Keep Your Laziness in Check

KENNETH FONER, University of Pennsylvania, USA
HENGCHU ZHANG, University of Pennsylvania, USA
LEONIDAS LAMPROPOULOS, University of Pennsylvania, USA

We introduce StrictCheck: a property-based random testing framework for observing, specifying, and testing the strictness of Haskell functions. Strictness 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 within Haskell.

Testing strictness requires us to 1) precisely specify the strictness of functions, 2) efficiently observe the evaluation of data structures, and 3) correctly generate functions with random strictness. We tackle all three of these challenges, designing an efficient generic framework for precise dynamic strictness testing. StrictCheck can specify and test the strictness 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 provide an expressive but low-level specification language as a foundation upon which to build future higher-level abstractions.

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 → Software testing and debugging**; *Software maintenance tools*;

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

ACM Reference Format:
Kenneth Foner, Hengchu Zhang, and Leonidas Lampropoulos. 2018. Keep Your Laziness in Check. *Proc. ACM Program. Lang.* 2, ICFP, Article 102 (September 2018), 30 pages. https://doi.org/10.1145/3236797

1 INTRODUCTION

Lazy evaluation gives great power to functional programmers, enabling them to program with infinite data structures [Abel et al. 2013], 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 fall victim to 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, a program with undesired strictness properties is often very similar to a program with the correct ones. 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

Authors' addresses: Kenneth Foner, University of Pennsylvania, USA, kfoner@seas.upenn.edu; Hengchu Zhang, University of Pennsylvania, USA, hengchu@seas.upenn.edu; Leonidas Lampropoulos, University of Pennsylvania, USA, llamp@seas.upenn.edu.

2475-1421/2018/9-ART102
https://doi.org/10.1145/3236797

semantic divergence nor poor performance are necessary results of a strictness bug—so neither conventional property-based testing nor benchmarking are sufficient to fully diagnose these issues.

How, then, can we test and prevent these 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 strictness bugs in Haskell programs.

StrictCheck allows the programmer to

- **Specify** strictness precisely, describing exactly what portion of its inputs a function is meant to evaluate,
- **Observe** the strictness of a function and reify this into a Haskell data structure using only a constant factor of overhead, and
- **Test** whether a function matches its strictness specification, reporting a minimal reproducible counterexample 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, and GADTs, given the appropriate typeclass instances.

1.1 Strictly Defining Our Terms

In a lazy language like Haskell, a value-yielding computation—a *thunk*—is executed only when that value is itself needed for some further computation. Consider the following program, where `f` and `g` are some existing functions:

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

The program above `prints` all of `f`'s result. To produce all of `f`'s result, we need to evaluate some portion of `g`'s result. In turn, to produce that portion of `g`'s result, we need to evaluate some portion of `list`. Lazy evaluation proceeds by tracing the transitive dependencies of a series of such nested *evaluation contexts*.

A *demand* is the portion of a value required by some context. One possible demand on the list `[1, 2, 3, ..]` above is `(_ : 2 : _)`. This notation says that some context evaluated the first two `(:)` constructors, as well as the second element of the list (the integer `2`). The underscores indicate that neither the first element of the list nor the tail of the list were evaluated by the context. For our purposes, a demand represents a portion of a particular value, describing what actually happened to that value in one specific evaluation context. At a high level, you could view a demand as a pattern to match against: the pattern `(_ : 2 : _)` evaluates the first two cons cells of a list, its second element, and nothing else.

The *strictness* of a function is a description of the demand that function exerts on its inputs, given a particular demand on its output and the particular values of its inputs. Strictness 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 evaluating the second if the first was was `True`; etc. Notice that any implementation of `f` must be functionally equivalent to the constant function `\x y → ()`, modulo its behavior on `undefined` inputs. That is, two functions which are equivalent for all total inputs may have distinguishable strictness.

1.2 Describing Strictness, and Testing It!

A *strictness specification* is a precise characterization of a function's strictness. Such a specification is a function itself, taking as input a particular demand on some function's output as well as a list of the function's inputs, and returning a list of predicted demands on those inputs. For example, if we have some function

```
f :: a → b → ... → z → result
```

then we might specify its demand behavior using another 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, whose architecture is shown below in Figure 1.

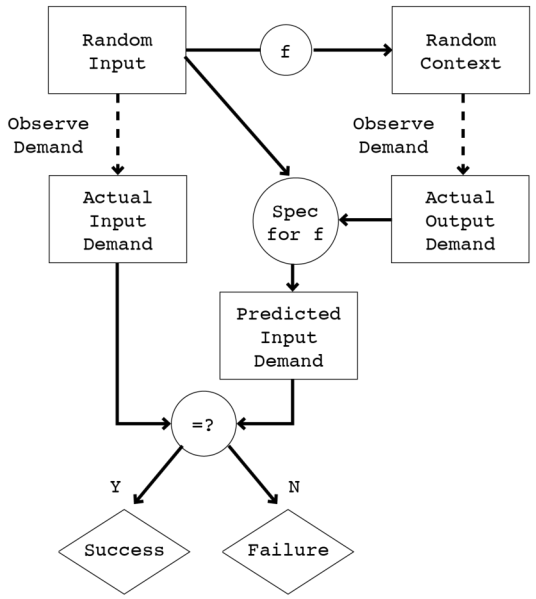

Fig. 1. Architecture of StrictCheck

In order to check whether a specification `spec_f` correctly predicts the demands on the inputs of a function `f`, we will

- generate random input(s) to `f`: `(a, b, ..., z)`,
- generate a random evaluation context `c` for the result of `f`,
- evaluate `f` in the context `c` by forcing evaluation of `(c (f a b ... z))`,
- observe the demand `d` which the context `c` exerted on the result of `f`,
- observe the demands `(da, db, ..., dz)` induced upon the inputs `(a, b, ..., z)`, and
- 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.

In the next section we show how StrictCheck concretely represents specifications, using as an example the function `take :: Int → [a] → [a]`.

In the rest of the paper, we present our technical contributions:

- We develop a method for observing the strictness of a function at runtime, and describe how we use generic programming to apply this mechanism to all algebraic data types (Section 3). Our technique uses only a small constant factor of overhead compared to the cost of evaluating a function without observing its strictness.
- We introduce a *precise* approach for specifying the strictness of functions. Such specifications are expressive enough to capture exactly how much evaluation occurs within a data structure during the evaluation of a function (Section 4). The current specification framework is relatively low-level, but crafted to allow future higher-level abstractions to build upon it, without the need to modify the core StrictCheck library.
- We work through a more complex example, Okasaki's purely functional queues [Okasaki 1995], demonstrating some of the difficulties of programming with laziness and how our approach can be used to alleviate them (Section 5).
- As a side contribution, we identify a limitation of QuickCheck's technique for generating random functions: namely, it only generates strict functions. We develop an alternate technique to generate functions with arbitrary strictness (Section 6).
- We implement StrictCheck, a Haskell framework for automatically testing functions against such specifications, available today at https://hackage.haskell.org/package/StrictCheck.

Section 7 discusses related work. We conclude and draw directions for future work in Section 8.

2 STRICTNESS SPECIFICATIONS, CONCRETELY

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 its length:

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

In StrictCheck, a value of the `Spec` type represents a strictness specification for some function. The specification of `take` has the type

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

This tells us that `take_spec` specifies a function with `take`'s type signature, taking two arguments of types `Int` and `[a]` and returning a list of type `[a]`.

2.1 Writing Specs in a Curry

In general, the `Spec` type can represent specifications for functions of any arity, with arguments and results of any type, including higher-order ones. Our first draft of the `Spec` type used heterogeneous lists to generalize over the arity of the function being specified. While this worked, the syntactic overhead of pattern-matching and constructing such lists made specifications more difficult to read. StrictCheck simplifies this syntax using *arity-polymorphic currying*. We briefly describe this technique before moving on with our example.

