Keep your Laziness in Check

ANONYMOUS AUTHOR(S)

We introduce StrictCheck: a property-based random testing framework for observing, specifying, and testing the strictness behaviors of Haskell functions. Strictness behavior is traditionally considered a non-functional property—StrictCheck allows it to be tested as if it were one, by reifying demands on data structures so they can be manipulated and examined internally within Haskell.

Testing strictness requires us to 1) efficiently observe the evaluation of data structures, 2) precisely specify strictness behaviors, and 3) correctly generate functions with random strictness. We tackle all three of these challenges, deriving an efficient, expressive, and generic framework for precise dynamic strictness testing. StrictCheck can specify and test the strictness behavior of any Haskell function—including higher-order ones—with only a constant factor of overhead, and requires no boilerplate for testing functions on Haskell-standard algebraic data types.

We demonstrate a non-trivial application of our library, developing a correct specification of a data structure whose properties intrinsically rely on subtle use of laziness: Okasaki's constant-time purely functional queue.

CCS Concepts: • **Software and its engineering → General programming languages**;

Additional Key Words and Phrases: Random Testing, Laziness, Haskell, Generic Programming

ACM Reference Format:
Anonymous Author(s). 2018. Keep your Laziness in Check. 1, 1 (March 2018), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Lazy evaluation gives great power to functional programmers, enabling them, among other things, to define their own control structures[citation needed], transparently and efficiently memoize computations [Hinze 2000], and decompose programs into modular pipelines [Hughes 1989]. However, programming with laziness can come with its own unique frustrations. Incorrect use of laziness can result in subtle bugs: if a program is too lazy, it may suffer from memory leaks; if it is too strict, it may suffer from asymptotic performance degradations and even infinite loops.

In practice, such bugs are often quite difficult to detect and diagnose. Seemingly trivial changes to one function can break the strictness of another function far away in a codebase. Moreover, programs with an undesired strictness are often nearly indistinguishable from those with the correct one. They may differ from the desired implementation only when tested on infinite or diverging input data, or may be semantically equivalent but inferior in performance. Unfortunately, neither semantic divergence nor poor performance are necessary results of a laziness bug—so neither conventional property-based testing nor benchmarking are sufficient to fully diagnose these issues.

How, then, can we test and prevent laziness bugs during development? In this paper, we present StrictCheck: a property-based random testing framework extending QuickCheck [Claessen and Hughes 2000] to catch arbitrarily complex laziness bugs in Haskell programs.

StrictCheck allows the programmer to:

XXXX-XXXX/2018/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- **Observe** the strictness of a function and reify it into a Haskell data structure using only a constant factor of overhead,
- **Specify** strictness precisely, capturing exactly how much a function should evaluate its inputs, and
- **Test** whether a function matches its strictness specification, reporting a minimal reproducible counter-example if it does not.

StrictCheck can test **any function**—including higher-order ones—defined over **any data type**, requiring **no boilerplate** for any Haskell 2010 algebraic data type. Moreover, StrictCheck is general enough to test functions over abstract types, existential types, GADTs, and more, given the appropriate typeclass instances.

1.1 Strictly Defining Our Terms

In a lazy language like Haskell, a computation yielding a value is only performed when that value is itself needed for some future computation. Computation only begins when the final result of a program—whatever that may be—is forced to be evaluated, e.g. by beginning to print it to the screen. Consider the following program, where `f` and `g` are some given functions:

```
list = [1, 2, 3, ..]   -- infinite lazy list
main = print (f (g list))
```

When the result of `f` is required by `print`, a certain portion of the result of `g` must be evaluated. The shape of this portion is dependent on what `f` requires in order to produce its result. In turn, to produce what `f` needs, `g` must evaluate some corresponding portion of `list`. Lazy evaluation proceeds by tracing the transitive dependencies of a series of such nested ***evaluation contexts***.

Evaluating a value in a given context means placing a ***demand*** upon it—the portion of that value required by the context. One possible demand on the list `[1, 2, 3, ..]` we saw above is the demand `_ : 2 : _`. This notation says that some context forced the first two `(:)` constructors to be evaluated, as well as the second element of the list (the integer `2`). It's important to note that we use the word "demand" to mean a sub-shape of a *particular* value—it represents what *actually happened* to that single value in a specific evaluation context. At a high level, you could view a demand as a pattern match on the value: pattern matching against `_ : 2 : _` forces the first two cons cells of a list, as well as its second element, but nothing else.

The ***strictness*** of a function is a description of the demand exerted on the inputs of that function, given a particular demand on its output. Strictness here is not a mere boolean—evaluated or not—but a large spectrum of possible behaviors. For instance, some function `f :: Bool -> Bool -> ()` could have one of nine different strictness behaviors, given a non-trivial demand on its result. Some of these include: evaluating neither argument; evaluating only the second argument; evaluating the first argument, and only if it was `True`, the second argument as well; etc. Notice that any implementation of `f` must be functionally equivalent to `\x y -> ()`, modulo its behavior on `undefined` inputs—two functionally equivalent functions may have distinguishable strictness behavior.

1.2 Describing Strictness, and Testing It!

Similarly, a precise characterization of a function's strictness is a function itself, which takes as input a particular demand on a function's output as well as a a list of the function's inputs, and returns a predicted list of demands on those inputs. For example, if we have some function

```
f :: a -> b -> ... -> z -> result
```

then we can specify the demand behavior of `f` using a function

```
spec_f :: Demand(result)                    -- demand on function result
       -> (a, b, ..., z)                    -- input values to function
       -> (Demand(a), Demand(b), ..., Demand(z))   -- demands on inputs
```

The structure of a specification suggests a natural flow for our testing framework. In order to check whether a specification `spec_f` correctly predicts the demands on the inputs of a function `f`, we will

(1) Generate random input(s) to `f`: `(a, b, ..., z)`,
(2) Generate a random evaluation context `c` for the result of `f`,
(3) Evaluate `f` in the context `c` by forcing evaluation of `(c (f a b ... z))`,
(4) Observe the demand `d` exerted on the result of `f` by the context `c`,
(5) Observe the demands `(da, db, ..., dz)` induced upon the inputs `(a, b, ..., z)`, and finally,
(6) Compare the actual observed demands `(da, db, ..., dz)` with the predictions of the specification `(pda, pdb, ..., pdz) = spec_f d a b ... z`,

repeating this until we've convinced ourselves that the specification holds. This architecture is shown below in Figure 1.

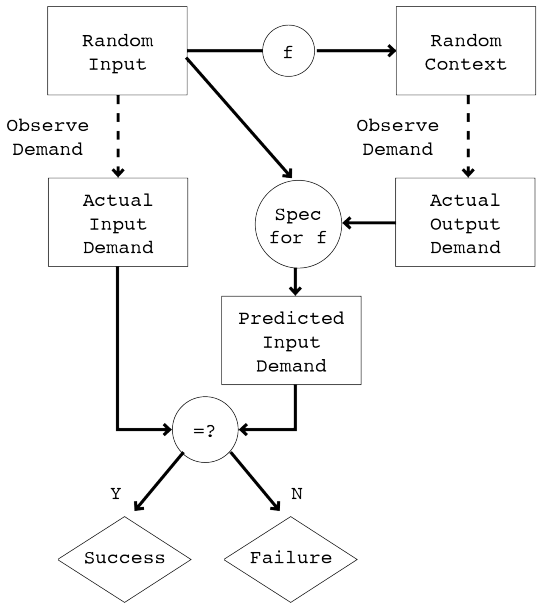

Fig. 1. Architecture of StrictCheck

For concreteness, suppose we want to specify the function `take`, which returns the first `n` elements of a given list, or the entire list if `n` is greater than the length of the list.

```
take :: Int -> [a] -> [a]
take n _ | n < 1 = []
take _      []   = []
take n (x : xs)  = x : take (n-1) xs
```

A specification for `take` will have the type

```
take_spec :: Spec '[Int, [a]] [a]
```

This specification type means that we will be specifying a function taking two arguments, `Int` and `[a]`, and returning a list of type `[a]`.

