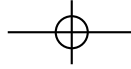
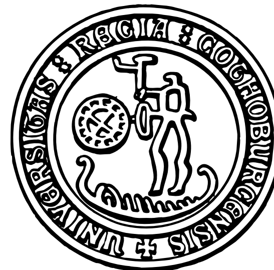


Thesis for the Degree of Doctor of Philosophy

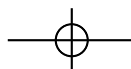
Random Structured Test Data Generation for Black-Box Testing

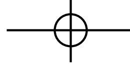
Michał H. Pałka

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computer Science and Engineering
Chalmers University of Technology
and Göteborg University
Göteborg, Sweden 2014





Random Structured Test Data Generation for Black-Box Testing
Michał H. Pałka
ISBN 978-91-7385-996-7

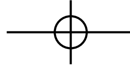
© 2014 Michał H. Pałka

Ny serie nr 3677

Technical Report 106D
ISSN 0346-718X
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology
and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

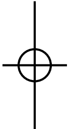
Printed at Chalmers
Göteborg, Sweden 2014

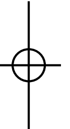


Abstract

We show how automated random testing can be used to effectively find bugs in complex software, such as an optimising compiler. To test the GHC Haskell compiler we created a generator of simple random programs, used GHC to compile them with different optimisation levels, and then compared the results of running them. Using this simple approach we found a number of optimisation bugs in GHC.

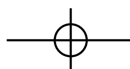
This approach for finding bugs proved to be very effective, but we found that implementing a generator of random programs by hand required a large amount of effort. Therefore, we developed an automatic method for deriving random generators of complex test data based on computable boolean predicates that specify the well-formed values of the data type. Defining such a predicate is usually much quicker than implementing a dedicated generator, even if its performance might be comparably lower.

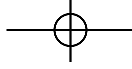


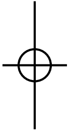
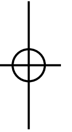


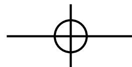
In addition, we discovered that the pseudorandom number generator used by us for random testing is unreliable, and that no reliable contruction exists that supports our particular requirements. Consequently, we designed and implemented a high-quality pseudorandom number generator, which is based on a known and reliable cryptographic construction, and whose correctness is supported by a formal argument.

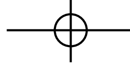
Finally, we present how random testing can be used to rank a group of programs according to their relative correctness with respect to their observed behaviour. The ranking method removes the influence of the distribution of the random data generator used for testing, which results in a reliable ranking.









Contents

Contents **v**

Introduction **1**

Paper I: Testing an Optimising Compiler by Generating Random Lambda Terms **17**

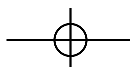
1 Introduction 17
2 Related work 20
3 Structure 21
4 Generation method 23
5 Shrinking 34
6 Applications 50
7 Related Work 68
8 Future work 73
9 Conclusions 74

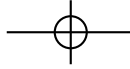
Paper II: Generating Constrained Random Data with Uniform Distribution **79**

1 Introduction 79
2 Generating Values of Algebraic Datatypes 82
3 Predicate-Guided Indexing 85
4 Experimental Evaluation 90
5 Related Work 95
6 Discussion 97

Paper III: Splittable Pseudorandom Number Generators using Cryptographic Hashing **101**

1 Introduction 101
2 Splittable PRNGs 105
3 Proposed construction 109
4 Correct hashing 113

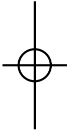


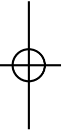


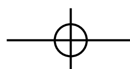
5 Linear generation 118
6 Performance 122
7 Discussion and future work 127
8 Related work 129
9 Conclusion 131
A Appendix. Definitions and Proofs 132

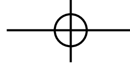
Paper IV: Ranking Programs using Black Box Testing **137**
1 Introduction 137
2 The Experiment 139
3 Evaluation Method 143
4 Analysis 147
5 Related Work 154
6 Conclusions 155

Bibliography **157**







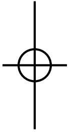


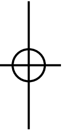
Acknowledgements

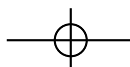
I would like to thank my supervisors, Koen Claessen and John Hughes, for their guidance and support. Both of them were a continuous source of ideas and contagious enthusiasm. I also wish to thank my examiner, Mary Sheeran for providing valuable feedback throughout my studies.

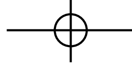
I would like to thank all of my colleagues at the department for making it such an open and friendly environment. I was lucky to have a chance to interact with many extraordinary individuals, many of whom contributed suggestions that helped me in writing this thesis.

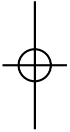
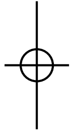
I also want to thank all my friends for their encouragement and friendship. I would like to especially thank Laleh for her friendship, care and understanding. Laleh, you are great! Finally, I would like to thank my parents for all the love and support throughout my life.

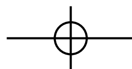


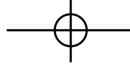










Introduction

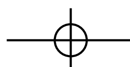
This thesis is concerned with finding software bugs using random testing. Our main focus is providing tools that programmers can use to find and remove bugs from their programs in an effective way. The principal application of the random testing techniques presented here is finding optimisation bugs in the GHC optimising Haskell compiler.

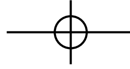
Software testing Software testing is the most common method of ensuring the quality of software, accounting for at least 50% of the budget of most software projects [Beizer, 1990, Myers et al., 2012]. Systematic software testing is widely used for *validation* [Sommerville, 2010] of developed software. At the same time, it finds even more use during the development process [Myers et al., 2012, Sommerville, 2010, Zeller, 2005], when it is performed in order to find and remove software bugs.

Testing and removing bugs (debugging) are time-consuming activities [Myers et al., 2012, Zeller, 2005], whose success is critical to software project completion. For example, a study in 2005 showed that 30–40% of all changes in Eclipse and Mozilla projects were bug fixes [Śliwerski et al., 2005]. Bringing improvements to testing and debugging may decrease the time spent in these activities, but more importantly also reduce the impact of poor software quality [Tassey, 2002].

This work is concerned with black-box testing, which treats the software under test as a black box, and is only concerned with observing the results of its execution [Beizer, 1990, Myers et al., 2012].

Why test software? A common focus of testing is to use it for demonstrating the correctness of software. However, for systems of realistic sizes only an approximate judgement about correctness can be made based on testing, as the number of possible inputs is much larger than the number tests that can be run. Thus, as famously stated by Dijkstra [1969]: "Program testing can be used to show the presence of bugs, but never to show their absence!"





Nevertheless, the need to give stronger guarantees about the correctness of software using testing led to the development of code coverage criteria, which are a way of measuring which parts of the program are exercised by the tests. Unfortunately, practical coverage criteria are not precise enough to ensure that a test suite that satisfies them can catch all bugs, and thus can only demonstrate that the test suite is insufficient [Myers et al., 2012].

In general, testing can only provide a weak guarantee about the correctness of software. While it is undoubtedly useful, making validation the primary goal of testing might put its effectiveness at risk. Since some bugs might only be triggered by test inputs that are rare and complex, finding them may require ingenuity. If the goal of the testing process is to demonstrate the absence of failing test cases, the people performing it might be encouraged to try less hard to find bugs thus undermining the testing effort [Myers et al., 2012].

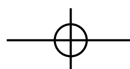
Indeed, while testing is used for validation, its importance has shifted towards finding bugs during the whole software development process [Myers et al., 2012, Sommerville, 2010, Zeller, 2005]. In our opinion, testing with the intention of finding bugs is more constructive than attempting to demonstrate their absence, and consequently we would like to make it our focus.

Debugging Every software project of a certain size contains bugs during its development, and their successful removal is critical to the project's completion. Debugging can be a challenging and error-prone process. For example, compared to ordinary changes, bug fixes are more likely to induce bugs [Mockus and Weiss, 2000].

Despite its difficulty, debugging receives comparatively little attention. According to Myers et al. [2012], "Of all the software development activities, debugging is the most mentally taxing", but at the same time "Compared to other software development activities, comparatively little research, literature, and formal instruction exist on the process of debugging". A similar observation about the hardness of debugging has been made by Kernighan et al.:

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? [Kernighan and Plauger, 1982, Chapter 2]

Debugging starts with discovering a program failure, and its aim is to diagnose the underlying problem that caused the failure and fix it. This usually requires completing the following steps:



1. Finding a failing test case (counterexample),
2. reproducing the failure,
3. identifying its root cause, and
4. fixing the bug that caused the failure.

While reproducing the found failure might pose problems due to uncontrollable parts of a program's external environment or its non-deterministic execution, identifying and removing the underlying cause of the failure tend to be the most difficult parts of debugging [Zeller, 2005]. Finding the problem might be especially difficult if the bug is caused by many errors scattered over different parts of the program, or a fundamental design flaw.

Solving a difficult debugging problem may sometimes seem like looking for a needle in a haystack. Using a systematic approach might be necessary to find the underlying cause of the failure. One approach is to use the *scientific method* [Zeller, 2005], which involves formulating *hypotheses* about the system, and testing them using experiments.

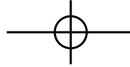
Given the difficulty of debugging, we believe that supporting this activity should be one of the principal goals of a testing method. Testing should be a programmer's tool for solving debugging problems.

Test automation Creating and executing test cases are labour-intensive tasks, which could benefit from automation. According to Beizer [1995], "Manual test execution doesn't work", as it allows very few test runs at a high cost, and is prone to errors, which puts the test results into question. Manually created test suites also require much effort, are troublesome to maintain, and tend to get out of sync with the code.

Automated test execution is becoming increasingly popular, dramatically reducing the labour involved in testing [Sommerville, 2010]. In particular, effective regression testing requires simply too much effort if test execution is not automated. The main obstacle for automating test execution is having insufficient control over the environment of the tested software, for example when testing GUI applications [Zeller, 2005].

Automated test case generation is unfortunately less widespread, and can be performed by generating individual test cases from a test case 'template'. Running automatically generated tests requires a test oracle, which is an effective procedure for deciding whether the test has passed based on the responses of the tested system to test data.

Automating test case generation requires more infrastructure compared to just automating test execution, but can also yield more benefits. One promising approach to automating test case generation is random testing.



Random testing *Random testing* refers to testing software with 'arbitrary' random data aimed at exercising diverse program executions in order to trigger bugs [Zeller, 2005]. The term *fuzz testing* may also be used to describe this technique, although it often indicates more specifically testing programs using random, and mostly incorrect input [Miller et al., 1990]. Random testing defined in this way is our focus.

Confusingly, the term *random testing* may also refer to two other approaches to testing software, which is not always made clear. Originally random testing referred to *ad-hoc*, unsystematic testing performed manually [Hamlet, 1994]. Another meaning of random testing is to perform system testing against a randomly generated test suite, whose distribution approximates the *operational profile* of the application [Hamlet, 1994]. This way of testing may give concrete reliability guarantees, but requires knowing the distribution of real data with which the system will be run.

