`falsify`: Internal Shrinking Reimagined for Haskell

Edsko de Vries
Well-Typed LLP
United Kingdom, London
edsko@well-typed.com

Abstract

In unit testing we apply the function under test to known inputs and check for known outputs. By contrast, in property based testing we state *properties* relating inputs and outputs, apply the function to *random* inputs, and verify that the property holds; if not, we found a bug. Randomly generated inputs tend to be large and should therefore be minimised. Traditionally this is done with an explicitly provided *shrinker*, but in this paper we propose a way to write generators that obsoletes the need to write a separate shrinker. Inspired by the Python library `Hypothesis`, the approach can work even across monadic bind. Compared to `Hypothesis`, our approach is more suitable to the Haskell setting: it depends on a minimal set of core principles, and handles generation and shrinking of infinite data structures, including functions.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Keywords: property based testing, internal shrinking, functional programming

ACM Reference Format:
Edsko de Vries. 2023. `falsify`: Internal Shrinking Reimagined for Haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium (Haskell '23), September 8–9, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3609026.3609733

> These considerations suggest that not the *verifiability* but the *falsifiability* of a system is to be taken as a criterion of demarcation.
>
> *Karl Popper, The Logic of Scientific Discovery*

1 Introduction

Unit tests typically supply a function with specific inputs, and then verify the result against specific outputs. This tends to be an excellent way to confirm one's cognitive bias: after all, if a programmer is not taking a specific edge case into account in their code, why would they think to include that edge case in their unit tests? It also scales poorly; typically, for a system with n features, we write $O(n)$ unit tests, a few tests per feature. However, often bugs arise from *interaction* between features; while writing $O(n)$ tests is merely tedious, writing $O(n^2)$, $O(n^3)$, …, tests becomes unfeasible. Thus the motto of property based testing, due to its godfather John Hughes: "do not write tests: generate them" [9].

Rather than supplying a function with specific inputs and checking for specific outputs, we instead define *properties* the function should have, supply the function with *randomly generated* inputs, and verify the results against the property.[1] If the property is not satisfied, the test fails and we have found a *counterexample*. This has proven highly effective; first made popular by the `QuickCheck` library for Haskell [5], the approach has since been ported to countless other programming languages, functional or otherwise (section 8).

An important part of property based testing is *shrinking*: randomly generated test data tends to to be large and contain irrelevant details. To make debugging easier, it is therefore important to try and shrink the counterexample as much as possible, ideally until *only* relevant details remains.

Despite widespread success, QuickCheck is not without its problems, especially in its approach to shrinking, as we shall see in section 2. Two libraries in particular have put forth alternative approaches. First, QuviQ QuickCheck [3] is based on *integrated shrinking* (section 3). Second, Hypothesis [14] is based on *internal shrinking*, and is the direct inspiration for our work on `falsify`; we will discuss our own approach in section 4, and compare it to Hypothesis in section 7.

Our contributions are as follows:

- A key abstraction in `Hypothesis` is the *choice sequence*, essentially an unrolled PRNG (section 7.1). Instead of this linear representation, we propose a tree-shaped representation which we call a *sample tree* (section 4.2).
- Hypothesis uses a relatively large and growing list of reduction passes for this choice sequence (section 7.2). In `falsify` there is only one[2]: shrink a sample in the tree (section 4.4). This makes shrinking predictable and gives users more control.

Haskell '23, September 8–9, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0298-3/23/09…$15.00
https://doi.org/10.1145/3609026.3609733

[1] Some property based testing approaches use exhaustive enumeration (for small domains) rather than random generation. In this paper our focus is on random generation; we will briefly come back to the enumeration case when we discuss related work (section 8.5).

[2] Two if you consider the optimisation where we replace *all* samples in a subtree by zero at once (section 4.4) to be a separate pass.

- Our sample tree gives us "hierarchical debugging" for free: we can shrink generators independently from each other. To do the same, Hypothesis needs to *recover* some kind of tree structure (section 7.1).
- We show how we can advantage of *selective functors* to further improve the independence of generators (section 6.1).
- We show how we can combine internal shrinking with manually written shrinkers (section 6.2.1).
- Our representation allows the generation of *infinite* data structures, and we show how to take advantage of this to generate functions (section 6.2.2).
- We discuss that even with internal shrinking, it is still important to *test* shrinking, and we show how we can do that (section 6.3).

2 Manual Shrinking

In order to better understand the motivation for and success of the internal shrinking approach, we will first summarise its predecessors. In this section we will discuss the traditional QuickCheck approach, which we dub here *manual shrinking*. Section 3 will discuss integrated shrinking, at which point the stage is set for introducing internal shrinking in section 4. We will focus on principles rather than implementation details, and set things up so as to facilitate comparison between the different approaches.

At the core of all of these approaches lies the definition of a *generator* of random values. In QuickCheck, a generator consists of a function to *produce* a random value given a pseudo random number generator (PRNG), and a function to *shrink* such values:

```
data Gen_Q a = Gen_Q { gen    :: PRNG → a
                     , shrink :: a → [a] }
```

The `shrink` function should return a list of possible shrunk values. When the test driver finds a counterexample, it calls `shrink` repeatedly until either none of the values constitute a counterexample (we have shrunk too much), or `shrink` returns no more options (the example is already minimal).

Here is an example generator for unsigned integers:

```
word_Q :: Gen_Q Word
word_Q = Gen_Q {gen = ..., shrink = shrinkWord}
```

where `shrinkWord` is defined as

```
shrinkWord :: Word → [Word]
shrinkWord 0 = []
shrinkWord n = [0 .. n - 1]
```

This definition of `shrinkWord` is "optimal" in the sense that *if* a smaller value of n exists it will find it, since it tries *all* smaller values. This is expensive, however, and shrinkers for integers typically use some kind of binary search; however, such concerns are orthogonal to our current discussion.

More pertinent is the fact that $\texttt{Gen}_\texttt{Q}$ only supports very limited compositionality. For example, here is how we might construct a generator for pairs:

```
pair_Q :: Gen_Q a → Gen_Q b → Gen_Q (a, b)
pair_Q g1 g2 =
  Gen { gen    = λr → let (r1, r2) = split r
                      in (gen g1 r1, gen g2 r2)
      , shrink = λ(x, y) →
            [(x', y ) | x' ← shrink g1 x]
          ++ [(x , y') | y' ← shrink g2 y] }
```

When producing a value, we split the PRNG in two [21], and feed the pieces to the two generators; in shrinking, we *either* shrink the left value *or* the right value in the pair (remember that shrinking just returns a single step; we might shrink the left value in step one, and then the right value in step two).[3]

Unfortunately, this cannot be the basis for an `Applicative` instance, since $\texttt{Gen}_\texttt{Q}$ is not a functor:

```
instance Functor Gen_Q where
  fmap f g = Gen_Q { gen    = f . gen g
                   , shrink = not definable }
```

No implementation of `shrink` exists because in addition to the argument $\texttt{f} :: a \to b$ we get in `fmap`, we would also need an $\texttt{f}' :: b \to a$ before we can apply `shrink` on type a.

This makes $\texttt{Gen}_\texttt{Q}$ less composable than one might like; for example, we cannot construct a generator for even numbers by doubling the results of a generator for arbitrary numbers, nor we can construct a generator for lists of elements that satisfy some predicate p by filtering the results of a generator for arbitrary lists.