A StrictCheck specification `Spec` is a function taking as arguments—in addition to a result demand and input values—a continution `predict`, which the writer of the specification will call on their predictions for the input demands, in the same order as inputs to `take`. Formulating specifications in continuation-passing style lets us make the user interface simpler, allowing us to curry all the involved functions.

In this way, we may specify our predicted strictness behavior for `take`:

```
take_spec :: Spec '[Int, [a]] [a]
take_spec =
  Spec $ \predict resultDemand n xs ->
    predict n resultDemand
```

This specification states that `take`'s integer argument `n` will always be evaluated, and that the demand `take` will place on its list argument `xs` will be identical to whatever demand is placed upon `take`'s result. Using QuickCheck [Claessen and Hughes 2000] as its backend, StrictCheck generates random inputs to the function (integers `n` and lists `xs`), as well as demands. Then, StrictCheck shrinks any counterexamples found to a minimal form and reports it.

When we test our specification for `take` using StrictCheck, we find that we've made a specification error:

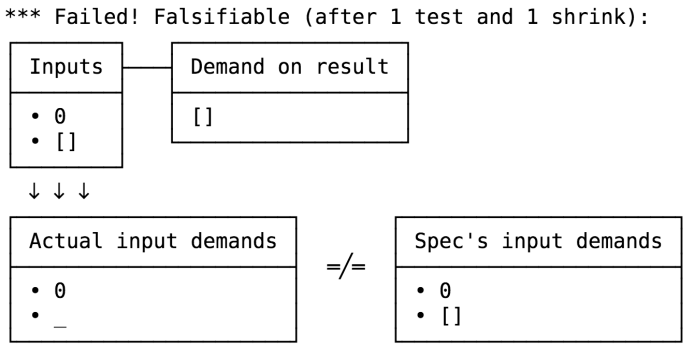

Fig. 2. Counterexample for an incorrect specification of `take`

This counter-example tells us that when we evaluated `take 0 []` and fully demanded its result `[]`, our specification predicted that we would place that same demand on the input list. However, no demand was placed on the input list! This is because `take` first checks if `n` is zero, and if so, immediately returns `[]` without evaluating the list at all.

To fix our specification, we need to account for what happens when we run `n` down to zero before we get to the end of the list. In that case, we don't evaluate the rest of the list, so the end of the demand on `xs` ought to be a thunk, not `[]`. We can correct the specification to account for this:

```
take_spec :: Spec '[Int, [a]] [a]
take_spec =
  Spec $ \predict d n xs ->
    predict n (if n > length xs then d else d ++ thunk)
```

This specification passes StrictCheck's battery of random tests.

What kinds of errors can our now-correct specification catch? Suppose that instead of our original definition of `take`, we wrote the following implementation:

```
take' :: Int -> [a] -> [a]
take' _      [] = []
take' n (x : xs)
  | n > 0     = x : take' (n-1) xs
  | otherwise = []
```

Under ordinary property-based testing, there is no way to distinguish between `take` and `take'`. They both compute the same function—but `take'` is stricter than `take` in a subtle way. [LEO: Maybe add: Can you spot where?] When we test `take'` against the correct specification for `take`, we see:

```
*** Failed! Falsifiable (after 1 test and 1 shrink):
```

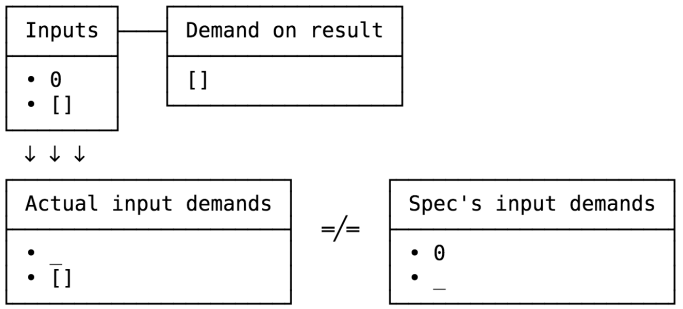

Fig. 3. Counterexample for `take'` under the correct specification `take_spec`

This counterexample quickly identifies the laziness bug in `take'`—while `take` evaluates `n` first, then conditionally evaluating `xs`, `take'` always evaluates the beginning of `xs`, conditionally evaluating `n` if `xs` is non-empty. This means that `take'` always evaluates one more constructor of its input list than it needs to. [KWF: Motivate why this is demonstrative of more serious laziness bugs—what if `xs` is expensive to compute?]

In the next section we work through another example on functional queues, demonstrating both the difficulties of programming with laziness and how our approach can be used to alleviate them. [KWF: Actual roadmap here.] [LEO: We do need a roadmap/contributions again I think. Putting back the old one. But we do need to match them to specific sections]. In the rest of the paper, we present our technical contributions:

- We develop a method for observing the strictness of a function at runtime with only a constant factor overhead, and describe our generic programming-based impementation to apply the observe mechanism to all algebraic data types(Section 3).
- We introduce a novel approach to specifying the strictness of functions that is *precise*. Such specifications are expressive enough to capture exactly how much evaluation occurs within a data structure (Section 4).
- As a side contribution, we identify a weakness in QuickCheck's function generation approach, namely that it only generates strict functions, and improve upon it to generate functions with arbitrary strictness behavior (Section 5).
- We implement a Haskell framework for automatically testing against such specifications, called StrictCheck, that uses QuickCheck for random generation of inputs and demands.

Section 6 discusses related work. We conclude and draw directions for future work in Section 7.

2 STRICTNESS SPECIFICATIONS, BY EXAMPLE

For a more detailed example, we turn to a popular data structure: purely functional queues. Such queues are typically implemented with two lists—a front list holding the queue elements in FIFO order, and a back list holding the rest of the elements in reversed order. When the front list is empty, a dequeue operation must reverse the back list to refill the front list. These specific dequeue operations take $O(n)$ time to finish and are much slower than any operation for imperative queues, a performance degradation that is unacceptable for many real-time systems, and can furthermore be compounded by persistence—since the $O(n)$ operation could be repeated many times.

Amongst his work on purely functional data structures [Okasaki 1998], Okasaki cleverly improved upon this design using lazy lists; incrementally performing the reversal of the back list, and at the same time incrementally appending the reversal of the back list onto the front list [Okasaki 1995]. Okasaki's queue relies on the following `rotate` function:

```
rotate :: [a] -> [a] -> [a] -> [a]
rotate      []       []  as =                        as
rotate      []  (b : bs) as =     rotate [] bs (b : as)
rotate (f : fs)      []  as = f : rotate fs []      as
rotate (f : fs) (b : bs) as = f : rotate fs bs (b : as)
```

If we computed the expression `rotate front back []`, its value will be equivalent to `front ++ (reverse back)`. However, unlike the naive reverse-and-append approach, a queue implemented with `rotate` avoids $O(n)$ operations by incrementally reversing the backlist with each dequeue operation. A `reverse` function has to evaluate the entire structure of the back list, which is not incremental. On the other hand, `rotate` produces a list immediately as long as the front list is not empty—a property implied by Okasaki queue's invariant.

Since `rotate front back []` is functionally equivalent to `front ++ (reverse back)`, a programmer could have implemented it naively and never discover the mistake through traditional testing! The two only differ in their strictness. Fortunately, we can use StrictCheck to write a specification that makes this property of `rotate` testable.

Before we describe the strictness behavior, we need to introduce a few of StrictCheck's types. First of all, we use a type `Thunk` to capture whether a particular field is forced or not.

```
data Thunk a = E a | T
```

Similarly to a `Maybe`, the `Thunk` data typehas two constructors: a nullary `T` corresponding to thunks and a single-argument `E` corresponding to evaluated values.

