

Keep Your Laziness In Check

Kenny Foner, Hengchu Zhang, and Leo Lampropoulos

November 20, 2017

University of Pennsylvania

2017-11-17

Keep Your Laziness In Check

Keep Your Laziness In Check

Kenny Foner, Hengchu Zhang, and Leo Lampropoulos
November 20, 2017
University of Pennsylvania

. . . sometimes it leads to unintended consequences.

2017-11-17

Keep Your Laziness In Check

└ Being lazy is fun and useful, but. . .

. . . sometimes it leads to unintended consequences.

- Kenny:
- Enqueue $O(1)$, Dequeue $O(1)$ NOT AMORTIZED because of laziness!
- Guarantees valid persistently; thunks are only evaluated once
- How do we know we have implemented the Okasaki Queues correctly?
- Correctness \neq functional correctness or micro-benchmark performance:
- \Rightarrow the right amount of lazy

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.

Keep Your Laziness In Check

└─Wouldn't it be nice to QuickCheck laziness?

- QuickCheck gives functional correctness
- But laziness can't be observed by QuickCheck
- What if we make it observable? Then it's just part of functional correctness!

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.”

2017-11-17

Keep Your Laziness In Check

StrictCheck

“We actually can do that thing.”

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

└ 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)
```

- Hengchu:
- instrumentListWithRef injected a snippet of code at every cons cell
- and it returns a new “loaded” list
- the injected code is executed as the “loaded” list gets pattern-matched on

Strictness doesn't exist in a vacuum

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

2017-11-17

Keep Your Laziness In Check

└ 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
```

2017-11-17

Keep Your Laziness In Check

└ 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
                []      -> ()
                (_:_) -> ()
```

2017-11-17

Keep Your Laziness In Check

└ 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

```
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)
```

Keep Your Laziness In Check

└ 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  
              []      -> ()  
              (_:_) -> ()  
  
spineStrict =  
  \xs -> whnf (foldl' (\ys y -> y : ys) [] xs)
```

- The context type gives a function that returns a trivial value of the unit type
- It might look like there's only one such function `const ()`
- There can actually be all kinds of varying strictness behavior
- Forcing the unit value triggers the strictness behavior on the input `a`
- This gives us an easy way to observe the strictness behavior of a context

Demanding an answer, lazily

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

2017-11-17

Keep Your Laziness In Check

└─Demanding an answer, lazily

Demanding an answer, lazily

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

```
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
```

└─Demanding an answer, lazily

```
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
```

- The seemingly trivial evaluate function forces the unit value returned by a context, thus triggering evaluation of the demanded part of observableList
- demandCount takes a context, and a function that operates over lists and the input list
- it applies instrumentListWithRef on the input list, producing an observableList, and applies the function and context over the observableList, triggering the injected instrumentation code

Examples of demandCount

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

2017-11-17

Keep Your Laziness In Check

└─ Examples of demandCount

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

Examples of demandCount

```
ghci> let f = take 6  
ghci> demandCount lazy f [1..5]
```

2017-11-17

Keep Your Laziness In Check

└─ Examples of demandCount

```
ghci> let f = take 6  
ghci> demandCount lazy f [1..5]
```

Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
```

2017-11-17

Keep Your Laziness In Check

└─ Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
```

Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
```

2017-11-17

Keep Your Laziness In Check

└─ Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
```

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
1
```

2017-11-17

Keep Your Laziness In Check

└─ Examples of demandCount

Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
1
```



```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
1
ghci> demandCount spineStrict f [1..5]
```

2017-11-17

Keep Your Laziness In Check

└─ Examples of demandCount

Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
1
ghci> demandCount spineStrict f [1..5]
```

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
1
ghci> demandCount spineStrict f [1..5]
5
```

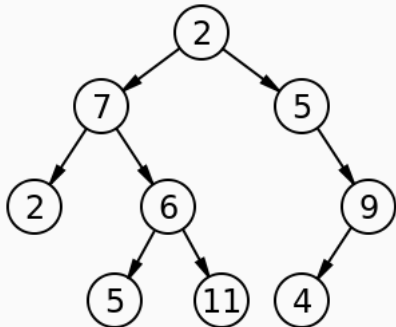
└─ Examples of demandCount

```
ghci> let f = take 6
ghci> demandCount lazy f [1..5]
0
ghci> demandCount whnf f [1..5]
1
ghci> demandCount spineStrict f [1..5]
5
```

- Just some examples showing demandCount
- Note that we simply treat `f` as a black box
- We don't need to know anything more about `f` other than the fact it typechecks with demandCount!