2.1.1 Arity-Polymorphic Currying. To generalize currying to any arity, we define the type family `(args ··→ result)` to compute a curried function type taking `args` as inputs and returning `result`.

```
type family (args :: [Type]) ··→ (result :: Type) where
  '[]        ··→ result = result
  (a : args) ··→ result = a → (args ··→ result)
```

In the inverse direction, we define two type families `Args` and `Result` to compute respectively the list of arguments to a curried function, and its (non-function-type) result. For any type `f`, observe that `f ~ (Args f ··→ Result f)`.

```
type family Args (f :: Type) :: [Type] where
  Args (a → rest) = a : Args rest
  Args x          = '[]

type family Result (f :: Type) :: Type where
  Result (a → rest) = Result rest
  Result r          = r
```

To actually curry functions, we need to pick some particular concrete type of heterogeneous lists (`List :: [Type] → Type`). In terms of this type, we define the `Curry` typeclass to curry/uncurry with our `List` data type:

```
class Curry (args :: [Type]) where
  uncurry :: (args ··→ result) → (List args → result)
  curry   :: (List args → result) → (args ··→ result)

instance Curry '[] where ...
instance Curry xs ⇒ Curry (x : xs) where ...
```

StrictCheck uses this machinery in several ways, including its specification interface.

2.1.2 Variadic Curried Specs. Because StrictCheck's `Spec`s are defined in terms of variadic curried functions, the test author does not need to explicitly manipulate heterogeneous lists. Instead, they work within the context of a `Spec`, indexed by the argument and result types of the function it is meant to correspond to:

```
newtype Spec (args :: [Type]) (result :: Type)
  = Spec (forall r. (args ··→ r) → result → (args ··→ r))
```

A `Spec` contains a specification function like those in Section 1.2, whose number of arguments is determined by the arity of the function `f` it specifies. The wrapped specification function takes, in order:

- a curried continuation we will name `predict`, accepting all of `f`'s argument types in order,
- an implicit representation of a demand on `f`'s result (as described in Section 2.1.4), and
- all of `f`'s original argument types in order.

The specification function will call `predict` on a representation of the demands which the specification author predicts `f` will exert on its inputs, given the demand exerted on `f`'s output. In Section 2.2, we'll see examples of such specifications in action.

2.1.3 Another Flavor of Curry. As we saw above, a `Spec` is indexed by two types: a *list of arguments* to the function specified, and the *result* type of that function. We might assume that a function type like `(A → B → C)`, with more than one argument, corresponds to multiple equivalent `Spec` types: `(Spec '[A, B] C)`, or `(Spec '[A] (B → C))`.

Perhaps surprisingly, a useful `Spec` type should *never* have a function as its `result` type. Recall from Section 1.2 that a strictness specification maps from a demand on a function's result, as well as a list of its inputs, to a predicted demand on those inputs. Suppose we tried to specify the curried function

```
zip :: [a]                  -- first argument
    → ([b] → [(a, b)])      -- result (partially applied function)
```

in terms of a mapping

```
zip_spec
  :: Demand ([b] → [(a, b)])  -- demand on (partially applied) result
  → [a]                       -- first argument
  → Demand [a]                -- demand on first argument
```

Any such specification would not be able to precisely express `zip`'s strictness. Why? There are only two possible demands on a function: evaluated, or not evaluated. Yet the shape of the demand on `zip`'s first argument `[a]` depends on how much of its result `[(a, b)]` is evaluated by some context. A specification like this can be predicated only on whether or not the result function `[b] → [(a, b)]` is itself evaluated—which isn't enough information to make an exact prediction of the demand on `zip`'s input `[a]`.

Extending this reasoning, any exact specification of a curried function with non-trivial strictness is only expressible by uncurrying that function. In StrictCheck, types like `(Spec '[A] (B → C))` are prohibited in favor of the strictly more expressive form `(Spec '[A, B] C)`. This conversion is handled automatically by the library; the user doesn't need to manually uncurry functions to specify them.

2.1.4 Demands as Ordinary Values? In the executable examples to follow, notice that we will overload a value of some type `t` as *both* an ordinary value and *also* as a representation of a demand on some value of that type. This might be surprising—as discussed in Section 1.1, there are more demands on values of a type than there are mere values of that type!

We will fully resolve this apparent paradox in in Section 4, but it's worth defusing some of the suspense now: when writing a specification, we *embed* in some value of type `t` a representation of `(Demand t)` on that same type by stubbing out portions of that value with the specially "tagged" bottom value `(thunk :: forall a. a)`. In the context of a `Spec`, a total value of type `t` corresponds to the fully strict demand on that value, while a partial value containing one or more `thunk`s corresponds to some non-strict demand on that value.

2.2 Let's Check Some Specs

We're now ready to write a strictness specification for `take`. But what should that specification say? A good first guess might be that `take` will always evaluate its integer argument `n` regardless of the demand on its result, and that it will evaluate its list argument `xs` to the same degree as whatever demand is placed upon its result. Let's put these words into code (with type annotations for clarity):

```
take_spec_first_attempt :: Spec '[Int, [a]] [a]
take_spec_first_attempt =
  Spec $ \(predict :: Int → [a] → r)  -- called upon predicted demands on inputs
          (resultDemand :: [a])       -- given demand on result
          (n  :: Int)                 -- given input value
          (xs :: [a]) →               -- given input value
            predict n resultDemand    -- prediction of demands on inputs
```

Is this specification right? Using QuickCheck [Claessen and Hughes 2000] as its backend, StrictCheck generates random inputs to the function (integers `n` and lists `xs`), as well as random contexts in which to evaluate its result. It observes whether the evaluation induced upon the inputs to `take` exactly matches the prediction of the specification. Then, it shrinks any found counterexamples to a minimal form and reports them. To check our specification, we invoke StrictCheck from within GHCi:[1]

[1]Like QuickCheck, StrictCheck cannot test polymorphic functions, so we need to instantiate any type variables before testing them. Usually, the quickest way to do this is with explicit type application—as we do here to specialize `take` to `Int`.

```
ghci> strictCheckSpecExact take_spec_first_attempt (take @Int)
```

StrictCheck quickly informs us that our specification does not match the actual behavior of `take`, printing a diagram to illustrate the mismatch (Figure 2).

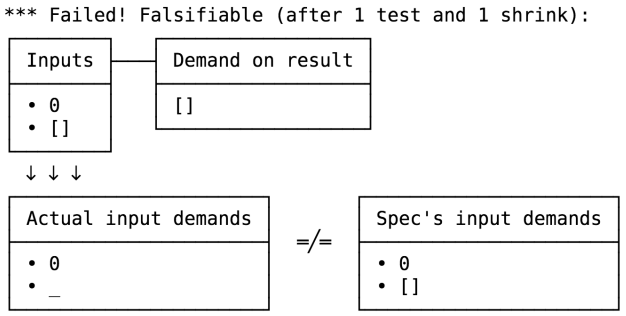

Fig. 2. Counterexample for `take` under the incorrect specification `take_spec_first_attempt`

This counterexample tells us that when we evaluated `(take 0 [])` and fully demanded its result, `[]`, our specification predicted that `take` would place that same demand, `[]`, on the input list. However, `take` placed *no* demand on the input list, because if `n` is zero, `take` immediately returns `[]` without evaluating the list at all.

While StrictCheck can't tell us whether a test failure indicates a bad specification or a bad implementation, in this case it is our specification is incorrect. To fix it, we need to account for what happens when `take` runs `n` down to zero before it gets to the end of the list. In that case, it doesn'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 `Spec` to account for this:

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

This specification passes StrictCheck's battery of random tests:

```
ghci> strictCheckSpecExact take_spec_correct (take @Int)
+++ OK, passed 100 tests.
```