Secondly, StrictCheck introduces the notion of a *demand*: a shape that describes the used part of a data structure. For instance, we use the type `ListD` to represent demands on lists.

```
data ListD a = NilD
             | ConsD (Thunk a) (Thunk (ListD a))
```

In `ListD`, `Thunk` wraps around all the list constructor fields, and this allows us to represent lack of further evaluation using the `T` constructor.

For example, given the list `[1, 2, 3]`, the value `ConsD T (E (ConsD T T))` represents the *demand* that requires the first 2 `(:)` constructors of the list, but does not evaluate any other parts of it. To improve readability of demand values, we will use the notation presented in the introduction: the syntax `_ : _ : _` will denote the demand above. In this expression, the first two underscores represent unevaluated elements in the list, while the last underscore represents unevaluated tail of the list. We also use metavariables $\texttt{X}_\texttt{i}$ to name either a thunk or an evaluated field. For instance, $\texttt{X}_\texttt{1}$

`: X2 : _` can denote the above 2-cons demand `_ : _ : _`, a similar demand where first two values are evaluated `1 : 2 : _`, or a mix of both.

As we saw in the introduction, in Haskell, to trigger the evaluation of a lazy function, we need to exert some demand on the result of that function, and the strictness specification should calculate the demands on the input given demands on the output. Moreover, the demand on the input could depend on the *value* of the inputs as well, just like in the `take` example.

Putting these pieces together, we can specify the strictness of `rotate`. Since `rotate` is always used by supplying the empty list `[]` as the accumulator argument, we define `rot fs bs = rotate fs bs []`, and show a specification of `rot`. Before giving the actual specification however, let's try to build up some intuition through a series of example demands for the naive reverse-and-append version:

```
rot_naive :: [Int] -> [Int] -> [Int]
rot_naive fs bs = fs ++ reverse bs
```

We present example demands with the following form:

Inputs	Result	Output Demand	Demand On Inputs
`fs : [1,2,3]` `bs : [6,5,4]`	`[1,2,3,4,5,6]`	`X1 : X2 : _`	`fs : X1 : X2 : _` `bs : _`

The fist column shows the input lists `fs` and `bs`; here they are `[1,2,3]` and `[6,5,4]` respectively. The second column shows the result of `rot_naive fs bs` when fully evaluated; here `[1,2,3,4,5,6]`. The third column shows the actual output demand exerted; just like we described above, the demand $X_1 : X_2 : \_$ means that we only force the first two cons cells and, depending on whether the X_i are thunks or not, we might also force the first two elements. The first and third column together are the actual inputs to our specification. The second column is there for the reader's convenience.

The last column is the demand on the inputs predicted by our specification. In this first example, where the output demand is smaller than the input list `fs`, the demand on the first argument `fs` is equal to the demand on the output, while the demand on the second list `bs` is the empty demand. Indeed, if concretely the output demand is `1 : 2 : _` (which could for example be obtained by printing `take 2 $ rot_simple [1,2,3] [6,5,4]`), then only the first two `(:)` constructors of `fs` will be forced, and both will have their integer fields evaluated.

On the other hand, if the output demand is larger than the first list, then the *entire spine* of the second argument has to be evaluated.

Inputs	Result	Output Demand	Demand On Inputs
`fs : [1,2,3]` `bs : [6,5,4]`	`[1,2,3,4,5,6]`	`X1 : X2 : X3 : X4 : _`	`fs : [ X1 , X2 , X3 ]` `bs : [ _ , _ , X4 ]`

The predicted demand on the first list will be the prefix of the output demand with size equal to `length fs`. The predicted demand on the second list will be the reverse of the remaining demand, padded with thunks if necessary.

There is one last case we need to consider: when the demand is equal to the first list and the second one is empty.

Inputs	Result	Output Demand	Demand On Inputs
`fs : [1,2,3]` `bs : []`	`[1,2,3]`	`[X1, X2, X3]`	`fs : [X1,X2,X3]` `bs : []`

In this case, since the demand requires the result to be fully evaluated, we need to also evaluate the back list to ensure it is indeed empty.

These cases above give rise to the following StrictCheck specification, where `isCapped` and `cap` are helper functions that check if the tail of the demand is a `thunk` and replace that `thunk` with an empty list respectively.

```
rot_naive_spec :: Spec '[[Int], [Int]] [Int]
rot_naive_spec =
  Spec $ \predict d fs bs ->
    let demandOnFs
          | length (cap d) > length fs =
              take (length fs) d
          | otherwise = d
        demandOnBs
          | length (cap d) > length fs
          || null bs && isCapped fs =
              reverse $ take (length bs)
                      $ drop (length fs) (cap d)
                        ++ repeat thunk
          | otherwise = thunk
    in predict demandOnFs demandOnBs
```

Just like the `take_spec` of the introduction, this specification uses the constructor `Spec` applied to a function. That function has as arguments a continuation `predict`, the output demand `d`, as well as the actual input lists `fs` and `bs`. In its body, it `predicts` the demand on `fs` and `bs`. In particular, if we don't demand more from the output of `rot_naive` than the length of `fs` then the demand on `fs` is just `d` itself, while `bs` is left completely unevaluated. On the other hand, if we do, only the first half of the demand is actually relevant to `fs`, while the rest constitutes the demand on `bs` after being appropriately padded with thunks and reversed.

We can now move on to the more complex `rot` function from Okasaki. Once again, let's build up intuition about the strictness behavior by looking at usage examples. First, let's consider the same examples as with `rot_naive`.

Inputs	Result	Output Demand	Demand On Inputs
`fs : [1,2,3]` `bs : [6,5,4]`	`[1,2,3,4,5,6]`	$X_1 : X_2 : \_$	`fs` : $X_1 : X_2 : \_$ `bs` : `_ : _ : _`

When we demand the first two elements from the result of `rot`, the demand propagates to a corresponding demand on the front list `fs`. In fact, the demand behavior of the front list is identical to the one of `rot_naive`! The difference between the two functions lies only in the incremental strictness when reversing the back list `bs`. In this example, `rot_naive` didn't evaluate the back list at all, since we demanded fewer elements than there exist in the front list. On the other hand, `rot` evaluates the same number of `(:)` cells as it evaluated in `fs`. That is precisely the intuition behind Okasaki's queues: we do more work incrementally, at each step, to avoid costly $O(n)$ operations.

In our second example, the demand was larger than both lists:

Inputs	Result	Output Demand	Demand On Inputs
`fs : [1,2,3]` `bs : [6,5,4]`	`[1,2,3,4,5,6]`	$X_1 : X_2 : X_3 : X_4 : \_$	`fs :` [X_1 , X_2 , X_3] `bs :` [`_` , `_` , X_4]

The demands in this case are identical to `rot_naive`: we evaluate both lists fully.

These two examples were enough to gauge almost all the strictness behavior of `rot_naive`. We could, then, write the following spec for `rot`, which is very similar to the one for `rot_naive`, with

the differences in the `demandOnBs` being precisely the ones explained above, and check to see if `rot` adheres to it.

```
rot_spec :: Shaped a => Spec '[[a], [a]] [a]
rot_spec =
  Spec $ \predict d fs bs ->
    let demandOnFs
          | length (cap d) > length fs =
              take (length fs) (cap d)
          | otherwise = d
        demandOnBs
          | numCtrForced d > length fs =
              reverse $ take (length bs)
                      $ drop (length fs) (cap d)
                        ++ repeat thunk
          | otherwise =
              (reverse $ drop (length fs) (cap d)
                        ++ replicate (length (cap d)) thunk) ++ thunk
    in predict demandOnFs demandOnBs
```

In addition to the `cap` helper function we saw above, this code uses another, `numCtrForced`, that just counts the number of evaluated constructors in the demand, both `(:)` and `[]`.

When StrictCheck is called to test this spec it comes up with a counterexample: when the `length` of `bs` is greater than the length of `fs`. [LEO: Show counter?] That case is out-of-scope for Okasaki: his queue implementation maintains the invariant that `length fs >= length bs`. However, since we didn't specify that in StrictCheck, it identified a counterexample. The full version of the specification, including the case that violates the invariant, is shown in Figure 5, side-by-side with the full spec of `rot_naive` for easy comparison.

