Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer

David R. MacIver
Imperial College London, United Kingdom
david@drmaciver.com

Alastair F. Donaldson
Imperial College London, United Kingdom
alastair.donaldson@imperial.ac.uk

Abstract

We describe *internal test-case reduction*, the method of test-case reduction employed by Hypothesis, a widely-used property-based testing library for Python. The key idea of internal test-case reduction is that instead of applying test-case reduction *externally* to generated test cases, we apply it *internally*, to the sequence of random choices made during generation, so that a test case is reduced by continually re-generating smaller and simpler test cases that continue to trigger some property of interest (e.g. a bug in the system under test). This allows for fully generic test-case reduction without any user intervention and without the need to write a specific test-case reducer for a particular application domain. It also significantly mitigates the impact of the *test-case validity* problem, by ensuring that any reduced test case is one that could in principle have been generated. We describe the rationale behind this approach, explain its implementation in Hypothesis, and present an extensive evaluation comparing its effectiveness with that of several other test-case reducers, including C-Reduce and delta debugging, on applications including Python auto-formatting, C compilers, and the SymPy symbolic math library. Our hope is that these insights into the reduction mechanism employed by Hypothesis will be useful to researchers interested in randomized testing and test-case reduction, as the crux of the approach is fully generic and should be applicable to any random generator of test cases.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases Software testing, test-case reduction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2020.13

Category Tool Insights Paper

1 Introduction

When generating test cases to discover bugs in a system under test (SUT), it is common to use *test-case reduction* [12, 23], where large and difficult to read test cases are transformed into smaller and more readable versions, as an aid to debugging the problems discovered. Tools for automating this process are called *test-case reducers*, or reducers for short. Test-case reducers are especially important when using random test-case generation (henceforth "random generation"), which often produces large and messy initial test cases [2, 23].

This presents a particular problem for *property-based testing* libraries [2, 1] which augment unit tests with randomly generated test cases, as each type of generated test case typically requires its own test-case reducer. When generating domain-specific types with no predefined test-case reducer, users who want test-case reduction must either write their own or use one of various approaches to generic test-case reduction which attempt to derive a suitable reducer automatically.

We present an alternative approach that we call *internal test-case reduction* (henceforth "internal reduction"), which allows one to build reduction into the generation process itself. Our presentation is based on the implementation of internal reduction in Hypothesis [18], a

© David R. MacIver and Alastair F. Donaldson;
licensed under Creative Commons License CC-BY

widely[1] used Python library for property-based testing. Internal reduction has been the only supported method of test-case reduction in Hypothesis since early 2016, so we consider it to be a mature and well-established technology, but it has not previously been described in the literature and does not appear to be widely known. The aim of this paper is to explain the idea of internal reduction in detail to the research community, provide insights into how it is used within Hypothesis, and illustrate the practical pros and cons of the approach via an experimental comparison with various other test-case reducers.

The key idea of internal reduction is to manipulate the underlying source of randomness consumed by a random generator, in order to cause the generator to produce smaller test cases automatically. The final reduced test case is constructed as if the generator had been implausibly lucky and produced a small and readable test case by chance.

The advantages of internal reduction over other approaches are twofold. First, given an existing internal reducer, every test-case generator comes with test-case reduction *for free*, without the need to write an external reducer. Second, because internal reduction works by re-generating test cases, any reduced test case is one that *could* have been generated. If the generator has been carefully engineered to guarantee that all generated tests satisfy certain properties, these properties will be satisfied *automatically* by the reduced test case. This helps users avoid the *test-case validity problem* [23], where reduced test-cases fail to satisfy necessary preconditions for the test. As a result, users hardly need to know that test-case reduction exists, and can just take the fact that test cases are presented in a reduced form as a given. Users must of course still ensure that their generators only produce valid test cases, but this is a problem they must solve anyway, and in practice it is often easier to construct a valid test case than it is to verify whether an arbitrary test case is valid.

Note that reduction quality or performance are *not* included among the major advantages of internal reduction. Anecdotally, and as we will provide evidence for in Section 4.3, test-case reduction in Hypothesis produces moderately better results than that found in other property-based testing libraries, but can be a fair bit slower. Based on extensive conversations with users of Hypothesis and other property-based testing libraries, we consider the user experience benefits to be worth the performance cost (and the slightly better results to be a nice bonus on top of that), but we cannot present any particularly compelling argument for this trade off beyond that experience.

Our main contributions are:

- The key idea of internal reduction, which we cast as a *shortlex optimization* problem over the choices made during generation (Section 2).
- A description of its implementation in Hypothesis (Section 3).
- A large evaluation demonstrating that Hypothesis's internal reduction is reasonably competitive with other test-case reducers, based on bugs found in the clang and gcc compilers by Csmith [25] (a generator of C programs), differential testing of two Python autoformatters, a set of experiments testing SymPy[2] (a symbolic algebra library) using TSTL (a domain-specific language for testing [10]), and a reimplementation of a set of synthetic benchmarks for QuickCheck's reduction that were proposed in [22] (Section 4).

We also discuss threats to the validity of our results (Section 5), related work (Section 6) and present our conclusions and future goals (Section 7).

---

[1] Usage is difficult to measure precisely, but it is used in thousands of open source projects and has over 100,000 downloads per week. See [18] for more details of usage.

[2] `https://www.sympy.org`

2 Foundations of Internal Reduction

In this section we present the key idea of internal reduction. We start with a brief account of test-case reduction more broadly (Section 2.1), then discuss how these ideas can be applied to the decisions made during generation to implicitly reduce a generated test case (Section 2.2), then finish with a worked example showing how a generated test case is transformed in the course of reduction (Section 2.3).

2.1 Test-Case Reduction Fundamentals

The starting point of test-case reduction is that we have some user-specified *interestingness test* that takes a test case and determines whether it is in some sense "interesting" – generally whether it triggers a specific bug in some system under test – and some known interesting test case. The goal is to find an interesting test case which is "more readable" than the initial one, which is typically quite large and complicated.

Exactly what counts as more readable is fairly under-defined. The ultimate goal is to improve the user's debugging experience, but this is hard to quantify. Past work on test-case reduction has identified three key features that seem generally helpful: 1) Smaller test cases are better [12, 23], 2) Users should be able to predict what features of a test case the reducer can remove, as this allows them to infer that any remaining features in the reduced test case are important [1], and 3) Test-case reducers should ideally *normalize* their input to a canonical interesting test case for each interestingness test [9].[3]

In support of these goals, we find it useful to think of any given test-case reducer as having a *reduction order*: a total order over all test cases ordering them from best to worst. The goal of reduction is then to find the reduction-order-minimal interesting test case.

A normalizing reducer would always find this minimal test case, but this requires brute force enumeration, which is typically infeasible. Reducers used in practice are instead only local minimizers, making small transformations to an interesting test case and checking if the transformed version is still interesting. Typically these transformations are organized into "reduction passes" and the reducer runs until it finds an interesting test case that no pass is able to reduce further.

2.2 Internal Reduction as Shortlex Optimization

Internal test-case reduction works by manipulating the underlying "random" behaviour in random generation, so we first discuss the structure of a random generator.

A random generator can be thought of as a true random variable taking values in some domain, but in practical implementations they are otherwise-deterministic functions that take a pseudo-random number generator (PRNG) and return some value. A PRNG provides an interface that the generator requests bits from, with each bit corresponding to a nondeterministic binary decision (a "coin flip"). As the PRNG is the only source of nondeterminism, any generated test case can be deterministically recreated from these binary decisions that led to it.[4] We call these sequences of binary decisions *choice sequences*, and view random generators as parsers of choice sequences, with the PRNG as a stream interface for reading the next bits from some underlying choice sequence.

[3] In practice no test-case reducers satisfy this condition in general, and the goal is only to approximate it in common cases.

[4] This is essentially a variant on the widely known observation that you can recreate the generated value from the seed that produced it.

A PRNG can produce infinitely many choices, but we can turn a generator into a parser of *finite* choice sequences by raising an exception when the generator reads too many bits, treating a too-short choice sequence as a parse error in the language defined by considering the generator as a parser. A generator must terminate after having made only finitely many choices, so any generated test case is the result of parsing some finite choice sequence.

Internal reduction works by performing test-case reduction not on the generated test case, but on the choice sequence that lead to it, with the hope that the test case corresponding to the reduced choice sequence is an improvement on the original. Although we cannot expect this to be true in every case, in this section we argue why, given a suitable reduction order, it is plausible that it would work for most "natural" random generators.

In our implementation of internal reduction in Hypothesis, the reduction order is the well-known shortlex order [24]: For choice sequences s and t, s is shortlex-smaller than t if $|s| < |t|$ or if $|s| = |t|$ and s is lexicographically smaller than t. Thus, internal reduction is *shortlex optimization* over the choice sequences leading to interesting generated test cases.