Such generators can be defined in a non-compositional manner, of course, but when we do, we find that the same logic is often present in the function to produce values and in the function to shrink them. For example, here is how we might write $\texttt{filter}_\texttt{Q}$ (where `filter` is the standard filter function on lists):

```
filter_Q :: (a → Bool) → Gen_Q [a] → Gen_Q [a]
filter_Q p g =
  Gen_Q { gen    = filter p . gen g
        , shrink = map (filter p) . shrink g }
```

3 Integrated Shrinking

The QuviQ QuickCheck library for Erlang set out to solve the problems we discussed in the previous section. In the Haskell world, this approach is popularised by the library `Hedgehog`. Their *integrated shrinking* approach makes it possible to define `Functor` and `Applicative` instances for generators, and duplication of logic is avoided. However, while a `Monad` instance is *possible*, it does not work very well; we will need to turn our attention to *internal* shrinking in section 4 before we can address that problem.

[3] To avoid combinatorial explosion, we not try to shrink both values at once.

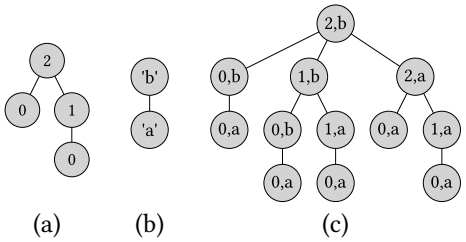


Figure 1. Applicative Composition

3.1 Shrink Trees

Instead of manually pairing each generator with a shrinker, in integrated shrinking the generator itself produces a value and all possible ways to shrink that value:

```
type Gen_H a = PRNG → Tree a
data Tree a = Node a [Tree a]
```

Here `Tree` is the type of *n*-ary trees: the root corresponds to the value to be produced, and values further down the tree are possible ways to shrink it. For primitive generators, the easiest way to define a generator is still by supplying a function to produce a value and a function to shrink a value:

```
gen_H :: Gen_Q a → Gen_H a
gen_H g = unfoldTree (λx → (x, shrink_Q g x)) . gen_Q g
```

As an example, fig. 1(a) shows the tree that might arise from generating the value 2, shrinking integers to all smaller values, like we did in section 2.

This definition has a number of important benefits. For starters, it can easily be given a `Functor` instance, using the `Functor` instance for `Tree`:

```
instance Functor Gen_H where
  fmap f g = fmap f . g
```

This alone is already very useful. For example, we can now write `filter` without any duplication of logic:

```
filter_H :: (a → Bool) → Gen_H [a] → Gen_H [a]
filter_H = fmap . filter
```

Like for manual shrinking, it is not difficult to write a combinator for pairs of generators:

```
pair_H :: Gen_H a → Gen_H b → Gen_H (a, b)
pair_H g1 g2 r =
    let (r1, r2) = split r in go (g1 r1) (g2 r2)
  where
    go :: Tree a → Tree b → Tree (a, b)
    go l@(Node a as) r@(Node b bs) =
      Node (a, b) $ [ go a' r  | a' ← as ]
                  ++ [ go l  b' | b' ← bs ]
```

This implements the same logic as $\texttt{pair}_Q$: the root of the tree corresponds to the original pair, and then we either shrink the left or the right value.

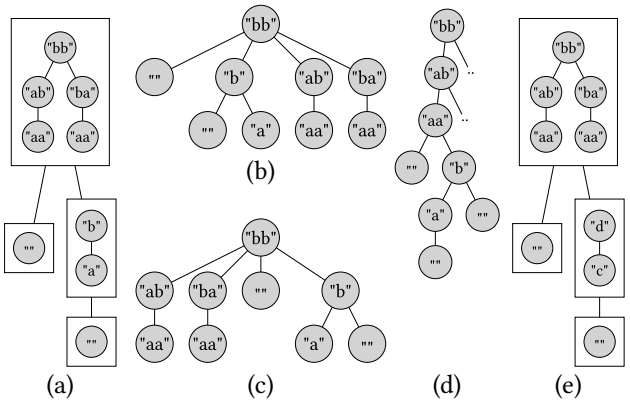


Figure 2. Monadic Composition

Figure 1 illustrates the process: (a) is the tree for integers we saw before; (b) shows a tree for characters starting at `'b'` and shrinking to `'a'`, and (c) shows the result of pairing these two.

We can now equip $\texttt{Gen}_H$ with an `Applicative` instance; for `pure`, we return a tree with the given value as its root and no possible shrinks:

```
instance Applicative Gen_H where
  pure x _  = Node x []
  g1 <*> g2 = fmap (uncurry ($)) (pair g1 g2)
```

3.2 Monadic Composition

This is all looking very promising so far, but trouble starts when we start to think about monadic composition. Monadic composition is required when the behaviour of a generator depends on previously generated values; it arises naturally and frequently. For example, consider generating a list of random length containing random characters:[4]

```
randomList :: Gen_H [Char]
randomList = do n ← word ; replicateM n char
```

Monadic composition boils down to implementing

```
(>>=) :: Tree a → (a → Tree b) → Tree b
x >>= f = ...
```

The obvious first thing to try is to apply f at every node in tree x, resulting in a tree of type `Tree (Tree b)`. Figure 2(a) shows what this would look like for `randomList`, assuming the `int` and `char` trees from fig. 1(a) and (b): the outer tree has the same structure as fig. 1(a), with a shrink tree for n characters replacing the value n in the original tree. The question is how to flatten or "join" this tree.

There are two natural ways in which we might define this function, which we might call *left-biased* join and *right-biased* join; their effects are illustrated in fig. 2(b) and (c):

[4] `replicateM` repeats an action n times, collecting the results; despite the name, it uses the applicative interface only.

```
lb (Node (Node x xs) xss) = Node x (map lb xss ++ xs)
rb (Node (Node x xs) xss) = Node x (xs ++ map rb xss)
```

In our example, left-biased join will shrink the length of the list first, and right-biased join will shrink the list elements first. However, notice something unfortunate: *once we start shrinking elements, we never go back again to shrink the list length.* Between these two choices then left-biased would the better, though neither is particularly good.

Looking at fig. 2(a), we might wonder[5] if there might be a third option, flattening a tree by inserting an edge from every node in a subtree to the roots of the flattened trees below; fig. 2(d) shows an excerpt from the resulting tree. The problem with this approach is that *a priori* we have no reason to believe there is any relation at all between the nodes of one subtree and the nodes of another. Notice how we might shrink from "bb" ⇝ "ab" ⇝ "aa" ⇝ "b"; it is as if we forgot that we previously shrunk that `'b'` to an `'a'`.

Perhaps looking at a slightly different example will bring this into sharper focus: fig. 2(e) shows an example where in the case of a list of length 1 we use a *different* generator for the element. Applying the same approach, we might shrink "bb" ⇝ "ab" ⇝ "aa" ⇝ "d". It is not so much that this is an *invalid* shrink step, but there is a lot of wasted effort.

Zooming out, if we ponder the type of bind again

```
(>>=) :: Gen_H a → (a → Gen_H b) → Gen_H b
```

it is actually clear that we cannot shrink the second generator independently from the first, since we *have* no generator without applying it to the result of the first. Yes, we can shrink the results of the second generator for a while and then go back and try to shrink the results from the first, but if we do, we need to reapply f and all start over again, and our previous shrinking efforts will simply be wasted; this is another way to understand what is happening in fig. 2(d).