The practice of random testing enjoys a mixed reputation. According to Myers et al. [2012], "In general, the least effective methodology of all is random-input testing", as opposed to selecting test data "more intelligently". Despite this criticism, successful applications of random testing point at its usefulness in a range of domains, such as testing UNIX utilities [Miller et al., 1990], compilers [Faigon, 2005, Lindig, 2005, McKeeman, 1998, Yang et al., 2011], and telecom software [Arts et al., 2006]. Random testing was shown to be effective in testing both small 'units' of code [Claessen and Hughes, 2000] and large programs [Yang et al., 2011].

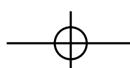
There are three commonly mentioned shortcomings of using random testing for finding and removing software bugs:

1. Simple random data generators, such as generating random strings, may not be able to exercise significant part of program code [Beizer, 1990].

2. A test oracle is needed to perform testing [Hamlet, 1994, Zeller, 2005].

3. Randomly generated failing test cases are often large and complicated, which makes them hard to understand [Yang et al., 2011, Zeller, 2005]. For example, Yang et al. [2011] observed that simplifying random test cases is essential for reported C compiler bugs to be fixed.

Successful application of random testing usually depends on addressing at least one of these concerns [Claessen and Hughes, 2000, Faigon, 2005, McKeeman, 1998, Yang et al., 2011]. Thus, we would like to concentrate on these three problems in order to make the benefits of random testing more widely available.



Property-based testing *Property-based testing* is a particular style of automated testing, which is implemented by QuickCheck [Claessen and Hughes, 2000]. QuickCheck is a random testing tool for Haskell based on *properties*, which is another name for test case templates. The following snippet is a property that checks whether the `insert` function applied to an element and a sorted list returns a sorted list.

```
prop_insert :: Int -> [Int] -> Property
prop_insert x xs =
  ordered xs ==> ordered (insert x xs)
```

A property is a Haskell function, which can be applied to different arguments to produce different test cases. In order to test the property, QuickCheck generates random instances of lists of integers and integers, and executes the resulting test cases. Thanks to the conditional `==>` operator, its right hand side will be tested only for sorted argument lists. The body of the property calls the tested code (`insert`), and defines an oracle that decides about the result of each test case using `ordered`.

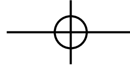
QuickCheck delivers the advantages of automated test case generation by allowing test suites to be specified in a very concise way, which also makes modifying them much less time-consuming, and much more likely to be adapted to changing code than traditional test suites.

QuickCheck properties are defined using a domain-specific language, which is the Haskell language extended with additional operators like `==>`. Each property may be regarded as a *hypothesis* about the tested code, which can be tested, and possibly disproved, by randomly testing the property.

When a random test fails, QuickCheck reports a counterexample. Instead of displaying the original counterexample, which might be rather large, QuickCheck applies an iterative *shrinking* procedure to find a simpler test case that also fails the property. Shrinking uses simple, structural reduction rules, which are provided for common data types, and can also be defined by users.

QuickCheck also provides another domain-specific language for defining random test data generators, as well as a number of predefined generators for common data types, which can be combined together, for example for generating lists of integers.

However, defining generators for custom data types is still manual, and may be challenging for some data types. As effective random testing requires good random test data distribution, significant effort might be needed to define a suitable generator. While automatic derivation of uniform generators for arbitrary algebraic data types has been recently demonstrated [Duregård et al., 2012], but the solution does not support data types with complex invariants.



Random number generation Generating complex random test data requires a high-quality random generator [Hamlet, 1994]. Since the software under test may perform arbitrary computations on its input data, they may uncover any existing flaws of the random number generator, which could compromise the results of testing.

Pseudorandom number generators (PRNGs) are often used for random testing due to their ability to replay randomly generated numbers. Unfortunately, the quality of PRNGs is difficult to assess, and some common PRNGs have been shown to be flawed [McCullough, 2009]. Empirical statistical tests are a common way of evaluating PRNGs, but passing a number of statistical tests is a poor predictor for passing other statistical tests.

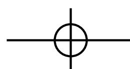
In addition to requiring a high-quality pseudorandom number generator, QuickCheck requires it also to be *splittable*. That is, each generator instance has an operation `split`, which creates two deterministically derived generator instances that return uncorrelated pseudorandom number streams. The `split` operation can be used freely to create any number of independent, yet deterministically derived instances.

QuickCheck uses splitting to efficiently generate random *lazy* data structures, whose different parts are computed on demand as they are accessed. Splitting can also be used to create a large number of independent pseudorandom streams for use in parallel computations.

To our knowledge, the problem of designing a fast, high-quality splittable PRNG that is supported by something more than statistical tests has until recently been unsolved.

This thesis Automated random testing has proven to be an effective tool for finding software bugs, but using it effectively requires addressing three main challenges: generating good random test data, providing a test oracle, and returning understandable counterexamples. With these challenges in mind, the aim of this thesis is to lower the barrier of entry to using random testing by improving random testing tools. In particular, we want to provide tools that programmers will use, without much effort, to make observations about the execution of their code, test hypotheses, track down failures, and gain understanding of the code. The primary application of random testing considered in this work is testing compilers.

In an ideal situation, we would like to have a push-button tool for random testing. Property-based testing and QuickCheck are largely such a push-button solution. In particular, many simple Haskell functions can be effectively tested by writing simple properties and using the predefined random data generators. However, defining test oracles (properties) and generators for testing more complex code is far from easy. For example,



testing a compiler requires generating very complex data (programs), and having an oracle that decides whether the result of compilation is correct, which is in general very difficult.

In this thesis, we propose to approach this problem in the following way:

1. Since defining a test oracle requires having a correctness criterion, we propose using weak correctness criteria that are easy to define. Oracles based on weak correctness criteria are not able to catch all bugs, but can be an economical solution for finding many of them.

2. We propose an automatic way of constructing random test data generators that generate random values of complex data types. The generators yield predictable distributions of values, and despite being less efficient than optimised hand-written alternatives, they are significantly easier to develop.

The first two papers of this dissertation demonstrate the usefulness of this approach to testing the GHC optimising Haskell compiler.

Another problem that is addressed in this work is the lack of high-quality pseudorandom number generators required by QuickCheck. As part of its infrastructure, QuickCheck requires a high-quality splittable PRNG to perform reliable random testing. We present a splittable PRNG construction, which is efficient and comes with a formal correctness argument, thus increasing the confidence in the results of random testing.

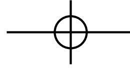
Finally, we show how random testing can be applied to create an unbiased test suite in order to rank a group of programs according to their relative correctness with respect to a reference program. The ranking method removes the influence of the distribution of the random data generator used for testing.

This thesis makes these specific contributions:

- We present a hand-written generator of random simply-typed lambda terms, based on performing random local choice and backtracking. The generator is used to test the GHC compiler using weak correctness criteria, such as *differential testing*, and successfully find optimisation bugs. (Paper I)

- We use simple, structural reduction rules to effectively reduce failing test cases found by random testing. (Paper I)

- In order to reduce the effort needed to develop a custom generator, we present an automatic method for deriving generators of random values of algebraic data types that satisfy a given predicate. As an



important feature of the derived generators, the distributions of values returned by them are predictable. (Paper II)

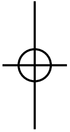
- We present an efficient construction of a high-quality splittable PRNG together with a formal correctness argument. (Paper III)
- We provide a method for ranking a group of programs according to their relative correctness using an unbiased test suite, which is obtained using random testing. (Paper IV)

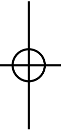
The remaining part of this introduction discusses the individual papers that form the thesis.

Work that led to this thesis has been supported by the Resource-Aware Functional Programming (RAW FP) grant awarded by the Swedish Foundation for Strategic Research.

Paper I: Testing an Optimising Compiler by Generating Random Lambda Terms

The paper presents a hand-written generator of random simply-typed lambda terms with polymorphic constants, and its application for testing the GHC optimising Haskell compiler.

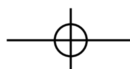




The GHC compiler was tested using *differential testing* [McKeeman, 1998], which involves compiling the same randomly-generated program using two different optimisation levels, and comparing the results of execution of both programs. Another variant of differential testing considered was to compile two different programs with equivalent semantics and look for discrepancies. In both cases differential testing was based on weak correctness criteria in the sense that satisfying the criteria gives very little assurance about the correctness of the compiler. However, we managed to demonstrate the usefulness of test oracles based on these criteria for finding bugs in GHC.

As the counterexamples returned by testing were large and hard to understand, we implemented simple, structural shrinking rules in order to automatically reduce them. Three shrinking rules were used: (1) replacing an expression with a proper subexpression, (2) inlining an argument into the function to which it was applied, and (3) replacing a complex expression with a constant. Shrinking using these simple rules proved to give very good results for all bugs that we found, and yielded understandable test cases, in addition to reducing their variety.

The project demonstrated that it is possible to find bugs in a complex optimising compiler using a relatively simple language as test data, and



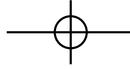
properties based on weak correctness criteria. Simply-typed lambda calculus with polymorphic constants, whose random terms were generated, is much simpler than the Haskell language. The differential testing setup used by us is based on a weak correctness criterion, for which a test oracle is easy to implement. Thus, starting with relatively simple random test data and correctness criterion may be an economical way of finding bugs even in complex software. At the same time, we found that the main effort was spent on developing the random test data generator, which took months to develop in this case. Making this process easier could considerably reduce the effort associated with random testing.

Related work Differential testing of compilers using random test data has been pioneered by McKeeman [1998], and subsequently used by Yang et al. [2011]. Both approaches apply it to testing C compilers by looking for discrepancies in the results of compilation between different compiler versions, and both use relatively complex hand-written test data generators. Additional complexity of these generators comes from the fact that they have to avoid undefined behaviour in generated C programs, which is avoided much more easily in typical Haskell programs.

Automated failing test case reduction has been explored before [Zeller and Hildebrandt, 2002]. Regehr et al. [2012] experimented with different reduction methods for shrinking C programs, and found that structural reduction gives the best overall results. Due to possible undefined behaviour introduced by reduction, semantic checks had to be performed on the reduced test cases to filter out invalid ones.

Fuzz testing [Miller et al., 1990] is an early example of using weak correctness criteria in random testing. Fuzz testing works by feeding random input data to programs and checking if they crash, and can be used to find security vulnerabilities.

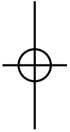
Statement of contributions This paper is a slightly revised version of the author's Licentiate Thesis, which in turn is an extended version of a paper that appeared in the International Workshop on Automation of Software Test (AST), 2011. The original workshop paper was coauthored with Koen Claessen, Alejandro Russo and John Hughes. Koen Claessen came up with the original idea behind the paper. The implementation of the test data generator and the testing was performed mostly by Michał H. Pałka. The workshop paper was joint work of all authors.

