Keep Your Laziness In Check

Kenny Foner, **Hengchu Zhang**, and **Leo Lampropoulos**

November 20, 2017

University of Pennsylvania

Keep Your Laziness In Check

Kenny Foner, Hengchu Zhang, and Leo Lampropoulos

November 20, 2017

University of Pennsylvania

Lazy Evaluation?

```
fibonacci =
  0 : 1 : (zipWith (+) fibonacci (tail fibonacci))
```


Lazy Evaluation?
fibonacci =
  0 : 1 : (zipWith (+) fibonacci (tail fibonacci))

Lazy Evaluation?

```
fibonacci =
  0 : 1 : (zipWith (+) fibonacci (tail fibonacci))
```

```
ghci> fibonacci !! 10
ghci> 55
```


Lazy Evaluation?

```
fibonacci =
  0 : 1 : (zipWith (+) fibonacci (tail fibonacci))
ghci> fibonacci !! 10
ghci> 55
```

Lazy Evaluation?

```
fibonacci =
  0 : 1 : (zipWith (+) fibonacci (tail fibonacci))
```

```
ghci> fibonacci !! 10
ghci> 55
```

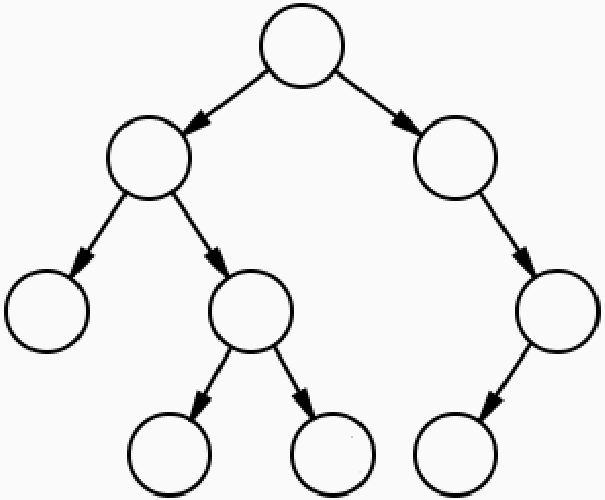
```
ghci> fibonacci !! 11
ghci> 89
```

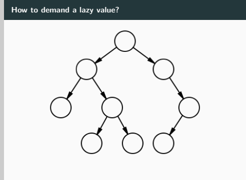

Lazy Evaluation?

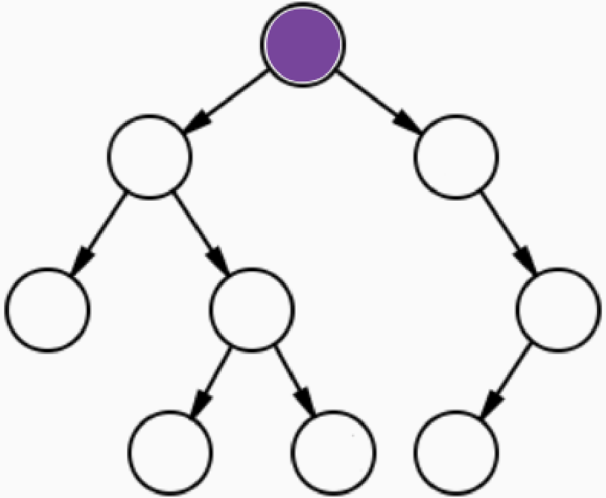
fibonacci =
  0 : 1 : (zipWith (+) fibonacci (tail fibonacci))
ghci> fibonacci !! 10
ghci> 55
ghci> fibonacci !! 11
ghci> 89

- Let's start with an example
- `fibonacci` builds a Haskell list, where : is list cons constructor
- `fibonacci` builds a circular structure that contains a reference to itself
- This is an infinite list
- but its definition doesn't cause the interpreter to go to an infinite loop
- Core idea is: code doesn't execute unless output is used

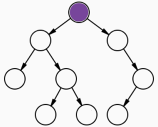
How to demand a lazy value?