4 Internal Shrinking: `falsify` Style

We now get to the heart of this paper, introducing *internal shrinking* as implemented in Haskell by the `falsify` library[6].

4.1 Generation versus Parsing

So far we have *split* the random number generator whenever we needed to compose generators. There are good reasons for doing this, and we will return to this approach soon. Let us however briefly consider an alternative, where generators return an updated PRNG, and we pass the PRNG linearly from generator to generator (shrinking is unchanged); this is illustrated in fig. 3.

If we "unroll" the PRNG to a stream of samples, we can shift our perspective and rather than thinking of `gen` as a generator, instead think of it as a *parser* of this stream:

```
parse :: [Sample] → (a, [Sample])
```

[5]Suggestion due to Gabriel Scherer.

[6]The library with author names removed is included in the submission.

```
data Gen_L a = Gen_L {gen :: PRNG → (a, PRNG), ...}

pair_L :: Gen_L a → Gen_L b → Gen_L (a, b)
pair_L g1 g2 = Gen_L { gen = λr0 →
                       let (a, r1) = gen g1 r0
                           (b, r2) = gen g2 r1
                       in ((a, b), r2), ... }
```

Figure 3. Linear Use of a PRNG

The key insight from `Hypothesis` is that we do not need to shrink values at all: we can shrink the *stream of samples*, and then re-run the parser/generator. In the remainder of this section we will now describe our approach, which takes this core idea from `Hypothesis` but is otherwise quite different.

4.2 Sample Tree

For reasons that will become clear later, we will not use a stream (list) of samples, but rather a *tree* of samples.

```
data STree  = STree Sample STree STree | Minimal
data Sample = Sample { value :: Word }
```

The intention of `STree` is that we *either* use the sample *or* the subtrees (corresponding to a split PRNG).[7] Conceptually, a sample tree is always infinite; `Minimal` corresponds to the tree that is zero everywhere:

```
view :: STree → (Sample, STree, STree)
view (STree s r1 r2) = (s, r1, r2)
view Minimal = (Sample 0, Minimal, Minimal)

pattern Inf :: Sample → STree → STree → STree
pattern Inf s r1 r2 ← (view → (s, r1, r2))
```

Given an interface for a (splittable [21]) PRNG

```
sample :: PRNG -> Sample
split  :: PRNG -> (PRNG, PRNG)
```

we can easily construct a sample tree from a PRNG:

```
sTree :: PRNG → STree
sTree r = let (r1, r2) = split r
          in STree (sample r) (sTree r1) (sTree r2)
```

The `Minimal` constructor is introduced during shrinking; we will discuss this in section 4.4.

A generator then is a parser of this sample tree. In our approach, these parsers cannot fail; every sample tree is a valid input (just like a generator cannot fail to produce a value depending on the PRNG seed). Moreover, we mandate that generators are subject to the following contract:

> When run against the minimal sample tree, a generator should produce its simplest value.

[7]With linear logic we would be able give `STree` a more precise type: `STree :: Sample & (STree ⊗ STree) ⊸ STree`.

4.3 Generator

A generator is a function which, when given a sample tree, produces a value as well all possible ways that the sample tree can be shrunk:

```
newtype Gen a = Gen { run :: STree → (a, [STree]) }
```

Sample tree shrinking is not independent from generation of values because sample trees are infinite: the generator defines which parts of the sample tree are of interest. It also enables some of the extensions discussed in section 6. However, users of the library almost *never* need to think about or deal with the underlying sample tree at all.

The list of shrunk sample trees corresponds to a single shrinking step. As in QuickCheck, the shrinking algorithm then is greedy, repeatedly taking the first candidate shrunk sample tree (where a shrunk sample tree is considered a candidate if it still corresponds to a counter-example):

```
shrink :: (a → Bool) → Gen a → (a, [STree]) → [a]
shrink p g (a, shrunk) =
    case filter (p . fst) $ map (run g) shrunk of
      []  → a : []
      c:_ → a : shrink p g c
```

To find the initial input to `shrink`, a test driver generates different sample trees from different initial PRNG seeds, until it finds a seed that corresponds to a counter-example to the property being tested.

The most primitive generator just returns the next sample; for simplicity, we will use `shrinkWord` as the shrinker.[8]

```
prim :: Gen Word
prim = Gen $ λ(Inf s l r) →
  ( value s
  , (λs' → STree (Sample s') l r) <$>
      shrinkWord (value s)
  )
```

4.4 Monad Instance

The `Monad` instance for `Gen` is as follows (this is only "morally" and not *strictly* a monad; we will discuss the monad laws in section 4.7):

```
instance Monad Gen where
  return x = Gen $ λ_ → (x, [])
  x >>= f  = Gen $ λ(Inf s l r) →
      let (a, ls) = run x l
          (b, rs) = run (f a) r
      in (b, comb s (l : ls) (r : rs))
```

We start with two sample trees l and r that we use for x and f, respectively. When we run x, we get a list of shrunk sample trees l', and when we shrink f (using the *unshrunk* sample tree r), we get a list of shrunk sample trees r'.

[8] As discussed in section 2, usually binary search is a more suitable choice, and it is the default in `falsify`, although this can be overridden in the rare cases that this is necessary.

```
comb :: Sample → [STree] → [STree] → [STree]
comb s (l : ls) (r : rs) = shortcut $
       [STree s l' r  | l' <- unlessMinimal l ls]
    ++ [STree s l  r' | r' <- unlessMinimal r rs]
  where unlessMinimal :: STree → [a] → [a]
        unlessMinimal Minimal _  = []
        unlessMinimal _       xs = xs

        shortcut :: [STree] → [STree]
        shortcut [] = []
        shortcut ts = Minimal : ts
```

Figure 4. Combining Shrunk Sample Trees

Critically, although we need the generator in order to shrink, we do not reintroduce problems of the kind that integrated shrinking suffers from. Since we are shrinking the sample trees that *feed into* the generators and not the values *produced by* the generators, we can go freely back and forth between shrinking the left subtree and right subtree.

The final step is to combine these sample trees; the code for `comb`, shown in fig. 4, is a bit subtle. Sample trees are infinite, and so we have to be careful to ensure termination. We do this by introducing a "shortcut" which replaces entire subtrees by `Minimal`, and then stop shrinking once a tree is minimal. Since any given property will only require a finite part of the sample tree, we must always be able to reduce an infinite sample tree to a finite tree in this way.