We now outline why this choice of shortlex order is a natural one.

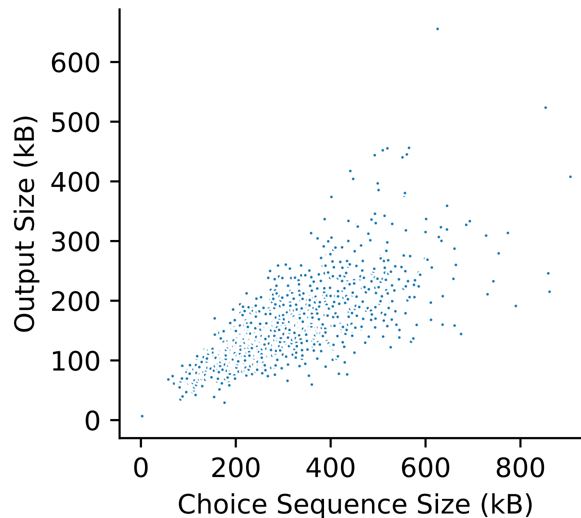

Figure 1 Input size vs output size for Csmith.

First, we justify that reducing the length of a choice sequence will typically reduce the size of the corresponding generated test case. This is fairly intuitive: Any part of the generated test case has to be constructed by the generator, and this will usually involve a series of nondeterministic choices, so parts of the test case that contribute to its size will correspond to regions of the choice sequence where they were generated. In the other direction, regions of the choice sequence correspond to decisions made during generation, so will usually appear as some part of the generated test case.

To see an example of this in practice, in Figure 1 we show the relationship between choice sequence length and generated program size in bytes for Csmith [25], a widely used generator of C programs. Here, the Pearson's correlation coefficient between choice sequence and test case size is 0.73, i.e. the length of the choice sequence is a strong but not perfect predictor of the test case size.

The relationship between choice sequence and test case size can break down for a number of reasons. For example, it is common to use *rejection sampling* during generation. In rejection sampling, one retries generating some value until it satisfies some predicate. For

example to generate a number between 0 and 9 one might generate a 4 bit integer (which will be between 0 and 15) and discard the generated integer and try again if it is greater than 9. This rejection process may in principle be repeated many times, which can result in many choice sequences of different sizes, all producing the same value.

An additional common example where choice sequence size is not reflected by output size is that integers will typically be generated as a fixed number of bits. We might, reasonably enough, want reduction to reduce integers towards zero (and doing so will reduce the size of their text representation), but all possible values have the same size of underlying choice sequence, so this reduction cannot be made by reducing the number of bits drawn, only by changing their contents.

This example informs what our reduction order should be between two choice sequences of the same length. If we want fixed width integers to reduce towards zero then, depending on whether we draw these integers in big or little endian order, choice sequences that differ only in regions corresponding to a single integer should be ordered based on either the lexicographic or co-lexicographic (i.e. lexicographic from right to left) order for that region.

It is natural to extend this to the whole choice sequence, suggesting that among choice sequences of the same length we should prefer either the lexicographically or co-lexicographically smaller of the two. The choice between the two is fairly arbitrary, but we picked the lexicographic ordering in Hypothesis because it corresponds with the “time ordering” of random generation, by prioritizing decisions made earlier in the generation process, as they potentially have more impact on the generated test case.

2.3 Shortlex Optimization by Example

We now show a worked example of how the generated test case might change as the underlying choice sequence is reduced through a series of local shortlex optimization. The transformations we will show in this section do come from an actual run of Hypothesis, but we defer discussion of how these specific transformations might have been chosen to Section 3.

Our example is as follows: Suppose we have a system under test (SUT) that takes binary trees as inputs, and that it crashes when given a height imbalanced tree (i.e. some branch of the tree has two children whose heights differ by more than one).

We could test this SUT using the Python code of Figure 2 to randomly generate inputs to it. After running the SUT against several generated inputs, we might discover the tree shown along with its associated choice sequence at the top-left of Figure 3.

```
1 class Tree(Generator):
2     def do_draw(self, source):
3         if source.getbits(1):
4             return Branch(source.draw(self), source.draw(self))
5         else:
6             return Leaf()
```

Figure 2 A simple binary tree generator. This code assumes `Branch` and `Leaf` classes for internal and leaf nodes. For ease of presentation, this generator has expected infinite size; a better one would be slightly leaf-biased.

This initial tree is moderately complicated, so we wish to find a smaller, simpler, tree, that will help us understand this bug. Rather than using *external* reduction, operating on the trees themselves, our *internal* reducer instead transforms the choice sequences producing them. We show the these choice sequences in Figure 3, along with the corresponding trees produced when running the generator of Figure 2 on them.

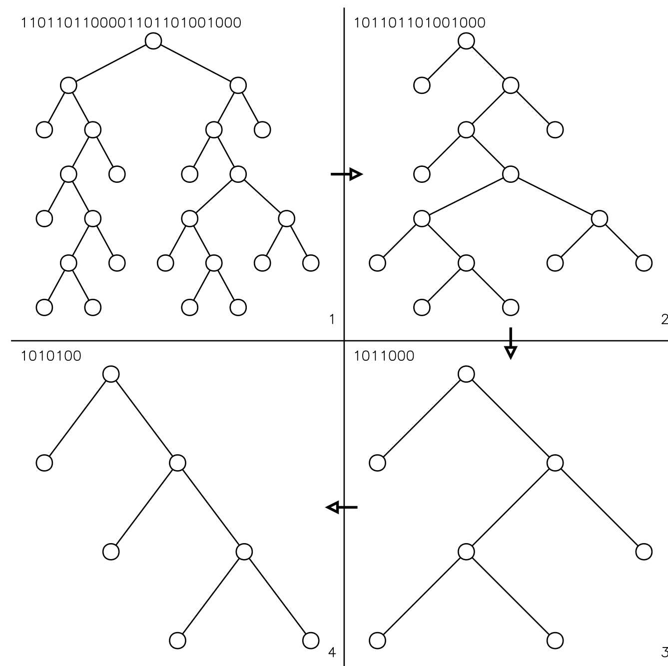

Figure 3 Successive reductions of choice sequences leading to unbalanced trees.

These transformations proceed as follows: Starting from our initial randomly generated choice sequence, labeled 1, the reducer performs the transformations $1 \to 2 \to 3$ by replacing long sequences of bits with shorter sequences of zero bits, first transforming "1101101100001…" to "101…", then "1011011101001000" into "10110000". These transformations correspond to collapsing a subtree into a single leaf node, but we emphasize that the transformations operate on the underlying choice sequences, without reference to generated data. Finally, in the transformation $3 \to 4$, the reducer swaps two bits, transforming "10110000" into "10101000". This swaps two subtrees, but once again is performed without any knowledge of the SUT's data domain.

In this case, the reducer in fact finds the shortlex minimal choice sequence leading to an unbalanced tree, which can be seen in quadrant 4 of Figure 3, although in general the result will only be locally minimal.

In the course of finding these transformations, the reducer will have tried many other "failed reductions" – choice sequences that did not yield interesting test cases, either due to generating balanced trees or providing too few bits for generation of a complete test case to succeed (the left and right examples of Figure 4 respectively).

3 The Design of the Hypothesis Reducer

In this section we outline some interesting details of the Hypothesis reducer, and the rationale behind them. It will likely be of greatest interest to readers who want to learn about the intricate details of implementing a test-case reducer.[5] Readers who care more about our high level claims may wish, at least initially, to simply regard the Hypothesis reducer as a black box and to skip to Section 4 for our evaluation of its effectiveness.

[5] The *very* interested reader may also wish to consult the source code, which is self-contained and reasonably well documented. `https://github.com/HypothesisWorks/hypothesis/blob/master/hypothesis-python/src/hypothesis/internal/conjecture/shrinker.py`

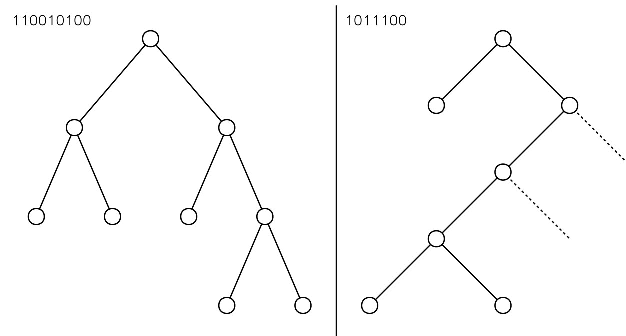

Figure 4 Some failed choice sequences arising during the reductions in Figure 3. The dashed lines represent branches that could not be generated due too short choice sequences.

