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.

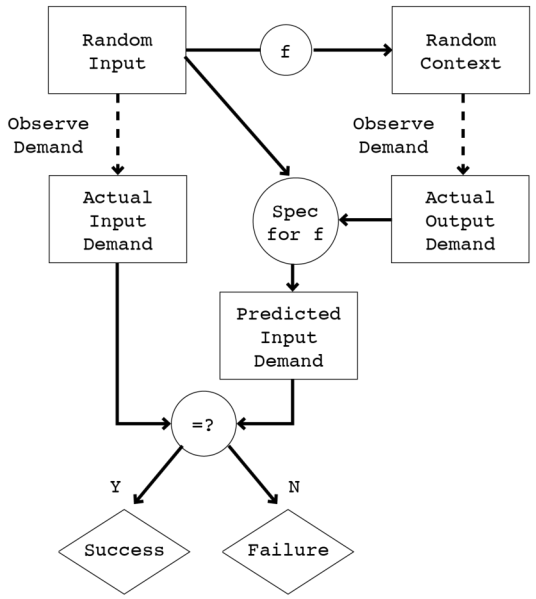

Fig. 1. Architecture of StrictCheck

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`.

```
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