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

```
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
- 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
- In this slide, we use call-by-name ML for simplicity of presentation, but it can be implemented in OCaml using the Lazy module
- 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 be observed by QuickCheck
- Say that 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

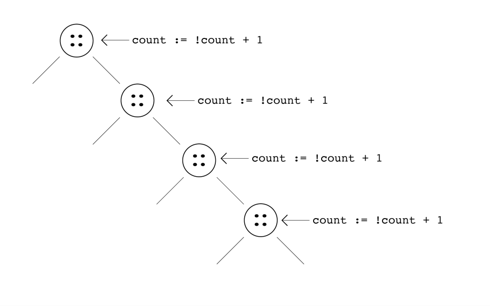
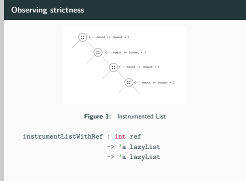


Figure 1: Instrumented List

```
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

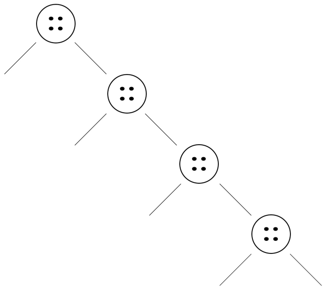


Figure 2: No Demand

Strictness doesn't exist in a vacuum

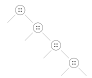


Figure 2: No Demand

Strictness doesn’t exist in a vacuum

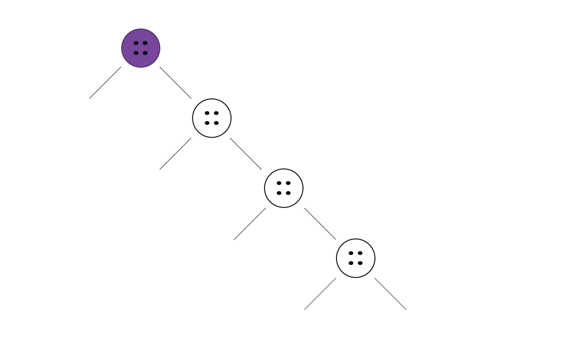


Figure 3: Weak Head Normal Form

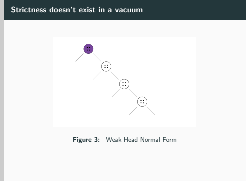

Strictness doesn’t exist in a vacuum

Figure 3: Weak Head Normal Form

Strictness doesn't exist in a vacuum

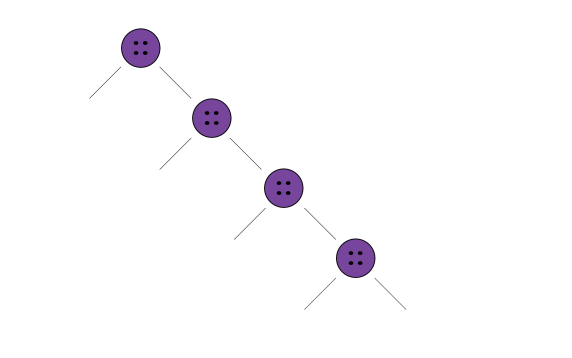
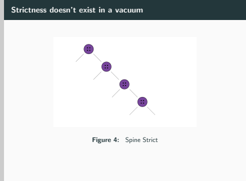


Figure 4: Spine Strict



- 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

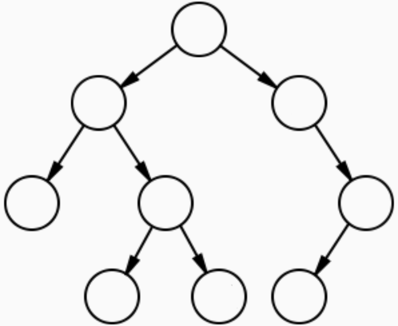
Examples of `demandCount`

```
> 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
```

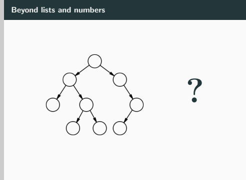

Examples of demandCount
> let f xs = takeLazy (lazy 6, xs)
> let list = toLazyList [1; 2; 3; 4; 5]
> demandCount noDemand f list
0
> demandCount firstCons f list
1
> demandCount spineStrict f list
5