Our correct specification of `take` can be used to catch a variety of errors. Suppose that instead of our original definition of `take`, we had written the seemingly equivalent function `take'`:

```
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'` has different strictness than `take` in a fairly subtle way. Can you spot how?

Testing `take'` against the correct specification for `take` reveals that its behavior does differ from the specification (Figure 3).

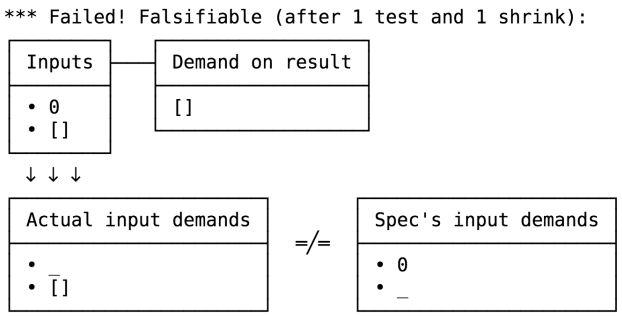

Fig. 3. Counterexample for `take'` under the correct specification `take_spec_correct`

This counterexample illustrates the strictness bug in `take'`. The function `take` evaluates `n` first, conditionally evaluating `xs` if `n` is positive; by contrast, the function `take'` always evaluates the beginning of `xs`, conditionally evaluating `n` if `xs` is non-empty. This means that `take'` evaluates one more constructor of its input list than `take` does.

While this bug might seem trivial, it is representative of a large class of errors, many of which can be quite serious. Accidental strictness like this can lead to arbitrary extra amounts of computation. Indeed, using an overly-strict function to define a self-referencing data structure can lead to non-termination—the too-strict function can cause a thunk within the structure to depend on its own value, forming a cyclic evaluation loop.

3 OBSERVING STRICTNESS IN HASKELL

In order to run the above tests, StrictCheck needed to compare the predictions of a `Spec` with its own observation of how much input was actually evaluated by a function. In this section, we provide an efficient mechanism to record this runtime behavior.

3.1 Getting Our Data in Shape

In Section 1.1, we saw that a demand on a value of some type is a sub-shape—specifically, a prefix—of that value. If we were to write down the types of demands on various ordinary types, we'd find ourselves repeating a pattern. The type `D` of demands on the values of some ordinary type `S`:

- has the same number of constructors as `S` does,
- has the same number of fields in each constructor as `S` does, and
- each field in `D` is either empty, or contains some demand on values of that field's type in `S`.

To actually represent such types, we define the type `Thunk`, to represent the possibility that some portion of a data structure has not been evaluated:

```
data Thunk a
  = T
  | E a
  deriving (Functor)
```

Using this type, we can say: the type of demands on values of some type is the same shape as the original type, but interleaving the `Thunk` type at every field, recursively throughout the entire type's shape.

Knowing this pattern, we could define the types of demands on our favorite data types, like this demand type for lists:

```
data ListDemand a
  = NilDemand
  | ConsDemand (Thunk a) (Thunk (ListDemand a))
```

If we were to do this, though, we'd quickly be overwhelmed with the amount of boilerplate required to define such types. What's more, we'd have to write still more boilerplate to allow the demand observation mechanism to manipulate these values. We already understand the relationship between these demand types and the ordinary types to which they correspond—we ought to use this knowledge to unify these ad-hoc definitions.

To do this, we take as inspiration the concept of *recursion schemes*, first described by [Meijer et al. 1991] and now implemented as the modern Haskell library `recursion-schemes` [Kmett 2017]. This work observes that any (regular) recursive data type is isomorphic to the explicit fixed-point of some corresponding *non-recursive* `Functor`. The shape of the correct such `Functor` should mirror the outermost layer of the original data type, with that `Functor`'s type parameter substituted wherever the original data type contains a recursive occurrence of itself. By factoring out the recursive structure of the original data type, the authors are able to define once and for all a suite of folds and unfolds for any data type that implements this isomorphism.

While this approach is elegant and useful, it is nevertheless not quite what we need. In particular, it is limited to expressing folds over the outermost recursive structure of the data type in question. In StrictCheck, we need to observe the evaluation of an entire data structure, into its innermost parts. The demand representation of a data type needs to interleave a `Thunk` at *every* field, not merely those which are a recursive occurrence of the original data structure.

To accomplish this, we will develop a new flavor of recursion scheme which works with structures shaped like some data type, but with a `Functor` (like `Thunk`) interleaved at every field. Our variation will let us express folds and unfolds across the entire structure of a data type, regardless of the types of its intermediate fields.

As in the original account of recursion schemes, we define a relationship between some possibly recursive type and some non-recursive type whose fixed point is shaped like the original type. We say that the `Shape` of a type `S`, parameterized by some `Functor f`, is the same shape as a value of `S`, but with `f` wrapped around every field in every constructor of `S`.

```
class Shaped a where
  type Shape a :: (Type → Type) → Type
  project :: (forall x. Shaped x ⇒ x → f x) → a → Shape a f
  embed   :: (forall x. Shaped x ⇒ f x → x) → Shape a f → a
```

We also require every `Shaped` type to define projection and embedding functions to witness the relationship between the original type and its `Shape`. To put all this together, let's consider the data type `D` and a valid corresponding `Shape`, `DShape`:

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

data DShape x y (f :: Type → Type)
  = AS (f x) (f Int)
  | BS (f y)
```

To `project` out the outermost layer of some `D` out, revealing its `Shape`, we take some polymorphic projection function

```
p :: forall x. Shaped x ⇒ x → f x
```

and call it on each field of the original value. Similarly, to embed an outer `Shape` back into the original type `D`, we take some polymorphic embedding function

```
e :: forall x. Shaped x ⇒ f x → x
```

and call it on each field of the given `Shape`:

```
instance Shaped (D x y) where
  type Shape (D x y) = DShape x y
  project p d = case d of
    A x i → AS (p x) (p i)
    B y   → BS (p y)
  embed e ds = case ds of
    AS fx fi → A (e fx) (e fi)
    BS fy    → B (e fy)
```

All of the above is non-recursive, dealing with only one layer of a value's structure at a time. We re-introduce recursion with an explicit fixed-point to recover the structure of our initial type. For some functor `f`, a value of type `f % a` is `Shaped` like a value of type `a`, but with an `f` interleaved at every field (and around the outside of the entire value as well):

```
newtype (f :: Type → Type) % (a :: Type) where
  Wrap :: f (Shape a ((%) f)) → f % a
```

Unlike the isomorphism in [Meijer et al. 1991], though, there may be *more* (or fewer) values of this fixed-point type than of the original type, depending on the functor `f` we interleave.

The methods of `Shaped` can be used to lift `project` and `embed` to operate recursively over entire interleaved structures. We call these functions `interleave` and `fuse`, respectively:

```
interleave :: (Functor f, Shaped a) ⇒ (forall x. x → f x) → (a → f % a)
fuse       :: (Functor f, Shaped a) ⇒ (forall x. f x → x) → (f % a → a)
```

These are special cases of more general derived folds and unfolds.

As we noted before, a `Demand` recursively interleaves `Thunks` into the `Shape` of the original data type. We represent this as the type synonym

```
type Demand = (%) Thunk
```

Using the generic programming framework `Generics.SOP` [de Vries and Löh 2014], we provide a default definition for the entire `Shaped` class which works for any Haskell 2010 algebraic data type. A user of StrictCheck who wishes to test a function on their own data type `T` need only ensure that it implements the `Generic` typeclass, then declare an empty instance of `Shaped`:

```
instance Shaped T
```

An instance of `Shaped` for some data type is sufficient to derive observation, shrinking, and pretty-printing of demands upon its values. In the next section, we'll see how to use it for the most vital of these: observation.

3.2 Observation, Shapefully

Without privileged access to the runtime system, how can a Haskell program observe the evaluation of a piece of data? The "safe" subset of Haskell won't allow us to do that, so we'll need to use unsafe primitives which break referential transparency.[2] In the end, we will very carefully wrap up these unsafe pieces to define a pure function `observe` which reports the precise demands exerted during

[2]We must take care to prevent compiler optimizations—many of which assume referential transparency—from breaking the functionality of these operations. For legibility, we elide these compiler hints.

some function's evaluation. To illustrate this safety boundary, we'll typeset referentially opaque operations in `red`.