3.1 A Summary of Reduction Passes

The Hypothesis reducer follows the common pattern of dividing reduction into different passes, each of which perform different classes of transformation designed to reduce interesting test cases in the shortlex order. We now provide a brief summary of these passes.

In Hypothesis 5.15.1 (which was recent at the time of this writing), the reducer contains 15 passes consisting of:

1. Six passes that delete contiguous regions of the choice sequence.
2. A pass that replaces a contiguous region of the choice sequence with a sub-region.
3. A pass for replacing a contiguous region of the choice sequence with a, possibly shorter, zeroed sequence of choices.
4. Four passes for pure lexicographic reduction.
5. Three passes for common patterns that require simultaneously lexicographically reducing some parts of the choice sequence while deleting others.

These passes tend to accumulate organically over time, based on examples we encounter that we feel the reducer should be able to handle and can't. Several of them are quite specific, but most are generic, and the combination seems to produce good results on most generators we encounter. The most specific of these by far is that one of the lexicographic passes is entirely a special case for Hypothesis's floating point generator. We discuss this further in Section 3.3.

3.2 Generator-directed Reduction

One of the biggest obstacles with test-case reduction on sequences (e.g. choice sequences for internal reduction, or file-based external test case reducers) is finding transformations that preserve some sense of syntactic validity, as syntactically invalid test cases will rarely be interesting. A classic technique here is hierarchical delta debugging (HDD) [19], which uses a grammar to find regions to delete.

In comparison to formats designed for human consumption, the choice sequence format is relatively forgiving – the only way a choice sequence can be invalid is for it to be too short.[6] This gives the reducer a reasonable amount of leeway in making changes, and often allows it to find valid reductions by accident when some change at the choice sequence level makes essentially arbitrary changes to the generated test case.

[6] In Hypothesis a generator may also explicitly declare a choice sequence to be invalid. We have omitted details of this for clarity of presentation.

```
class Generator(object):
  def do_draw(self, context):
    raise NotImplementedError()


class Source(object):
  def __init__(self, prefix=()):
    """A Source object such that the i'th
    call to getbits returns prefix[i] (possibly
    truncated) and after that is random."""
    self.prefix = prefix

    # Records the bits drawn
    self.record = []
    self.draw_stack = []
    # Records (start, end) positions for draws
    self.draws = []

  def getbits(self, n):
    """Returns an n-bit integer."""
    i = len(self.record)
    if i < len(self.prefix):
        result = self.prefix[i] & ((1 << n) - 1)
    else:
        result = random.getrandbits(n)
    self.record.append(result)
    return result

  def draw(self, gen):
    """Returns the result of gen.do_draw(self)"""
    self.draw_stack.append(len(self.record))
    result = gen.do_draw(self)
    self.draws.append((
      self.draw_stack.pop(), len(self.record)))
    return result
```

Figure 5 A simplified implementation of the Hypothesis API.

Set against this, many transformations that make perfect sense at the level of the generated test case may be highly non-obvious at the level of the choice sequence without additional information about how it will be used. As we saw in the binary tree example of Section 2.3, we might be able to collapse a subtree into a single leaf by replacing some sequence of bits with a single zero bit. However, there are $O(n^2)$ possible contiguous subsequences of the choice sequence, and if we don't know how long a sequence of 0 bits to use this adds an additional $O(n)$ possibilities, giving $O(n^3)$ transformations to consider for what we would be $O(n)$ in the size of the tree for an external reducer!

If the reducer had structural information about what regions of the choice sequence corresponded to a subtree, it could similarly restrict its attention to only $O(n)$ suitable regions of the choice sequence. The key observation that Hypothesis uses to get access to this boundary information is that although we do not have a *grammar* for the language, we do have a *parser* – the generator itself – and by instrumenting the API it uses we can implement something akin to HDD, allowing us to discover transformations of the choice sequence that would be difficult to discover otherwise.

We outline this instrumented API in Figure 5. Generators are constructed as an instance of a `Generator` class, which are passed to a `draw` method on a `Source` object. The `Generator` object records the results of `getbits` calls and how these correspond to `draw` calls, which

```
def zero_draw(source):
  """Attempt to replace regions corresponding to draw calls with
  sequence of all zero bits, if doing so would not increase the length."""

  i = 0
  while i < len(source.draws):
    u, v = source.draws[i]
    prefix = source.record[:u]
    suffix = source.record[v:]

    # Attempt to replace the draw with a zero sequence of the same length
    attempt = Source(prefix + [0] * (v - u) + suffix)
    if is_interesting(attempt) and len(attempt.record) <= len(source.record):
      source = attempt
    else:
      # If the number of bits was wrong, try again with the right number.
      u2, v2 = attempt.draws[i]
      if v2 < v:
        attempt = test_function(prefix + [0] * (v2 - u2) + suffix)
        if (
          is_interesting(attempt) and
          len(attempt.record) <= len(source.record)
        ):
            source = attempt
    i += 1
  return source
```

Figure 6 Replacing a draw with all zero bits.

can be used to suggest modifications to the choice sequence. In particular, for our recursive generator of Figure 2, each subtree corresponds to a single `draw` call whose start and end points are recorded on the `Source`.

A useful analogy is to consider the `draw` calls as defining the grammatical structure of the choice sequence format, while the `getbits` calls define the lexical structure. This structure often allows us to make transformations at the choice sequence level that naturally mirror the ones that a dedicated external reducer would have made to the generated test cases, without knowing any further details about what those generated test cases are.

In Figure 6 we present Python pseudo-code that shows how a reduction pass might try to replace all `draw` calls with a (possibly shorter) sequence of zero bits, one of the passes we mention in Section 3.1. Unlike the brute force $O(n^3)$ approach, running this pass attempts only $O(n)$ possible transformations[7].

In our worked example in Section 2.3, the pass of Figure 6 is what allows us to replace any subtree with a leaf: e.g. First it might try transforming "1101101100001..." to "1000000000001...", which would produce a valid but uninteresting choice sequence, and then it would observe that fewer choices were made in the target draw than expected, so it would try again with the single zero bit that was used, leading to the sequence "101..." that we saw in Figure 4.

[7] This assumes that every `draw` call contains at least one `getbits` call, but where this is not the case the results can be cached, a detail we omit here.

3.3 Generator / Reducer Co-design

There is a certain amount of co-design between Hypothesis's library of generators and its reducer. We show in Section 4.1.2 that this co-design isn't strictly necessary, in that the Hypothesis reducer produces reasonable results without it, but we have nevertheless found it useful.

The co-design occurs when we encounter an example that reduces poorly, requiring us to modify one or both of the generator or the reducer. Typically, when the example is user provided we will modify the reducer, and when it is part of the Hypothesis library of generators, we will modify the generator to be more "reduction friendly", but in some cases it is still better handled by modifying the reducer.

In particular, as we mention in Section 3.1, there is a special case for our floating point generator. This generator is designed so that lexicographic reduction will produce "visually simpler" floating point numbers. This is important because if a float was generated as its IEEE representation it would instead reduce towards 0.0, which tends to produce reduced test cases that look pathological. e.g The most reduced non-zero double precision float would be 5e-324, when ideally we would like to reduce non-zero floats to 1.0.

This results in certain transformations that look very natural to a human reader but are quite complicated at the choice sequence level. e.g. 9.0 is represented as a 64-bit integer value of 9 in our internal float encoding, but 9.1 is represented as 9237896145653045656. Although going from the latter to the former is an obvious reduction to a human reader, and is a lexicographic reduction at the choice sequence level, it would be quite hard for the reducer to discover on its own. As a result, it was worth adding a special case to our implementation to make it aware of transformations that were obvious at the floating point level but not at the underlying choice sequence level.

The floating point generator is the only case we've encountered that required this level of special casing, and this was largely only needed due to the relative complexity of the floating point format. Additionally, it was only worth it because it is such a foundational generator: If it had not been part of our core library, it would likely not have been worth investing much time in it, and so it would have been left with the default behaviour which, while suboptimal, was still relatively adequate.

More commonly, it is worth designing core generators to aid the performance of test-case reduction, because some designs make it easier to find relevant reductions. Users are not expected to need to do this, but the cost-benefit trade off is different for the core Hypothesis library of generators, as they are more widely used and we have greater expertise in the behaviour of the reducer.

To illustrate this, in Figure 7 we show an example of how one might[8] generate lists using Hypothesis. This generator arranges matters so that an element of the list can be deleted by deleting a contiguous region of the choice sequence, corresponding to the `getbits` call followed by a subsequent `draw`. Deleting the region corresponding to these two calls effectively causes the loop to skip over the iteration where the generated element would previously have been added.