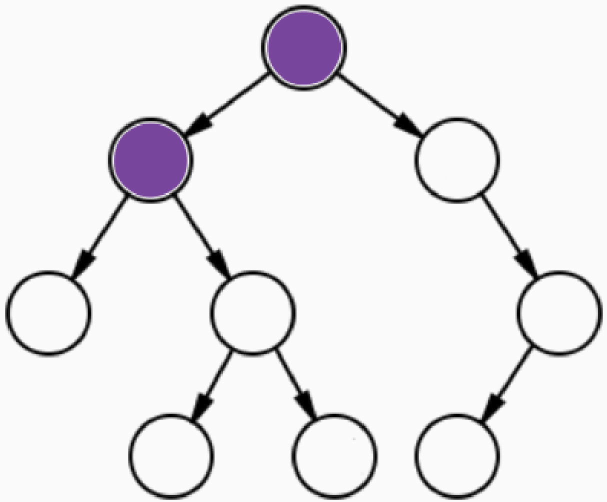




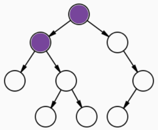


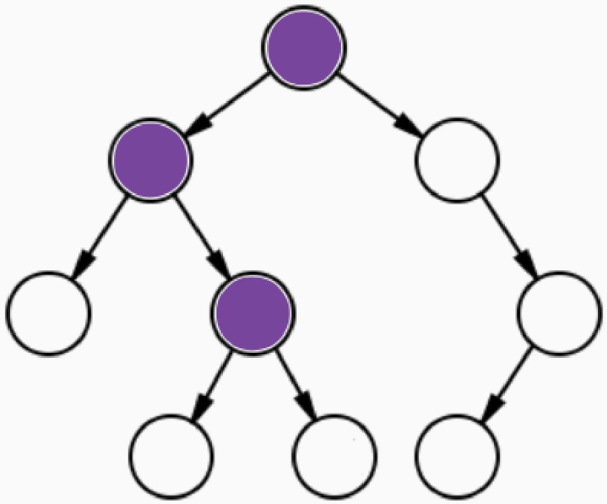
Keep Your Laziness In Check



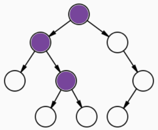


Keep Your Laziness In Check





Keep Your Laziness In Check



- A lazy value can be thought of as a pointer graph with regions that have not matierialized/evaluated yet
- In this graph, we use white regions to represent parts of the value that have not been evaluated yet
- We evaluate more of the value by following the pointers

Pattern Matching

```haskell
tail :: [Int] -> [Int]
tail xs = case xs of
            [] -> []
            (_ : xs) -> xs
```


Pattern Matching

tail :: [Int] -> [Int]
tail xs = case xs of
[] -> []
(_ : xs) -> xs

- In a functional language like Haskell, we traverse the pointer graphs of lazy values by performing pattern matching on them
- In the `tail` function for lists, we evaluate the input list to its first constructor
- Useful: think about Randoop, it iteratively generates new API calls, and alternates between generating and testing.
- If Randoop were able to generate these call sequences lazily, then from Randoop developer's perspective there can just be a module that focuses on generation and one that focuses on testing
- Since unused generated API calls are not actually evaluated, it doesn't cost additional CPU resource

Being lazy is sometimes vital, but...

...it can lead to unintended consequences.

Being lazy is sometimes vital, but...

...it can lead to unintended consequences.

- Ubiquitous in Haskell programming, used in OCaml programming
- (FRP) In the broader world, it is intrinsic in web-based functional reactive programming
- (Spark) and in the field of big-data processing, for example, Spark

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 only specify and check functional correctness properties

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

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

- PBRT is a technique for randomized testing which allows users to write down an executable specification for a function and test whether it holds for randomly generated inputs.
- QuickCheck gives functional correctness
- But laziness can’t currently be observed by tools like QuickCheck
- Laziness is not generally considered part of functional correctness much like how asymptotic runtime is not part of functional correctness because they are not directly observable
- What if we make it observable? Then it’s just part of functional correctness!

StrictCheck

"We actually can do that thing."

2017-12-03

Keep Your Laziness In Check

StrictCheck

"We actually can do that thing."

Observing strictness

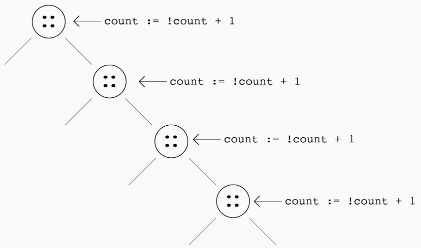
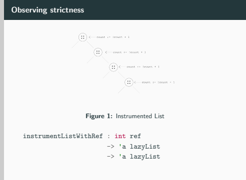


Figure 1: Instrumented List

```
instrumentListWithRef : int ref
                     -> 'a lazyList
                     -> 'a lazyList
```

- Hengchu:
- `instrumentListWithRef` clones the structure of the original list, and attaches a lazy effectful computation to each constructor in the list. So as the list is evaluated, each step of evaluation triggers one update