Internally, we observe evaluation by mutating reference cells during the course of a thunk's evaluation. We use the referentially opaque primitive

```
unsafePerformIO :: IO a → a
```

to convert an `(IO a)` action into a thunk which secretly runs that action and returns its result—if and when it happens to be evaluated. This operation lets us create data whose value depends on its own order of evaluation.

```
entangle :: a → (a, Thunk a)
entangle a =
  unsafePerformIO $ do
    ref ← newIORef T
    return ( unsafePerformIO $ do
               writeIORef ref (E a)
               return a
           , unsafePerformIO $ readIORef ref )
```

Calling `entangle` on some value of type `A` returns a pair: a copy of the original `A`, and a `(Thunk A)`. The copy of the value returned by `entangle` is instrumented so that if and when it is evaluated, it will write its own value to a newly allocated reference cell. When the `Thunk` returned by `entangle` is evaluated, it reads from that same reference cell and returns whatever it contains at that moment. The effect of all this: if the returned `A` is evaluated before the corresponding `Thunk`, that `Thunk` will contain a copy of that value; if the `Thunk` is evaluated first, it will merely be an empty `T`. We can see this behavior by experimenting with `entangle` in GHCi:

```
ghci> x = "ab" ++ "cd"
ghci> (x', tx) = entangle x
ghci> x'
"abcd"
ghci> tx
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, its value will forever be `T`:

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

Instead of entangling just one value and a corresponding `Thunk`, we can use generic folds derived from `Shaped` to produce an instrumented copy of some value and an entire `Demand` reflecting the exact shape of its evaluated prefix.

To define this operation, we need to introduce one more fold, a generalized `unzipWith`:

```
unzipWith
  :: (All Functor [f, g, h], Shaped a)
  ⇒ (forall x. f x → (g x, h x))
  → (f % a → (g % a, h % a))
```

Using `unzipWith` as well as the `fuse` and `interleave` functions described in Section 3.1, we can recursively entangle every piece of a value.

```
entangleShape :: Shaped a ⇒ a → (a, Demand a)
entangleShape =
  first (fuse runIdentity)
  . unzipWith entangle'
  . interleave Identity
  where
    entangle' :: Identity x → (Identity x, Thunk x)
    entangle' =
      first Identity . entangle . runIdentity
```

In the above, we use the `Identity` type

```
data Identity a
  = Identity { runIdentity :: a }
  deriving (Functor)
```

The `entangleShape` operation recursively unzips the structure of a value to create a mutable shadow of it. Evaluating any thunk within the resultant instrumented value triggers the update of a mutable pointer at the corresponding location in the returned `Demand`. When we evaluate the entangled `Demand`, we implicitly freeze this pointer structure, creating a reified representation of whatever sub-shape of the original value happens to be evaluated by the time the `Demand` itself was evaluated.

Although `entangleShape` is still referentially opaque, we can use it to define `observe1`, a pure function which observes the evaluation of a unary function on an input in a given context. We specify that context as a function `context :: b → ()`. This function will evaluate some part of the function's result before returning `()`. The caller of `observe1` can choose what `context` function to provide, thus determining what demand will be placed on the function's result during observation.

```
observe1 :: (Shaped a, Shaped b)
         ⇒ (b → ())              -- evaluation context for the result
         → (a → b)               -- function to be observed
         → a                     -- input to the function
         → (Demand b,            -- demand exerted by the context
            Demand a)            -- demand induced upon the input
```

To observe the evaluation of a function on some input and in some context, `observe1`:

(1) entangles the input to the function, yielding an instrumented `input'` and an observation of the demand upon it, `inputDemand`;
(2) entangles the result of the function when applied to that input, yielding an instrumented `result'` and an observation of demand upon it, `resultDemand`;
(3) evaluates the instrumented result `result'` in the evaluation `context`; and
(4) returns the demand on the result and the demand this has induced upon the inputs.

```
observe1 context function input =
  let (input', inputDemand)  =
        entangleShape input              -- (1)
      (result', resultDemand) =
        entangleShape (function input')  -- (2)
  in let !() = context result'           -- (3)
  in (resultDemand, inputDemand)         -- (4)
```

For example, `observe1` can show us that `reverse` is spine-strict in its input list when its result is evaluated to weak-head normal form:

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

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

To observe the evaluation of functions with more than one argument, StrictCheck provides the more general function `observe` which uses the variadic currying machinery in Section 2.1.

```
observe :: (All Shaped (Args function),
            Shaped (Result function),
            Curry (Args function))
        ⇒ (Result function → ())
        → function
        → Args function
        ··→ (Demand (Result function), NP Demand (Args function))
```

Recall that `(args ··→ result)` calculates the curried function type with arguments `args`, returning `result`. The type `(NP Demand (Args function))` is a heterogeneous list, each of whose elements is the induced demand observed on the function's corresponding argument. To extract these individual input demands, we pattern-match on its constructors `(:*)` and `Nil` .

Using `observe`, let's examine the strictness of a function with multiple arguments. Consider the function `productZip`:

```
productZip :: [Int] → [Int] → [Int]
productZip xs ys =
  zipWith (*) xs ys
```

From GHCi, we observe that in the fully strict evaluation context `normalize`,[3] the function `productZip` is asymmetrically strict:

```
ghci> (result, (xs :* ys :* Nil)) = observe normalize productZip [10, 20] [30, 40]
ghci> printDemand result
300 : 800 : []
ghci> printDemand xs
10 : 20 : []
ghci> printDemand ys
30 : 40 : _
```

This tells us that `zipWith` is stricter on the left than the right: it lets the right-hand input list go untouched after it exhausts the left-hand input list.

Performance of `observe`. We claimed in the introduction that observing the strictness of a function has the same asymptotic performance as the original function. Previously described methods for strictness observation feed partially-`undefined` inputs to a function until observing that its result is not `undefined` [Anders Danielsson and Jansson 2004]. That method requires re-evaluating the function on many partial inputs. In the worst case, determining the function's exact strictness on some input may require evaluating it an exponential number of times (relative to the size of that input). By contrast, `observe` uses `entangleShape` to capture the exact evaluated shape of its input

[3] StrictCheck defines `normalize` as a generic operation in terms of the `Consume` typeclass from Section 6.2. In the context above, it suffices to think of `normalize` as the function `foldr seq ()`.

and result using only one invocation of the observed function. Exploring the resultant observed demand structure requires the same work to be done as would evaluating the function's ordinary result, plus a slight extra overhead to manage the mutable shadow copy of the inputs.

Purity of `observe`. While `entangle` and `entangleShape` are referentially opaque, `observe` is referentially transparent. As we saw, `entangle`'s results are dependent on the order in which you evaluate them. On the other hand, `observe` takes an evaluation context as *input* and crucially, does not return the instrumented result of the observed function—it returns just the reified `Demands`. This means that any evaluation of the result of the function occurs within `observe`, prior to it returning any observation of demand. As such, all the updates to the mutable shape have concluded prior to our exploration of the resultant (and now frozen) demands. Because of this, the order of evaluation outside `observe` does not influence the value of its output.We can safely expose `observe` to the outside world, and fear not that it will cause untoward side effects.

4 MANIPULATING DEMANDS IN SPECIFICATIONS

StrictCheck specifications need to manipulate demand values in order to produce their predictions. Section 3 introduced the type `(Demand :: Type → Type)`, which maps types to their demand types, and described how to obtain demand values by observing the evaluation of a function in some context. Now we focus on how to manipulate these values to craft precise specifications.

The writer of any kind of formal specification deserves a well-developed library for writing those specifications. If strictness specifications are to be phrased in terms of reified demands what tools will we have to manipulate these values as we build our specifications?

Take for example Haskell's standard `Data.List` module. Each of its many operations could be given an analogue which works over *demands* on lists, and each of these analogous operations might be useful to manipulate values of type `Demand [a]` in some specification. In principle, we could tediously implement each of these analogous operations from `Data.List` for `Demands` on lists. However, this places tremendous overhead on us, the library designers—and that's only considering specifications concerning demands on lists! We cannot reasonably expect to re-implement the rich Haskell ecosystem for our new landscape of demands.

Instead, it would be excellent if we could reuse existing functions to manipulate their corresponding demand values. However, demand values inhabit a different type. What's more, this difference in type is more than merely a difference in name: for almost all types `a`, there are many more inhabitants of `(Demand a)` than there are of `a`. How, then, could we automatically lift an existing function over some type `a` to work over the structurally distinct type `(Demand a)`?

4.1 Embedding Demands in Values

Our inspiration comes from realizing a combinatorial connection between demand types and lifted values. At every point in a `Demand` where a `Thunk` could be equal to `T`, there is a corresponding point in a corresponding non-total value where some sub-part could be `undefined`. If we count the elements of a type (taking into account that lazy values may contain embedded bottoms), we find that there are the same number of *partial* inhabitants of a type as there are *total* inhabitants of its type of demands.

Our specifications so far have already made use of the term `thunk`. As alluded to in Section 2.1.4, this term is a bottom value which, when evaluated, throws a pure exception:

```
thunk :: forall a. a
thunk = throw Unevaluated

data Unevaluated = Unevaluated
  deriving (Show, Exception)
```

Consider the demand value `(_ : 2 : _)` on lists of integers. Using `thunk`, we can represent this demand as itself a partial inhabitant of the type `[Int]`: `(thunk : 2 : thunk)`. Here the first `thunk` represents the unevaluated `Int`, and the second `thunk` represents the unevaluated tail of the list. We can manipulate this embedded demand representation with existing list operations; for example, applying `(take 1)` to this list would return the singleton list `[thunk]` containing the first `thunk`.

As we might now expect, StrictCheck can losslessly translate between these new "implicit" demands and the explicit demands we've seen already. In the forward direction, it converts from a `(Thunk a)` to a partial value of type `a` by `throw`ing the `Unevaluated` exception at each `T` we encounter:

```
fromThunk :: Thunk a → a
fromThunk (E a) = a
fromThunk T = thunk
```

In the reverse direction, it converts back from a partial value of type `a` to a `(Thunk a)` by `catch`ing the `Unevaluated` exception and converting it into a `T`:

```
toThunk :: a → Thunk a
toThunk a = unsafePerformIO $
  catch
    (let !_ = a in return (E a))
    (\(_ :: Unevaluated) → return T)
```

These functions are inverses, provided that every reified demand value is non-partial—that is, it contains no embedded exceptional values.

Within specifications, we may need to test whether a particular implicit demand is a `thunk` or not. Using `toThunk`, we define `isThunk` to detect this:

```
isThunk :: a → Bool
isThunk a =
  case toThunk a of
    T → True
    _ → False
```

Much as `entangleShape` lifts `entangle` to operate recursively over entire demands, we can lift the above pair of inverse functions to convert entire data structures between the two representations:

```
toDemand   :: Shaped a ⇒ a → Demand a
fromDemand :: Shaped a ⇒ Demand a → a
```

Given the current `Spec` interface to StrictCheck, most specification authors will never need to use these conversions between demand representations. During testing, explicit demands expressed in terms of `Shape`s are converted to implicit demands embedded in values and back again at the boundaries of a `Spec`.

4.2 Writing Specs Using Embedded Demands

Specifications written with the `Spec` type are provided partial values produced by `fromDemand` by default. In all of the examples shown in Section 5, we manipulate 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.

When programming with these partial list values, we must be mindful of where `thunk` may occur. As we have seen, `thunk` is a divergent term. If a computation tries to evaluate `thunk`, its entire result becomes equal to `thunk`. For example, if we apply `length` to the partial value `(1 : 2 : thunk)`, we would get back a `thunk` of type `Int` since `length` needs to keep evaluating the list until it reaches the `[]` constructor. Because of this, our examples will use the function `(cap :: [a] → [a])`, which

replaces a `thunk` at the tail of a list with `[]`. This can help us prevent spine-strict functions from behaving incorrectly when lifted to implicit demands. We will see details of `cap` in Section 5.

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. However, if the specification author wishes, parts of a `Spec` can also work with explicit `Demands`—the author merely needs to invoke the `toDemand` and `fromDemand` functions.

5 CASE STUDY: PURELY FUNCTIONAL QUEUES

In Section 2.2, we wrote a specification for the simple function `take`. Now, let's work through a more complex example: a specification for purely functional queues. Such queues are often implemented with two lists: a front list holding the queue elements in first-in-first-out 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, which takes $O(n)$ time to finish. While an amortized analysis shows that the expected time for dequeueing is constant, this is not true if the queue is used persistently, since this $O(n)$ operation could be repeated arbitrarily many times if references to old values of the queue are re-used. In his work on purely functional data structures [Okasaki 1998], Okasaki improves upon this design using a clever application of laziness. Instead of reversing the back list all at once, Okasaki's queue incrementally performs the reversal of the back list, at the same time incrementally appending that reversed list to the end of the front list [Okasaki 1995]. Okasaki's queue relies on a function called `rot`:

```
rot :: [a] → [a] → [a]
```

For any `front` and `back` lists, the expression `(rot front back)` is functionally indistinguishable from `(front ++ reverse back)`. However—unlike this naïve implementation—a queue implemented with `rot` avoids the occasional $O(n)$ reversal, cleverly using laziness to distribute the cost of reversing the `back` list across each pattern-match on the `front` list. We'll soon see how this works in Section 5.2.

When implementing such a queue, we might accidentally break Okasaki's invariants—yet because the two implementations differ only in their strictness, we would fail to discover the mistake through traditional property testing. Fortunately, we can use StrictCheck to write a specification that lets us automatically test the strictness invariants of `rot`.

Before working through the tricky example of Okasaki's queue, let's first build some intuition by understanding and specifying the naïve reverse-and-append implementation of queue rotation.

5.1 Warmup: Naïve Queues

As earlier, there's an easy way to implement the `rotate` function, though this naïve version's strictness properties leave something to be desired:

```
rot_naive :: [a] → [a] → [a]
rot_naive fs bs = fs ++ reverse bs
```

Before writing down a complete specification for it, it's easier to build up a picture of our predictions by considering several possible cases we could encounter while evaluating its result.

Notating Demands. Below, we'll continue to use the demand notation presented in Section 1.1. Additionally, we'll use metavariables $\mathtt{D_i}$ to name either a thunk or an evaluated field. For instance, we will allow the list demand $(\mathtt{D_1}\mathtt{:}\mathtt{D_2}\mathtt{:\_})$ to denote several simultaneous possibilities, including the above 2-cons demand `(_:_:_)`, a similar demand where first two values are evaluated `(1:2:_)`, or a mix of both. This notation is useful to illustrate the relationship between two different demands.

For instance, we could represent that some list demand is the reversal of another list demand by writing the first as `(D1:D2:D3:[])` and the second as `(D3:D2:D1:[])`.

We will show the relationship of several sets of possible demands to `rot_naive` by arranging them into a table, using metavariables `D1`, `D2`, etc. to show how they are connected. For example, consider what happens when we make some demand `(D1:D2:_)` upon the result of calling `rot_naive` on the lists `[1,2,3]` and `[6,5,4]`:

Inputs	Output	Demand On Output	Demand On Inputs
`fs = [1,2,3]` `bs = [6,5,4]`	`[1,2,3,4,5,6]`	`D1:D2:_`	`fs = D1:D2:_` `bs = _`

In this table, the first 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]`. By using the metavariables `D1` and `D2`, we represent several possible observations in a single table: the third column shows the demand exerted on the output, while the last column shows the demands on the inputs induced by that output demand. 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. For example, if the output demand is `(1:2:_)`, only the first two `(:)` constructors of `fs` will be evaluated, 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, since determining the value of the first element of a reversed list requires reversing the entire list, all at once:

Inputs	Output	Demand On Output	Demand On Inputs
`fs = [1,2,3]` `bs = [6,5,4]`	`[1,2,3,4,5,6]`	`D1:D2:D3:D4:_`	`fs = D1:D2:D3:[]` `bs = _:_:D4:[]`

In this case, our specification should predict that the demand on the first list will be the prefix of the output demand equal in length to `fs`. It should also predict that the demand on the second list will be the reverse of the remaining demand on the result, prefix-padded with thunks if necessary.

There is one last case we need to consider: when the demand on the result is equal in size to the first list, and the second list is empty:

Inputs	Output	Demand On Output	Demand On Inputs
`fs = [1,2,3]` `bs = []`	`[1,2,3]`	`D1:D2:D3:[]`	`fs = D1:D2:D3:[]` `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. As such, our specification should predict that the back list will be fully evaluated, and that the demand on the front list will be identical to the demand on the result of the function.

5.1.1 Specifying Naïve Queues. Let's translate this careful case-wise analysis to write a StrictCheck specification. We use the two functions