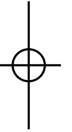


Paper II: Generating Constrained Random Data with Uniform Distribution

This paper presents a method for automatically deriving generators of complex random test data. The derived generators produce random values of an algebraic data type that satisfy a given predicate, which is provided as a lazy Haskell function. The main advantage of the derived generators is that defining a predicate is typically much simpler than creating a dedicated generator of values satisfying that predicate. In addition, the derived generators can provide concrete guarantees about the distribution of generated values, which reduces the risk of important test data being underrepresented.

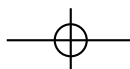
We evaluated the technique on a number of examples, including repeating testing the GHC compiler using one of the properties from the previous paper. We found that the performance of the derived generator was lower than that of the hand-written one for the sizes of values that the testing required, and prevented it from finding bugs at a competitive rate. However, creating a suitable predicate for the derived generator took a matter of days, instead of months that it took to develop the dedicated generator.





The project demonstrated that generating values of a certain size satisfying computable predicates selected uniformly at random is practical in many circumstances. Moreover, developing generators based on computable predicates takes much less effort than dedicated generators. We found that the performance of the derived generators may be considerably influenced by the order of evaluation of the predicate, which means that creating suitable predicates requires some thought, but is still much easier than developing dedicated generators. Even though we found the derived generators to be too slow in some of our benchmarks, we consider these initial results to be encouraging, as addressing the performance issues may make derived generators an effective and widely applicable random test data generation tool.

Related work Bounded-exhaustive testing is a strategy where software is tested for all values up to a certain size, and can be regarded as an alternative to random testing. The rationale behind trying all small values for triggering bugs is the *small scope hypothesis* [Jackson, 2006, Marinov, 2005, Runciman et al., 2008], which maintains that "Most bugs have small counterexamples" [Jackson, 2006]. Bounded exhaustive testing makes test data generators simpler, as they work by enumerating values [Marinov, 2005, Runciman et al., 2008]. Furthermore, the enumerators can also be adapted to efficiently generate values that satisfy a computable predicate



using a similar, but slightly simpler technique than ours [Marinov, 2005, Runciman et al., 2008].

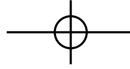
Bounded-exhaustive testing is an attractive way of finding minimal counterexamples, which might be more efficient than finding a random counterexample. On the other hand, the practice of random testing has shown that finding a large counterexample and reducing it is often easier than finding a small failing test case [Andrews et al., 2008, Lei and Andrews, 2005, Regehr et al., 2012]. Thus, we hypothesise that both bounded-exhaustive testing and random testing could show their advantages in different situations, and it is useful to have both of them at hand.

A slightly different approach to generating complex test data has been explored by Reich et al. [2012], who used computable predicates to generate canonical programs of a simple functional programming language. The generation procedure combined a predicate that allows only well-formed programs with other predicates, which ensure that the programs are in a particular canonical form. Generating canonical forms, which are single representative programs from sets pf equivalent programs, led to efficient pruning of the search space and generating a relatively small number of interesting test values, which were then used to falsify properties.

White-box testing [Myers et al., 2012] uses knowledge about the internal logic of a program to generate test inputs. Using that knowledge, white-box testing is able to decide that some test inputs are equivalent, or generate test inputs that lead to particular branches of the program being executed. A disadvantage of white-box testing is the heavy machinery that the testing tools need in order to support it, and that they are often restricted to a particular programming language, or low-level code architecture.

An interesting variant of white-box testing is *white-box fuzz testing* [Bounimova et al., 2013]. White-box fuzz testing starts by choosing a random input and running the program on it using a symbolic execution and collecting the path constraints of the chosen execution path. Then, one of the path constraints is negated, and a concrete input is generated that satisfies the negated path constraint, and all the preceding ones, so that executing the program with the new input will trigger a new execution path. This process is repeated many times, leading to exercising many different program paths. White-box fuzz testing has been successfully used to find many potential security vulnerabilities in Microsoft products.

Statement of contributions This is a slightly revised version of a paper to be presented at the International Symposium on Functional and Logic Programming (FLOPS), 2014. The original paper was coauthored with Koen Claessen and Jonas Duregård. Koen Claessen came up with the original



idea behind the paper and the initial implementation. All three authors were involved in the subsequent implementation and improvement of the generator, and contributed equally to writing the paper.

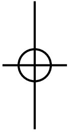
Paper III: Splittable Pseudorandom Number Generators using Cryptographic Hashing

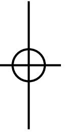
The paper presents the design of a high-quality splittable pseudorandom number generator (PRNG) and a formal argument about its correctness.

Traditional, non-cryptographic PRNGs are developed according to rather *ad-hoc* criteria of pseudorandomness, notably that they have a sufficiently long period, and pass a number of empirical statistical tests [L'Ecuyer and Simard, 2007]. These criteria are relatively fragile, and do not ensure that a PRNG passing them will not fail another statistical test.

We found that this notion of pseudorandomness is not precise enough to guide the design of a splittable PRNG. In particular, evaluating the designs by running a battery of statistical tests takes hours or days and gives very little insight about what a good design should be.

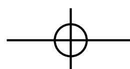
On the other hand, we found that one of the standard cryptographic constructions can be used as foundation for a reliable splittable PRNG. Keyed hash functions are flexible cryptographic primitives that generate pseudorandom output, and are used for example as Message Authentication Codes (MACs). By using a standard cryptographic construction, we managed to outsource the concern of providing pseudorandom output.

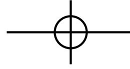




Keyed hash functions are typically built on top of cryptographic block ciphers. To gain confidence about the correctness of their construction, security reduction proofs are provided [Bellare et al., 1996]. The proofs can guarantee a certain degree of pseudorandomness, while making an assumption about the pseudorandomness of the underlying block cipher. This assumption must be asserted, as there are no proofs of pseudorandomness of block ciphers. However, relying on a standard property of a well-studied cryptographic primitive is the best bet that we can make in this situation.

The project resulted in a design of an efficient, high-quality splittable PRNG that is justified by a convincing correctness argument. We found that the cryptographic definition of pseudorandomness is simple and can serve as a robust correctness criterion for pseudorandom number generators. We also found that there was no other readily available framework that could be used to create a reliable splittable PRNG. Despite the common belief that PRNGs based on cryptographic primitives are inefficient, the project demonstrated that modern cryptographic primitives are efficient enough to offer competetive performance to traditional PRNGs. Finally, we found that the problem of creating a keyed hash function is closely related to creating





a splittable PRNG, and if there is any progress in developing new keyed hash function constructions, then it can also be carried over to splittable PRNGs.

Statement of contributions This is a slightly revised version of a paper that appeared in the Haskell Symposium, 2013. The original paper was coauthored with Koen Claessen. Both authors contributed to the design of the generator. Michał H. Pałka implemented the generator and was behind using a security reduction proof to justify its correctness. Both authors contributed to the writing of the paper.

Paper IV: Ranking Programs using Black-Box Testing

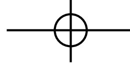
The paper presents an experiment in which students' programming solutions are ranked using a random test suite selected to avoid the bias caused by a particular random distribution of test data.

The original goal of the experiment was to compare the performance of students solving Haskell programming puzzles while using property-based testing, and unit testing, respectively. Unfortunately, the results of the experiment were inconclusive, due to the low number of students that took part in it.

Analysis of students' solutions led to a novel method of ranking different solutions to the same problem based on the observed results of testing against a reference solution. The method looks at the discrepancies between different solutions and the reference solution, and tries to explain them by the smallest number of bugs. Each of the solutions is then assigned the number of inferred bugs based on its test results.

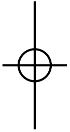
The solutions can be ranked by the numbers of bugs they have, or presented as a partial order where more correct solutions are above the more buggy ones. Each inferred bug is represented by a small test case, and therefore looking at these test cases can give an idea about the differences between the solutions.

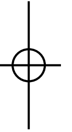
As part of the experiment, the students were also asked to provide test suites for their code, in the form of properties or unit tests, depending on the group. The test suites were evaluated based on how many of the incorrect solutions they can catch. This led to an interesting finding that while unit testing resulted in higher average score for test suites, property-based testing resulted in test suites that are either very ineffective or very effective. Our hypothesis about this result is that properties are more effective, but also more difficult to write. We would like to test this hypothesis in a future experiment.

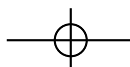


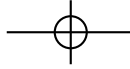
The main outcome of the project is an analytical technique of evaluating and ranking programs based on the level of correctness of their observed behaviour. We found that ranking solutions in this way can be a useful tool for analysing the differences in behaviour of a set of programs. For example, analysing students' solutions using it quickly brought into attention common mistakes made by many students simultaneously. A possible future application of this method might be regression testing, where a number of versions of the same software would be benchmarked against each other. Furthermore, we found the rankings to be relatively stable across different runs of the algorithm. Finally, even though the performed experiment involved too few students to reach conclusive results, we believe that performing a similar experiment on a larger scale would be interesting.

Statement of contributions This is a slightly revised version of a paper that appeared in the International Workshop on Automation of Software Test (AST), 2010. The original paper was coauthored with Koen Claessen, John Hughes, Nick Smallbone and Hans Svensson. John Hughes came up with the idea for the experiment and was the initiator of the work that led to the paper. Each of the authors contributed to running the experiment, performing the analysis and writing the paper.





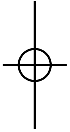
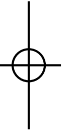




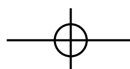
Paper I

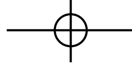
Testing an Optimising Compiler by Generating Random Lambda Terms

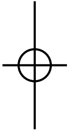
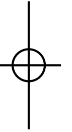
Michał H. Pałka **Koen Claessen** **Alejandro Russo**
John Hughes

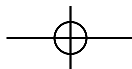



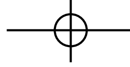
This in an extended version of a paper that appeared in the International Workshop on Automation of Software Test (AST), 2011.









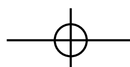
Paper I: Testing an Optimising Compiler by Generating Random Lambda Terms

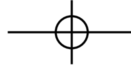
Abstract

This work tries to improve on the relatively uncommon practice of random testing of compilers. Random testing of compilers is difficult and not widespread for two reasons. First, it is hard to come up with a generator of valid test data for compilers, that is a generator of programs. And secondly, it is difficult to provide a specification, or test oracle, that decides what should be the correct behaviour of a compiler. This work addresses both of these problems. Existing random compiler test tools do not use a structured way of generating well-typed programs, which is often a requirement to perform comprehensive testing of a compiler. This thesis proposes such a method based on a formal calculus. To address the second problem, this thesis proposes using two variants of differential testing, which allows for detecting bugs even when a very limited partial specification of the tested compiler is available. This setup is evaluated practically by performing effective testing of a real compiler.