Strictness doesn't exist in a vacuum

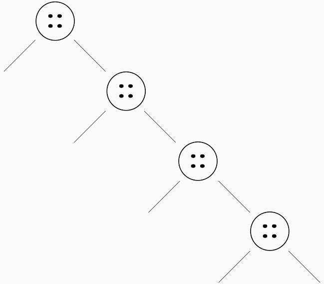
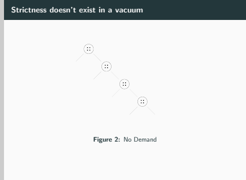


Figure 2: No Demand



Strictness doesn't exist in a vacuum

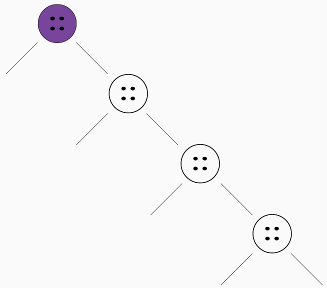
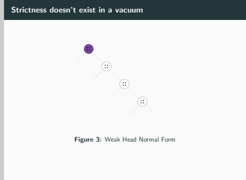


Figure 3: Weak Head Normal Form



Strictness doesn't exist in a vacuum

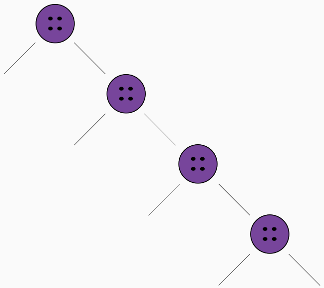
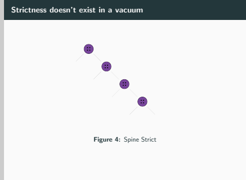


Figure 4: Spine Strict



- These pictures are illustrations of a demand context on a lazy list
- This gives us an easy way to observe the strictness behavior of a context
- While these pictures might look like a data structure, they are meant to describe the behavior of a context of evaluation on a lazy data structure of this shape

Demanding an answer, lazily

```
demandCount context f xs =
    let count = ref 0
        observableList =
          instrumentListWithRef count xs
    in context (f observableList); !count
```


Demanding an answer, lazily

demandCount context f xs =
let count = ref 0
observableList =
instrumentListWithRef count xs
in context (f observableList); !count

- 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
- context is exerting some demand on the output of f

Examples of `demandCount`

```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
```


Examples of demandCount

> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]

Examples of `demandCount`

```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]

> demandCount noDemand f list
```


Examples of demandCount

> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list

Examples of `demandCount`

```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
```


Examples of demandCount
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0

Examples of `demandCount`

```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]

> demandCount noDemand f list

0

> demandCount firstCons f list
```


Examples of demandCount

> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list

Examples of `demandCount`

```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]

> demandCount noDemand f list

0

> demandCount firstCons f list

1
```


Examples of demandCount

> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1

Examples of `demandCount`

```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list
```


Examples of demandCount
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list

Examples of `demandCount`

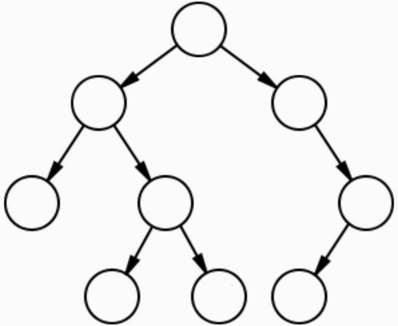
```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list
5
```


Examples of demandCount

> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list
5

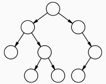
- Just some examples showing demandCount
- takeLazy is a function that takes the specified number of elements from a lazy list and returns a lazyList
- Note that we simply treat `f` as a black box
- We don't need to know anything more about `f` other than the fact it typechecks with demandCount!

Beyond lists and numbers



?

Beyond lists and numbers



?

- Kenny:
- Arbitrary lazy algebraic datatypes
- 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

Reifying demand on lazy lists

```
type 'a lazyList =
  | Cons of ('a Lazy.t *
             'a lazyList Lazy.t)
  | Nil
------------------------------------
type 'a thunk = T | E of 'a

type 'd listDemand =
  | ConsD of ('d thunk *
              'd listDemand thunk)
  | NilD

type intDemand = IntD
```


- Now, we can exactly characterize a demand on a list of Ints by composing the type ListDemand and IntDemand
- The type ListDemand is a reification of the demand placed upon a lazy list during the course of evaluation in some context

Examples

```
type 'd listDemand =
  | ConsD of ('d thunk *
              'd listDemand thunk)
  | NilD

type intDemand = IntD
----------------------------------------------------
ConsD (T,
  ConsD (E IntD,
    ConsD (T, T)))

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


Examples

type 'd listDemand =
  | ConsD of ('d thunk *
              'd listDemand thunk)
  | NilD

type 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

Reifying demand on lazy trees