In contrast, if we generated lists by first drawing a length parameter and then drawing that many elements, deleting an element of the list would require first lowering that length parameter and then deleting a later part of the choice sequence. Identifying all such pairs would require $O(n^2)$ transformations.

[8] For simplicity, this generator elides details which control the expected size of the list. The real version also contains a hint to the reducer about what regions are worth deleting.

```
1  class ListGenerator(Generator):
2      def __init__(self, elements):
3          self.elements = elements
4
5      def do_draw(self, data):
6          results = []
7          while True:
8              more = data.getbits(1)
9              if more:
10                 results.append(data.draw(self.elements))
11             if not more:
12                 break
13         return results
```

Figure 7 A simplified list generator.

Hypothesis does in fact have a reduction pass that does this, because such patterns are common in user code, but its performance is comparatively poor due to the large number of transformations to be tried, and so we have used the implementation that allows for more efficient reduction.

An additional benefit of this is that, because it is relatively easy to transform the choice sequence in ways that preserve the structure of the generated list, other reductions become possible. For example, due to trying to delete short subsequences of the choice sequence, when generating lists of lists, Hypothesis will try merging adjacent lists (e.g. transforming $[[1, 2], [3, 4]]$ into $[1, 2, 3, 4]$), because this corresponds to deleting the choices in two adjacent calls to `getbits`.

4 Case Studies and Experiments

In this section we present data on internal reduction in Hypothesis, comparing the cost of reduction and final size of reduced test cases to those of existing external reducers.

Our goal is not to show that internal reduction is especially impressive on these metrics. As we discuss in Section 1, the primary benefits of internal reduction are not its performance or the quality of the end results, but that it provides adequate reduction for any generated test case, while avoiding the test-case validity problem. As such our evaluation is mostly intended to be descriptive, and to increase the plausibility of our claim of adequacy.

We structure our evaluation around the following research questions:

1. How does the size of the final test case obtained through internal reduction compare to that obtained through external reduction? (RQ1)
2. How expensive is internal reduction compared to external? (RQ2)
3. How much overhead does the process of going through the generator introduce? (RQ3)

In addressing these research questions we primarily focus on future-proof metrics that are independent of our particular experimental setup: the number of SUT and generator invocations. Unlike the metrics, wall clock time is sensitive to specific implementation choices in Hypothesis and the tools against which we compare, and other engineering issues such as the choice of implementation language. Furthermore, to make our large study feasible, experiments were performed in parallel on a multi-core machine, with associated impact on wall clock time variance.

Our main three evaluations use Hypothesis to find and reduce real bugs in three classes of real world software:

- We used a modified version of Csmith to allow Hypothesis to generate C programs, which we used to trigger bugs in old versions of the open source C compilers, gcc and clang (Section 4.1);
- We wrote a custom generator of Python programs and used it to perform differential testing of yapf [7] and black [17], two open source Python autoformatters (also Section 4.1);
- We implemented a Hypothesis-based test harness to trigger bugs in SymPy, an open source symbolic algebra library (Section 4.2).

For completeness, we also compared Hypothesis on a series of synthetic benchmarks used in [22] to evaluate SmartCheck, a proposed generic test-case reducer for QuickCheck (Section 4.3).

We note that while we have been able to apply Hypothesis to this relatively diverse range of applications, in order to compare with a number of different test-case reduction tools, no one of the tools that we compare with could be easily applied to all of these case studies. This is an important selling point for internal reduction: it works at the level of choice sequences, and any randomized generator can be relatively easily adapted to consume a choice sequence instead of using a pseudo-random number generator, thus internal reduction has wide applicability.

We have made the code for reproducing the data for these experimental results available at `https://github.com/mc-imperial/hypothesis-ecoop-2020-artifact`.

4.1 Evaluation on Generated Programs

We designed a system for running controlled reduction experiments on Hypothesis-generated examples and used it to run tests on real world bugs found by two different program generators:

1. A patched version of Csmith,[9] which uses Hypothesis as its source of entropy, where any calls to methods named `make_random` were wrapped in a macro so as to show up as if they were a generator passed to `draw`.
2. A generator of syntactically valid Python programs that we wrote ourselves using Hypothesis's library of generators.

For each of these generators we wrote interestingness tests that would use the generated test cases to look for bugs in some real world software. For Csmith-generated programs, these were crashes or wrong code bugs in old versions of gcc and clang. For Python programs produced by our generator, we used them to test a Python autoformatter, yapf, and checked its output for style violations.

For each of these generators and their corresponding interestingness test, we built a corpus of 200 choice sequences that resulted in interesting test cases. We then ran reduction for each of these starting points using each of: 1) Internal reduction provided by Hypothesis; 2) *C-Reduce* [23], a test-case reducer primarily designed for C programs but suitable for any text format; 3) *Picire* [13], a modern implementation of the classic delta-debugging algorithm.

We explain this experimental setup in more detail in Section 4.1.1, and then present the results in Section 4.1.2

[9] `https://github.com/HypothesisWorks/csmith`

4.1.1 Experiment Design

For each experiment we defined a class of bugs we were looking for, with precise interestingness tests for identifying each possible bug.

For the generator of Python programs, we used it to perform differential testing of yapf, a Python source code autoformatter developed at Google, against black, a more recent and more widely used autoformatter. The test we performed was that we ran black on the generated source, followed by pycodestyle,[10] a style checker for conformance to PEP8, the official Python style guide. If there were no style errors, we then ran yapf on the black-formatted source. Any style errors introduced constituted a bug in yapf, as it had taken source code that it was possible to format correctly and introduced a style violation.

For Csmith, we ran a large number of old versions of gcc and clang,[11] at four different optimization levels (-O0, -O1, -O2, -Os). This could produce three distinct types of bug: The compiler could crash, the compiled binary could crash when run, or there could be a miscompilation, determined when the output differed from that on gcc 8.3.0 (the latest of the compilers tested) compiled at -O0. Whenever an example triggered multiple bugs we associated it with the bug it triggered in the latest compiler, at the lowest optimization level for that compiler, as this seemed like a reasonable proxy for how interesting the bug was.

The need for a validity oracle for C-Reduce and Picire. Csmith guarantees generating C programs that are free from undefined behaviour by construction [25]. When we drive Csmith via Hypothesis, test-case reduction involves using Csmith to generate successively simpler programs, each of which is thus free from undefined behaviour by construction. In contrast, neither C-Reduce nor Picire provides such a guarantee. In order to use these reducers we had to define a *validity oracle* that detected if the program was likely to be free from undefined behavior. The validity oracle that we used compiled the program with clang and GCC and checked for warnings likely to indicate undefined behavior, as recommended in the C-Reduce documentation,[12] and in addition ran the generated binary under UBSan[13] to look for non-trivial undefined behavior that was only detectable at run time. We did not apply the validity oracle when reducing compiler crash bugs, as the execution result of the program is not relevant in such cases.

We generated a corpus for each experiment by sampling choice sequences of length up to 8KB (Hypothesis's default maximum size) until we had 200 choice sequences that triggered bugs for each experiment. For the Python generator, this small buffer size was not a problem, but for Csmith this was a significant restriction – when generating a corpus without this size restriction we found only about 2% of choice sequences corresponding to programs triggering bugs were under 8KB. This is corroborates previous observations in [25] that Csmith is most effective when generating large programs. We attempted to run the Csmith experiment with a larger buffer size, but unfortunately Hypothesis is not currently well designed for larger sizes and we hit some memory limitations, so we decided to restrict ourselves to examples within Hypothesis's normal operational parameters.

For each corpus member and each reducer, we ran the reduction to completion, instrumented so as to record SUT calls and report on successful reductions.

---

[10] `https://pycodestyle.pycqa.org/en/latest/`
[11] All of those installed by `https://github.com/mattgodbolt/compiler-explorer-image/blob/master/update_compilers/install_compilers.sh`. This included gcc versions ranging from 4.1.2 to 8.3.0 and clang versions ranging from 3.9.1 to 7.0.0, but not every patch release in that range.
[12] `https://embed.cs.utah.edu/creduce/using/wrong1/test1.sh`
[13] `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

	Csmith	Python
No Reduction	1963.9 (1750.3–2230.2)	417.2 (365.8–483.0)
C-Reduce	120.0 (114.2–126.2)	70.9 (64.5–76.8)
Hypothesis	812.3 (786.8–843.1)	71.8 (64.7–78.8)
Picire	345.09 (321.3–375.5)	75.7 (68.8–82.4)

Figure 8 Mean sizes, measured as number of bytes after formatting, of final examples for each reducer.