4.5 Comparison with Integrated Shrinking

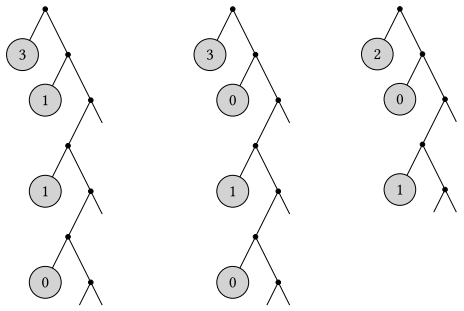
Let us consider a similar generator to the one that was giving us trouble with integrated shrinking:

```
randomList :: Gen [Word]
randomList = do n ← prim ; replicateM n prim
```

Suppose we are testing the property that "the elements of a list are all equal", and we find counter-example `[1, 1, 0]`; fig. 5(a) shows the corresponding sample tree. At this point, we cannot shrink the list length, because the prefix `[1, 1]` is not a counter-example; we *can* however shrink the first element of the list to get `[0, 1, 0]`. We can now "go back" and shrink the size of the list, ending up with `[0, 1]`; figures 5(b) and 5(c) show the corresponding sample trees. Unlike with integrated shrinking, we can go back and forth between shrinking elements of the list and shrinking the list length, and will not "forget" which lists elements we already shrunk.

4.6 Limitations

We must be careful to recognise what this approach can and cannot achieve. Shrinking worked out well in section 4.5, because the generator for a list of length n (`replicateM` n g) will look at a *subtree* of the sample tree as we decrease n.



(a): `[1, 1, 0]` (b): `[0, 1, 0]` (c): `[0, 1]`

Figure 5. Internal Shrinking

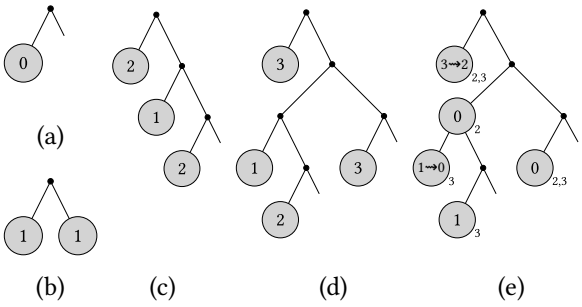


Figure 6. Non-uniform Sampling Trees

But suppose we defined a custom list generator like this:

```
list :: Word → Gen a → Gen [a]
list 0 _ = pure []
list 1 g = (λx     → [x]     ) <$> g
list 2 g = (λx y   → [x, y]  ) <$> g <*> g
list 3 g = (λx y z → [x, y, z]) <$> g <*> g <*> g
```

On the surface, this looks pretty regular too, but in fact it is not, due to the left-associative nature of `<*>`; figure 6(a–d) shows the sample trees that a generator for lists of length between 0 and 3 would look at.

Let us reconsider the example we studied in the previous section, but with this new list generator. Suppose we start with the sample tree from fig. 6(e), where we have marked the samples in the tree with a "2" if they are used when generating a list of length two, and with a "3" for lists of length three; fig. 6(e) corresponds to starting list `[1, 1, 0]`.

As before, we can shrink this to `[0, 1, 0]`, but now we cannot shrink this further: if we tried to reduce the list length, we would look at a different part of the sample tree, which in this case happens to correspond to the list `[0, 0]`, which is not a counter-example to the property that all elements of a list must be equal.

Let us ponder monadic bind once more; have we really achieved something?

```
(>>=) :: Gen_H a → (a → Gen_H b) → Gen_H b
```

When we considered this in section 3.2 in the discussion of integrated shrinking, we said that we *cannot* shrink the right hand side independently from the left because we do not *have* anything to shrink. With internal shrinking, this is not true: we can shrink the right subtree. The question is whether or not this is useful, and the answer is "it depends".

The answer is affirmative in a well-behaved example like `randomList` from section 4.5, but we are not always so lucky. When we are not, two things can happen, depending on the specifics of the generator. First, we might explore a part of the sample tree we did not look at before, essentially restarting shrinking, like we might in integrated shrinking. Second, we might continue with a previously shrunk tree, but interpret it in the context of a different generator; whether or not this will allow us to make further progress is again generator dependent. Partly the answer to this is to try and arrange things so that we *are* "lucky", writing our generators with this in mind: shrinking *never* truly comes for free!

4.7 Monad Laws

Technically speaking, the `Monad` instance for `Gen` does not satisfy the monad laws: the structure of a generator dictates which parts of the sample tree it uses. This is similar to the `Gen` monad in `QuickCheck`: "`Gen` is only morally a monad: two generators that are supposed to be equal will give the same probability distribution, but they might be different as functions from random number seeds to values".[9]

We should admit, however, that the fact that internal shrinking works directly with the random samples does make this a little worse. For example, consider

```
contrived :: Gen Word
contrived = do x ← prim
               if x == 0 then prim
                         else prim >>= return
```

If the initial value of x is non-zero, `prim` will look at one part of the tree; if then at later stage x is shrunk to zero, the `prim >>= return` branch will look at a *different* part of the sample tree, and any shrinking we did in the `prim` branch will be lost. Put another way, "modulo a choice of sample tree" is perhaps a stronger side condition than it might seem.

5 Example Generators

To build some intuition for how we can take advantage of internal shrinking, this section will discuss some example generators. The key is to consider how values produced by a generator shrink as the values produced by the generators it depends on shrink. The only primitive generator is `prim`, which produces a `Word` and shrinks towards 0.

[9] From `Test.QuickCheck.Gen.Unsafe`.

5.1 Booleans

Here is how we might produce a boolean:

```
bool :: Gen Bool
bool = (≥ (maxBound `div` 2)) <$> prim
```

The values returned by `prim` are drawn uniformly from the full range of `Word`; this means that `bool` chooses between `True` and `False` with equal probability. Moreover, as the value returned by `prim` shrinks towards zero,[10] `bool` will shrink towards `False`. To get a generator that shrinks towards `True` instead, we can simply take the complement.

5.2 Limited Range

Let us consider writing a generator that produces a value in the interval $[0, n]$, for $n <$ `maxBound`. Here is a non-solution:

```
wrongBelow :: Word → Gen Word
wrongBelow n = (`mod` (n + 1)) <$> prim
```

Although this produces values in the right range, it does not have the correct shrinking behaviour.[11] Suppose $n = 10$; as the value produced by `prim` shrinks from 22 to 21, the value produced by `wrongBelow` "shrinks" from 0 to 10.

We must instead somehow *compress* the range of `prim`. One way to do this is to first generate a random fraction in the range $[0, 1]$:

```
fraction :: Gen Double
fraction = frac <$> prim
  where frac :: Word → Double
        frac x = fromIntegral x / range

        range :: Double
        range = fromIntegral (maxBound :: Word)
```

All we need to do now is scale the fraction to the right range:

```
below :: Word → Gen Word
below n = (round . (* fromIntegral n)) <$> fraction
```

5.3 Signed Fractions

In section 5.2 we designed a generator for fractions in the range $[0..1]$. As our final example we will consider how we might write a generator for signed fractions in the range $[-1, 1]$, shrinking towards 0. First, another non-solution:

```
wrongSignedFraction :: Gen Double
wrongSignedFraction = do
  neg ← bool
  if neg then negate <$> fraction
         else fraction
```

We generate an additional boolean `neg`, telling us whether or not we should negate the fraction. The problem with this definition is that this boolean will shrink towards `False`, introducing a bias towards positive fractions; for example, it would be impossible to shrink from $+0.4$ to -0.2.

One possible correct implementation is to two generate *two* fractions, one to serve as positive fraction and to serve as a negative one, and choose whichever is closest to zero:

```
signedFraction :: Gen Double
signedFraction = aux <$> fraction <*> fraction
  where aux :: Double -> Double -> Double
        aux x y | x ≤ y     = x
                | otherwise = negate y
```

6 Extensions

In this section we will consider some extensions to the basic theory that significantly increase its expressivity, without complicating the core ideas.