1 Introduction

Correctness of compilers is crucial to most software projects, while at the same ensuring their correctness is relatively difficult. While there exist examples of formally-verified compilers, such as the CompCert optimising compiler [Leroy, 2009] and the Racket compiler and virtual machine [Klein et al., 2010a], such compilers are not common. Instead of using modern testing and verification techniques, writers of production compilers rely mostly on testing based on test cases manually created by themselves or contributed by the users.





Formal verification is an attractive approach to reliable software, but its cost and complexity are still sky-high, which makes it applicable only to very specialised compilers. For instance, over 3/4 of the code of CompCert is devoted to verification [Leroy, 2009]. Software testing, on the other hand, is the most prevalent and economical way of assuring quality, which, if done right, may result in quicker software development [Tassey, 2002].

Thus, compiler writers resort to test suites[1], which are run continuously during development. But these test suites consist largely of test cases that were taken from bug reports submitted by the users, which means that they tend to find few new bugs.

Having an automatic testing tool for a compiler, on the other hand, would give the advantage of finding bugs early. One possibility is to use random property-based testing. However, this requires having a generator of random programs.

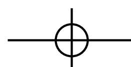
Generating good test programs is not an easy task, since these programs should have a structure that is accepted by the compiler. As compilers often employ multi-stage processing before producing compiled code, in order to test later stages, earlier ones must be completed without error. The requirements for passing a compilation stage can be as basic as a program having the correct syntax, or more complex such as a program being type-correct in a statically-typed programming language.

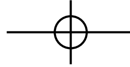
In this thesis, we investigate generation of random type-correct Haskell programs for the purpose of testing the GHC Haskell compiler, and in particular, its optimising middle-end [Peyton Jones, 1996].

We chose the *simply-typed lambda-calculus* [Pierce, 2002] as the underlying model of well-typed programs as it is the simplest calculus that has the notion of variable binding and type-correctness, as well as first-class functions, which makes it adequate for representing simple Haskell programs. On top of that, we support polymorphic constants, which makes it possible to generate simple Haskell programs that use polymorphic library functions as well as to encode in them some programming language constructs.

The presented generator of random simply-typed lambda-terms has been successfully applied to testing the GHC Haskell compiler. This compiler contains a particularly sophisticated optimising middle-end, which performs many stages of intermediate-language transformations, such as inlining, let-floating, lambda lifting, specialisation and common subexpression elimination [Peyton Jones, 1996]. Such elaborate processing could easily be a source of intricate bugs, making it especially interesting to test.

[1]For example, the GCC compiler test suite: `http://gcc.gnu.org/onlinedocs/gccint/Testsuites.html`; or the GHC test suite: `http://hackage.haskell.org/trac/ghc/wiki/Building/RunningTests/Adding`.





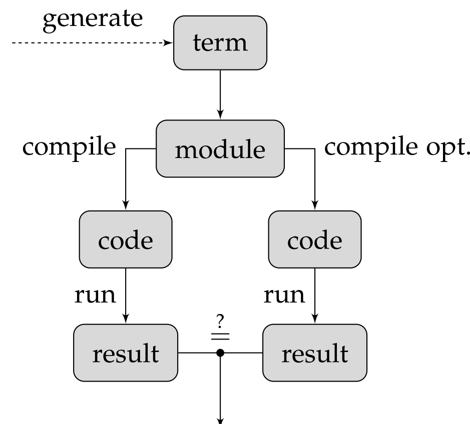


Figure 1.1: Differential testing

The GHC compiler has been tested with randomly generated Haskell programs using *differential testing* [McKeeman, 1998], which involved compiling the same program with different optimisation settings and comparing its observed behaviour. Figure 1.1 shows a diagram of the testing process. Testing uncovered four interesting bugs, three of which were fixed based on our bug reports[2]. We also learned interesting facts about valid optimisations performed by the GHC.

Reported failing test cases were often reasonably small due to the process of *shrinking*, which automatically reduces the failing test case as much as possible. Below is one of the found terms that provoked a failure.

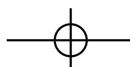
```
(\a -> seq a (seq (a []) id)) (\a -> seq undefined (+1))
```

This term can be manually rewritten as follows, to obtain a program that provokes a bug:

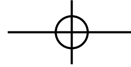
```
a = \x -> seq undefined (+1)

main = do
  print $ (a [] ‘seq‘ id) [0]
```

This short program turned out to be miscompiled by the tested version of GHC when optimisation was turned off. The failure has been reported as ticket 5625 and the bug causing it was fixed. More information about this failure can be found in Section 6.1.

[2]The remaining one has been fixed as a result of another bug report.



thesis March 27, 2014 10:04 Page 20



2 Related work

Even though random testing is not commonplace in compiler development, there are accounts of its successful application. The work of McKeeman [1998] and the CSmith tool [Yang et al., 2011] are both such examples. Both tools employ *Differential testing* where results of programs compiled with different C compilers are compared to spot bugs. Generating random programs was a central part of both of these efforts, and in both cases the problem was solved using an elaborate, but ad-hoc generator.

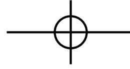
McKeeman focuses on testing all stages of a C compiler by generating programs of different level of conformity to the C language: lexically-correct, syntax-correct, etc., and much effort is put into creating program generators for each of these levels. CSmith, on the other hand, focuses on testing optimising middle-ends of the compilers and most effort was put into generating C programs that do not depend on undefined or unspecified behaviour. Assuring this semantic property is, of course, difficult, which is why CSmith uses an array of heuristics and checks for assuring it.

Notwithstanding the effort put into creating these program generators, both test tools found a big collection of previously unreported bugs in the tested compilers. The test cases reported by these tools are usually large and hard to analyse. Only McKeeman discusses automated reduction of size of counterexamples—in case of CSmith they have to be reduced by hand, which can be very labour-intensive.

Lindig [Lindig, 2005] created a random testing tool to test whether the C function calling convention is followed by compilers. The generation of programs is much simpler in this case, as only types of functions to be test-called need to be randomly generated. The rest of the program is skeleton code that is generated algorithmically. Despite its simplicity, the tool managed to find a number of discrepancies between compilers that manifested when a function was called from code compiled with another compiler. Automated testing of unexpected cases was important.

Random program generators have also been successfully applied to testing Java libraries [Klein et al., 2010b]. This work defines a formal calculus, whose random terms are then transformed into programs and is able to generate programs containing higher-order features.

Verified compilers The CompCert optimising compiler [Leroy, 2009] and the Racket compiler and virtual machine [Klein et al., 2010a] are notable for their quality assurance. The former is a full-fledged optimising compiler that was constructed with formal verification in mind. However, it only supports a subset of C and the catalogue of optimisations performed by it is limited. Furthermore, even though the most important of its parts are



formally-verified, which was a costly task, testing was able to find in it previously unknown bugs [Yang et al., 2011].

The Racket compiler is verified using lightweight formal verification, which is realised by testing the compiler against a formal model. While the compiler supports unrestricted version of the Scheme programming language, it is only capable of performing basic optimisations.

Term generators Generation of random lambda-terms has attracted moderate attention. There were attempts at generating random untyped lambda terms [Bodini et al., 2011, Wang, 2005], where it is already difficult to obtain a reasonable distribution. Two notable attempts at generating simply-typed lambda terms are based on bit-encoding schemes [Vytiniotis and Kennedy, 2010] and enumeration [Rodriguez Yakushev and Jeuring, 2010], but it is unclear whether it is possible to adapt the latter for random generation.

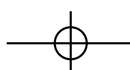
Unfortunately, none of these works considers any practical applications for randomly generated lambda-terms—the only application considered is examining statistical properties of random lambda terms [Bodini et al., 2011, Wang, 2005]. Also, none of the typed generators handles parametric polymorphism.

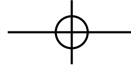
3 Structure

The rest of this section introduces *property-based testing*, which is the way of structuring test data generators and test oracles used throughout the paper. In the next section we explain our approach to random generation of simply-typed lambda terms. Section 5 describes the method of structurally shrinking the generated terms aimed at reducing the sizes of reported counterexamples. We identify two interesting design decisions that we can make in the shrinking method and evaluate their performance based on experimental benchmarks. Section 6 presents the failing test cases found for GHC and properties used to find them. We also analyse the consequences of the noticed discrepancies for the programmers using GHC. Section 7 describes related work, and Sections 8 and 9 present future work and conclusions.

Contributions We claim the following contributions:

- We present a generator of random simply-typed lambda terms based on performing random local choice and backtracking. The generator makes use of tailored generation rules and heuristics to avoid excessive backtracking and skewing the distribution of generated





terms too much. The generator also supports generating terms with polymorphic constants. (Section 4)

- The generator is applied successfully for finding bugs in GHC optimising compiler using differential testing. (Section 6)
- We perform differential testing using two expressions that should give equivalent results and find discrepancies that are independent of optimisation options. (Section 6)
- We present a method for shrinking simply-typed lambda terms, which is used for reducing counterexamples found by us during testing. The method proposed by us reduces the terms structurally, but is very effective in reducing the test cases despite the complexity of the processing performed by GHC. Shrinking makes finding the root cause of a bug based on a counterexample dramatically easier. (Sections 5 and 6)

3.1 Property-based testing

To obtain test oracles for random testing we employ *property-based testing*, which allows us to derive test oracles naturally from logical properties. For example, consider the commutativity law for integers.

$$\forall n\, m.\ n + m = m + n$$

We may turn it into a testable property by writing a Haskell function that checks this equality for two given numbers.

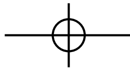
```
prop_comm :: Integer -> Integer -> Bool
prop_comm n m = n + m == m + n
```

This function is an executable version of the above logical property and may be used as an oracle in random testing. We use the word 'property' to denote such executable properties throughout this paper.

A single test on the above property is performed by generating two random numbers `n` and `m`, evaluating the function and checking if its result is true.

Testing this property involves running the function on a finite number of inputs when the number of all inputs is infinite, so testing can only result in *disproving* the property, by finding a counterexample, or leaving its validity undecided.

The function `prop_comm` implements both the tested code (the `+` operator) and the test oracle (the `==` operator), while the random test case generator is separated.



Variables	$x, y, \ldots$		
Constants	$\mathbf{c}, \mathbf{d}, \ldots$	::=	`head`, `tail`, `(+)`, `0`, `1`, ...
Types	$\sigma, \tau, \ldots$	::=	`Int` | `Bool` | `ListInt` | $\cdots$ | $\sigma \to \tau$
Terms	$M, N, \ldots$	::=	$x \mid \mathbf{c} \mid \lambda x{:}\sigma.\ M \mid MN$

Figure 1.2: Syntax for simply-typed λ-calculus

The properties used by us to test the GHC compiler employed differential testing, described in Section 6, and an example diagram illustrating the process is shown in Figure 1.11 in that section.

QuickCheck [Claessen and Hughes, 2000] is a Haskell library that provides comprehensive support for property-based testing. QuickCheck contains a combinator library for building composable properties as well as random generators for basic Haskell data types. We implemented our random simply-typed term generator in QuickCheck using these basic generators and performed testing of the GHC compiler using properties also written in QuickCheck.