4.1.2 Experimental Analysis

In order to answer RQ1, we have to define a suitable notion of size. The number of bytes is the obvious choice, but one subtlety to consider is that many size reductions are both impossible in internal reduction, and also undesirable! For example, removing whitespace is often a valid reduction in size that reduces readability. In order to offset this, instead of raw size we consider formatted size. For each experiment we used a standard automatic formatter, clang-format[14] for C programs and black for Python programs, and consider the size of the formatted result. We also strip comments from the C programs.

We justify this as a reasonable metric by observing that the purpose of test-case reduction is not actually to reduce size, but rather to ease debugging. A formatter is designed to improve the readability of the code (and it is often worth formatting reduced test cases to understand them better), and a human reader is unlikely to pay attention to the comments unless they are an aid to understanding, so this is a truer representation of the size a human reader sees.

We calculated the size of the reduced test case for each test case and reducer, and report the mean size in Figure 8 alongside 95% bootstrap confidence intervals. A permutation test for difference of means shows that the differences between these means are significant for all three reducers on the Csmith experiment ($p < 10^{-5}$ for C-Reduce vs each of the others, $p \approx 0.0003$ for Hypothesis vs Picire), and non-significant at a threshold of 0.05 for all pairs on the formatting example.

We discuss RQ1 separately in the context of the Python formatting experiments and the Csmith experiments, as the findings are substantially different.

Python formatting results for RQ1. Figure 8 shows that, for the Python formatting case study, Hypothesis, Picire and C-Reduce perform comparably well (with overlapping confidence intervals regarding reduced test case size, and nonsignificant differences in means). As we discuss above, our claim is that internal reduction should work well enough to be useful, not that it should out-perform other reduction approaches, and these results support that claim.

Csmith results for RQ1. The results of Figure 8 show that C-Reduce and Picire are able to achieve substantially smaller reduced programs than Hypothesis on average. Regarding our aim that internal reduction should be good enough to be useful: the reduction factors associated with Hypothesis in Figure 8 would certainly be worth having for debugging purposes if no other reducer were readily available, and the fact that test cases retain the Csmith guarantee of validity when reduced using Hypothesis is a potentially important bonus (especially for wrong code bugs) that the size results of Figure 8 do not show.

[14] `https://clang.llvm.org/docs/ClangFormat.html`

```
1  #include "csmith.h"
2  static long __undefined;
3  static int8_t func_1(void);
4  static int8_t func_1(void) {
5    int8_t l_2 = 0L;
6    return l_2;
7  }
8  int main(int argc, char *argv[]) {
9    int print_hash_value = 0;
10   if (argc == 2 && strcmp(argv[1], "1") == 0)
11     print_hash_value = 1;
12   platform_main_begin();
13   crc32_gentab();
14   func_1();
15   platform_main_end(crc32_context ^ 0xFFFFFFFFUL, print_hash_value);
16   return 0;
17 }
```

Figure 9 The minimal size Csmith program that Hypothesis could find, which is essentially the smallest program that Csmith can generate.

Nevertheless, Hypothesis does produce substantially larger reduced test cases than the external reducers, and the reasons for this provide various insights into the limitations of internal reduction.

The first reason to note is that Csmith-generated C programs have a certain amount of "necessary size", due to boiler plate that every Csmith program contains. Because internal reduction reduces test cases by re-generating them, the minimum size of the reduced test cannot be lower than the smallest test the generator can produce.

Effectively, Hypothesis is reducing against a harder validity oracle: It has to produce an interesting test case that Csmith could have generated, while C-Reduce and Picire merely have to produce an interesting test case that is a valid C program which appears free of undefined behaviour. For most uses of Hypothesis in property-based testing, this sort of constraint is mild and perhaps actively desirable, but in this case it results in a significantly larger final test case. In particular, it is impossible for Hypothesis to prevent Csmith from generating its standard boiler plate code, so there is a certain baseline difference between Hypothesis and an external reducer that it can never do better than.

In order to determine this baseline, we ran Hypothesis on each starting example, reducing the choice sequence subject only to the constraint that it successfully generates a program. The smallest program found by Hypothesis during these reductions is shown in Figure 9, which we know (from familiarity with how Csmith works) is essentially the smallest program Csmith is capable of generating. This gives us a baseline minimum size for Hypothesis reduced programs of 410 bytes. In contrast, C-Reduce and Picire are perfectly capable of producing an empty file, or a trivial 14-byte main function definition if we require that the file can produce an executable (which we do for wrong code bugs). This already accounts for a sizable proportion of the difference in size. Adjusting for these baselines, Hypothesis generates a mean final size of around 402 bytes, and Picire of around 331 bytes. This difference is still statistically significant, but much more reasonable.

In order to understand the size difference above and beyond this baseline, we ran C-Reduce on the Hypothesis final example for the smallest examples of a crashing bug and a wrong code bug respectively. We show examples of these in Figure 10 and 11.

The difference on the crash bug, where C-Reduce is less constrained, largely comes from the larger baseline we discuss above, while the difference on the wrong code bug demonstrates a number of other issues: Csmith will always pre-declare union definitions, and uses long identifiers, both of which C-Reduce is able to fix.

We also see an advantage of Hypothesis in this example: In Figure 11, C-Reduce has produced a call to printf with too many arguments. This is defined behavior, as the extra arguments are ignored, but is suspicious and likely to be distracting when debugging. In contrast, the reduced program produced by Hypothesis is Csmith-generated, and has no such issues.

Hypothesis reduced:

```
1  #include "csmith.h"
2  static long __undefined;
3  static const volatile int32_t g_2 = (-1L);
4  static const int8_t func_1(void);
5  static const int8_t func_1(void) {
6    volatile int8_t l_3 = (-1L);
7    l_3 = g_2;
8    return g_2;
9  }
10 int main(int argc, char *argv[]) {
11   int print_hash_value = 0;
12   if (argc == 2 && strcmp(argv[1], "1") == 0)
13     print_hash_value = 1;
14   platform_main_begin();
15   crc32_gentab();
16   func_1();
17   transparent_crc(g_2, "g_2", print_hash_value);
18   platform_main_end(crc32_context ^ 0xFFFFFFFFUL, print_hash_value);
19   return 0;
20 }
```

C-Reduce run on the Hypothesis output:

```
1  #include "csmith.h"
2  const volatile a;
3  b() { volatile int8_t c = a; }
```

Figure 10 The smallest Hypothesis reduced crash bug.

To emphasise the above discussion regarding Csmith boiler plate: in both of these examples we can see that Hypothesis is quite close to being constrained by a fundamental limitation of this approach. The limitation is not its ability to reduce further (although in Figure 11 we do see what is likely a missed reduction at the choice sequence level – the third field of the union is successfully removed by C-Reduce but not by Hypothesis even though it plausibly could be), but the fact that it guarantees that reduced examples are ones that could be generated means that the reduced examples must include certain features that Csmith will always generate: e.g. variables will always be initialized, functions will always be pre-declared, and union types are always declared separately from their usage.

These larger sizes are certainly a minor downside of internal reduction, in that for ease of debugging a smaller program is usually more useful. That said, in the case of Csmith, being limited to reducing to programs that Csmith can generate means that even reduced tests will be executable programs that follow a well-known structure and are guaranteed to be free from undefined behaviour, which might make them ideal as end-to-end tests for addition to a compiler regression test suite (rather than e.g. tests that simply check whether a compiler crashes, but that are otherwise meaningless). It is also arguable that since most of the extra size is easy-to-understand boiler plate, its presence has little practical significance.

Hypothesis reduced:

```
#include "csmith.h"
static long __undefined;
union U1 {
  const int32_t f0;
  const unsigned f1 : 17;
  const volatile signed : 0;
};
static const union U1 g_2 = {-1L};
static const union U1 func_1(void);
static const union U1 func_1(void) { return g_2; }
int main(int argc, char *argv[]) {
  int print_hash_value = 0;
  if (argc == 2 && strcmp(argv[1], "1") == 0)
    print_hash_value = 1;
  platform_main_begin();
  crc32_gentab();
  func_1();
  transparent_crc(g_2.f0, "g_2.f0", print_hash_value);
  transparent_crc(g_2.f1, "g_2.f1", print_hash_value);
  platform_main_end(crc32_context ^ 0xFFFFFFFFUL, print_hash_value);
  return 0;
}
```

C-Reduce run on the Hypothesis output:

```
#include "csmith.h"
union {
  int32_t a;
  unsigned b : 17;
} c = {-1L};
int main() {
  printf("%d\n", c.a, c.b);
  return 0;
}
```

Figure 11 The smallest Hypothesis reduced wrong code bug, with additional reduction provided by C-Reduce.

Either way, RQ1 has a clear answer on the Csmith experiments: Hypothesis produces examples in the same order of magnitude as, but still substantially larger than, those produced by specialized reducers such as C-Reduce that are well-adapted to the problem domain, but produces of examples of comparable but slightly larger size to those found by more generic reducers.