?



?

- Kenny:
- This gives us a bare example that counts the number of cons cells forced, but what if we want more fine grained information of arbitrary algebraic data types?
- Numbers can't directly capture the structure of a tree, so it can't capture the nodes of a tree that get forced in a computation

```

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

```

└─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

```

- Now, we can exactly characterize the demand on a list of Ints by composing the type ListDemand and IntDemand

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

```
data IntDemand = IntD
```

```
ConsD T
  (ConsD (E IntD)
    (ConsD T T))
```

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

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

- The 2nd Int is forced, and we force 3 cons cells, we don't force anything in the rest of the list
- The 1st and 3rd Int is forced, and the list's spine is forced
- Note that these represent concrete observations instrumented on given inputs, they do not represent the general behavior of contexts

└TreeDemand

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

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

- The demand type for a binary Tree
- We can either demand the value at an internal node, or one of the subtrees
- The demand type represents all of the possible prefixes/subshapes of its corresponding data type

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

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

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

- Notice the similarity between ListDemand and TreeDemand with their corresponding data types
- In general, the demand type of a data type simply interleaves a Thunk at every constructor field

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

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

2017-11-17

Keep Your Laziness In Check

└ Computing demand, generically

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))
```



```
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))
```

└ Computing demand, generically

- FIX THE MARGIN ISSUES
- There is a generic transformation between a data type and its corresponding data type representing a demand upon it
- Demand is a type level function that calculates the demand type
- We return a Thunk of demand because the input might not be demanded at all
- As the input value is traversed by the function, which is driven by the context, the evaluation of each thunk builds a pointer based data structure which is isomorphic to the demand data type, which is then frozen into a demand representation

```
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))
```

```
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)
```

```
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)
```

- For simplicity, we speak of types in terms of sums and products
- FuncDemand is isomorphic to IntDemand in that it only captures whether the function was used at all

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

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

- Having demonstrated how to reify demand behaviors into value, we now need to figure out how to write down a specification to check if whether a particular run of the program has the specified laziness according to the specification
- Specifications goes backwards from demands on the output to demands on the inputs of the function. This is because how much input is demanded depends on how much output is demanded, hence the arrows go the other way.
- Note that takeSpec takes the same inputs as take would. This is necessary in general because the demand behavior of programs can depend on their inputs, as it is the case for take!

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

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

PLACEHOLDER FOR
PICTURE

2017-11-17

Keep Your Laziness In Check

Connecting specification with observation
PLACEHOLDER FOR
PICTURE

└ Connecting specification with observation

- To QuickCheck takeSpec, we use mechanisms from QuickCheck to fuzz an `Int` and a `[a]` as inputs for `take`
- We use the `CoArbitrary` mechanism to generate random contexts that exerts non-trivial demand on the output values of `take`
- We use the generic `demand` function to kick off the entire computation
- We have now observed the demand on the input and also the demand on the output of `take`, so we can just straightforwardly compare that to what the specification expects.
- If we find a counterexample, we shrink the inputs first, and then shrink the reified demand, and re-exert that new demand on the output.

PLACEHOLDER FOR
PICTURE

2017-11-17

Keep Your Laziness In Check

└ Higher-order specifications

Higher-order specifications
PLACEHOLDER FOR
PICTURE

- There is in fact also a mechanical transformation between the type of a function to the type of its demand specification
- Moreover, this transformation works even with high-order functions
- Higher-order specifications will take specification of their input functions because their demand behavior is parameterized over the demand behavior of their input functions
- At testing time, these specifications can be fuzzed through the same CoArbitrary mechanism

Contributions

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

- **Observe** laziness 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.

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

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

- **Observe** laziness 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.

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

StrictCheck is a lightweight tool that adds a very small overhead for instrumentation and observation of functional programs in Haskell. It also provides infrastructure for writing specifications and checking