QuickCheck also provides generic support for *shrinking* [Claessen and Hughes, 2000][3] that tries to reduce the size and complexity of the reported counterexamples. We implemented a shrinking method for the simply-typed lambda terms for use in our testing.

The method for generation of random simply-typed lambda terms is independent from QuickCheck, however many other aspects of the testing are heavily influenced by property-based testing, for example the properties and shrinking.

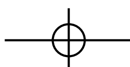
4 Generation method

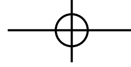
To generate random programs for use in testing we consider simply-typed lambda terms [Pierce, 2002] that can contain constants in addition to variables. Their syntax is shown in Figure 1.2. Typing rules, shown in Figure 1.3, are standard and constants are typed in the same way as variables. We require that all variables and constants in environments have distinct names.

The aim of the generator is to produce a well-typed term of a certain type, which can contain free variables and constants from a given environment. Of course there are combinations of target type and environment for which no term can be constructed.

Simple generator One possible generation method can be obtained by reading the typing rules shown in Figure 1.3 backwards. To generate a

[3] Shrinking is not referred to by name in this paper, but is realised by the function 'smaller'.





Typing judgements $\Gamma \vdash M : \sigma$
Environment $\Gamma \quad ::= \quad \{x_1 : \sigma_1, x_2 : \sigma_2, \mathbf{c_1} : \sigma_3, \ldots\}$

$$(\text{Var})\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad (\text{Cnst})\ \frac{\mathbf{c} : \sigma \in \Gamma}{\Gamma \vdash \mathbf{c} : \sigma} \qquad (\text{Lam})\ \frac{x : \sigma, \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau}$$

$$(\text{App})\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Figure 1.3: Typing rules

term that is in the consequence of a rule it is firstly necessary to generate terms that are in its premises, if they exist, and then combine them. In other words, the *goal* of generating a term might involve generating the *subgoals* recursively, leading to a procedure that works top-down and produces a term together with its typing derivation. Of course, the derivation must be finite. The rules ensure that the resulting terms are well-typed.

Suppose that we want to generate a simply-typed λ-term of type `Int` while having access to the following constants: `z:Int` and `s:Int` $\to$ `Int`.

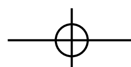
The simplest term that can be generated is just `z`, as this constant has the right type. This term is generated by applying the Cnst rule with **c** *instantiated* with the constant `z`. This rule does not have any recursive premises, only a side condition that `z` must be in the environment, so no subgoals must be generated to finish the generation. Successful generation yields a type derivation tree for the generated term, shown below. We define Γ to be the initial environment $\{\texttt{z:Int}, \texttt{s:Int} \to \texttt{Int}\}$.

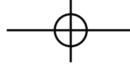
$$(\text{Cnst})\ \frac{\texttt{z:Int} \in \Gamma}{\Gamma \vdash \texttt{z:Int}}$$

We can generate another term if we apply `s` to `z` and obtain the term `s z`, which also has the right type. To do that by following the typing rules, we must first apply an instance of the App rule where both σ and τ are instantiated with `Int`. Applying this rule requires two subgoals to be generated recursively with their own *generation contexts*, that is their target types together with their environments. Below we show the part of the derivation tree that is determined after selecting the App rule.

$$(\text{App})\ \frac{\Gamma \vdash ?_1 : \texttt{Int} \to \texttt{Int} \qquad \Gamma \vdash ?_2 : \texttt{Int}}{\Gamma \vdash ?_1\ ?_2 : \texttt{Int}}$$

The question marks ($?_1$ and $?_2$) represent the subterms that will be generated as subgoals.





The first subgoal has the same environment as the original term, but a different target type, which is $\texttt{Int} \to \texttt{Int}$. This subgoal is generated using the Cnst rule instantiated with the constant s.

The second subgoal receives the same generation context as the original term and is generated using the Cnst rule.

Solving the subgoals yields the missing parts of the term and its derivation tree.

$$(\textsc{App})\ \dfrac{(\textsc{Cnst})\ \dfrac{\texttt{s}:\texttt{Int} \to \texttt{Int} \in \Gamma}{\Gamma \vdash \texttt{s}:\texttt{Int} \to \texttt{Int}} \qquad (\textsc{Cnst})\ \dfrac{\texttt{z}:\texttt{Int} \in \Gamma}{\Gamma \vdash \texttt{z}:\texttt{Int}}}{\Gamma \vdash \texttt{s z}:\texttt{Int}}$$

In the same way it is possible to generate more complex terms, for example $\texttt{s}\ (\texttt{s}\ (\texttt{s z}))$. The terms can also contain locally-defined functions in the form of λ-expressions. In this way, we can create more type-correct λ-terms, for example $(\lambda x:\texttt{Int}.x)\ (\texttt{s z})$, or $(\lambda x:\texttt{Int}.\texttt{s}\ (\texttt{s}\ x))\ \texttt{z}$. The Lam typing rule, which is needed for generating a locally-defined function, adds one variable to the environment of its subgoal, which makes it possible to refer to that variable in the body of the function.

Thus, interpreting the typing rules as *generation rules* allows us to generate well-typed terms in a top-down fashion. The four rules are capable of generating every well-typed term—after all a term is well-typed only if there exists a typing derivation for it.

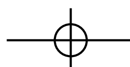
Using a generation rule to generate a part of a term involves choosing its specific instance. Choosing a suitable instance of the Var and Cnst rules is straightforward, as the type of the chosen variable or constant must be the same as the target type. If such variable or constant is not available, then the rules cannot be applied.

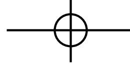
Similarly, the Lam rule can only be applied if the target type is functional and type σ from the rule must be equal to the argument of the target type.

This is not the case, however, when the App rule is applied, as the type of the argument is not determined by the generation context and can be chosen freely.

Often a specific instance of the App rule is required to occur in a derivation tree of a term. For example, suppose that a derivation tree might only be constructed by using constant **c** that has type $\sigma_1 \to \sigma_2 \to \sigma_3 \to \tau$ to solve a goal of type τ. The App rule must be then instantiated with σ_3. Furthermore, two more applications of the App rule are required, also with specific types.

In addition, some instances of the App rule cannot occur in a valid derivation tree. Notably, if it is instantiated with an argument type, for





which a term cannot be constructed, then no valid derivation containing it exists.

Thus, the derivation is *sensitive* to the way the App rule is instantiated.

Concrete realisation The generation rules give us a *non-deterministic* generation procedure as in a given generation context it might be, and frequently is, possible to use more than one instance of the rules. Unfortunately, we can not have the luxury of allowing the generator to select an arbitrary viable instance, as bad choices may lead the generator to a dead end or non-termination, even if making another choice would result in successful generation.

These problems can be solved by using a procedure that performs backtracking. Whenever a bad choice is made, the procedure will fail and backtrack to another choice. Given that it is possible to run into an infinite loop, a limit on the number of recursive invocations is imposed.

Unfortunately, since the success or failure of the generation depends so much on the way the App rule is instantiated, such generation procedure suffers from excessive backtracking, which is why we propose a different set of generation rules.

4.1 Alternative rules

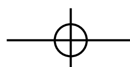
We chose to use an alternative set of generation rules instead of adopting the typing rules for that purpose. Consider following method where a term can be generated in two ways:

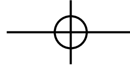
- We may introduce a lambda expression if the target type is functional. The body of the lambda expression must then be generated to complete the term.

- We may use a symbol from the environment, constant or variable, that possibly needs some arguments to be applied to it to match the target type. The needed arguments become new goals that have to be generated. If the type of the symbol is the same as the target type, no arguments are needed and generation is finished.

The first tactic is captured by the Lam rule in Figure 1.3 and the formal rule for the second one is given below:

$$(\textsc{Indir}_{\mathrm{V}})\ \frac{f : \sigma_1 \to \ldots \to \sigma_n \to \tau \in \Gamma \quad \Gamma \vdash M_1 : \sigma_1 \ \cdots \ \Gamma \vdash M_n : \sigma_n}{\Gamma \vdash f M_1 \ldots M_n : \tau}$$





There is also a sister rule for constants, which we omit here. Please note that τ can be any type, also functional, and that the Indir rules supersede Var and Cnst if $n = 0$.

This choice of generation rules gives us a generation method with interesting properties. First, it is not possible to generate all terms with it, as it never generates any β-redexes. Secondly, it is nevertheless complete, as it will be able to generate a term if a well-typed term exists for a given combination of the target type and environment. And finally, the problem of generation being very sensitive to the way rules are instantiated is reduced. The reason for this is that applying the Indir rule does not involve choosing any types, and while choosing the wrong variable or constant might result in failed generation there is a finite number of choices to make.

In a way, the Indir rule is a 'guided' App rule, which chooses the argument types to suit a specific variable or constant from the environment.

Keeping the App rule For the purpose of practical generation, we nevertheless keep the App rule in addition to the Indir rule to be able to generate β-expanded terms. While the Indir rule gives higher chances of successfully generating a term without excessive backtracking, the App rule is capable of generating β-expanded terms.

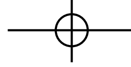
4.2 Polymorphic constants

So far we discussed the simply-typed lambda calculus with monomorphic constants, which means that constants have specific types. However, typical Haskell programs make use of *parametric polymorphism*, which requires a richer term representation.

The type system of Haskell is quite complex, but a large part of code in Haskell uses only one aspect of parametric polymorphism, notably *polymorphic constants*[4]. Other variations of polymorphism in Haskell, such as *higher-rank types* or polymorphic let definitions require a more complicated type system, which is why we decided to support only polymorphic constants.

Polymorphic constants can be used to represent functions operating on polymorphic data structures, such as lists. They are also needed to build expressions that use advanced Haskell libraries, including ones involving monads, applicative functors and arrows.

[4] In particular, Haskell 98 programs are restricted to Hindley-Milner polymorphism with monomorphic let bindings if no explicit type signatures are provided [Peyton Jones, 2003].



Variables	$x, y, \dots$	
Constants	$\mathbf{c}, \mathbf{d}, \dots$	::= `head`, `tail`, `+`, `0`, `1`, ...
Type variables	$\alpha, \beta, \dots$	
Types	$\sigma, \tau, \dots$	::= `Int` | `Bool` | $\sigma \to \tau$ | α | `List` σ | $\cdots$
Polymorphic types	$\Sigma, \Upsilon, \dots$	::= $\forall \alpha\beta\gamma \cdots .\sigma$
Terms	$M, N, \dots$	::= x | $\mathbf{c}$ | $\lambda x{:}\sigma.\ M$ | MN

Figure 1.4: A simple λ-calculus with polymorphism

Polymorphic constants can also be used to encode some programming language constructs, such as if-then-else, which means that support for them does not have to be hard-coded into the generator.