To evaluate RQ2, we recorded the number of SUT evaluations made during the run of these experiments. We show the results of this in Figure 12

Here all differences are statistically significant at $p < 10^{-5}$. Hypothesis is thus about four times faster[15] than Picire on the Csmith experiment, and about 50% slower on the formatting example. We haven't investigated this in detail but expect that the latter is because Hypothesis makes a number of lexicographic transformations to the choice sequence that don't impact the final size of the generated test case, but result in e.g ensuring generated string literals contain only zeroes.

[15] In terms of SUT calls that is. In terms of wall clock time it was actually slower due to the high cost of how we invoked Csmith.

	Csmith	Python
C-Reduce	3968.0 (3731.8–3216.7)	863.3 (797.5–937.0)
Hypothesis	762.0 (701.8–829.6)	1284.565 (1106.56 1556.175)
Picire	3138.9 (2970.9–3348.9)	529.23 (483.61, 579.205)

Figure 12 Number of SUT invocations for each reducer.

To answer RQ3, we also recorded the number of generator evaluations, and for each experiment calculated the ratio of generator evaluations to SUT calls (every generator evaluation leads to an SUT call, so the former is always larger than the latter). We calculated a 95% bootstrap confidence interval for the geometric mean of these ratios (the geometric mean being chosen as the appropriate mean to use for comparing ratios). For the Csmith experiment this gave us a confidence interval of $2.78 - 2.94$, and for the formatting experiment the interval was $1.21 - 1.30$. i.e. for Csmith we performed nearly three times as many generator invocations as SUT invocations, while for Python we performed up to about 30% more. The difference is likely accounted for by the fact that the Python generator was built on top of Hypothesis's core library of generators which as we describe in Section 3.3, are designed to behave well with lexicographic reduction in general and Hypothesis's reducer in particular.

How much overhead this corresponds to in practice depends significantly on the generators and SUTs in question. Our interface to Csmith was quite slow, so there generation time probably dominated even without any overhead, but for most cases we would expect the generator to be significantly faster than the SUT.

4.2 Case Study: SymPy

TSTL [11] is a domain specific language defined for testing APIs written in Python. Actions using the API are described using the TSTL language, and it builds tests as sequences of actions, expressed as fragments of Python code that are evaluated against a model of the SUT.

Reduction in TSTL consists of attempting to find shorter sequences of actions that can trigger the same bug. Previously work on TSTL's reducer [11] tested SymPy, a symbolic algebra library for Python, and we adapted these tests to use Hypothesis in order to evaluate internal reduction for this use case.

One downside of comparing with TSTL is that a TSTL test is always valid – any action which should not be run is simply ignored – so the benefit of guaranteed validity associated with internal reduction is not relevant, but it is still a reasonable point of comparison for reducer effectiveness.

4.2.1 Experiment Design

We implemented a backend that takes a TSTL-generated harness and runs it with Hypothesis, which we used to run the TSTL tests for SymPy from its examples directory. We ran these tests against version 1.1.1 of SymPy, which is slightly older than the latest version, as we knew that the test harness was capable of finding many bugs in this older version, providing us with a variety of example bugs on which to evaluate reduction.

This backend does not implement TSTL's checks, which run a number of equivalence checks on the generated SymPy programs to assert that various expressions that are expected to give the same result do in fact do so. These checks were minimally useful for SymPy [8], and were prohibitively slow, thus they would have limited the amount of data we could have collected.

As a point of comparison, we used a custom implementation of delta debugging which we adapted to take advantage of two structural features of TSTL: It would automatically discard any actions that were no longer able to run, and prune all steps after the failing one. We did not compare to the TSTL reducer due to wanting to ensure we matched the slightly different semantics of our backend implementation, and for convenience when instrumenting it, but believe this modified delta debugging should work similarly well to its standard reducer. We did not however implement anything equivalent to its test-case normalization features [9].

To enable us to gather a large corpus of data, we aggressively pruned slow tests by removing test cases where an individual step took more than two seconds to run. This implicitly removed a large class of errors, as it appears to be very easy to trigger `RecursionError` bugs in SymPy which, for some reason (possibly a high cost associated with each recursive call), always resulted in the triggering step exceeding this timeout. This potentially impacts the generality of our results, but there were sufficiently many other errors in SymPy that it seemed unproblematic to exclude them.

Additionally, we found a number of the SymPy test cases were *flaky* – that is, they did not reliably produce the same exception when run with different random or hash seeds. We don't entirely understand why this would be the case (we expect it is something internal to SymPy's implementation) but we didn't spend a great deal of time investigating . We know from experience that flaky test cases tend to lead to poor performance in most test-case reducers, and we wished to avoid these dominating the results, so we attempted to remove any test cases where reduction passed through a flaky test case. We removed test cases where any of the original generated test case or either of the internally or externally reduced final test cases were flaky, but flakiness checking was fairly expensive so we did not check all intermediate results.

Starting from an initial generated corpus of 3000 distinct failing test cases, removing flaky tests left us with 2930 interesting test cases. These were spread across 33 distinct errors, which we distinguished based on error type and line number, and had a mean length of 64.5 (which gave a 95% confidence interval for the population mean of 63.6 – 65.4). Notably, this is somewhat larger than the mean size of 44.7 reported in [11]. While we did not investigate the cause of this in detail, there were a number of small differences in our experimental setup which could account for it, such as the exclusion of the relatively easy to trigger `RecursionError` bugs.

For each of these initial test cases, we ran both the Hypothesis reducer and our delta debugging implementation for TSTL, subject to the interestingness test that an exception was raised with the original exception type and line number. We recorded the number of SUT calls made by each, and the final size of the reduced test cases.

4.2.2 Experimental Analysis

On average (geometric mean), Hypothesis made 20.6 (95% confidence 20.3 – 21) times as many SUT calls as delta debugging, resulting in tests that were 83% (95% confidence 82% – 84%) of the size produced by delta debugging.

This is relatively expensive for a marginal gain. However, that seems to be less a feature of internal reduction and more one of the problem domain: As part of the work on test-case normalization in [9], they implemented normalization passes which performed similar external transformations to those enabled by Hypothesis's lexicographic internal reduction, and when these normalization passes were enabled reduction took about thirty times as long and obtained test cases that were about 55% of the size of those obtained without normalization.

We think it likely that the performance of Hypothesis could be substantially improved on this experiment, but resisted the urge to optimize for this use case for now, letting the experimental results stand as they are. Brief investigation suggested that Hypothesis's heuristics for reduction pass ordering do not work very well on these examples, which lead to it doing a significant amount of lexicographic reduction when it could still have usefully been trying to reduce the size of the choice sequence.

4.3 Evaluation against QuickCheck and SmartCheck

The only previous evaluation of test-case reduction in QuickCheck we are aware of comes from [22], which defined SmartCheck, a generic reducer for algebraic data types, and introduced a set of five synthetic benchmarks to compare it to QuickCheck. Each of these benchmarks consists of some data type to generate to test some code that has a known (deliberately inserted) bug in it. We have reimplemented these benchmarks in Python to evaluate Hypothesis on them and compare its behavior to that of QuickCheck and SmartCheck.

The five benchmarks are "bound5", "binheap", "calculator", "parser", and "reverse". We updated these from the originals to improve QuickCheck's behavior, mainly by replacing some ineffective custom reducers with QuickCheck's `genericShrink`. We also changed the "binheap" benchmark to add a precondition that prohibited invalid heaps, as we noticed that much of SmartCheck's performance on that benchmark came from very rapidly reducing to small but invalid heaps (an instance of the test-case validity problem).

Experiment	Hypothesis	QuickCheck	SmartCheck
binheap	9.02 (9.01–9.03)	9.00 (9.00–9.00)	9.42 (9.37–9.48)
bound5	2.08 (2.07–2.10)	11.30 (10.91–11.76)	6.02 (5.79–6.29)
calculator	5.00 (5.00–5.00)	5.11 (5.07–5.15)	5.00 (5.00–5.01)
parser	3.31 (3.28–3.34)	3.99 (3.98–4.01)	4.08 (4.01–4.14)
reverse	2.00 (2.00–2.00)	2.00 (2.00–2.00)	2.00 (2.00–2.00)

Figure 13 Mean size of reduced examples on synthetic benchmarks. Each data type has a different notion of size associated with it, but it typically means something like number of nodes in the tree.

Experiment	Hypothesis	QuickCheck
binheap	170.31 (166.14–174.76)	88.22 (86.90–89.55)
bound5	95.13 (93.57–96.91)	1438.89 (1282.34–1811.64)
calculator	72.41 (70.57–74.32)	30.97 (29.92–32.37)
parser	126.50 (124.11–128.90)	34.23 (33.63–34.81)
reverse	50.84 (50.40–51.29)	17.68 (17.27–18.10)

Figure 14 Mean number of test cases tried while reducing synthetic benchmarks.

