Keep Your Laziness In Check

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

November 20, 2017

University of Pennsylvania

2017-11-19

Keep Your Laziness In Check

Keep Your Laziness In Check

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

November 20, 2017

University of Pennsylvania

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

Lazy lists

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


Lazy lists

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


- In this talk, we'll analyze functions that operate over lazy data structures
- In OCaml, Lazy is a standard library module that provides operators for suspending computations into lazy values
- for instance, a lazy list like this
- A lazy list is just a list with all of its parameters wrapped in a lazy computation

Lazy queues

```
let enQ a (front,                     back) =
          (front, lazy (Cons (lazy a, back)))

let deQ (front, back) =
  match Lazy.force front with
    | Cons (a, front') -> (a, (front', back))
    | Nil ->
        let Cons (a, front') = reverseLazyList back
        in (a, (front', lazy Nil))
```


Lazy queues

```
let enQ a (front,                     back) =
          (front, lazy (Cons (lazy a, back)))

let deQ (front, back) =
  match Lazy.force front with
    | Cons (a, front') -> (a, (front', back))
    | Nil ->
        let Cons (a, front') = reverseLazyList back
        in (a, (front', lazy Nil))
```

- As a micro example to motivate why thinking about laziness can be non-trivial, let's look at this queue
- This has amortized O(1) performance cost.
- This queue is lazy as long as you don't empty the front, then it is fully lazy in the structure of the back list. It only forces the spine of the back list when the front list is emptied.

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."

StrictCheck
"We actually can do that thing."

Observing strictness

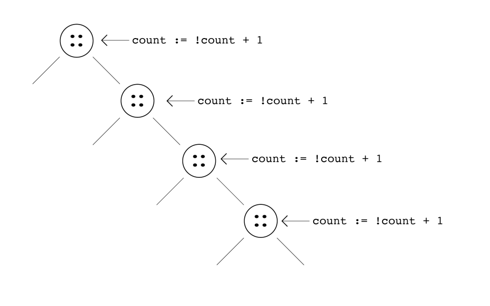
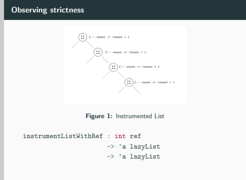


Figure 1: Instrumented List

```
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

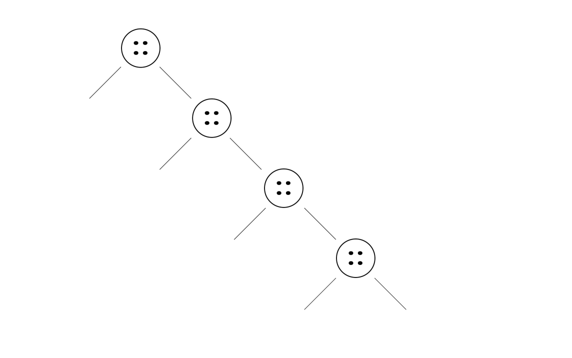
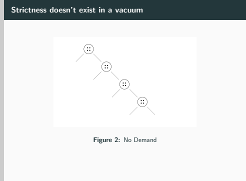


Figure 2: No Demand



Strictness doesn't exist in a vacuum

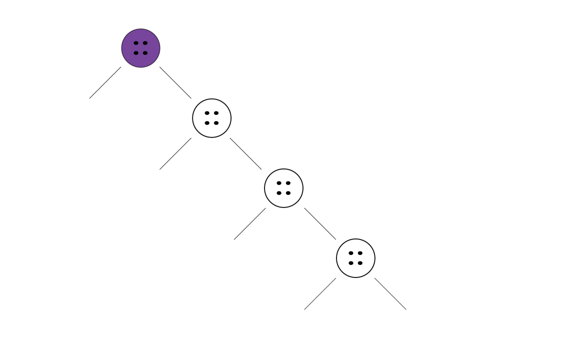
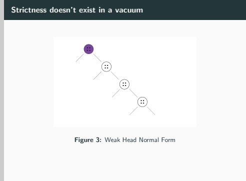


Figure 3: Weak Head Normal Form



Strictness doesn’t exist in a vacuum

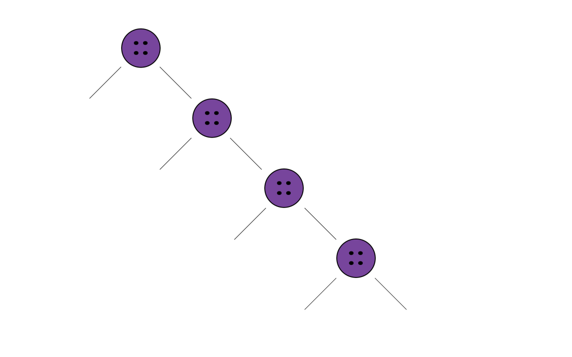
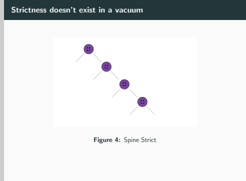


Figure 4: Spine Strict



- 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`

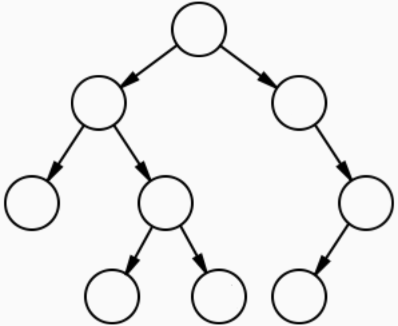
```
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list
5
```


Examples of demandCount

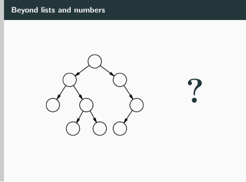
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list
5

- 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



?



- 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
```