6.1 Selective Functors

Consider this combinator that chooses randomly between one of two generators, shrinking towards the left:

```
chooseSuboptimal :: Gen a → Gen a → Gen a
chooseSuboptimal l r = do b ← bool
                          if b then l else r
```

Suppose the value of b is `True`, and we are shrinking the sample tree for l. If at any point b shrinks to `False`, we will now use the sample tree as l left it, and use that *same* tree for r; but it is not at all given that any shrinking we did for l was in any way meaningful for r.

We might attempt to improve the situation as follows:

```
chooseWrong :: Gen a → Gen a → Gen a
chooseWrong l r = aux <$> l <*> r <*> bool
  where aux :: a → a → Bool → a
        aux x y b = if b then x else y
```

In this version we generate values from *both* l and r, and then choose. Since `Gen` is lazy, this will not *actually* run both generators; only one value will be demanded in any given test run. This version is nonetheless wrong: it shrinks very poorly. Suppose b is currently `True`, and we are using the output of l. Since the sample tree used by r is unused at this point, it can be replaced by `Minimal` during shrinking; where in the suboptimal version r would reuse the sample tree left by l, here the only tree available to r is the minimal one.

What we want to do instead is *skip* shrinking of the left subtree as long as we are not using it, but the fact that it is not used is currently invisible to the shrinker. Fortunately, Mokhov et al. taught us how to selectively skip effects: we need to make `Gen` a selective functor [16]; the definition is shown in fig. 7.

[10] The full `falsify` library improves on this basic design for `bool`, and for `below` (section 5.2), by allowing `prim` to take bigger steps. This improves the performance of shrinking, but does not otherwise change its semantics.

[11] It could be argued that this does not matter: although `wrongBelow` will cycle, it will *eventually* shrink towards zero. However, this is only true if temporarily increasing the value still results in a counter-example. It also makes shrinking much less intuitive.

```
instance Selective Gen where
  select e f = Gen $ λ(Inf s l r) → do
    let (ma, ls) = run e l
    case ma of
      Left  a → let (f', rs) = run f r
                in (f' a, comb s (l : ls) (r : rs))
      Right b → (b, comb s (l : ls) [r])
```

Figure 7. `Selective` Instance for `Gen`

Like in the definition of `>>=` in the `Monad` instance, we must run the left generator before we can proceed with the right generator. However, we do not need the full result: we just need to know if the right generator is required at all. If we do not need it, we do not attempt to shrink it.

Any selective functor supports `ifS`:

```
ifS :: Selective f ⇒ f Bool → f a → f a → f a
```

This means that we can define `choose` simply as

```
choose :: Gen a → Gen a → Gen a
choose = ifS bool
```

In the general case, as the value produced by the first argument to `ifS` toggles between `True` and `False`, we will toggle between the left and right generator, shrinking them independently from each other, but never losing progress: if we switch from left to right and then back to left, shrinking will continue where it left off.

As an example use case, we can write a better list generator which, unlike the simple definition from section 4.5, can drop elements from anywhere in the list:

```
list :: ∀a. Int → Gen a → Gen [a]
list len g = catMaybes <$> replicateM len g'
  where g' :: Gen (Maybe a)
        g' = choose (pure Nothing) (Just <$> g)
```

6.2 Overriding Shrinking

Generators are entirely stateless: the only state is the sample tree, and after every shrink step the generator is simply re-run. However, if we make one small tweak to the sample tree, we significantly increase the expressiveness of generators: we will record whether a sample has been shrunk.

```
data Sample = ... | Unshrunk { value :: Word }
```

This allows us to define `shrinkTo`, shown in fig. 8. This is an important generator, which will produce x initially, then shrink to any of the xs, and can then shrink no further. The lines marked `(*)` are there to ensure that the generator can produce valid results when run with the minimal sample tree (zero everywhere) and a sample tree resulting from a *different* generator, respectively. This is a very useful combinator; we will sketch two applications below.

```
shrinkTo :: ∀a. a → [a] → Gen a
shrinkTo x xs = Gen $ λ(Inf s l r) →
    let setNext s' _ = STree (Sample s') l r in
    case s of
      Unshrunk _ → (x, zipWith setNext [0..] xs)
      Sample   i → (index i xs, [])
  where
    index :: Word → [a] → a
    index _ []     = x  -- *
    index _ [y]    = y  -- *
    index 0 (y:_)  = y
    index n (_:ys) = index (n - 1) ys

toNothing :: Gen a → Gen (Maybe a)
toNothing g = shrinkTo Just [const Nothing] <*> g
```

Figure 8. User-defined Shrinking

6.2.1 Manual and Integrated Shrinking.

It is sometimes useful to be explicit about shrinking, à la QuickCheck; for example, this will allow us to take advantage of previously hard-earned insights on shrinking a particular type of value.

We need two ingredients to do this. First, we need a way to change a generator so that it does not shrink at all: we are interested in its initial value only:

```
initial :: Gen a → Gen a
initial (Gen g) = Gen $ second (const []) . g
```

The second ingredient is a way to turn a shrink tree into a generator; for this we need `shrinkTo` (fig. 8):

```
fromShrinkTree :: Tree a → Gen a
fromShrinkTree (Node x xs) = do
    next ← Nothing `shrinkTo` map Just xs
    case next of
      Nothing → return x
      Just x' → fromShrinkTree x'
```

It is now easy to provide a way to do manual shrinking: we get the initial value of the generator, construct the full shrink tree, and then turn this into a new generator:

```
shrinkWith :: (a → [a]) → Gen a → Gen a
shrinkWith f gen = do
    x ← initial gen
    fromShrinkTree $ unfoldTree (λx' → (x', f x')) x
```

6.2.2 Generating Functions.

To illustrate the importance of generating infinite data structures, we will consider how to generate and shrink functions, following the approach proposed by Claessen for QuickCheck [4]. The details are fiddly; we will only show how Claessen's approach can be applied in the context of `falsify`, and refer the reader to [4] for additional details. The approach is based on a type of *concrete* functions:

```
data (:->) :: Type → Type → Type where
  Nil   :: a :-> b
  Unit  :: b → a :-> b
  Table :: Eq a => [(a, b)] → a :-> b
  Prod  :: a :-> (b :-> c) → (a, b) :-> c
  Sum   :: a :-> c → b :-> c → Either a b :-> c
  Map   :: (a→b) → (b→a) → b :-> c → a :-> c
```

A concrete function is an explicit, possibly partial, map from inputs to outputs: `Nil` is the empty description, `Unit` describes the constant function, `Table` is used to to tabulate (part of) the function's domain, and `Prod` and `Sum` allow to decompose functions for products and sums respectively. Finally, `Map` is used to define functions on b in terms of functions on a, given an isomorphism between a and b.

When first generated, concrete functions are typically infinite: they record the output of the function for every value in the function's domain. The key insight from Claessen is that in any given test, the function will only ever be applied to a *finite* number of inputs. Thus, we generate the function, relying on laziness to only demand the values that are used, and then use shrinking to cut away the parts of the concrete function that are not used, until we are left with a finite description of the function. This is not unlike the way that we are shrinking the sample tree!