We ran each benchmark 1000 times for each library. We present the mean sizes of the reduced examples in Figure 13, and the mean number of SUT evaluations made in Figure 14. We ran into some technical difficulties obtaining the number of SUT evaluations made by SmartCheck and, as it omits many classes of transformation that both Hypothesis and QuickCheck consider (e.g. reducing the value of generated integers) and did not do particularly well on the size evaluation besides, didn't feel it was especially useful to invest more time on the problem.

By a permutation test, all differences in mean SUT invocations are significant at $p < 10^{-5}$. For sizes, differences were significant at $p < 10^{-4}$, with the following exceptions:

- All implementations reliably produced the minimal size example for "reverse" so there was no difference in means.
- Hypothesis and SmartCheck on the "calculator" example ($p \approx 0.5$)
- Hypothesis and SmartCheck on the "parser" ($p \approx 0.02$).
- Hypothesis and QuickCheck on the "binheap" benchmark ($p \approx 0.003$).

To account for multiple testing we set a significance threshold at $p < \frac{0.05}{30} \approx 0.0017$ (by applying the Bonferonni correction – there are three pairs of comparisons for each benchmark, for each of size and SUT count, so thirty tests), so these should all be considered nonsignificant.

The only case where Hypothesis produced worse average results than QuickCheck (significant or not) was the "binheap" benchmark, where it did very slightly worse than QuickCheck (9.02 vs 9.0). We haven't investigated why but suspect it's due to difference in the distribution of initial test cases (Hypothesis tends to produce larger examples) rather than the reducer. Whatever the reason, the difference, though statistically significant, is tiny.

We note that the behaviour of QuickCheck on the "bound5" example is pathologically bad, both in size and performance, in large part because it was constructed to be so. Hypothesis fares well on this example without modification, showing one of the advantages of having a more sophisticated reducer by default.

Thus on RQ1 Hypothesis fares well compared to QuickCheck, generally producing similar or better results. On RQ2, Hypothesis proves more expensive than QuickCheck by a factor of 2–3, depending on the benchmark.

5 Threats to Validity

The main empirical claims of our paper are that our model of internal reduction through shortlex optimization is viable, and in particular that it provides results that are competitive with alternative reducers that might be used in its place.

As we have been using it in the context of a widely deployed testing library for more than four years, we are quite confident of its viability, at least within our application domain, and our empirical results in Section 4 support the claim that it performs reasonably with respect to alternatives.

The main threat to validity is how well these results generalize. Although we have presented four reasonably diverse case studies, three of which were on their own larger than most previous evaluations of test-case reduction, the range of software and generators used in practice is naturally larger yet. It is plausible that there are reduction problems that we have simply never run into that present their own challenges.

A common factor in all of our experiments is that the starting points were not especially large – Hypothesis by default only considers choice sequences of at most 8KB, and we retained that restriction in our analysis. As we discuss in Section 4.1.1, this was a particularly notable restriction in the case of Csmith.

Our intuition, which is backed by a certain amount of anecdotal evidence, is that most test-case reducers experience problems at larger scales that they do not see at smaller ones, because larger test cases offer more opportunities to get stuck in local minima. Additionally, often large test cases trigger bugs in SUTs that were difficult to trigger at smaller scales – either because they are intrinsically connected to test case size (a scenario that tends to reduce very poorly in general) or because they simply happen with too low probability at small sizes. Between these two factors, we expect interesting new difficulties to arise at larger scales, requiring more work on Hypothesis's reducer.

This also points to the other major limitation of our results: Although our claim is that internal reduction as a general model is viable, our empirical results are restricted to its implementation in Hypothesis. This suffices as an existence proof, but the Hypothesis reducer has been the subject of considerable engineering effort, and our results do not determine how much of the viability of internal reduction is only because of that engineering effort.

However, part of why the Hypothesis reducer is so sophisticated is because internal reduction rewards that: Because one reducer can serve many different types of test case, it was worth investing that effort into it, and the reducer can in principle be used in many different contexts, so even if it turns out that internal reduction is only viable with this engineering work, we don't consider that to be a major point against it.

6 Related Work

There are several categories of work related to ours, which we now describe: Test-case reduction in general (Section 6.1), test-case reduction in property-based testing (Section 6.2), use of the choice sequence model to improve generation (Section 6.3), and finally other users of internal reduction (Section 6.4).

6.1 Test-Case Reduction

Our work on internal reduction naturally builds on prior work on test-case reduction.

Test-case reduction was first described in the original papers on *delta debugging* [12, 26]. Most subsequent research has been focused on continuing delta debugging's goal of reducing the size of the test case, with other reductions such as our lexicographic passes being treated as of secondary interest.

This work on reducing size has generally focused on taking advantage of the structure of particular input formats. The major examples of this in the literature include *hierarchical delta debugging* (HDD) [19], which makes use of a grammar for the test-case format, and C-Reduce [23], which is extensively specialized to features common in C and C-like languages.

As we discuss in Section 3, the Hypothesis reducer is a similarly specialized reducer designed for the class of languages parsed by generators, and its design has been inspired by this prior work. In particular, the approach we describe in Section 3.2 of marking out regions of the choice sequence corresponding to parts of the test case very strongly resembles HDD's use of a grammar to do the same, and the pass-based approach we describe in Section 3.1 strongly resembles the architecture of C-Reduce.

One exception to the prior focus on reducing size is [9], which introduced the notion of test-case normalization as an important property of reducer. Additionally, although this was not made explicit, the normalization passes suggested in [9] can be regarded as optimizing for the lexicographic ordering, which makes their approach another example of our suggested goal of shortlex optimization. However, this was in the context of an external reducer, not an internal one.

6.2 Test-Case Reduction in Property-Based Testing

Test-case reduction has been an important feature in property-based testing since the early work on QuickCheck [2]. In property-based testing, test-case reduction is usually called *shrinking*, but for consistency we will continue to use the term test-case reduction.

In the original QuickCheck, and other property-based testing libraries closely based on it, test-case reduction follows the external reduction model, with reducers run on the generated test cases once an interesting one has been discovered, with an appropriate reducer selected based on the type of the generated data or provided by a user.

Originally these reducers were hand-written ones. However, most users do not particularly want to write their own test-case reducers, so this lead to the introduction of generic test-case reducers. These are particularly popular in Haskell, where most data is represented with algebraic data types, and good generic programming libraries allow for automatically deriving reducers for most data types that are "good enough" (any test-case reduction will tend to improve the utility of property-based testing, and to be worth the effort, hand-writing a reducer has to be less work than the debugging effort it saves). Indeed, the derivation of such reducers has been used to motivate the development of some of these generic programming libraries [14]. Generic reduction was also explored in [22], but the suggested approach does not appear to have been widely adopted.

However both manual and generic approaches to test-case reduction suffer a variant of the test-case validity problem: After reduction has been performed, the final reduced test case may be one that could not have been generated. This tends to be counterintuitive to users, who consider it a bug or missing feature.[16] There is no straightforward way to derive an oracle for whether something could be generated, so this problem is essentially insoluble without a different approach.

In aid of this, several property-based testing libraries have introduced what is called *integrated shrinking*,[17] where test-case reduction is "bundled" with the generators, so that every generator contains information about how to reduce its generated test cases. In this sense, the internal reduction model we describe in this paper can be thought of as a form of integrated shrinking.

There is however another more widely used implementation of integrated shrinking, the *rose tree*[18] method [4, 5] This approach works by having generators generate a (lazily evaluated) tree consisting of an initial value and possible reductions of it, so that generated values can be reduced by walking the tree. The rose tree method has been implemented in test.check (Clojure)[19] and Hedgehog (Haskell)[20] among others.

Such generators can easily be implemented by pairing a normal random generator of test cases with a test-case reducer, but they save implementation effort by allowing for composition with user defined functions. In particular by supporting the monadic [20] bind operator to chain generators together, one can in principle generate anything, as monadic bind can be used to express arbitrary computation. Unfortunately in practice the rose tree approach produces poor reductions when bind is used,[21] so generally this approach to integrated shrinking only works well with a relatively restricted set of generators.