Instead of supporting polymorphic constants, we could replace them with a number of their instances, for example:

$$\mathtt{head}_{\mathtt{Int}} : \mathtt{List\ Int} \to \mathtt{Int}$$
$$\mathtt{head}_{\mathtt{Int} \to \mathtt{Int}} : \mathtt{List}\ (\mathtt{Int} \to \mathtt{Int}) \to \mathtt{Int} \to \mathtt{Int}$$
$$\dots$$

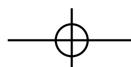
However, this would lead to an explosion of constants and, given that there are infinitely many instances of each (non-trivial) polymorphic type, it is not clear which and how many instances should be included.

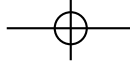
To accommodate polymorphic constants we must extend our core calculus, as shown in Figure 1.4. Polymorphic types are written as $\forall \alpha\beta\gamma \cdots .\sigma$, where $\alpha, \beta, \dots$ are type variables, which are allowed to occur in σ and are replaced by types during instantiation. It is illegal for a type variable to occur in a non-polymorphic type even though the syntax allows this.

The typing rules, shown in Figure 1.5, are only slightly different to previous typing rules. All terms have monomorphic types. Lambda-bindings and, in consequence, variables are monomorphic. Also, the occurrences of constants are fully instantiated, and thus all have concrete types. The only rule that has changed is Cnst, which now performs instantiation of the constant's polymorphic type.

Generation rules for terms with polymorphic constants also need only small changes compared to previous ones. The rule that changes is Indir_{C}, which introduces constants:

$$\Sigma[\alpha \mapsto \rho_1, \dots] = \sigma_1 \to \dots \to \sigma_n \to \tau$$
$$(\text{Indir}_{\text{C}})\ \frac{\vdots \quad \mathbf{f}{:}\Sigma \in \Gamma \qquad \Gamma \vdash M_1{:}\sigma_1 \quad \cdots \quad \Gamma \vdash M_n{:}\sigma_n}{\Gamma \vdash \mathbf{f} M_1 \dots M_n{:}\tau}$$





Typing judgements			$\Gamma \vdash M : \sigma$
Environment	Γ	::=	$\{x_1 : \sigma_1, x_2 : \sigma_2, \mathbf{c_1} : \Sigma_1, \ldots\}$
Type substitution			$[\alpha \mapsto \tau_1, \beta \mapsto \tau_2, \ldots]$

$$(\textsc{Var})\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad (\textsc{Cnst})\ \frac{\mathbf{c} : \forall \alpha\beta \cdots . \sigma \in \Gamma}{\Gamma \vdash \mathbf{c} : \forall \alpha\beta \cdots . \sigma[\alpha \mapsto \tau_1, \beta \mapsto \tau_2, \ldots]}$$

$$(\textsc{Lam})\ \frac{x : \sigma, \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau} \qquad (\textsc{App})\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Figure 1.5: Typing rules for simple λ-calculus with polymorphism

Constant **f** can be used if its polymorphic type Σ can be instantiated so that its result is the same as the target type τ, which is realised by the side condition marked with vertical dots.

Instantiation

Of course, in order to use the $\textsc{Indir}_C$ rule, both **f** and an instantiation of its type must be chosen. Depending on the circumstances, this might be easy or difficult. The following examples examine different cases.

Example 1 Consider that the following constant is present in the environment:

$$\texttt{tail} : \forall \alpha . \texttt{List}\ \alpha \to \texttt{List}\ \alpha \in \Gamma$$

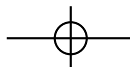
and that the target type is `List Int`. If we want to use `tail` to generate this term we have to find an instance of the $\textsc{Indir}_C$ rule. But the only instantiation that can possibly be used here is $[\alpha \mapsto \texttt{Int}]$, and thus the generation step might look as follows:

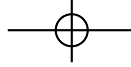
$$(\textsc{Indir}_C)\ \frac{\begin{array}{c}\Sigma[\alpha \mapsto \texttt{Int}] = \texttt{List}\ \alpha \to \texttt{List}\ \alpha \\ \vdots \end{array} \quad \texttt{tail} : \Sigma \in \Gamma \quad \dfrac{\cdots}{\Gamma \vdash \mathbf{?} : \texttt{List}\ \alpha}}{\Gamma \vdash \texttt{tail}\ \mathbf{?} : \texttt{List}\ \alpha}$$

Therefore, in this case the instantiation is completely guided by the environment and target type.

Example 2 A slightly more complicated situation occurs when the identity function is to be instantiated. The identity function has following type:

$$\texttt{id} : \forall \alpha . \alpha \to \alpha$$





Suppose that, for example, the type of the expression that is to be generated is `Int`. Then, the most obvious choice is to instantiate α with `Int` and generate `id` applied to an `Int` argument. However, it is not the only choice. We may as well instantiate α with $\texttt{Bool} \rightarrow \texttt{Int}$ in which case applying `id` to an argument yields a *function* of type $\texttt{Bool} \rightarrow \texttt{Int}$. Thus, `id` effectively becomes a *two-argument function* that can be applied to arguments $\texttt{Bool} \rightarrow \texttt{Int}$ and `Bool`. Continuing this way, we may instantiate α with $\texttt{Bool} \rightarrow \texttt{Bool} \rightarrow \texttt{Int}$ and obtain an instance of `id` that takes *three* arguments. We may carry on adding arguments to `id` like this forever, even if the resulting terms would look uncommon.

This demonstrates that whenever a constant has type that looks like $\forall\alpha \cdots . \cdots \rightarrow \alpha$, there will be infinitely many possible ways of instantiating it. Fortunately, it is not likely that constants applied to many 'extra arguments' are relevant for generating interesting terms.

Example 3 Consider the function map and that we want to use it to generate a list of integers. Map is represented with the following constant:

$$\texttt{map}:\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow \texttt{List}\ \alpha \rightarrow \texttt{List}\ \beta$$

Since the resulting term has the `List` type, the constant must be applied to two arguments. Furthermore, the target type of `List Int` dictates that β in the type of `map` must be instantiated with `Int`. However, nothing constrains how α must be instantiated, making it possible to use any instantiation of it.

Therefore, many instantiations of `map` are possible, and the same problem occurs with several important constants that we might want to use for generating terms, such as:

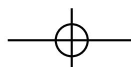
$$\begin{aligned} \texttt{ap}\ &: \forall\alpha\beta.\texttt{A}\ (\alpha \rightarrow \beta) \rightarrow \texttt{A}\ \alpha \rightarrow \texttt{A}\ \beta \\ \texttt{bind}\ &: \forall\alpha\beta.\texttt{M}\ \alpha \rightarrow (\alpha \rightarrow \texttt{M}\ \beta) \rightarrow \texttt{M}\ \beta \end{aligned}$$

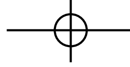
The problem is important as specific instances of `map` and similar constants are likely to be required to generate some interesting terms. For example, `bind` has to be used (and instantiated) in order to build complex monadic expressions.

Also, not surprisingly, if we include a constant that represents function application, exactly the same problem would occur as we would have to select α in the following type:

$$\texttt{(\$)}:\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

Therefore, the problem of instantiating such constants is more general then the problem of instantiating the App rule.





These three examples show that the problem of instantiating polymorphic types in the Indir rule is (a) easily solvable in some cases and (b) choosing arbitrary types might be needed in other cases, which is not crucial for generation, but (c) there are also important cases where instantiation is critical.

We were not able to solve this problem completely, but we present a partial solution of it, which is based on heuristics, in our generation algorithm.

Generation algorithm

The algorithm generates terms recursively top-down by applying generation rules. To avoid non-terminating generation each recursive invocation of the algorithm uses a *size parameter*, which is decreased in subsequent invocations.

The first step of each recursive invocation is to create a list of admissible instances of the generation rules, that is the instances that can be used in the current context.

When the size parameter is 0 the list is restricted only to non-recursive rule instances, and thus the generation is forced to succeed or fail without further recursion.

The list is then permuted at random and the first rule instance from it is applied. If the rule instance requires subterms to be generated, these get turned into subgoals and the generation procedure is invoked recursively with new generation contexts.

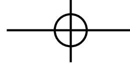
When the recursive invocations succeed, their results are combined into the resulting term according to the generation rule and the current invocation of the generator concludes. When they fail, the generator performs backtracking by selecting the next admissible instance of some generation rule from the list. If there are no more instances left, the original recursive invocation fails.

The size parameter used to generate the subgoals of a goal is decreased using the following equation, where p is the number of immediate subterms that need to be generated and s_{g} is the size parameter of the goal.

$$s_{\mathrm{sg}} = \left\lfloor \frac{(s_{\mathrm{g}} - 1)}{p} \right\rfloor \tag{4.1}$$

Example Consider an environment containing constants `tail`:$\forall\alpha.$`List` $\alpha \to$ `List` α and `x:List Int`. Note that we omit writing $\forall$ when a polymorphic type does not have type parameters. If the type of the term to be generated is `List Int`, the admissible rule instances are the following:

thesis March 27, 2014 10:04 Page 32



- The INDIR rule may be instantiated with `tail` and $n = 1$, which requires generating a subterm of type `List Int`. The instantiation involves a substitution $[\alpha \mapsto \texttt{Int}]$ and both τ and σ_1 are instantiated with `List Int`. The instance is presented below in full:

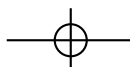
$$\frac{\begin{array}{c}\forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha[\alpha \mapsto \texttt{Int}] = \texttt{List Int} \to \texttt{List Int} \\ \vdots \quad \texttt{tail}{:}\forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha \in \Gamma \qquad \Gamma \vdash M_1{:}\texttt{List Int}\end{array}}{\Gamma \vdash \texttt{tail}\ M_1{:}\texttt{List Int}}\ (\textsc{Indir}_C)$$

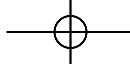
- The INDIR rule may similarly be instantiated with `x` and $n = 0$.
- The APP rule may be instantiated with τ mapped to `List Int` and with σ mapped to any type.
- There is no admissible instantiation of the LAM rule.

If the size parameter is 0, then INDIR instantiated with `x` becomes the only admissible instance, as it is the only one that is non-recursive.

The algorithm described above is an ordinary randomised search algorithm with backtracking and size limit. The important part of it is how the set of admissible rule instances in a given context is selected. Ideally, we should select all the admissible rule instances, but because of the possible many instantiations of some constants, this set might be infinite and therefore we must approximate it by selecting the subset of all possible instantiations. The following procedure is used:

- Admissible instances of $\textsc{Indir}_V$ and LAM are selected completely as there is a finite number of them. Variables are monomorphic and the type of the λ-bound variable is uniquely determined by the generation context.
- Applying the APP rule involves choosing the argument type. To instantiate the rule, $N_{\mathsf{app_tries}}$ types are chosen at random and instances of APP based on these types are included in the set.
- There are three cases involving the $\textsc{Indir}_C$ rule, inspired by the examples that we discussed previously:
    1. When the instantiation of a given constant is unique, that instantiation is chosen.
    2. When the constant's type looks like $\forall\alpha \cdots . \cdots \to \alpha$, the constant can be applied to a number of extra arguments. Up to $N_{\mathsf{extra_arg}}$ extra arguments can be applied, and any types that are not determined are chosen at random.