To conclude this section, we use StrictCheck to test the reverse and append implementation against the `rot_spec` specification, which immediately produces the following counterexample:

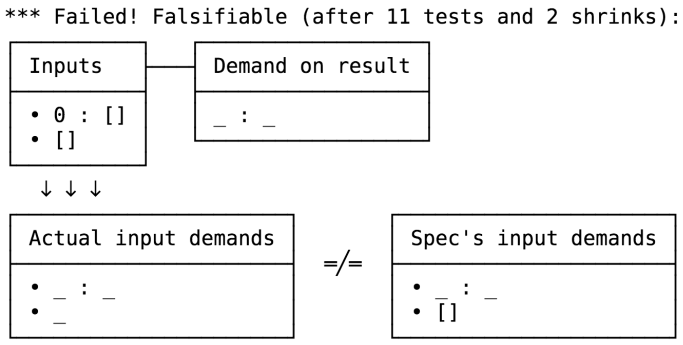

Fig. 4. Counterexample for `rot_naive` under `rot_spec`

As expected, the incremental `rot` specification requires the second list to be evaluated when `rot_naive` is in fact lazy. As such, we could use this specification for example in a real-time application that uses queues, to ensure that the non-functional correctness property of incremental strictness holds.[LEO: This is kind of weak?]

```
rot_simple_spec :: Spec '[[Int], [Int]] [Int]
rot_simple_spec =
  Spec $ \predict d fs bs ->
    let demandOnFs
          | length (cap d) > length fs =
              take (length fs) d
          | otherwise = d
        demandOnBs
          | length (cap d) > length fs
            || null bs && isCapped fs =
              reverse $ take (length bs)
                      $ drop (length fs) (cap d)
                        ++ repeat thunk

          | otherwise = thunk

    in predict demandOnFs demandOnBs
```

```
rot_spec :: Spec '[[Int], [Int]] [Int]
rot_spec =
  Spec $ \predict d fs bs ->
    let demandOnFs
          | length (cap d) > length fs =
              take (length fs) (cap d)
          | otherwise = d
        demandOnBs
          | numCtrForced d > length fs =
              -- Identical to rot_simple case
              reverse $ take (length bs)
                      $ drop (length fs) (cap d)
                        ++ repeat thunk
          -- Only needed when length bs > length fs
          | length (cap d) > length bs =
              reverse $ drop (length fs) (cap d)
                       ++ replicate (length bs) thunk
          -- Force part of bs even if not demanded
          | otherwise =
              (reverse $ drop (length fs) (cap d)
                       ++ replicate (length (cap d)) thunk)
              ++ thunk
    in predict demandOnFs demandOnBs
```

Fig. 5. StrictCheck specifications of `rot_simple` and `rot` side by side.

3 OBSERVING STRICTNESS IN HASKELL

[KWF: TODO: This is my job to revise.] Having specifications by themselves would not be useful, if we could not compare predictions with actual runtime behavior of Haskell functions. This requires a method of recording how much input is consumed in the execution of a function. StrictCheck provides an efficient and generic mechanism for observing this runtime behavior.

3.1 Getting Our Data in Shape

[KWF: I vote (pretty strongly?) that we front-load this material. I realize it's technically complicated, but I don't like that we give the reader a fiction about how demands really are represented until the end of the paper. I'm going to attempt a refactor once I finish this section, to see how it flows.]

We previously claimed that StrictCheck is able to operate over any Haskell 2010 algebraic data typewith **no boilerplate**. We fulfill this promise by using generic programming to implement observation for arbitrary algebraic data structures.

Recall that a demand on a value is merely a sub-shape, specifically a prefix, of that value. In the case of lists, we described a demand with the type `ListD`:

```
data ListD a = NilD
             | ConsD (Thunk a) (Thunk (ListD a))
```

There is a mechanical relationship between the shape of an algebraic data typeand the type of demands on that data type. In general, a demand on a data typehas the same number of constructors, and the same number of fields for each constructor. The difference between the original type and its corresponding demand type is that for each field of some type `a` in the original type, the corresponding field in its demand type is of type `Thunk a`.[LEO: This can be parsed as the type argument a of List. I added "some", still not happy.]

[KWF: Cite "Generalized bananas, lenses and barbed wire" by Erik Meijer, Maarten Fokkinga and Ross Paterson, and `recursion-schemes`, below.]

To capture this structural relationship explicitly, we choose to represent the demand upon a data structure as a fixed point. In [Meijer et al. 1991], the authors distill the essence of a monomorphically recursive data structure as the fixed point of a suitably defined functor. Consider the ordinary list:

```
data [a] = [] | a : [a]
```

In order to translate this data typeinto a more generic form, we define the data type`Fix`, which we will use to encode recursion:

```
newtype Fix f = Fix (f (Fix f))
```

Using this type, we can define a data typeisomorphic to our original list data type, but phrased in terms of `Fix`:

```
data ListF x r
  = Nil
  | ConsF x r


type List' a = Fix (ListF a)
```

In the above, the occurrences of the type variable `r` signify the points at which a recursive occurrence of `ListF` should be inserted by the fixed-point.

Other data structures, e.g. `Trees`, follow the same pattern:

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

```
data TreeF x r
  = LeafF
  | NodeF r x r

type Tree' a = Fix (TreeF a)
```

If we supply such data typeswith a pair of functions translating between themselves and their isomorphic fixed points, a sophisticated suite of folds, unfolds, and other transformations can be automatically derived "for free"!

Our goals in StrictCheck are similar, but not identical, to the solution offered by this formulation of recursion schemes. In general, a demand data structure needs to have a `Thunk` interleaved at *every* field, not merely those which are a recursive occurrence of the original data structure.

Yet, the general idea of recursion schemes is still useful to us. While the usual formulation of recursion schemes takes the fixed point of `Functors` of kind `* -> *`, our representation is accurately expressed as the fixed point of a different kind of functor: that of `(* -> *) -> *`.

We define the typeclass `Shaped` to supply the primitive operations necessary to translate and manipulate demands:

```
class Shaped a where
  type Shaped a :: (* -> *) -> *
  ...
```

The `Shape` of a data type`G` is a generalization of the pattern we saw earlier with demands. In the type `Shape G`, each field in `G` of some type `x` is replaced with a field of type `(f x)`. For example, the `Shapes` of lists and binary trees are:

```
data ListShape a f
  = NilS
  | ConsS (f a) (f [a])

data TreeShape a f
  = LeafS
  | NodeS (f (Tree a)) (f a) (f (Tree a))

instance Shaped [a] where
  type Shape [a] = ListShape a

instance Shaped (Tree a) where
  type Shape (Tree a) = TreeShape a
```

This same pattern applies to non-polymorphic fields as well. Consider the (contrived) data type`D`:

```
data D x y
  = A x Bool
  | B Int y
```

The `Shape` of `D` would be:

```
data DShape x y f
  = AS (f x) (f Bool)
  | BS (f Int) (f y)
```

```
instance Shaped (D x y) where
  type Shape (D x y) = DShape x y
```

As in the original recursion schemes work, we need to re-introduce recursion by means of an explicit fixed point to recover the structure of our initial type. We do so with the `(%)` type (pronounced "interleave"):

```
newtype (f :: * -> *) % (a :: *) :: * where
  Wrap :: f (Shape a ((%) f)) -> f % a
```

A `Demand` on a data structure is a special case of an interleaved `Shape`, where the `f` parameter is `Thunk`! That is, our original `ListD a` type is isomorphic to `Thunk % [a]`.

```
type Demand a = Thunk % a
```

The `Shaped` class defines a handful of operations in total. Two of them, `project` and `embed`, are direct analogues to those in [Meijer et al. 1991]:

```
project :: (forall x. Shaped x => x -> f x) -> a -> Shape a f

embed   :: (forall x. Shaped x => f x -> x) -> Shape a f -> a
```

The method `project` allows us to transform a type `a` into its corresponding `Shape`, when given a function to inject an arbitrary `x` into some functor `f`. Its dual, `embed`, allows us to transform a `Shape a` back into its corresponding type `a`, when given a function to extract an arbitrary `x` from some functor `f`.