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

Recalling `deQ`

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

let deQ (front, back)
  : ('a Lazy.t * ('a lazyList Lazy.t *
                  'a lazyList Lazy.t)) =
  match Lazy.force front with
    | Cons (a, front') -> (a, (front', back))
    | Nil ->
        let Cons (a, front') = reverseLazyList back
        in (a, (front', lazy Nil))
```


Recalling deQ

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

let deQ (front, back)
  : ('a Lazy.t * ('a lazyList Lazy.t *
                  'a lazyList Lazy.t)) =
  match Lazy.force front with
    | Cons (a, front') -> (a, (front', back))
    | Nil ->
        let Cons (a, front') = reverseLazyList back
        in (a, (front', lazy Nil))
```

First-order specifications

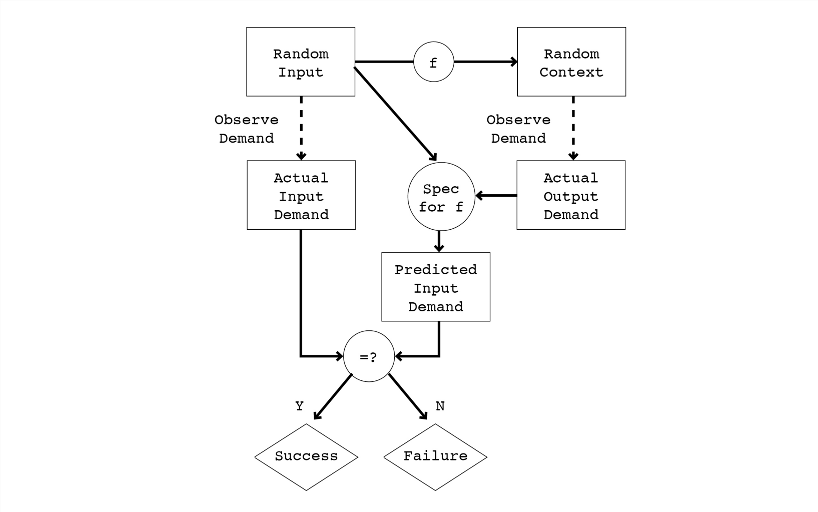
```
let spec_deQ (front, back) (dInt, (dFront, dBack)) =
  match Lazy.force front with
    | Cons (_, _) ->
        (E (ConsD (dInt, dFront)), dBack)
    | Nil -> (E NilD,
              spineStrictAs back
                (pad (length back - lengthD dFront)
                     (reverseD (ConsD dInt dFront))))
```


First-order specifications

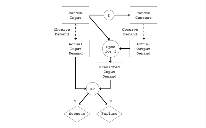
```
let spec_deQ (front, back) (dInt, (dFront, dBack)) =
  match Lazy.force front with
    | Cons (_, _) ->
        (E (ConsD (dInt, dFront)), dBack)
    | Nil -> (E NilD,
              spineStrictAs back
                (pad (length back - lengthD dFront)
                     (reverseD (ConsD dInt dFront))))
```


- 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 go 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.
- Specifications are passed in the actual values of the inputs because their demand behaviors can be dependent on those inputs
- spineStrictAs takes a lazy data structure, and unions a spine strict demand corresponding to the whole structure with another demand on some sub-part of that structure

Connecting specification with observation



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)
  ⋮