3. When the constant's type is not fully instantiated based on the target type, the remaining types are chosen at random.

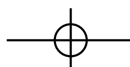
Limiting the number of instances of App and Indir_C is done by selecting arbitrary types at random in places where any type is allowed. The procedure for random selection of types chooses them based on the set of types available in the environment. First, a set of base types is created by considering different combinations of symbols from the environment and then a 'small' random type is generated from it.

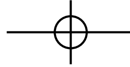
Parameter $N_{\text{app_tries}}$ is arbitrarily fixed to 5 and this number of instances of the App rule are created, as the rule has a high chance of failing if the wrong type is chosen. Higher values of $N_{\text{app_tries}}$ did not seem to be more effective in triggering bugs in our testing, but led to excessive backtracking, which slowed down the generator.

Parameter $N_{\text{extra_arg}}$ is set to 3. Values higher than 1 already did not give any advantage in testing. We decided to conservatively increase $N_{\text{extra_arg}}$ to 3, as the there is no performance hit associated with that. Choosing an arbitrary value for this parameter prevents some terms from being present in the distribution of the generator. However, we believe that these terms should occur very rarely in any reasonable distribution and omitting them is acceptable. Note that generation is not sensitive to this parameter as using extra arguments is never required for it to succeed.

Weights The random backtracking algorithm tries the rule instances in a random order in a given recursive invocation. In order to increase the variety of the generated terms, the rules have *weights* assigned to them. Instances of rules with higher weights have higher chances of being tried first. Weights assigned to rules are respectively 2 for rules that use locally-bound variables, 1 for rules that introduce constants and 4 for the rules for introducing an application or a λ-expression. We tried different combinations of weights and found that increasing the weights on the App and Lam rules increases the chances of triggering bugs during our testing. On the other hand, increasing the weight of the App rule beyond 4 (if other weights are low) causes excessive backtracking. We also chose to prefer local variables to constants as using them should lead to expressions with more 'interesting' semantics.

Instance selection Selecting viable instances of rules involving variables Lam and Indir_V, is relatively easy and involves checking for equality between monomorphic types. Choosing instances of Indir_C is done by performing unification of the target type with possibly modified types of constants.





Generating terms containing `seq` Some of our applications require generating terms that contain occurrences of the following constant.

$$\texttt{seq}:\forall\alpha\beta.\alpha \to \beta \to \beta$$

The purpose of the Haskell function `seq` is to change strictness of an expression without changing its semantics in any other way. The meaning of expression `seq a b` is the same as that of `b`, with the difference that the former is strict in `a` (however, `b` might be also strict in `a` by itself). The function `seq` is often used to eliminate space leaks in Haskell programs by making forcing some expressions to be evaluated. More discussion about function `seq` can be found in Section 6.2.

Using this constant expands the generation space greatly, as matching β against the target type leaves the choice of α completely unconstrained. Given that `seq` is most useful in our applications when its first argument contains locally-bound variables, instead of treating it as ordinary constant, the generator allows it to have only variables as first arguments.

Distribution

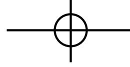
The generator described above works by performing *random local choice*, influenced by rule weights, and is restricted by the size parameter. The distribution of the generated terms is similar to that for a size-limited generator based on stochastic grammars.

Due to the fact that the size parameter is always distributed evenly between subterms (Equation 4.1), the distribution of the generated terms is biased towards full trees, compared to the uniform distribution of terms with 'n' nodes.

The current design of the generator results in that some terms whose generation involves guessing of types, as described in Section 4.2, might be underrepresented in the distribution if successful generation of subgoals of a rule depends on specific types being chosen in that instantiating that rule.

5 Shrinking

Counterexamples found by random generation of lambda terms often look strange and convoluted. However, in order to come up with a useful bug report the counterexample must be convincing to the developers working on the project.



The following term was found to violate a property, described in Section 6, that optimisation should not increase strictness.

$$
\begin{aligned}
&(\lambda a.\texttt{seq}\ a\ ((\lambda b.\texttt{seq}\ a\ (\lambda c.\texttt{seq}\ b\ \texttt{tail}))\ (a\ (\texttt{head undefined}))\\
&\quad(\texttt{case1}\ (\lambda b.\texttt{length})\ (\texttt{seq}\ a\ \texttt{2})\ (\texttt{seq}\ a\ \texttt{undefined}))))\\
&\quad(\lambda a.\texttt{seq}\ a\ ((\lambda b.\texttt{seq}\ a\ (\texttt{seq}\ a\ (\texttt{seq}\ a\ (\lambda c.\texttt{seq}\ c\ (\texttt{seq}\ b\ \texttt{undefined})))))\\
&\quad(\texttt{seq}\ a\ (\lambda b.\texttt{seq}\ b\ (\lambda c.a)))))
\end{aligned}
$$

The term is a tangible proof that there is *something* wrong with the tested compiler, however to say *what* goes wrong exactly is more difficult. In particular, tracing the execution of the compiler on it is likely to be very labour-intensive because of the size of the term, which would make identifying the root cause of the bug difficult.

Fortunately, it is quite likely that only part of the test case is relevant for triggering the problem and there probably exists a smaller term that triggers the same bug. We use a technique called *shrinking* [Claessen and Hughes, 2000] to search for a simpler counterexample that also triggers a bug. Shrinking was able to reduce the above counterexample to a simpler term that violates the same property:

$$(\lambda a.\texttt{seq}\ a\ (\texttt{seq}\ (a\ \texttt{undefined})\ \texttt{tail}))\ (\lambda a.\texttt{seq}\ \texttt{undefined}\ \texttt{(+1)})$$

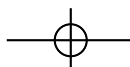
The shrunk term is much smaller than the original one, which makes its analysis much easier. It is also *minimal* in the sense that it cannot be further reduced, which suggests that the term contains only parts that are required for triggering the bug. Furthermore, shrinking reduces the variation of terms reported as counterexamples, as different terms originally found during testing often shrink to the same or very similar terms. This makes differentiation between different bugs easier as one bug is typically represented by a set of similar shrunk terms.

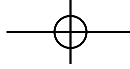
Moreover, looking at different shrunk terms that violate one property can give even stronger hints about the problem that is triggered. For example, most terms found by this property contained subterms that look like $\lambda a.\texttt{seq undefined}\ \ldots$, which define a function that is a defined value itself, but returns $\perp$ given any argument.

Given their size and understandability, shrunk terms are good candidates for including in bug reports as they are more likely to convince the developers that the reported bug is worth fixing.

5.1 Shrinking simply-typed lambda terms

Shrinking is done by searching for a smaller term similar to the original one that also causes the given property to fail. Smaller *shrinking candidates*





are created by reducing the term structurally. When one of the shrinking candidates triggers a failure shrinking will try to reduce it further. The example term shown earlier has been shrunk in 17 steps.

To illustrate a shrinking step, suppose that the term below provokes some failure.

$$(\lambda x.\texttt{id}\ (\texttt{tail}\ x))\ (\texttt{b ++ b})$$

Constants that appear in it represent common Haskell functions and have the following types:

$$\begin{aligned}\texttt{id} &: \forall\alpha.\alpha \to \alpha\\ \texttt{tail} &: \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha\\ \texttt{(++)} &: \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha\end{aligned}$$

Constant `b` is a list of integers and has type `List Int`.

In search of a smaller term that also provokes a failure, we may turn to looking at subterms of the original term. For example, subterm `b ++ b`, which has the same type as the original term, could be considered. Term $\lambda x.\texttt{id}\ (\texttt{tail}\ x)$ cannot be used as it has a different type. Terms $\texttt{id}\ (\texttt{tail}\ x)$, and $\texttt{tail}\ x$, although having the right type, contain an unbound variable[5] so we cannot use them as either.

However, it also makes sense to perform simplifications inside of the term. For example, `b ++ b` may be replaced with `b`, since they have the same type, which gives:

$$(\lambda x.\texttt{id}\ (\texttt{tail}\ x))\ \texttt{b}$$

Similarly, we may make a replacement inside of the lambda expression:

$$(\lambda x.\texttt{tail}\ x)\ (\texttt{b ++ b})$$

Also, the whole lambda expression, which has type $\texttt{Int} \to \texttt{Int}$, may be replaced with its subterm:

$$\texttt{tail}\ (\texttt{b ++ b})$$

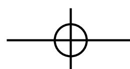
Another way in which we could simplify a term is by replacing its part with a constant of the right type. For example, if the environment contained constant `[]` (empty list), we could replace one of the list subterms with it:

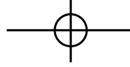
$$(\lambda x.\texttt{id []})\ (\texttt{b ++ b})$$

Finally, we can simplify a simply-typed lambda term by performing β-reduction on it, that is inline the function's argument into its body.

The formal properties of the simply-typed lambda calculus ensure that we can do this reliably as two important guarantees are made—that (a) the

[5]Terms with unbound variables are not well-typed and thus technically have no type.





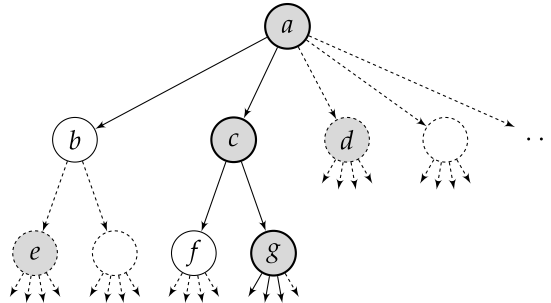


Figure 1.6: Shrinking process. Test cases that fail are marked in grey. Dotted test cases are not considered.

resulting term is well-typed and that (b) β-reductions always terminate. The second property is non-trivial as the size of a term might increase after a β-reduction.

The considered term contains one function applied to an argument, so it may be β-reduced in one way:

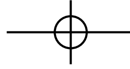
```
id (tail (b ++ b))
```

5.2 Shrinking using QuickCheck

The example above illustrates all ways of shrinking a term. This subsection presents a semi-formal description of the generic shrinking process performed by QuickCheck.

In each step of the shrinking process a number of shrinking candidates are tested, which are reduced versions of the original test case. When one of them fails, the shrinking step is concluded and the newly found counterexample becomes the starting point for the next shrinking step.

Figure 1.6 contains an illustration the shrinking process. Test case a is the original counterexample found during testing, which is subsequently shrunk. The first shrinking step considers a's shrinking candidates b, c, d, and so on, which are tested in turn. Test case b, which is considered first, succeeds and is discarded. Test case c fails and becomes the 'current' shrunk test case, whose shrinking candidates are tested in the next step. In this step g is found to falsify the property and the process continues with its shrinking candidates. Shrinking terminates when all current shrinking candidates succeed, or when a test case does not have any, and the last failing test case is reported.