Figure 9 shows an excerpt of how function generation works in `falsify`. Like in QuickCheck, we cannot write a single generator that can produce any type of function, but instead rely on a type class which dispatches on the type of the argument to the function. The use of `toNothing` in `table` and of `toNil` in the instance for sums is critical: it means that the initial function we generate will be able to respond to all inputs, but will then subsequently allow us to remove entire chunks of the function description. The effectiveness of Claessen's approach is quite astonishing. For example, consider this example from the demo of the main `falsify` library:

```
prop_listToBool :: Property ()
prop_listToBool = do
    Fn (f :: [Word8] → Bool) ←
      gen $ Gen.fun (Gen.bool False)
    assertBool $ f [3,1,4,2] == f [1,6,1,8]
```

The library will quite happily report one of two possible shrunk counter-examples here:

```
{[1,6,1,8]->True, _->False}
```

or

```
{[3,1,4,2]->True, _->False}
```

A true testament to the power of laziness, and a great example of why it can be really important to generate large values before shrinking them to small values (as opposed to try and exhaustively enumerate values of increasing size).

```
toNil :: Gen (a :-> b) → Gen (a :-> b)
toNil = fmap (fromMaybe Nil) . toNothing

table :: ∀a b. (Eq a, Enum a, Bounded a)
      => Gen b → Gen (a :-> b)
table g = Table . catMaybes <$>
            mapM aux [minBound .. maxBound]
  where aux :: a → Gen (Maybe (a, b))
        aux x = toNothing $ (x,) <$> g

class Fn a where
  fn :: Gen b → Gen (a :-> b)

instance Fn ()    where fn = fmap Unit
instance Fn Word8 where fn = table

instance (Fn a, Fn b) => Fn (a, b) where
  fn = fmap Prod . fn . fn

instance (Fn a, Fn b) => Fn (Either a b) where
  fn g = Sum <$> toNil (fn g) <*> toNil (fn g)

instance Fn a => Fn [a] where
  fn = fmap (Map f g) . fn
    where f []     = Left ()
          f (x:xs) = Right (x, xs)

          g = either (const []) (uncurry (:))
```

Figure 9. Generating and Shrinking Functions (excerpt)

QuickCheck depends on *two* classes to do this: `Function`, which constructs concrete functions from regular functions, and `CoArbitrary`, which is used to perturb the state of the random number generator. We use only a single class, which constructs the concrete function directly. We believe it is easier to understand: the user never has to think about the PRNG at all, but just writes generators as usual.

6.3 Testing Shrinking

With manual shrinking, we have to ensure that invariants established by the generator are preserved by the shrinker. This is not necessary when using internal shrinking: when the sample tree is shrunk, we re-run the generator, so any invariants established by generator *must* still hold.[12]

This does not mean we do not have to test shrinking. We already saw an example of an incorrect generator in section 5.2: `wrongBelow` generates a random number below a certain upper limit, but it does not shrink correctly. We would like to be able to *test* whether a generator shrinks correctly.

[12] Of course, this is not true when using `shrinkWith`.

This means we need a new primitive generator, that gives us the current sample tree:

```
peek :: Gen STree
peek = Gen $ λst -> (st, [])
```

This allows us to implement the inverse of `fromShrinkTree` that we saw in section 6.2.1:

```
toShrinkTree :: ∀a. Gen a -> Gen (Tree a)
toShrinkTree g = unfoldTree aux . run g <$> peek
  where aux :: (a, [STree]) → (a, [(a, [STree])])
        aux (x, shrunk) = (x, map (run g) shrunk)
```

With the shrink tree in hand, we can now generate a random path through this tree, and then verify that some property holds between every pair of adjacent values. We do not show the details here, but instead just show what this looks packaged up in the main `falsify` library:

```
testShrinking (>=) ((`mod` 100) <$> gen Gen.prim)
```

This property will result in a counterexample like this:

```
Invalid shrink: 59 ~> 80
Before shrinking: generated "9025540370474205959"
After shrinking: generated "4512770185237102980"
```

where we can see that the value generated by `prim` shrunk correctly, but the output of the generator *increased* instead.

7 Comparison with `Hypothesis`

The `Hypothesis` library taught us to change our perspective and instead of *generation* think of *parsing*, and rather than shrinking the values produced *by* the generators instead shrink the samples that feed *into* parsers. Apart from this key idea, however, our approach has little in common with `Hypothesis`. In this section we will discuss the differences.

7.1 Representation of Random Samples

The first major difference is that `Hypothesis` does not use a tree of samples but rather a list, known as a *choice sequence*, which is passed linearly from generator to generator (cf. section 4.1). An immediately consequence of this choice, and the reason why none of `QuickCheck`, `Hedgehog`, or `falsify` follow it, is that it becomes impossible to write generators for infinite datatypes (such as functions; section 6.2.2), which might require an unbounded number of random samples.

Another benefit is that we get *hierarchical delta debugging* or HDD [15] for free. Consider generating a pair of lists:

```
pairOfLists :: Gen ([Word], [Word])
pairOfLists = (,) <$> .. <*> ..
```

Ideally we should be able to shrink the first list without affecting the second. `Hypothesis` achieves this by marking the part of the choice sequence that a generator uses, and then attempts to respect these boundaries during shrinking. Since generators call other generators (hence "hierarchical"), this is recovering some kind of tree!

For us the two generators use different parts of the sample tree, and so shrinking one will not affect the other, without special precautions.

There is a user-visible benefit of our representation also. When `Hypothesis` first runs a test, it will give it as many random samples as it needs; this initial choice sequence is then fed to the shrinker. But this assumes that shrinking can never result in a demand for more samples; this is *typically* true, but not always. Indeed, the `Hypothesis` paper [13] points out that shrinking should produce "simpler" values, not "smaller", and that this is application dependent.

For example, suppose that we have `Maybe` a value in some test, where `Nothing` means the value is "missing" for some reason, and is considered an exceptional circumstance. In this case, `Just` a value might be "simpler", and we want a generator that can shrink from `Nothing` to `Just`:

```
towardsJust :: Gen (Maybe Word)
towardsJust = do
  missing ← bool
  if missing
    then pure Nothing
    else Just <$> prim
```

If `missing` is initially `True` we produce `Nothing` without asking for further samples. If `missing` now shrinks to `False`, the generator cannot produce `Just` a value, because no more samples are available. By contrast, in `falsify` the generator will simply look at a part of the sample tree it previously ignored, and `Nothing` can shrink to `Just` n for some n.

Section 4.6 showed an example where shrinking could not progress because doing so would result in exposing a different part of the sample tree (which happened not to correspond to a counter-example). Although this cannot *literally* happen in `Hypothesis` (there are no "unexplored parts of the sample tree"), something similar can happen: suppose we have a choice sequence $\ldots n \ldots n' \ldots$, where n causes a choice between parsers p and p'; it is possible that we cannot shrink n because running p' with n' would not result in a counter-example. It is a fundamental limitation of internal shrinking that shrinking sometimes cannot proceed because the samples we feed into a generator came from a unrelated context.

7.2 Shrinking

We now turn to the second major difference between the `Hypothesis` approach and ours: shrinking. Shrinking in `falsify` operates at the level of individual samples, and will never drop or reorder samples. This means that shrinking will never redistribute samples from one parser to another.[13]

[13] In the presence of monadic bind shrinking can of course result in parsers making different choices, thereby repurposing samples that were first used by one generator for use by another; however, this is local and under the user's control.