The remaining two methods in the `Shaped` class are called `render` and `match`. The method `render` provides for pretty-printing (necessary in order to display sensible output to the user of StrictCheck), while the method `match` provides a very generic notion of pattern-matching on `Shape`s.

Using `Generics.SOP` [de Vries and Löh 2014], we may provide a default definition for the associated type `Shape`, as well as every method in the class. A user of StrictCheck who wishes to test a function on some custom data type`G` need only declare an empty instance of `Shaped` for it:

```
instance Shaped G
```

Even this can be shortened by the use of GHC's `DeriveAnyClass` extension, which allows the user to add `Shaped` to the `deriving` clause at the definition site of their data type:

```
data G = ... deriving (Shaped)
```

From `Shaped`, we can derive a variety of very general higher-rank folds and unfolds, which are sufficient to generically implement observation, shrinking, pretty-printing, and more.

3.2 Observation, Shapefully

In particular, let's examine how `Shaped` allows us to observe evaluation. In order to observe the evaluation of a data structure, we need an unsafe (non-exported!) primitive which reifies the evaluation of a value into an observable `Thunk`. We call this `entangle`, as it creates a kind of "spooky action at a distance" [Einstein et al. 1935].[LEO: I'm not fond of this. Maybe allude to quantum entanglement instead? IDK.]

```
{-# NOINLINE entangle #-}
entangle :: forall a. a -> (a, Thunk a)
entangle a =
  unsafePerformIO $ do
    ref <- newIORef Thunk
```

```
      return ( unsafePerformIO $ do
                 writeIORef ref (Eval a)
                 return a
             , unsafePerformIO $ readIORef ref )
```

Calling `entangle` on a value returns a pair: a copy of the original value, and a `Thunk`. The copy of the value we return has been instrumented so that if and when it is evaluated, it will write a copy of itself to an `IORef`. The second element of the pair is, notionally, a `Thunk`, but in fact an unsafe read from that same `IORef`. If the value is evaluated before the `Thunk`, the `Thunk` will contain a copy of the value; if the `Thunk` is evaluated first, it will be merely a `T`.

We can see this behavior by experimenting with `entangle` in `GHCi`:

```
> x = "ab" ++ "cd"
> (x', tx) = entangle x
> do print x'; print tx
"abcd"
E "abcd"
```

Here, because we evaluated `x'` first (by printing it), its value was written to the `Thunk` `tx`. If instead we evaluate the `Thunk` first, it does not matter whether we evaluate the value afterwards; its value will forever be `T`:

```
> y = "wx" ++ "yz"
> (y', ty) = entangle y
> do print ty; print y'; print ty
T
"wxyz"
T
```

Using the generic machinery in `Shaped`, we can lift this operation to `Shapes`, producing a copy of a value and a `Demand`, each of whose component `Thunks` reflects the evaluation status of a particular *real* thunk within the original value.

```
entangleShape :: Shaped a => a -> (a, Demand a)
```

The effect of `entangleShape` is to create a mutable shadow of the entangled data structure. Each thunk which is evaluated within the observable `a` triggers the update of a mutable pointer within the `Demand`. Normalizing the entangled `Demand` implicitly freezes this pointer structure, resulting in a reified representation of the sub-shape of the original value which was evaluated at the time the `Demand` itself was evaluated.

This is the operation we really need in order to observe evaluation of recursive data structures. Using it, we can define `observe1`, which observes the evaluation of a unary function:

```
observe1 :: (Shaped a, Shaped b, _)
         => (b -> ())              -- evaluation context
         -> (a -> b)               -- function to be observed
         -> a                      -- input to function
         -> (Demand b, Demand a)   -- observed demands
observe1 context function input =
  let (input', inputD)  =
        entangleShape input              -- (1)
      (result', resultD) =
        entangleShape (function input')  -- (2)
```

```
    in let !() = context result'          -- (3)
    in (resultD, inputD)                  -- (4)
```

[KWF: Flow this in with the previous stuff about observation…]

[HZH: We need to explain why evaluation context returns unit somewhere around here, but I'm not sure where] To observe the evaluation of a function on some input and in some evaluation context, `observe1`:

(1) Entangles the input to the function, yielding an observable input, `input'`, and observation of demand upon it, `inputD`,
(2) Entangles the result of the function applied to that input, yielding an observable result, `result'`, and observation of demand upon it, `resultD`,
(3) Evaluates the observable result `result'` in the evaluation context `context`, and
(4) Returns the reified demand on the result and the reified demand this induced upon the inputs.

While `entangle` and `entangleShape` are emphatically unsafe (that is, referentially non-transparent), `observe1` is pure and referentially transparent. We can justify this by examining the reasons `entangle` is *not* pure. In particular, `entangle`'s results are dependent on the order in which you evaluate them! But `observe1` takes an evaluation context as *input*, and crucially, does not return the observable result of the function—it just returns the reified `Demand`s. This means we can be sure that any evaluation of the result of the function occurs within `observe1`, prior to it returning anything. As such, all the updates to the mutable shape have concluded prior to our exploration of the resultant entangled demands, which means that the order of evaluation outside `observe1` does not influence its results. Furthermore, because each invocation of `observe1` allocates a *new* mutable data structure (within each call to `entangleShape`), the context(s) in which we evaluate a call to `observe1` do not change its results. That which was impure has been purified![KWF: This is long and technical and dense. I don't know if we need to go into this kind of detail, but I wrote it all out just in case.][LEO: As a semi-outsider, I think this is EXACTLY the level of detail we need :)]

We can use `observe1` to see, for example, that `reverse` is spine-strict in its input list when its result is evaluated to weak-head normal form:

```
> (resultDemand, inputDemand) = observe1 (\x -> seq x ()) reverse "abc"
> printDemand inputDemand
_ : _ : _ : []
```

The demand printed shows us that when we only demanded the first constructor of the reversed list, every `(:)` and `[]` constructor in the spine of the list was evaluated, but none of the elements were.

StrictCheck also generalizes over arity, providing the function `observe`, which observes the evaluation of a (curried) n-ary function:

```
observe :: ( All Shaped (Args function)
           , Shaped (Result function), _ )
        => (Result function -> ())
        -> function
        -> Args function
        -...-> (Demand (Result function), NP Demand (Args function))
```

Above, `(-...->)` is a type family of kind `[*] -> *` which calculates the appropriate curried function type, given the arguments to the function in question. The type `NP Demand` is an ***N***-ary ***P***roduct of demands on the arguments of the function.

4 DEMANDS AND SPECIFICATIONS

StrictCheck specifications need to manipulate demand values in order to produce their predictions. Section 3 introduced the type constructor `Demand :: * -> *`, which maps types to their demand types, and also explained how to obtain these demand values through `observe`. Now we focus on how to use these values in specifications.

The Haskell `Data.List` module provides many useful functions over lists, some compute the length of lists, some append lists, some reverse lists, and many more. Each of these operations can be useful for values of the `Demand [a]` type. For example, in `rot_spec` we need to compute the length of evaluated sub-list of the output, and split list demands.

We could, in principle, implement analogous operations from `Data.List` for `Demand [a]`. However, the overhead is tremendous just for writing specifications over functions on lists! Since StrictCheck's observe mechanism works generically, there could be countless boilerplate to manually write in order to compute over demand values of many different types. We certainly should reuse existing functions to manipulate demand values. However, demand values inhabit a different type, and they may contain thunks throughout its structure. Since thunks contain no extra information other than representing the lack of evaluation, a demand value of type `Demand a` do not contain enough information to reproduce the original value of type `a`. However, we can still produce a partial value if we have a special term for thunks. StrictCheck provides such a term `thunk` with type `forall a. a`, which we've already seen in the introductory examples, so that it can represent thunks for any type.