Because monadic bind can be used to express arbitrary computation, the question of whether internal reduction can work well in these scenarios is essentially equivalent to the question of whether it can work well with arbitrary generators, to which the answer is that it depends. It is certainly possible to construct generators that Hypothesis finds difficult

[16] `https://github.com/typelevel/scalacheck/issues/129`
[17] `https://hypothesis.works/articles/integrated-shrinking/`
[18] A rose tree is a tree where each branch node can have any number of children.
[19] `https://github.com/clojure/test.check`
[20] `https://hedgehog.qa/`
[21] `https://github.com/clojure/test.check/blob/master/doc/growth-and-shrinking.md#unnecessary-bind`

to reduce, but as we saw in Section 4.1 it tends to work well with even large and complex generators written without internal reduction in mind. Also, a key difference is that when Hypothesis has difficulty reducing a generator, typically this is a limitation of its reducer rather than the model: Generally there is some shortlex smaller sequence that would reduce the generated value, but the reducer is unable to find it. Such situations can often be resolved by improving the reducer with no modifications to user code. In contrast, the rose tree model offers no alternative but to add a custom external reducer.

Nevertheless, at present the rose tree model is significantly more widely used than internal reduction. Partly this is just because it predates the internal reduction model, but it is also significantly simpler to implement and easier to understand. Nevertheless, we believe that the benefits of internal reduction are worth the increased implementation complexity, and hope this paper will aid readers' understanding of its model.

6.3 Choice Sequences to Improve Generation

We have introduced the term *choice sequence* to refer to the binary decisions made during random generation, which we use to regard random generators as deterministic parsers of sequences of bits. Similar approaches have been used elsewhere. Most other usage has focused not on test-case reduction but instead on improving the quality of generated test cases by using coverage guided fuzzing. For example, crowbar [3] is an OCaml library for providing property-based testing built on top of the AFL Fuzzer[22] using this approach. DeepState [6] is a unit testing library for C++ which supports either symbolic execution or coverage guided fuzzing. These effectively use a choice sequence, encoded as a sequence of bytes provided by the fuzzer, to make nondeterministic decisions.

Recent work in Zest [21] also uses a choice sequence model to improve the quality of generated test case, but uses the term "parameters" to refer to the individual bits. We prefer our "choice sequence" terminology, as the interpretation of a given bit can change during reduction (e.g. a bit that once chose whether to terminate a list might become part of a generated value) so we find thinking of them as parameters a little misleading.

6.4 Other Uses of Internal Reduction

The idea of internal reduction as shortlex optimization originates with Hypothesis, but the idea of manipulating a generator to produce smaller results predates it. The main prior art of which we are aware is Seq-Reduce [23], a Csmith mode that attempts to reduce the length of the choice sequence by regenerating parts of it. Seq-Reduce was only designed to work with Csmith, and was abandoned due to disappointing results, while we have shown that with internal reduction is both broadly applicable and can work well with Csmith in particular. We have not investigated why we see such a substantial difference between the two approaches, but think it likely that its approach of randomly regenerating parts of the test case was unlikely to work without more structural information such as we describe in Section 3.2.

In addition, there are two significant production implementations of internal reduction that have appeared subsequent to Hypothesis, in both cases explicitly based on its approach. These are DeepState, which we also mention in Section 6.3, and theft[23], a property-based testing library for C. Both share our approach of internal reduction as shortlex optimization, but have their own reducer implementations.

[22] `https://github.com/google/AFL`
[23] `https://github.com/silentbicycle/theft`

7 Conclusion and Future Work

We have presented internal reduction, an approach that performs test-case reduction on generated test cases by manipulating the behavior of the generator that produced them.

The key advantages of internal reduction over conventional, external, reduction are that, by operating solely on the behavior of the generator, it a) provides "free" reduction for arbitrary generators, saving the need to write a new reducer, and b) ensures that reduced test cases are ones that could have been generated, avoiding the test-case validity problem.

As demonstrated by our experimental results, the size of the reduced test cases found by internal reduction is competitive with that found by general purpose reducers such as delta debugging or the reducers typically found in property-based testing libraries, at a moderate increase in reduction cost. Unsurprisingly, there is still a large size gap between its results and those of more specialized reducers such as C-Reduce. We expect that this will continue to be the case, and do not suggest internal reduction as the best model when it is worth investing significant engineering effort in a specialized reducer for a particular test-case format.

Nevertheless, by providing good quality test-case reduction "for free", internal reduction has significantly improved the user experience of property-based testing in Hypothesis, and the other testing tools we mention in Section 6.4, and is likely to be useful to other users of random generation, especially those not currently using test-case reduction.

Internal test-case reduction has been used in Hypothesis for over four years now, and we consider it a mature and proven technology. Future work on the Hypothesis reducer will seek to improve its performance, and likely will see the development of further reduction passes and heuristics that expand the set of generators it works well for. We're particularly interested in exploring whether we can improve its performance on larger initial choice sequences, and hope to do further work based on attempting to lift the 8KB buffer size restriction we saw in our experiments with Csmith-generated programs in Section 4.1.

Another exciting line of research is the use of the choice sequence model to implement other functionality. As we discuss in Section 6.3, there are a number of implementations that use this idea to provide coverage-guided fuzzing. Hypothesis has some limited support for this, which we are intending to expand further in future. Additionally, Hypothesis has an implementation of targeted property-based testing [15], which guides generation towards test cases maximizing or minimizing some objective function. The advantages of the choice sequence model for targeted property-based testing are much the same as that for test-case reduction: It provides a fully generic mechanism that requires no user intervention, and ensures that all provided test cases are ones that could have been generated, significantly easing the validity problem. In contrast, prior attempts at fully automating targeted property-based testing (that is, implementing it without requiring user provided mutation functions) in [16] required a great deal of care to ensure valid test cases were produced.

In general, the choice sequence model has proven flexible and powerful, allowing us to implement advanced features with minimal negative impact on users, and without requiring any user expertise in the subject. This makes Hypothesis a powerful tool for creating production implementations of software testing research ideas. We intend to continue using it as such, and encourage other researchers to do the same.

References

1. Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. Testing telecoms software with quviq quickcheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006. doi:10.1145/1159789.1159792.
2. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000. doi:10.1145/351240.351266.
3. Stephen Dolan and Mindy Preston. Testing with crowbar. In *Proceedings of the OCaml Users and Developers Workshop*, September 2017. URL: https://ocaml.org/meetings/ocaml/2017/extended-abstract__2017__stephen-dolan_mindy-preston__testing-with-crowbar.pdf.
4. Reid Draper. Proposal: free shrinking with quickcheck. https://mail.haskell.org/pipermail/libraries/2013-November/021674.html, 2013. Accessed: 2020-05-25.
5. Reid Draper. Writing simple-check. http://reiddraper.com/writing-simple-check/, 2013. Accessed: 2020-05-25.
6. Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.
7. Google. yapf: Yet another python formatter, 2018. URL: https://github.com/google/yapf.
8. Alex Groce. private correspondence.
9. Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 1–11. ACM, 2017. doi:10.1145/3092703.3092704.
10. Alex Groce and Jervis Pinto. A little language for testing. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2015. doi:10.1007/978-3-319-17524-9_15.
11. Alex Groce, Jervis Pinto, Pooria Azimi, and Pranjal Mittal. TSTL: a language and tool for testing (demo). In Michal Young and Tao Xie, editors, *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 414–417. ACM, 2015. doi:10.1145/2771783.2784769.
12. Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In Debra J. Richardson and Mary Jean Harold, editors, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, pages 135–145. ACM, 2000. doi:10.1145/347324.348938.
13. Renáta Hodován and Ákos Kiss. Practical improvements to the minimizing delta debugging algorithm. In Leszek A. Maciaszek, Jorge S. Cardoso, André Ludwig, Marten van Sinderen, and Enrique Cabello, editors, *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016.*, pages 241–248. SciTePress, 2016. doi:10.5220/0005988602410248.
14. Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 204–215. ACM, 2005. doi:10.1145/1086365.1086391.
15. Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In Tevfik Bultan and Koushik Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 46–56. ACM, 2017. doi:10.1145/3092703.3092711.
16. Andreas Löscher and Konstantinos Sagonas. Automating targeted property-based testing. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 70–80. IEEE Computer Society, 2018. doi:10.1109/ICST.2018.00017.

17 Lukasz Langa. black: The uncompromising code formatter, 2018. URL: `https://github.com/ambv/black`.

18 David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, November 2019. `doi:10.21105/joss.01891`.

19 Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151. ACM, 2006. `doi:10.1145/1134307`.

20 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. `doi:10.1016/0890-5401(91)90052-4`.

21 Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 329–340. ACM, 2019. `doi:10.1145/3293882.3330576`.

22 Lee Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 53–64. ACM, 2014. `doi:10.1145/2633357.2633365`.

23 John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012. `doi:10.1145/2254064.2254104`.

24 Wikipedia contributors. Shortlex order, 2020. [Online; accessed 10-January-2020]. URL: `https://en.wikipedia.org/wiki/Shortlex_order`.

25 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. `doi:10.1145/1993498.1993532`.

26 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002. `doi:10.1109/32.988498`.