```
isCapped :: [a] → Bool
cap      :: [a] → [a]
```

to, respectively, check if the tail of the list demand is a `thunk`, and replace that `thunk` with an empty list if so.

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

Just like the specifications of `take` in the introduction, this specification uses the constructor `Spec` applied to a function. That function has as arguments a continuation `predict`, the result demand `resultDemand` represented implicitly as a value of type `[Int]`, (per Section 4), as well as the actual input lists `fs` and `bs`.

This specification correctly predicts the demand on `fs` and `bs`. In English, it says: "If we don't demand more from the output of `rot_naive` than the length of `fs`, then the demand on `fs` is just the result demand itself, while `bs` is left completely unevaluated. On the other hand, if we do demand more from the output than the length of `fs`, only the first piece of the demand is actually relevant to `fs`, while the rest constitutes the demand on `bs` after being appropriately padded with thunks and reversed."

5.2 A Trickier Spec: Okasaki's Persistent Queues

Now, let's specify the strictness of Okasaki's `rot`. The function `rot` is defined in terms of the more general `rotate`:

```
rot :: [a] → [a] → [a]
rot fs bs = rotate fs bs []

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)
```

Each element taken from the result of `(rot front back)` requires both the front and back lists to be evaluated more deeply by one constructor. As a result, the "effort" of reversing the back list is distributed across each `dequeue` operation, so that by the time the front list has been notionally exhausted, the back list has already been reversed and appended to it.

It's worth noting that `rotate` is neither stricter nor lazier than the naïve reverse-and-append operation `rot_naive`. It is stricter in the back list up until the point when the front list has been exhausted—and then, it is much lazier, as there is no more work to do upon the back list.

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

Inputs	Output	Demand On Output	Demand On Inputs
`fs = [1,2,3]` `bs = [6,5,4]`	`[1,2,3,4,5,6]`	$\texttt{D}_1\texttt{:D}_2\texttt{:\_}$	`fs =` $\texttt{D}_1\texttt{:D}_2\texttt{:\_}$ `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 exist in the front list. On the other hand, `rot` evaluates the same number of `(:)` cells in `bs` as it evaluated in `fs`. This 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	Output	Demand On Output	Demand On Inputs
`fs = [1,2,3]` `bs = [6,5,4]`	`[1,2,3,4,5,6]`	$\texttt{D}_1\texttt{:D}_2\texttt{:D}_3\texttt{:D}_4\texttt{:\_}$	$\texttt{fs = D}_1\texttt{:D}_2\texttt{:D}_3\texttt{:[]}$ $\texttt{bs = \_:\_:D}_4\texttt{:[]}$

The demands in this case are identical to the ones we saw in `rot_naive`: they require that the spines of both lists be fully evaluated.

5.2.1 Specifying Okasaki's Queues. From this reasoning, we might write the following spec for `rot` by making the modifications to the specification for `rot_naive` which correspond to the observations above:

```
rot_spec :: Shaped a ⇒ Spec '[[a], [a]] [a]
rot_spec =
  Spec $ \predict resultDemand fs bs →
    let demandOnFs
          | length (cap resultDemand) > length fs =
              take (length fs) (cap resultDemand)
          | otherwise = resultDemand
        demandOnBs
          | numCtrForced resultDemand > length fs =
              reverse $ take (length bs)
                      $ drop (length fs) (cap d)
                        ++ repeat thunk
          | otherwise =
              (reverse $ drop (length fs) (cap resultDemand)
                         ++ replicate (length (cap resultDemand)) 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 list demand, including both `(:)` and `[]` constructors in its total. This function is implemented using the `length` function over lists and the previously introduced `cap` function that replaces the tail thunk in a list demand with an empty list.

When we test this specification using StrictCheck, it presents us with a counterexample. Repeatedly running this test reveals that all such counterexamples occur when the `length` of `bs` is greater than the length of `fs`. However, this indicates a bug in our testing procedure, not in `rot`: Okasaki's queue implementation maintains the invariant that `length fs >= length bs`, and so this case does not arise in any actual invocation of `rot`. However, since we didn't specify that precondition, StrictCheck identified a counterexample. The full version of the specification, including coverage

for 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 a counterexample (Figure 4).

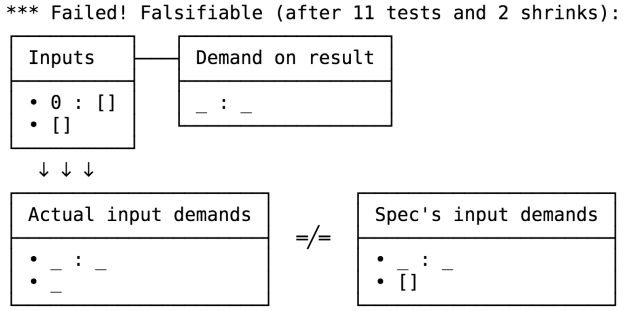

Fig. 4. Counterexample for `rot_naive` under `rot_spec`

As expected, the incremental `rot` specification requires the second list to be evaluated when `rot_naive` is in fact lazy. The specification `rot_spec` correctly distinguishes the incremental laziness behavior from that of `rot_naive`. A precise and executable specification for `rot` both serves both as live documentation of its intended strictness, and delivers high degree of confidence in the correctness of `rot` since users can easily verify its strictness behavior at the push of a button.

6 TESTING HIGHER-ORDER FUNCTIONS

So far, we've seen how StrictCheck can test demand specifications for first-order functions like `take`. Haskell is a higher-order language, though, and it would be a shame if we could only test the strictness of 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 (a : as) = f a : map f as
```

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)`. Readers familiar with QuickCheck might think to reach for its built-in `CoArbitrary` class, which indeed allows the user to generate random functions. As we will see, however, `CoArbitrary` is capable of generating only *fully strict* functions. This uniform strictness is fine for testing ordinary functional correctness properties, but it's a deal-breaker for StrictCheck. If we accidentally wrote a function `map'` that unconditionally evaluated every element of the input list, it would behave identically to the ordinary `map` for all fully strict function arguments. StrictCheck would only be able to distinguish it from a correct implementation of `map` by testing it with a non-strict function argument. If we want to use StrictCheck to test higher-order functions, we will need to generate functions with randomly varied strictness.

6.1 Generating (Strict) Functions with QuickCheck's `CoArbitrary`

In order to understand how StrictCheck generates random *non-strict* functions, it's helpful to first understand how QuickCheck generates its own random *strict* functions. Let's start at the beginning, by investigating its generator monad `Gen`.

Most users of QuickCheck only encounter `Gen` as an abstract type. To construct a random generator `(Gen A)` for a value of type `A`, a user combines QuickCheck's library of generator combinators (`choose`, `elements`, etc.) using `Gen`'s monadic operations. For example, the generator below returns pairs whose first element is in the range 0–10, and whose second element is a random boolean:

```
genMyPair :: Gen (Int, Bool)
genMyPair = do
  i ← choose (0, 10)
  b ← elements [False, True]
  return (i, b)
```

This interface lets a user describe how to build a random value without worrying about how the randomness is supplied to their generator and its component primitives. The `Arbitrary` typeclass defines a sensible "default" generator, meant to statistically cover every possible value in a type:

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

This approach works wonderfully for generating first-order data types. What's less clear is how to generate higher-order types like functions and data structures containing them. How might we randomly sample from a function type `(A → B)`?

One easy but unsatisfying answer: if we have an `Arbitrary` instance for `B`, we can generate constant functions, each of which always returns some pre-generated value of `B`:

```
genConst :: Arbitrary b ⇒ Gen (a → b)
genConst = do
  result ← arbitrary
  return (\_ → result)
```

This is unsatisfying indeed: constant functions represent only a sliver of the type `(A → B)`. There are many errors QuickCheck would never discover if it used only constant functions as higher-order inputs, as most useful tests would require us to generate functions with input-dependent results. To see how QuickCheck does this, let's peek under the hood of the `Gen` type to see how it really works.

In essence, a value of type `(Gen A)` is a function from a (pseudo-)random seed to some value of type `A`. When a program invokes a generator to actually create a random `A`, QuickCheck picks a random seed and feeds it into this function, thus returning a random `A`. As we've seen, the `Monad` instance for `Gen` lets users combine primitive generators to produce generators for larger data types. The monadic bind operation for generators

```
(>>=) @Gen :: Gen a → (a → Gen b) → Gen b
```

creates a generator `(Gen b)` which, when given an initial random seed `r`:

(1) *splits* `r` into two statistically independent random seeds `s` and `t`,
(2) feeds `s` to the generator `(Gen a)` to produce an `a`, and
(3) produces a `b` by calling the `(a → Gen b)` function, feeding the resultant generator with the second seed `t`.

In this way, QuickCheck ensures that different generators are not correlated.

Splitting is also the key to generating random functions with input-dependent results. To generate such functions, we need to take a generator for the function's output, and feed it with a *different*

random seed depending on the value of the function's input. As a result, the function will produce different random output when given different inputs.

How might this work? For example, let's make a generator for functions `(Gen (Bool → Int))`. Our generator, when given an initial random seed `r`, splits `r` into the seeds `s` and `t`. It then picks one of `s` or `t`, choosing based on whether the function's input `Bool` is `True` or `False`. Finally, it feeds that particular seed into the output generator `(Gen Int)`. Because the choice of seed for generating the function's output is dependent on the value of the function's input, this approach generates useful non-constant functions.

Instead of asking the user to explicitly manipulate random seeds, QuickCheck provides the `CoArbitrary` abstraction to simplify generating random functions.