Consider the demand value `_ : 2 : _` on lists of integers. We can obtain an isomorphic value of type `[Int]` using `thunk`—`thunk : 2 : thunk`, where the first `thunk` represents the missing `Int` value, and the second `thunk` represents the missing tail of the list. This allows us to poke at this partial list with list operations; for example, `take 1` from this list would return the singleton list, `[thunk]`, containing the first thunk. StrictCheck provides two generic operations `fromDemand :: (Shaped a) => Demand a -> a` and `toDemand :: (Shaped a) => a -> Demand a` that performs this conversion between demand values and partial values. StrictCheck also provides `isThunk :: (Shaped a) => a -> Bool` for testing whether a `Shaped` value is a `thunk` or not.

In fact, specifications written with the `Spec` type are provided these partial values as demands by default. In all of the examples shown in Section 2, we have manipulated demand values on lists with authentic functions from `Data.List`, and in these examples this approach is significantly more convenient than explicit manipulations of `Demand [a]` values. Programming with these partial list values, however, means we must be mindful of where `thunk` may occur. As `thunk`'s type suggests, it is necessarily a divergent term, which StrictCheck implements as a special exception. If a computation tries to evaluate `thunk`, then the entire computation becomes the exceptional value—`thunk`. For example, if we apply `length` to the partial value `thunk : 2 : thunk`, we would get back a `thunk` of type `Int` since `length` needs to keep evaluating the list until it reaches the `[]` constructor.

Although `thunk` is a divergent term, StrictCheck does not lose any type safety in the testing process since it immediately converts the partial values back to demand values once a specification returns. If a specification accidentally evaluated `thunk`, then the testing infrastructure would simply report a failed test case due to mismatch between predicted and observed demands. We acknowledge that this design makes writing specifications more error prone than they would be if instead they explicitly manipulated `Demand` values and explicitly handled at every pattern-match the possibility of encountering a `thunk`. However, we believe this weakness to be outweighed by the expressive benefit of manipulating `Demand` values with any available function defined over the original data type, and this conjecture has in our experience proven to be accurate.

5 GENERATING LAZY FUNCTIONS

So far, we've seen how StrictCheck can test demand specifications for functions such as `take` and `rotate`. However, Haskell is a higher-order language—and we would like not to be limited to testing specifications only of such first-order functions. However, testing the strictness of higher-order functions comes with its own challenges. For example, consider the standard `map` function:

```
map :: (a -> b) -> [a] -> [b]
map _ []       = []
map f (x : xs) = f x : map f xs
```

Just as with any other function, testing a strictness specification for `map` requires us to generate random inputs for it. In this case, we need both to generate a list of elements `[a]`, as well as a random function `(a -> b)`. To generate such random functions, QuickCheck users can leverage the so-called `CoArbitrary` mechanism, which we will describe in more detail later in this section. However, as we will see, the functions produced by this method are necessarily strict. For testing functional correctness properties, this assumption is entirely reasonable: the strictness of generated Haskell functions does not generally impact results. However, in the context of StrictCheck, generating functions with observably different strictness is absolutely necessary!

5.1 Generating (strict) functions using QuickCheck

Before addressing this problem in random generation, we need to briefly describe the current approach used by QuickCheck and identify its limitations with regards to laziness. At a high level, a generator for some type `A` in Haskell is just a function from some type `g` of random seeds (borrowed from `System.Random` [Claessen and Palka 2013]) to `A`. Function types `A -> B` are no different. Intuitively, if you know how to generate elements of type `B`, and also how, given some random seed `g`, to produce a (unique) new seed `g'` based on some `A`, then you can combine the two to produce a generator for `A -> B`.

For example, consider the simple case of functions with type `Bool -> Bool`. The generator for the result type is straightforward; to generate a value of type `Bool` you just use the random seed to pick between `True` and `False`. In QuickCheck, such choices are encoded using combinators, like `elements`:

```
genBool :: Gen Bool
genBool = elements [True, False]
```

The `Gen` monad is just a `Reader` monad that hides low-level manipulations of the random seed. For the purposes of this paper, we could treat `Gen A` as isomorphic to `g -> A`.

Under the hood, the main operation on random seeds is *splitting*:

```
split :: g -> (g,g)
```

which, according to its documentation, creates "two [statistically] distinct random number generators." This operation naturally gives rise to a way to obtain new random seeds given an input `Bool`. We split the original random seed: if the input `Bool` is `True`, we keep the first component of the pair, and if it is `False`, the second one. This ensures that subsequent uses of the resultant seed will produce different random outputs, dependent on the `Bool` in question.

```
modifyBool :: g -> Bool -> g
modifyBool g b = let (g1,g2) = split g in
                 if b then g1 else g2
```

While `modifyBool` is what a QuickCheck user would morally write, QuickCheck does not actually expose its random seed representation. Instead, it exports a function called `variant`, which takes

an integer argument and a generator, and alters the generator by supplying a new seed in similar fashion:

```
variant :: Int -> Gen a -> Gen a
```

Our boolean seed-modification function, often called `coarbitrary` in parallel to QuickCheck's `arbitrary` for generation, is then written as:

```
coarbitraryBool : Bool -> Gen a -> Gen a
coarbitraryBool True  g = variant 0 g
coarbitraryBool False g = variant 1 g
```

We can now combine `coarbitraryBool` and `genBool` to almost obtain a generator for functions `Bool -> Bool`:

```
boolGenBool :: Bool -> Gen Bool
boolGenBool b = coarbitraryBool b genBool
```

The final piece of magic that QuickCheck provides is `promote`, a function that can be used to capture a random seed in a closure and repeatedly use it, thereby lifting `boolGenBool` to an actual generator for functions.

```
promote :: Monad m => m (Gen a) -> Gen (m a)

genBoolBool :: Gen (Bool -> Bool)
genBoolBool = promote boolGenBool
```

It is not hard to see how this approach can generalize to other types. Each different constructor of a type corresponds to a different integer argument given to `variant`, while fields of a constructor translate to recursive calls to `coarbitrary`. QuickCheck provides a typeclass infrastructure for generation and seed-modification (`Arbitrary` and `CoArbitrary` respectively) to facilitate writing such generators.

```
class Arbitrary a where
  arbitrary :: Gen a

class CoArbitrary a where
  coarbitrary :: a -> Gen b -> Gen b
```

Because the structure of a correct `CoArbitrary` instance is mechanically derivable from the shape of a data type, QuickCheck uses generic programming to provide a default `CoArbitrary` instance for most algebraic data types.

5.2 Generating lazy functions with QuickCheck

Unfortunately, this approach only generates fully strict functions. The default `coarbitrary` method will recursively pattern match on every subfield of an input value, which ends up evaluating it completely. [LEO: There is a subtle point to be made here about interleaving consume and produce if we want to]

For most uses of QuickCheck, this is acceptable—if not outright desirable—since requiring the strict consumption of the function's input allows the maximum amount of randomness to be introduced into the function's output. When testing laziness properties, however, we should like to test that our specification holds when a function's higher-order arguments have arbitrary strictnesses. If we only test our specification by generating fully-strict functions, we will very likely miss important test failures. For instance, our test suite would be unable to distinguish between the standard implementation of `map`, and a function of the same signature which, in addition to

mapping the function across the input list, fully normalizes every element of that list while so doing.

It should therefore be clear that proper testing of higher-order strictness specifications will require us not only to generate random functions, but to generate random functions with *random strictness*. We cannot use `CoArbitrary` for this—instead, we must design a new mechanism.

An inspiration for our design will come from an alternate way of viewing partially-lazy functions. We call a lazy function *continuous* if evaluating any finite prefix of its output forces the evaluation of only a finite prefix of its input. Consider the function `zip`:

```
zip :: [a] -> [b] -> [(a, b)]
zip []       _        = []
zip _        []       = []
zip (a : as) (b : bs) = (a, b) : zip as bs
```

This function is continuous, because evaluating a finite part of its output list will only require the corresponding pieces of its input lists to be evaluated—even if all the lists concerned are infinite.