This is not true in Hypothesis, which considers the choice sequence to be one large value, of which individual parsers use chunks. The choice sequence is shrunk with respect to shortlex ordering [13, Section 2.2]: shorter sequences come before longer sequences, and sequences of the same length are ordered lexicographically.

This has important ramifications, some fundamental, and some that depend on the precise way that shrinking of choice sequences is implemented. First, lexicographic shrinking of the choice sequence has essentially unpredictable local effects on individual generators. For example, dropping a sample ("shifting up" all samples) or sorting a part of the choice sequence redistributes samples to different parsers entirely; this can make it quite difficult to understand why a specific test is shrinking in a particular way. Second, we argued in section 7.1 that this ordering is not always the right one: for some generators, a longer choice sequence might correspond to a "simpler" value.

In addition to these fundamental issues, we also need to consider the implementation of lexicographic shrinking. Hypothesis uses a series of shrinking passes, including passes which

- reduce some values in the choice sequence
- delete regions from the choice sequence
- sort parts of the choice sequence
- shrink one part of the choice sequence whilst delete another in the same step

There are even shrinking targeted at specific generators. The paper mentions that at the time of writing there were 15 such passes in total. This is of course rather ad-hoc, as the Hypothesis paper readily acknowledges [13, Section 3.1]:

> "These passes tend to accumulate organically over time, based on examples we encounter that we feel the reducer should be able to handle and cannot. Several of them are quite specific, but most are generic, and the combination seems to produce good results on most generators we encounter."

This reveals a fundamental difference in design philosophy. The passes in Hypothesis are designed so that even poorly written generators have a chance of shrinking reasonably well; in falsify the author of a generator has a greater responsibility. To a degree this is made less onerous by a suite of standard generators, but it is true that poorly written generators in falsify will probably shrink poorly.

On the other hand, when a carefully constructed generator is shrinking poorly, there is a better chance of being able to improve it in falsify, where the shrinking behaviour of a generator can always be reduced to first principles: one merely needs to consider how the values produced by a generator shrink as the values produced by the generators it depends on shrink. At the root of this dependency tree sits prim, which produces values that shrink towards zero.

There is no need to consider how an ever changing list of reduction passes affects your generator, nor is there a danger of a carefully designed generator suddenly shrinking much more poorly because some choice sequence reduction pass was added, removed, or changed.

8 Related Work

8.1 Aside: Type-based versus Value-based Generators

In this paper we distinguished between *manual* (section 2), *integrated* (section 3), and *internal* shrinking (section 4). There is an orthogonal distinction we can make, between a *type-based* and a *value-based* approach. In a type-based approach, a generator is associated with a type through a type class instance; this is the default in QuickCheck:

```
instance HasGenerator a where
  generator :: Gen a
```

No such class exists in a value based approach, where properties must list generators explicitly. There are good arguments for doing this, as Jacob Stanley discusses in an excellent presentation [20]; it is the default in Hedgehog, Hypothesis and falsify. However, these are *defaults* only, and the choice is mostly independent of the approach to shrinking. We *can* define a type class in Hedgehog that produces a default generator for any type, and conversely, we *can* specify explicit generators in QuickCheck.

This is by means of context for a blog post "Integrated vs type based shrinking"[14] by MacIver, author of Hypothesis. In this blog post MacIver makes the case that integrated is better than manual shrinking, because invariants of the generator are automatically preserved (cf. section 3.1), and that value-based is better than type-based, for many of the same reasons that Stanley points out. However, *integrated versus type based* is a false dichotomy: these are independent choices. This has led to some libraries claiming to be "like Hypothesis" because they used a value-based approach, *not* because they used internal shrinking. It is important to keep this confusion in mind when evaluating related work.

8.2 Manual Shrinking

QuickCheck was introduced in a seminal paper by Claessen and Hughes [5]. Interestingly, that paper put no emphasis on shrinking, relegating it to a short section on "pretty printing" [5, Section 5.4]; it got added to QuickCheck only in version 2. However, its critical importance is now canon (e.g., [9]).

We discussed manual shrinking in section 2. Proponents of integrated shrinking sometimes claim that it is strictly better than manual shrinking because, they claim, shrinking comes for free, "without any extra developer effort to implement a shrinking function"[15].

[14] https://hypothesis.works/articles/integrated-shrinking/

[15] This quote is from "QuickCheck, Hedgehog, Validity", Tom Sydney Kerckhove, 2019. Stanley makes a similar remark in his Lambda Jam presentation.

We saw in section 3 that this is often not the case, due to problems with monadic bind. Internal shrinking addresses this to some degree, although we discussed its limitations in section 4.6, and we saw in section 5 that we must still take care in how we write our generators.

Moreover, manual shrinking has *advantages* too. If we discover that a generator is not shrinking well *when we discover a bug*, it is helpful if shrinking is independent from the generator: modifying the generator may change which counterexample it finds, if indeed it still finds one at all.

Libraries in this category include[16] `junit-quickcheck` for Java, `FsCheck` [1] for .NET, `ScalaCheck` for Scala, `PropEr` [18] and `Triq` for Erlang, `JSVerify` for JavaScript, `kotest` for Kotlin, `QuickChick` [6] for Coq, `base_quickcheck` for O'Caml, `quickcheck` for Rust, `gopter` for Go, and others; there are too many to provide an exhaustive list here. They are mostly a straight-forward adaptation of QuickCheck.

8.3 Integrated Shrinking

Integrated shrinking was introduced in QuviQ QuickCheck for Erlang [3, 8]. After attending a course taught by John Hughes, Reid Draper ported the approach to `test.check` for Clojure (then called `simplecheck`); in the Haskell world it is mostly known from the library `Hedgehog`, which has also been ported to F# and C#, R, and Scala. In O'Caml, recent versions of `qcheck` are also based on integrated shrinking.

We can classify `proptest` for Rust also as implementing integrated shrinking, although the details are different as it is stateful rather than tree-based. But there are equivalents of `Functor`, `Applicative` and `Monad` instances, and here too the monad instance exhibits the poor shrinking problem that we discussed in section 3.2. The library claims to be "inspired by Hypothesis", but it does not implement internal shrinking; this is due to the confusion we discussed in section 8.1.

The situation with `jqwik` for Java and Kotlin is similar. It suffers from the same confusion; its user manual states "`jqwik`'s shrinking approach is called integrated shrinking, as opposed to type-based shrinking which most property-based testing tools use." Although it then refers to the blog post by MacIver, it does not implement internal shrinking, and the user manual warns that monadic bind ("flat mapping" in Java terminology) results in poor shrinking.

8.4 Internal Shrinking

Internal shrinking was introduced in Hypothesis [13, 14]. In addition, the Hypothesis author implemented a minimal library demonstrating the core ideas; known as `minithesis`, this library been ported to various other languages. Martin Janiczek ported the approach in the `elm-explorations/test` library in Elm, which is based on Hypothesis-style internal shrinking as of version 2.0.

[16]We include scientific references where they exist, but do not provide URLs to source code to save space.