```
type 'a lazyTree =
  | Node of ('a lazyTree Lazy.t *
             'a Lazy.t *
             'a lazyTree Lazy.t)
  | Leaf

type 'd treeDemand =
  | NodeD of ('d treeDemand thunk *
              'd thunk *
              'd treeDemand thunk)
  | LeafD
```


Reifying demand on lazy trees

```
type 'a lazyTree =
  | Node of ('a lazyTree Lazy.t *
             'a Lazy.t *
             'a lazyTree Lazy.t)
  | Leaf

type 'd treeDemand =
  | NodeD of ('d treeDemand thunk *
              'd thunk *
              'd treeDemand thunk)
  | 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

Comparing demands on lazy lists and trees

```
type 'd listDemand =
  | ConsD of ('d thunk *
             'd listDemand thunk)
  | NilD

type 'd treeDemand =
  | NodeD of ('d treeDemand thunk *
             'd thunk *
             'd treeDemand thunk)
  | LeafD
```


Comparing demands on lazy lists and trees

```
type 'd listDemand =
  | ConsD of ('d thunk *
             'd listDemand thunk)
  | NilD

type 'd treeDemand =
  | NodeD of ('d treeDemand thunk *
             'd thunk *
             'd treeDemand thunk)
  | 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

Observing demand, generically

```
demandList : 'b context -> (int lazyList -> 'b)
           -> int lazyList
           -> ('b * intDemand listDemand thunk)

demandTree : 'b context -> (int lazyTree -> 'b)
           -> int lazyTree
           -> ('b * intDemand treeDemand thunk)
```


Observing demand, generically

demandList : 'b context -> (int lazyList -> 'b)
-> int lazyList
-> ('b * intDemand listDemand thunk)

demandTree : 'b context -> (int lazyTree -> 'b)
-> int lazyTree
-> ('b * intDemand treeDemand thunk)

Observing demand, generically

```
demandList : 'b context -> (int lazyList -> 'b)
           -> int lazyList
           -> ('b * intDemand listDemand thunk)

demandTree : 'b context -> (int lazyTree -> 'b)
           -> int lazyTree
           -> ('b * intDemand treeDemand thunk)
------------------------------------------------
demand     : 'b context -> ('a -> 'b) -> 'a
           -> ('b * ('a DEMAND) thunk)
```

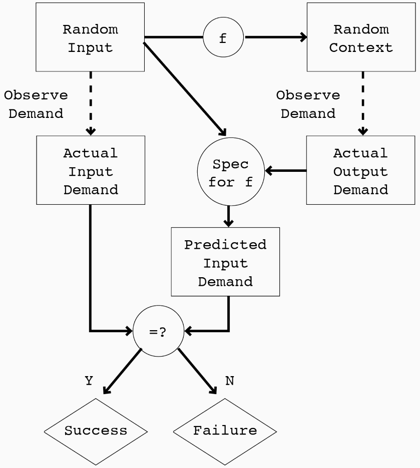

Observing demand, generically

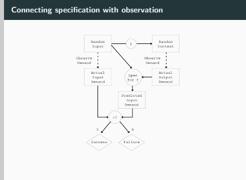
```
demandList : 'b context -> (int lazyList -> 'b)
           -> int lazyList
           -> ('b * intDemand listDemand thunk)

demandTree : 'b context -> (int lazyTree -> 'b)
           -> int lazyTree
           -> ('b * intDemand treeDemand thunk)
------------------------------------------------
demand     : 'b context -> ('a -> 'b) -> 'a
           -> ('b * ('a DEMAND) thunk)
```


- There is a determinate relation between a lazy data structure and its demand representation
- We return a `Thunk` of demand because the input might not be demanded at all
- We expect that the 'a parameter is some kind of lazy structure, and 'b is also some kind of lazy structure
- We use generic programming to implement the DEMAND type for all algebraic data types

Connecting specification with observation





- We randomly fuzz the inputs to the function, and
- We can generate random contexts that exert non-trivial demand on the output values
- We observe the demand on the input and also the demand on the output, so we can just straightforwardly compare that to what the specification expects.

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.

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

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.

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

Have questions?

Ask us about . . .

- Higher-order specifications
- Generic programming for any arity and datatype
- Random generation of partial demand contexts
- Shrinking of contexts and inputs
- Efficiency of our implementation
- Details of instrumentation
- What language are you actually implementing this in? (Hint: it's not ML)

  ⋮

Have questions?

Ask us about . . .

- Higher-order specifications
- Generic programming for any arity and datatype
- Random generation of partial demand contexts
- Shrinking of contexts and inputs
- Efficiency of our implementation
- Details of instrumentation
- What language are you actually implementing this in? (Hint: it's not ML)

  ⋮