An example of a *non*-continuous function is `sum`:

```
sum :: [Int] -> Int
sum []       = 0
sum (n : ns) = n + sum ns
```

This function is not continuous, because evaluating its output integer requires its entire input list to be evaluated—and if that list is infinite, evaluating its output to weak-head normal form will require strictly consuming an entire infinite data structure.[1] As such, we can view a continuous function as a procedure which alternates between evaluating some finite amount of input and producing some finite amount of output, until its entire input has been consumed and its entire output produced.

In StrictCheck, we model this explicitly. For instance, consider the special case of functions `(Nat -> Nat)`; we will again generalize to functions at arbitrary types later. For the `Nat` type, we use the type of (lazy) Peano naturals.[2]

```
data Nat = Z | S Nat
```

A continuous function from `Nat` to `Nat` will, for each `S` or `Z` evaluated of the output, force the evaluation of a finite number of constructors of the input.

We can model this explicitly by defining a pair of functions on QuickCheck `Gens`—one for consumption of input, and one for production of output. To randomly consume a finite portion of a `Nat`, we make a choice: either we don't destruct any of it and instead immediately produce some amount of output, or we consume one constructor of input *and then* produce some output.

```
consumeNat :: (Gen a -> Gen a) -> Nat -> Gen a
consumeNat p n =
  oneof [ p (consumeNat p n)
        , case n of
            Z    -> variant 1 (fix p)
            S n' -> variant 2 (p (consumeNat p n'))
        ]
```

[1] Technical note: While it might seem a shortcoming that we do not generate fully strict functions such as `sum`, we do generate functions which are arbitrarily close *approximations* of such. If a strictness bug occurs *only* when a function is infinitely strict, testing for its presence is uncomputable. As long as a strictness bug can be exhibited for some arbitrary-but-finite demand, we can in principle uncover it, which is the best we can hope to do.

[2] The mathematically precise reader may note that these are actually *co*-natural numbers, as they permit an additional non-finite inhabitant `omega`—the `Nat` consisting of only `S` constructors, never reaching a `Z` constructor.

Dually, when we are producing an output of a function, we have three choices: we may punt on generating any output and instead consume more input, return a `Z` constructor (and thus consume no more input, as we have produced all of our output), or produce an `S` constructor, continuing on to (perhaps) consume more input.

```
produceNat :: Gen Nat -> Gen Nat
produceNat r =
  oneof [ r
        , return Z
        , S <$> r
        ]
```

At each point in this process, we can make the choice to consume zero or one pieces of input or produce zero or one pieces of output. This means it's possible for us to consecutively consume any finite number of pieces of input prior to consecutively producing any finite number of pieces of output. This alternation continues until we randomly choose to finish producing output.

Notice that we define `produceNat` and `consumeNat` using open recursion, each expecting an input which we call recursively to produce more output or consume more input, respectively. Because of this, we can interleave the production of output and the consumption of input by tying the two functions together.

```
natFunction :: Gen (Nat -> Nat)
natFunction = promote (consumeNat produceNat)
```

As with `CoArbitrary`, `promote` takes a function returning a generator `(Nat -> Gen Nat)` and "lifts" it to become a generator returning functions.

While this approach works well to generate functions consuming `Nat`s, it doesn't generalize to the case of functions consuming more complex data types. Consider how we might attempt to implement a random generator for functions on (unlableled) binary trees.

```
data Tree = Leaf | Node Tree Tree
```

Writing a "produce" function for `Tree`s is straightforward enough. We follow the same pattern as we used for producing `Nat`s, choosing between immediately consuming more input, and producing any one of the possible constructors.

```
produceTree :: Gen Tree -> Gen Tree
produceTree r =
  oneof [ r
        , return Leaf
        , Node <$> r <*> r
        ]
```

We encounter a problem, though, when we try to write the corresponding "consume" function. As before, we wish to make a choice between immediately producing some output, and consuming input before producing any more output. Following the pattern established in the case of `Nat`, we almost can write the appropriate function—but not quite.

```
consumeTree :: (Gen a -> Gen a) -> Tree -> Gen a
consumeTree p t =
  oneof [ p (consumeTree p t)
        , case t of
            Leaf ->
              variant 1 (fix p)
```

```
            Node l r ->
              variant 2 (p (consumeTree p _____))
        ]
```

What should fill the blank above? In order for subsequent production of output to continue consuming input, the argument to the recursive call of `produce` must close over all the remaining data not yet consumed. However, there seems no easy way to accomplish this: we may either consume the left subtree or the right subtree as we continue to produce output, but we have no ability to consume pieces of both, as we can only pass one generator into `produce`.

In order to generate functions which consume input in truly arbitrary patterns, we need a way of passing *multiple* yet-to-be-consumed inputs forward into the computation. We can accomplish this by *homogenizing* our representation of inputs. When we produce outputs, the exact value of the consumed portion is not important. In the unlabeled binary tree example, we don't really care what the concrete values of `l` and `r` are; we only need the randomness derived from `l` and `r`. This motivates the following representation for the randomness derivable from data structures, which we call `Input`.

```
data Input =
  Input (forall a. Gen a -> Gen a) [Input]
```

Values of type `Input` are rose trees whose nodes contain randomness derived from different parts of the consumed data. We represent this randomness as perturbation function values of type `forall a. Gen a -> Gen a`: just like QuickCheck's `variant` combinator, these values perturb the randomness in a generator.

We derive `Inputs` from values by means of the `consume` function, defined in the `Consume` typeclass:

```
class Consume a where
  consume :: a -> Input
```

By way of example, here are the `Consume` instances for `Bool`, `Either a b`, and `(a, b)`:

```
instance Consume Bool where
  consume True  = Input (variant 0) []
  consume False = Input (variant 1) []

instance (Consume a, Consume b) => Consume (Either a b) where
  consume (Left a)  = Input (variant 0) [consume a]
  consume (Right b) = Input (variant 1) [consume b]

instance (Consume a, Consume b) => Consume (a, b) where
  consume (a, b) = Input (variant 0) [consume a, consume b]
```

In the same way as `CoArbitrary` instances, `Consume` instances have exactly one correct implementation, and this can be mechanically derived from the shape of the data typeto be `consume`d. In StrictCheck, we derive these instances automatically using generic programming.

Consuming functions (as well as other opaque types such as `IO` computations) merely entails forcing their evaluation, returning an empty `Input`.

```
instance Consume (a -> b) where
  consume !_ = Input (variant 0) []
```

Importantly, instances of `Consume` must be precisely the right strictness: they should require the top-most constructor of the `consume`d value to be evaluated *and nothing more*, before returning the

`Input` constructor. That is, as we evaluate an `Input`, the value from which it was created will be evaluated to precisely the same degree as the `Input`. This enables us to randomly evaluate a value in an incremental fashion, extracting randomness corresponding to the information contained in the prefix of that value which we have, by proxy, evaluated.

A single `Input` may contain multiple child `Inputs`—one for each field of the value from which that `Input` was created. As we produce the output of our random function, we interleave the consumption of not merely a single `Input`, but a list of `Inputs`, each of which corresponds to a currently-unevaluated "leaf" of some original input to the function. The dual to `Consume`, therefore, is `Produce`:

```
class Produce b where
  produce :: [Input] -> Gen b
```

An instance of `Produce` for some type with no fields should be equivalent to that type's `Arbitrary` instance. For example, the instance of `Produce` for `Bool`s is merely:

```
instance Produce Bool where
  produce _ = elements [False, True]
```

However, when a data typehas fields, we need to make sure that a function producing that type is able to consume some arbitrary prefix of its input prior to producing the value within a field. To do this, we need to introduce a new helper function, `build`.

```
recur :: Produce a => [Input] -> Gen a
```

The implementation of `recur`:

(1) Selects an arbitrary set of `Inputs` from the list provided it,
(2) Evaluates them to an arbitrary degree, and
(3) Collects the revealed randomness (i.e. the functions (`forall a. Gen a -> Gen a`) contained in the prefix of the input it just evaluated, then
(4) Uses this randomness to vary the current random seed, and
(5) Makes a recursive call to `produce`, feeding it the new list of yet-unevaluated leaves of the `Inputs`.