The only other library we are aware of that is based on internal shrinking is Theft for C, which introduced support for "auto-shrinking" in version 0.3. Its approach is similar to Hypothesis, manipulating a random bit stream, with various passes and heuristics to modify it, and using an analogue to `getbits` to demarcate sections of the random bit stream.

All of these libraries are explicit implementations of the ideas in Hypothesis, and so our discussion of the differences between `falsify` and `Hypothesis` in section 7 applies to all.

One interesting aspect of Hypothesis is that it supports targeted search [12] (as opposed to purely random generation of test inputs). This is a large area in its own right; it would be interesting to see how this research applies in our context.

8.5 No Shrinking

Some libraries for property based testing do not support any kind of shrinking at all. In some cases, this is because these are libraries that are designed to exhaustively explore small test cases; examples include `SmallCheck` for Haskell [19] and `GAST` for Clean [11]. This is a complementary approach, not a competing one.

Other packages do not attempt to be exhaustive, but try to generate examples of increasing size, hoping that this suffices. Randomised testing in PLT Redex [10], `QC` for C, `RUTE-J` [2] for Java, `crowbar` for O'Caml, and `quick` for Go fall into this category. Experience with QuickCheck and Hedgehog suggests however that this approach is limited: both of these libraries implement an increasing size parameter *in addition to* shrinking.

8.6 Other Related Work

The essence of internal shrinking is the interpretation of a *generator* of random values as a *parser* of samples from a random number source. Goldstein and Pierce [7] formalise this relationship. They focus on the representation of the generator and its properties; there is no analogue in their paper to our sample tree concept.

Property based testing is a huge research area; a full survey is well beyond the scope of this paper. However, we have no reason to believe this work is in any way dependent on the specific way that shrinking works.

In the world of imperative programming there is a lot of research on "parameterised unit tests" and the closely related topic "fuzzing". These approaches are also based on randomly generated inputs and so here too shrinking ("minimisation") is important. Generation tends to be done by the framework rather than by the programmer, however, so concepts such as "compositional generators" or "separating generation from shrinking" do not apply; this makes a comparison between systems of this kind and our work difficult. Similarly, there is work on automatically deriving generators and shrinkers for property based testing; since the purpose of this work is to derive rather than write generators, our work on internal shrinking is probably of limited value in such a setting.

Property based testing can be regarded as an optimisation problem: find the *smallest* counter-example to a property; shrinking is then an optimisation technique. Seen in this light, perhaps the work by Mu and Oliveira [17] can be used to put shrinking on a more formal footing. They define an operator $S \upharpoonright R$, defined in terms of a Galois connection, where intuitively S is the set of all solutions to a problem ("all counter-examples"), R ranks them ("simpler counter-example"), and $S \upharpoonright R$ picks the best one; indeed, they even pronounce this operator as "S *shrunk* by R". Exploring this connection further is future work.

9 Conclusions

The `Hypothesis` library for property based testing in Python showed that if we do not think of a generator as producing a value from a pseudo-random number generator but as a parser of the sampled produced *by* that PRNG, we no longer need to shrink values: we can just shrink the PRNG samples and re-run the generator. In this paper we ported this idea to Haskell; while inspired by Hypothesis, the details are quite different. In particular, shrinking behaviour in `falsify` is more predictable, more controllable (through the use of selective functors), and supports the generation of infinite data structures (such as functions).

Acknowledgements

The author was motivated to do this research by the talk "How to do Property-based Testing Shrinkers Right" by Martin Janiczek at Haskell Exchange 2022; Martin has also provided valuable feedback on drafts of this paper. The discussion of the history of manual and integrated shrinking in section 8 benefited from clarification by John Hughes. Finally, the author thanks Oleg Grenrus, Rodrigo Mesquita, Pi Delport, Jon Fowler as well as the anonymous reviewers for their valuable feedback and comments.

References

[1] B. K. Aichernig and R. Schumi. 2016. Property-Based Testing with FsCheck by Deriving Properties from Business Rule Models. In *ICSTW '16*. https://doi.org/10.1109/ICSTW.2016.24

[2] J. H. Andrews, S. Haldar, Yong Lei, and Felix Chun Hang Li. 2006. Tool Support for Randomized Unit Testing. In *RT '06*. ACM. https://doi.org/10.1145/1145735.1145741

[3] T. Arts, J. Hughes, J. Johansson, and U. Wiger. 2006. Testing Telecoms Software with Quviq QuickCheck. In *ERLANG '06*. ACM. https://doi.org/10.1145/1159789.1159792

[4] K. Claessen. 2012. Shrinking and Showing Functions. In *Haskell '12*. ACM. https://doi.org/10.1145/2430532.2364516

[5] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00*. ACM. https://doi.org/10.1145/357766.351266

[6] M. Denes, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*.

[7] H. Goldstein and B. C. Pierce. 2022. Parsing Randomness. *Proc. ACM Program. Lang.* 6 (Oct 2022). https://doi.org/10.1145/3563291

[8] J. Hughes. 2007. QuickCheck Testing for Fun and Profit. In *PADL '07*. Springer. https://doi.org/10.1007/978-3-540-69611-7_1

[9] J. Hughes. 2016. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane. In *LNTCS 9600*. Springer. https://doi.org/10.1007/978-3-319-30936-1_9

[10] C. Klein and Robert B. Findler. 2009. Randomized testing in PLT Redex. In *Workshop on Scheme and Functional Programming*. Cal Poly Technical Report CPSLO-CSC-09-03.

[11] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. 2003. GAST: Generic Automated Software Testing. In *IFL '03*, R. Peña and T. Arts (Eds.). Springer. https://doi.org/10.1007/3-540-44854-3_6

[12] A. Löscher and K. Sagonas. 2017. Targeted Property-Based Testing. In *ISSTA '17*. ACM. https://doi.org/10.1145/3092703.3092711

[13] D. R. MacIver and A. F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In *ECOOP '20*, R. Hirschfeld and T. Pape (Eds.). Schloss Dagstuhl. https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

[14] D. R. MacIver and Z. Hatfield-Dodds et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019). https://doi.org/10.21105/joss.01891

[15] G. Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *ICSE '06*. ACM. https://doi.org/10.1145/1134285.1134307

[16] A. Mokhov, G. Lukyanov, S. Marlow, and J. Dimino. 2019. Selective Applicative Functors. *Proc. ACM Program. Lang.* 3 (Jul 2019). https://doi.org/10.1145/3341694

[17] Shin-Cheng Mu and J. N. Oliveira. 2012. Programming from Galois connections. *The Journal of Logic and Algebraic Programming* 81, 6 (2012). https://doi.org/10.1007/978-3-642-21070-9_22

[18] M. Papadakis and K. Sagonas. 2011. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In *Erlang '11*. ACM. https://doi.org/10.1145/2034654.2034663

[19] C. Runciman, M. Naylor, and F. Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. *SIGPLAN Not.* 44, 2 (Sep 2008). https://doi.org/10.1145/1411286.1411292

[20] J. Stanley. 2017. Gens N' Roses: Appetite for Reduction. Presentation at Lambda Jam 2017.

[21] Guy L. Steele, Doug Lea, and Christine H. Flood. 2014. Fast Splittable Pseudorandom Number Generators. In *OOPSLA '14*. ACM. https://doi.org/10.1145/2660193.2660195

Received 2023-06-01; accepted 2023-07-04