```
class CoArbitrary a where
  coarbitrary :: a → (Gen b → Gen b)
```

An instance of `CoArbitrary` for some type `A` describes how to use some `A` to perturb the random seed of some other unknown generator `(Gen b)` in a way that depends on the value of that `A`. To aid in writing these instances, QuickCheck exports a function called `variant`, which takes an integer argument and a generator, and alters the generator by perturbing its random seed in a way that depends on that integer.

```
variant :: Int → (Gen b → Gen b)
```

To write a proper instance of `CoArbitrary` for `A`, the user should call `variant` at a different integer index for each different constructor of `A`, and compose the resultant perturbation function `(Gen b → Gen b)` with those produced recursively from the fields of that constructor. For example, here is the `CoArbitrary` instance for `(Maybe a)`:

```
instance CoArbitrary a ⇒ CoArbitrary (Maybe a) where
  coarbitrary Nothing  = variant 0
  coarbitrary (Just a) = variant 1 . coarbitrary a
```

QuickCheck uses generic programming to follow this same pattern, providing a default `CoArbitrary` instance for most data types. Note that all correct `CoArbitrary` instances should destruct the entirety of their input, so that every part of an input value has a chance to perturb an output generator's random seed.

Given a `CoArbitrary` instance for some input type `A` and an `Arbitrary` instance for some output type `B`, QuickCheck can generate random functions `(A → B)`. To do this, it uses the internal function

```
promote :: (a → Gen b) → Gen (a → b)
```

This operation takes a function returning generators `(a → Gen b)`, and transforms it into a generator returning functions `(Gen (a → b))`. The generator returned by `promote` captures its initial random seed and returns a closure `(a → b)` which has access to that seed. When that closure is given an input `a`, it calls the original function `(a → Gen b)` to produce a generator `(Gen b)`. It then feeds the captured random seed to this generator to return its output `b`.

The function `promote` is the final piece enabling QuickCheck to generate random functions from anything `CoArbitrary` to anything `Arbitrary`:

```
instance (CoArbitrary a, Arbitrary b) ⇒ Arbitrary (a → b) where
  arbitrary =
    promote $ \a →
      (coarbitrary a) arbitrary
```

Unfortunately for StrictCheck, this approach generates only fully strict functions. Evaluating even the smallest piece of this function's result means feeding some initial random seed to its call to `arbitrary`. Computing this seed requires evaluating the perturbation function defined by the call

to `coarbitrary`, which—if the instance of `CoArbitrary` is correctly defined—forces the entire input `a` to be fully evaluated.

Worse, even if a `CoArbitrary` instance didn't evaluate some of its input, we would still be unable generate functions with arbitrary strictness. Since all the random perturbation extracted from an input value is collected into a single atomic seed, evaluating that seed in turn evaluates anything that influences its value all at once. Therefore, functions generated this way exert the same demand on their input, regardless of the demand on their output.

6.2 Generating Non-Strict Functions for StrictCheck

To solve this problem, let's look at non-strict functions from a different perspective. We call a function *productive* if evaluating any finite prefix of its output requires 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 productive, because evaluating a finite part of its output list requires evaluating only the corresponding pieces of its input lists. An example of a *non*-productive function is `sum`:

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

This function is not productive, because evaluating its output `Int` requires its entire input list to be evaluated. If its input is an infinite list, any non-trivial demand on the result of `sum` will cause it to diverge, as it attempts to consume all of that infinite input.

We can view a productive function as a procedure which alternates between evaluating some finite amount of input and producing some finite amount of output, until it has produced its entire output. Functions generated by StrictCheck explicitly follow this pattern, randomly alternating between consuming pieces of input and producing pieces of output. At each step, the value of some piece of input can influence both the value of some future piece of output, as well as future choices about which pieces of input to consume next.

We generate functions using two complementary typeclasses: `Consume` and `Produce`. While the `CoArbitrary` typeclass collapses an input all at once into a random perturbation, the `Consume` typeclass transforms it into a tree of individual random perturbations. As we produce pieces of input, we'll compose and apply these pieces of randomness incrementally.

We represent an individual random perturbation with the type `Variant`, which wraps a universal transformation on generators just like the result from QuickCheck's own `variant` function.

```
newtype Variant
  = Variant { vary :: forall a. Gen a → Gen a }
```

This type is is a `Monoid`: its identity and combination function are `id` and `(.)`, respectively, lifted over the `newtype`. We assemble individual `Variants` into a rose tree called an `Input`:

```
data Input
  = Input Variant [Input]
```

The `Consume` typeclass, our equivalent to `CoArbitrary`, maps a value to an `Input`. The functions we'll generate will incrementally use the random perturbations it contains while producing their outputs.

```
class Consume a where
  consume :: a → Input
```

Instances of `Consume` create `Inputs` whose tree structures match those of the `consume`d values, with a `Variant` at each node corresponding to the randomness derivable from the corresponding value constructors. Just as `CoArbitrary` provides the function `variant` for writing its instances, we provide the function `constructor` for writing instances of `Consume`:

```
constructor :: Int → [Input] → Input
constructor i = Input (Variant (variant i))
```

Instances of `Consume` are defined in a similar manner to their corresponding `CoArbitrary` instances. For example, the `Consume` instance for `(Maybe a)` is:

```
instance Consume a ⇒ Consume (Maybe a) where
  consume Nothing  = constructor 0 []
  consume (Just a) = constructor 1 [consume a]
```

Importantly though, instances of `Consume` must be precisely the right strictness: evaluating the top-most `Input` constructor should require evaluating the top-most constructor of the `consume`d value, and nothing more. That is: as we evaluate an `Input`, the value from which it was created should be evaluated to precisely the same degree as that `Input`. In this way, evaluating an `Input` can act as a proxy for evaluating the original value to which it corresponds—a property StrictCheck relies upon for its own correctness. This rule implies that the `consume` method for functions must evaluate them, even though it will always return the trivial `Input`:

```
instance Consume (a → b) where
  consume !_ = constructor 0 []
```

Just like `CoArbitrary` instances, `Consume` instances have exactly one correct implementation. In StrictCheck, we derive these instances automatically using generic programming.

To complement `Consume`, we define the `Produce` typeclass:

```
class Produce b where
  produce :: [Input] → Gen b
```

Like `arbitrary`, `produce` generates a random output value in the `Gen` monad. However, it also is given a list of `Inputs`, each of which represents a "leaf" of some still-unevaluated value. As we'll see shortly, when `produce` needs to recursively generate some part of its output, it destructs some of these leaves and uses the `Variants` they contain to further randomize its output.

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

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

However, a function producing a data type with fields needs the opportunity to consume a random part of its input before it produces the value of a field. To enable this, we define a function `draws`, which randomly destructs some part of a list of `Inputs`. It collects the `Variants` from each `Input` node it traverses, returning their composition alongside the remaining leaves of the `Inputs`.