The implementation of `recur` used in StrictCheck is somewhat sophisticated—it uses a geometrically bounded depth-first traversal to ensure that it generates continuous functions almost-everywhere (that is, with 100% probability), as well as biases functions producing products toward consuming divergent paths through their input.

Using `recur`, we can write instances of `Produce` which properly interleave consumption with production. By way of example, here are the `Consume` instances for `Either a b` and `(a, b)`:

```
instance (Produce a, Produce b) => Produce (Either a b) where
  produce inputs =
    oneof [ Left  <$> recur inputs
          , Right <$> recur inputs
          ]

instance (Produce a, Produce b) => Produce (a, b) where
  produce inputs = do
    a <- recur inputs
    b <- recur inputs
    return (a, b)
```

Notice that we feed the same `[Input]` to the recursive calls generating `a` and `b`. This is because, in general, it's quite possible for two produced elements of a pair to require the evaluation of some shared portion of the inputs to a function. We wish to allow this behavior in our generated functions, and furthermore, we wish not to bias the order of evaluation toward one component of the pair or the other.

In order to `produce` random functions, we make use once more of `promote`. Here, producing a function entails returning a function which `consumes` its argument and adds the resultant `Input` to the `[Input]` fed through the production of the final output of the function.

```
instance (Consume a, Produce b) => Produce (a -> b) where
  produce inputs =
    promote $ \a ->
      produce (consume a : inputs)
```

By recursion, this instance generalizes to the production of functions of arbitrary arity.

Now, we can generate random functions with random strictness, by using `produce` to generate a function, feeding it the empty list of inputs to start. [KWF: Rephrase?]

```
lazyFunction :: (Consume a, Produce b) => Gen (a -> b)
lazyFunction = produce []
```

In fact, this generator is a special case—we can generate anything with a `Produce` instance in exactly the same way. When testing with StrictCheck, we generate all inputs (including first-order ones) using the polymorphic generator `lazy`:

```
lazy :: Produce a => Gen a
lazy = produce []
```

And with that, we can test the laziness properties of all functions, of any order we please.

6 RELATED WORK

StrictCheck is the first framework in Haskell that allows property-based testing of laziness with exact specifications. There is a rich body of work on property-based testing, observing lazy programs, and working with partial values in Haskell. We discuss their relationship to StrictCheck, and comment on StrictCheck's novelty compared to existing work.

Property-based testing. QuickCheck [Claessen and Hughes 2000] focuses on testing functional properties of Haskell programs through user-provided property specifications. StrictCheck uses QuickCheck's type-based random generator as its backend for generating random inputs, but focuses on testing strictness behavior (traditionally considered a non-functional property) against user-provided specifications. StrictCheck also provides a more flexible variant of the `CoArbitrary` typeclass from QuickCheck which is capable of generating functions with random strictness behavior.

SmallCheck and Lazy SmallCheck [Runciman et al. 2008] are similar to QuickCheck, and provide property-based testing of functional properties against a specification. They differ from QuickCheck by implementing a strategy of exhaustively checking properties on inputs up to a certain depth instead of, as QuickCheck does, randomly sampling from a large space of values. Lazy SmallCheck allows property-based testing by generating partial values as inputs, but there the purpose is to verify functional properties on partial inputs. Lazy SmallCheck does not provide exact strictness specification in the style of StrictCheck.

Observing Haskell programs. In [Gill 2001], Gill develops an (unnamed) library for observing the evaluation of Haskell values. Gill's work uses a similar technique of injecting effectful code into values through `unsafePerformIO`, but his work only records the evaluated values as strings, while StrictCheck uses a typed approach that fully reifies the evaluated structure as a first class value which may be manipulated by other Haskell programs. Gill's library provides programmers with runtime information to aid in manual debugging of lazy functional programs, whereas StrictCheck provides automated testing of strictness behavior on such programs.

Haskell libraries such as `ghc-heap-view` [Breitner 2014] and `Vacuum` [Morrow and Seipp 2009] provide functions for inspecting the heap representation of Haskell values at runtime. These libraries can reify pointer graphs describing the current heap state of the inspected value. StrictCheck's observation mechanism is different from these libraries in that we observe the strictness of a function rather than the evaluation structure of a data value. StrictCheck's observe mechanism is referentially transparent, whereas these libraries operate in the `IO` monad.

Programming with partial values. Danielsson et al. developed the `ChasingBottoms` library in Haskell in order to study program verification under the context of partial and infinite values [Anders Danielsson and Jansson 2004]. The `ChasingBottoms` library provides a set of functions to test whether a Haskell value is divergent. [KWF: Also cite `Control.Spoon`, which is newer, but effectively the same thing.] StrictCheck uses similar techniques to convert reified demand values to and from partial values, to the end of re-using existing functions while writing specifications.

Testing strictness. Chitil published a framework for testing the strictness of Haskell functions also named StrictCheck [Chitil 2011]. Chitil's work develops a notion of "least-strictness", and tests whether a function is least-strict by feeding partial values as inputs to the function. This only tests a very specific laziness property of Haskell functions, while our work allows users to precisely specify and test the laziness of a function. StrictCheck also generalizes to higher-order functions, a domain not addressed in this prior work.

7 CONCLUSION AND FUTURE WORK

In this paper, we identified a class of dynamic properties of interest in the functional programming community that are entirely out-of-scope for traditional testing: laziness properties. We described an approach to observe, specify and automatically test such properties and implemented it in an openly available Haskell library called StrictCheck.

Since this approach is entirely novel, there is a lot of space for improvement. In particular, both the specification language and the user interface can be improved, as StrictCheck is used to test more applications. In the future, we would also like to synthesize specifications of laziness for functions based on their observed dynamic behavior.

REFERENCES

Nils Anders Danielsson and Patrik Jansson. 2004. Chasing Bottoms - a Case Study in Program Verification in the Presence of Partial and Infinite Values. (05 2004).

Joachim Breitner. 2014. ghc-heap-view: Extract the heap representation of Haskell values and thunks. https://hackage.haskell.org/package/ghc-heap-view. (2014). [Online; accessed 14-March-2018].

Olaf Chitil. 2011. *StrictCheck: a Tool for Testing Whether a Function is Unnecessarily Strict.* Technical report 2-11. University of Kent, Kent, UK. 182–196 pages. http://kar.kent.ac.uk/30756/

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. https://doi.org/10.1145/357766.351266

Koen Claessen and Michal H. Palka. 2013. Splittable Pseudorandom Number Generators Using Cryptographic Hashing. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13).* ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2503778.2503784

Edsko de Vries and Andres Löh. 2014. True sums of products. In *WGP@ICFP.*

A. Einstein, B. Podolsky, and N. Rosen. 1935. Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? *Physical Review* 47 (May 1935), 777–780. https://doi.org/10.1103/PhysRev.47.777

Andy Gill. 2001. Debugging Haskell by Observing Intermediate Data Structures. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 1. https://doi.org/10.1016/S1571-0661(05)80538-9 2000 ACM SIGPLAN Haskell Workshop (Satellite Event of PLI 2000).

Ralf Hinze. 2000. Memo Functions, Polytypically!. In *Proceedings of the 2nd Workshop on Generic Programming, Ponte de.* 17–32.

J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. Springer-Verlag, 124–144.

Matt Morrow and Austin Seipp. 2009. vacuum: Graph representation of the GHC heap. https://hackage.haskell.org/package/vacuum. (2009). [Online; accessed 16-March-2018].

Chris Okasaki. 1995. Simple and efficient purely functional queues and deques. *JOURNAL OF FUNCTIONAL PROGRAMMING* 5, 4 (1995), 583–592.

Chris Okasaki. 1998. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08).* ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1411286.1411292