE

Contributions

With **StrictCheck**, you will be able to:

- **Observe** laziness efficiently from within Haskell
- **Specify** laziness properties as Haskell functions
- **Test** implementations against those specifications
- **For all types**,[1] including higher-order functions and data types containing functions

The implementation is a work in progress.

[1] Simple (i.e. non-indexed, non-existential) types