- 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

ListDemand

```
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
```


ListDemand

```
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 the demand on a list of Ints by composing the type ListDemand and IntDemand

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

```
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
```


TreeDemand

```
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

```
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
```


ListDemand vs TreeDemand
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

Computing 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)
```


Computing 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)

Computing 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)
```


Computing 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
- 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
- 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

First-order specifications

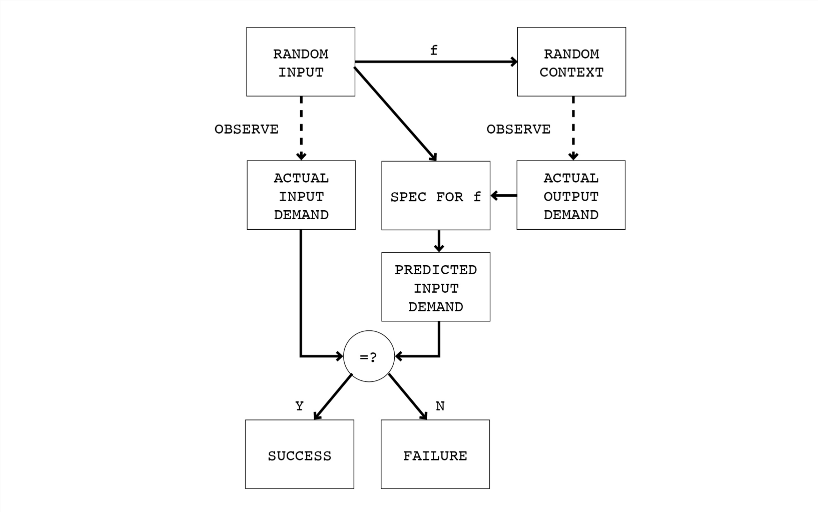
```
deQSpec : (int lazyList Lazy.t * int lazyList Lazy.t)
        -> (intDemand thunk *
             (intDemand listDemand thunk *
              intDemand listDemand thunk))
        -> (intDemand listDemand thunk *
            intDemand listDemand thunk)
deQSpec (front, back) (dInt, (dFront, dBack)) =
  match Lazy.force front with
    | Cons (_, _) -> (E (ConsD (dInt, dFront)), dBack)
    | Nil -> (dBack,
              (spineStrictD |back|)
               U ((spineStrictD
                   (|back| - |dFront|))
                  @ (reverse (ConsD dInt dFront))))
```

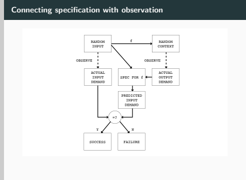

First-order specifications

```
deQSpec : (int lazyList Lazy.t * int lazyList Lazy.t)
        -> (intDemand thunk *
             (intDemand listDemand thunk *
              intDemand listDemand thunk))
        -> (intDemand listDemand thunk *
            intDemand listDemand thunk)
deQSpec (front, back) (dInt, (dFront, dBack)) =
  match Lazy.force front with
    | Cons (_, _) -> (E (ConsD (dInt, dFront)), dBack)
    | Nil -> (dBack,
              (spineStrictD |back|)
               U ((spineStrictD
                   (|back| - |dFront|))
                  @ (reverse (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 goings 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`!

Connecting specification with observation





- To QuickCheck `takeSpec`, we use mechanisms from QuickCheck to fuzz an Int and a [a] as inputs
- We can generate random contexts that exerts non-trivial demand on the output values
- 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.
- Higher-order also works! but we don't want to go into the details

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

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

Have questions?

Come ask us about . . .

- Higher-order specifications
- Generic programming
- Random generation of demand contexts
- Details of instrumentation
- What language are you actually implementing this in?
  (Hint: it's not ML)

  . . .

Have questions?

Come ask us about . . .

- Higher-order specifications
- Generic programming
- Random generation of demand contexts
- Details of instrumentation
- What language are you actually implementing this in?
  (Hint: it's not ML)

  . . .