```
draws :: [Input] → Gen (Variant, [Input])
```

We could implement `draws` in many ways, each of which would give a different statistical distribution to the strictnesses of our generated functions. In StrictCheck, `draws` uses a geometrically bounded depth first random traversal, which biases generated functions so that demanding different pieces of output tends to evaluate different pieces of input. We detail our implementation in Appendix B.

Using `draws`, we implement a function `recur`, which generates an output value whilst randomly consuming `Inputs`:

```
recur :: Produce b ⇒ [Input] → Gen b
recur inputs = do
  (v, inputs') ← draws inputs
  vary v $ produce inputs'
```

When `recur` is called on some list of `Inputs`, it invokes `draws` to partially consume some of those inputs. Using the random perturbation collected from the consumed `Input` nodes, it calls `produce` on the still-unconsumed leaves of `Input`. If `produce` is always mutually recursive with `recur`, then before a function produces a piece of output, it might randomly consume some input.

Using `recur`, writing a `Produce` instance is directly analogous to writing an `Arbitrary` instance. Where the user would previously have used `arbitrary` to produce a sub-field of a value, they substitute `(recur inputs)`. For example, here are the `Produce` 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 =
    (,) <$> recur inputs
        <*> recur inputs
```

Now, let's finally use `produce` to generate random functions. Here, as in QuickCheck, we use `promote` to transform a function returning generators into a generator returning functions.

```
instance (Consume a, Produce b) ⇒ Produce (a → b) where
  produce inputs =
    promote $ \a →
      recur (consume a : inputs)
```

To `produce` a function, we `consume` its argument and add it to the given list of unevaluated `Inputs`, calling `recur` on those `Inputs` to generate its result.

We generate anything with a `Produce` instance in exactly this way. When testing with StrictCheck, we generate all arguments (including first-order ones) using the polymorphic generator `nonStrict`:

```
nonStrict :: Produce a ⇒ Gen a
nonStrict = produce []
```

This generator is equivalent to `arbitrary` for first-order arguments, but as we've seen, generates higher-order arguments of arbitrary strictness.

6.3 Specifying Higher-Order Functions

Now that we can properly generate random non-strict functions, let's test the strictness properties of higher-order functions. For example, let's specify and test the strictness of `map`:

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

In order to specify `map`, we need to introduce a new specification combinator, `specify1`, which derives a trivially correct specification for a unary function:

```
specify1 :: (Shaped a, Shaped b) ⇒ (a → b) → (b → a → a)
specify1 function resultDemand input =
  let (_, inputDemand) = observe1 context function input
  in fromDemand inputDemand
  where
    context = toContext (toDemand resultDemand)
```

This function takes a function, an implicitly-represented demand on its result, and its input, using `observe1` to see what it actually does under the context corresponding to that demand. It uses the function (derived from `Shaped`)

```
toContext :: Shaped b ⇒ Demand b → (b → ())
```

to convert a demand into an evaluation context suitable for observation.

We'll use `specify1` to reify the strictness behavior of a generated higher-order argument, so our specification can depend on it. Consider the difference in strictness between the functions `(map (const 1))` and `(map id)`—a correct specification of `map` needs to depend on the strictness of `map`'s argument, as this will determine how strict `map` is as a whole on its input list. With `specify1` in hand, we can correctly specify `map`:

```
map_spec :: (Shaped a, Shaped b) ⇒ Spec '[a → b, [a]] [b]
map_spec =
  Spec $ \predict resultDemand f xs →
    predict
      (if all isThunk (cap d) then thunk else f)
      (zipWith (specify1 f) resultDemand xs)
```

This specification predicts that the function given to `map` will be evaluated only if the demand on `map`'s result evaluates at least one element of the list. Additionally, each element of the input list is evaluated to precisely the degree required by `map`'s function argument, under the particular demand placed on the corresponding element of the result list. This specification passes all tests.

7 RELATED WORK

StrictCheck is the first framework in Haskell for property-based testing of strictness 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 (traditionally considered a non-functional property) against user-provided specifications. StrictCheck also provides a more flexible variant of the `CoArbitrary` typeclass from QuickCheck, capable of generating functions with random strictness.

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 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 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: 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. StrictCheck uses similar techniques to convert reified demand values to and from partial values.

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 strictness property of Haskell functions, while our work allows users to precisely specify and test a broader category of strictness property. StrictCheck also generalizes to higher-order functions, a domain not addressed in this prior work.

8 CONCLUSION AND FUTURE WORK

In this paper, we identified a class of dynamic properties typically considered out of scope for traditional property-based testing: strictness properties. We described an approach to specify, observe, and automatically test such properties and implemented it in an openly available Haskell library called StrictCheck.

Since this approach is somewhat new, there is a lot of space for improvement. In particular, the specification language is relatively low-level. A more declarative and general specification language could be constructed atop StrictCheck's foundations, and we anticipate exploring this design space in future work. Even without a high-level specification language, StrictCheck can be used to test whether a function has the same strictness as a reference implementation. Additionally, the same technique could be used to identify buggy compiler optimizations that break strictness. Further, taking inspiration from [Claessen et al. 2010], we would like to synthesize strictness specifications for functions based upon dynamic observations of their behavior.

A COMPARING THE SIMPLE AND FULL SPECIFICATIONS OF QUEUE ROTATION

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

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

Fig. 5. StrictCheck specifications of `rot_simple` and `rot` side by side, as described in Section 5.2

B IN DETAIL: INTERLEAVING RANDOM EVALUATION INTO GENERATION

The function `draws` described in Section 6.2 evaluates a random sub-forest of some `[Input]` in a random depth-first order. In the end, it returns a `Variant` storing the combined entropy from all the nodes of the `Input`s it traversed, as well as a new forest of the yet-to-be-consumed "leaves" of the original `Input`s. The number of nodes of `Input` consumed by a call to `draws` is given by a geometric distribution with expectation 1. Below we list the full implementation of `draws`, presented alongside a selection of the `Produce` and `Consume` APIs for reference.

```
newtype Variant
  = Variant { vary :: forall a. Gen a → Gen a }

instance Monoid Variant where
  mempty = Variant id
  mappend v w = Variant (vary v . vary w)

data Input
  = Input Variant [Input]

class Consume a where
  consume :: a → Input

class Produce b where
  produce :: [Input] → Gen b

recur :: Produce b ⇒ [Input] → Gen b
recur inputs = do
  (v, inputs') ← draws inputs
  vary v $ produce inputs'

draws :: [Input] → Gen (Variant, [Input])
draws inputs = go [inputs]
  where
    -- Mutually recursive, traverse a tree-zipper 'levels' into a forest of inputs:
    go, inwardFrom :: [[Input]] → Gen (Variant, [Input])

    go levels =
      oneof                                -- uniform choice between:
        [ return (mempty, concat levels)   -- flatten 'levels' into a list of untouched leaves
        , inwardFrom levels ]              -- keep traversing input

    inwardFrom levels =
      case levels of
        [            ] → return mempty         -- if no more input: stop
        [  ] : outside → inwardFrom outside    -- if nothing here: backtrack up a level
        here : outside → do                    -- if something here:
          (Input v inside, here') ← pick here  -- pick a subfield (forcing corresponding thunk)
          vary v $ do                          -- let future traversal path depend on its value
            (entropy, levels') ← go (inside : here' : outside)  -- maybe traverse even deeper
            return (v <> entropy, levels')     -- add this node to the collection of entropy

    -- Pick a random list element: return it, and the remaining list
    pick :: [a] → Gen (a, [a])
    pick as = do
      index ← choose (0, length as - 1)
      let (before, picked : after) = splitAt index as
      return (picked, before ++ after)
```

ACKNOWLEDGMENTS

We are grateful to José Manuel Calderón Trilla, Stephanie Weirich, Benjamin Pierce, Mayur Naik, Katrina Xiaoyue Yin, Jennifer Paykin, Robert Rand, Antal Spector-Zabusky, Matthew Weaver, Ryan Trinkle, Daniel Winograd-Cort, Sandra Dylus, Juliette Martin, and the Penn PLClub for their useful comments and support. One of the authors developed an early version of this work in a streamed programming session; the authors would like to thank those who participated. This material is based upon work supported by the National Science Foundation under Grant Numbers 1421243, 1521523, 1703835, and 1319880. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2429069.2429075

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, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *Tests and Proofs*, Gordon Fraser and Angelo Gargantini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 6–21.

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

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

Edward A. Kmett. 2017. recursion-schemes: Generalized bananas, lenses and barbed wire. http://hackage.haskell.org/package/recursion-schemes. (2017). [Online; accessed 12-June-2018].

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