Note that the failing test case *e* from Figure 1.6 is never considered, because shrinking looks only at immediate shrinking candidates of a failing test case. Also, test case *d* is not considered, because it occurs after the failing *c* in a sequence of shrinking candidates. Shrinking is a *greedy algorithm* that performs local choice, which is not guaranteed to be optimal. The resulting shrunk test case is only a *local minimum*. However, good choice of a specific shrinking method tends to give results that are not far away from optimal.

A shrinking method for a particular data type is defined as a function that maps each element of that data type to a list of shrinking candidates defined as the following Haskell function:

```
class Arbitrary a where
  ...
  shrink :: a -> [a]
```

The shrinking candidates should be smaller than the original element, but the measure by which they are smaller can be freely chosen by the developer of the shrinking method. The only formal requirements on the function are that (a) it is total, (b) that lists of candidates are finite and (c) there are no infinite chains of shrinking steps. These requirements ensure that the shrinking process will always terminate, either when all shrinking candidates succeed, or when the current term has an empty list of shrinking candidates.

5.3 Shrinking method for simply-typed lambda terms

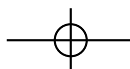
The particular shrinking method for the simply-typed lambda terms that works as described in the beginning of the section can be defined by performing 3 kinds of steps:

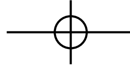
Rule 1. replace with subterms A subterm may be replaced with its proper subterm, if their types are equal. Care must be taken to avoid referring to variables bound by removed λ-bindings.

Rule 2. β-reduce A term may be β-reduced.

Rule 3. replace with constant Any subterm that is not a constant may be replaced with a constant if the types agree.

There is a possible optimisation that we could introduce in Rule 1—we may restrict it to only consider replacing subterms with their immediate proper subterms of the right type. For example, term `0 + (1 + 2)` may be





replaced by 1 using the unrestricted Rule 1, but not using the restricted one, because 1 is a proper subterm of 1 + 2 that can also be used.

The idea behind this restriction is that 1 will be tried anyway in a later shrinking step and that omitting unnecessary shrinking candidates prevents the list from being excessively long. This is important, for example, when shrinking the data type of binary trees where the total number of all subtrees might be rather large. Since shrinking has to respect types, we expect this optimisation to have a smaller effect in the case of simply-typed lambda terms, however we decided to take it into consideration. The effects of this restriction are investigated later in Section 5.8.

5.4 Properties of shrinking

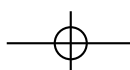
There are two important properties that must be established for our shrinking method: that (a) shrunk terms are well-formed and that (b) there can be no infinite sequence of shrinking steps.

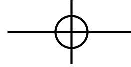
Shrinking produces well-typed terms It is easy to see that rules 1 and 3 turn well-typed terms into well-typed terms as their construction explicitly ensures that. For rule 2, which performs β-reduction, it is not trivial. However, a known result for the simply-typed lambda calculus called *subject reduction*[Pierce, 2002] states exactly that β-reduction is type-safe. Even though we allow polymorphic constants, our terms can still be modelled in the simply-typed lambda calculus as the constants are always fully instantiated. Thus, all three kinds of steps preserve well-typedness.

Shrinking terminates Again, it is easy to see that rules 1 and 3 are constructed in such way that they always make terms smaller, so they can never result in non-termination. Rule 2 is more tricky, as it may increase the size of the term due to duplication when the argument term is inlined into the function body. However, another standard result, strong normalisation, states that β-reduction terminates for the simply-typed lambda calculus[Pierce, 2002].

Unfortunately, even though no rule can cause non-termination by itself, it is not possible to extend this easily into proof that any combination of steps will always terminate. We expect that the proof can be extended, however the extra steps are non-trivial, and thus we leave it for future work.

Unfortunately, even though no rule can cause non-termination if used in isolation, it is not possible to extend this easily into proof that any combination of steps will always terminate. We expect that the proof can be extended, however the extra steps are non-trivial, and thus we leave it for future work.





5.5 Design choices of shrinking

The three ways of simplifying a term were chosen because they comprise generic ways of simplifying a term. Parts of a counterexample that are not relevant for triggering a failure might be removed by rules 1 and 3.

It is debatable whether rule 2 that inlines a function argument into its body is a simplification. In some cases the opposite, β-expansion, might lead to a simplified term. However, unrestricted β-expansion is non-terminating, whereas β-reduction terminates even when rules 1 and 3 are present, as we showed previously.

On top of that, the shrinking process turned out to be effective in practice as the shrunk counterexamples were not possible to be reduced further by hand in most cases.

5.6 Shrinking with batch test cases

In order to speed up testing, a single test case contains a list of terms that are tested together in one run, as described in Section 6. Usually 1000 terms are tested in one batch. Generating test cases that contain a list of terms is straightforward in QuickCheck, where it is enough to compose a generic generator of lists with a generator of terms. Similarly, a shrinking method can be derived from the generic one for lists. However, this solution has some shortcomings.

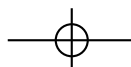
The generic shrinking method first tries to reduce the length of the list to one element using binary search, and then tries to shrink that element using its own shrinking function.

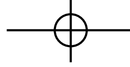
This method is inefficient in two ways. First, it does not use the information about which term in the list has failed and has to resort to binary search, adding about 10 unnecessary shrinking steps. And secondly, each shrinking candidate is compiled separately without taking advantage of compiling and running them in batches.

An optimised shrinking method would receive the index of the term that has failed the property, and generate a list of shrinking candidates of that term that could be compiled in a batch. Thus, it would avoid the initial search for the failing term and make shrinking that involves many failed attempts much faster.

It turned out that this can be implemented without changing the implementation of QuickCheck, although at a price of making the properties less composable. In order to create a property that uses batch test cases we propose using the following combinator.

```
forAllParShrink
  :: Int -- Number of test cases in one batch
```






```
-> Int -- Number of shrinking candidates in one batch
-> Gen a -- Test case generator
-> (a -> [a]) -- Shrinking function
-> (a -> String) -- Printing
-> ([a] -> Maybe Int) -- Batch property
-> Gen Prop
```

The batch property is passed as a function taking a list of test cases and returning a `Maybe Int` where `Nothing` indicates that all test cases succeeded, whereas `Just` n means that test case n was the lowest one that failed.

For each run of the property, the function creates a list of test cases whose length is specified by its first argument. When a failure occurs, the offending test case, which is pointed to by the index returned by the property, is passed to the shrinking function, which generates a list of shrinking candidates. A number of these test cases, limited by the function's second argument, is passed again as a batch to the property.

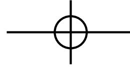
The batch property defined as a function of type `[a] -> Maybe Int` has a disadvantage when it comes to composability. Contrast it to the ordinary `forAll` combinator, which takes properties of type `Testable t => a -> t`. The property defined inside of `forAll` may itself use any of the QuickCheck property combinators, such as `collect`, `printTestCase` or `within`, whereas the one defined using `forAllParShrink` cannot, and must be defined monolithically.

On the other hand, the property inside of a `forAll` cannot give any extra information about the failure to the combinator, such as which test case failed in the batch, which is precisely why we have to use the type `[a] -> Maybe Int` for batch properties.

Shrinking using batch test cases implemented like this was sped up by factor of around 10–20 in typical cases, using batches that were limited to having 40 terms.

5.7 Weaknesses and possible improvements

Reducing the variety of constants The shrinking process is not able to replace a constant in a term with another one because of risk of nontermination. This leads to many similar terms being reported if a particular one is not required to trigger a failure. For example, one property generated



the following terms:

```
(-) a ((-) b 0)
(-) a ((-) b 1)
(-) a ((-) b 2)
(-) a ((+) b 0)
...
```

The counter-examples were shrunk to 12 variations, all using a different combination of constants. Given that shrinking should reduce, if possible, the number of reported counterexamples for a given bug, it would be beneficial if all these terms were *normalised* to a common representative. One way of doing that would be to establish an ordering on all constants and allow a 'larger' constant to be replaced by a 'smaller' one.

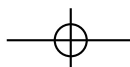
A slightly related idea would be to expand constants into their definitions. The expansion, obviously, has to be restricted as recursive definitions could be expanded forever. Such a rule might solve, for instance, the problem of transforming `1` into `0`, as `1` expanded to `(+1) 0` can be reduced to `0`. However, even this simple expansion is problematic because the whole `(+1) 0` can be transformed back to `1` using Rule 3 introducing the risk of non-termination. Also, expanding all non-recursive definitions might make the terms much larger. Thus, it is not clear how to restrict expansion of definitions so that it is useful for shrinking.

Retyping subterms Some cases would benefit from being able to replace a subterm with its proper subterm if it can be *retyped* to match the type of the term that it replaces. For example, if subterm $\lambda x.$`[]` has type `Bool` $\rightarrow$ `List Int` it could also be retyped with `Int` $\rightarrow$ `List Bool` and used to replace a larger term of this type.

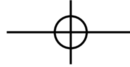
Small improvements Two potential improvements can be experimented with that would introduce only small changes to the method. Firstly, subterms could be replaced with variables that are in scope with the right type. And secondly, the shrinking candidates could be ordered in a more complicated way involving, for instance, interleaving of candidates generated by different rules.

5.8 Shrinking parameters

There are two possible choices in the shrinking method. The first one is the order in which the shrinking candidates generated by different rules



thesis March 27, 2014 10:04 Page 43



should be considered, which in some cases might influence the result of shrinking.

The second choice is whether to optimise shrinking by restricting Rule 1 to only consider replacing subterms with their immediate proper subterms of the right type, as described in Section 5.3.

To assess the effects of these choices on shrinking, we parametrised our shrinking method over these choices and conducted an experiment to measure its performance with different parameters. The experiment was performed by collecting pools of randomly generated unshrunk counterexamples for two properties, shrinking them with different combinations of parameters and examining their performance based on the results of shrinking and its speed. The two properties were the ones discussed in Sections 6.1 and 6.2 of the next chapter and each of the pools contained 200 terms. While it is certainly worth extending the experiment to cover more properties, or even other applications than just testing GHC, the data that we obtained should give us some idea about the influence of parameters.

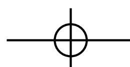
The results of the experiment suggest that restricting Rule 1 yields no negative effects on the quality of shrunk terms while improving the speed of shrinking by a small margin. The outcomes are less conclusive about which order of shrinking rules should be used. All orderings deliver comparable quality of shrunk terms, but differ in the numbers of shrinking steps and failed shrink attempts performed during shrinking. Although we can spot orderings that are clearly better than others, it is not possible to nominate the best one, if we assume that batch shrinking (described in Section 5.6) is used.

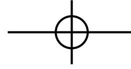
In this section we present the data that we gathered together with our analysis. We also propose recommendations based on that, but some of our conclusions are not definitive and for that reason we present our analysis in detail to allow drawing alternative conclusions from it.

Immediate subterms

First, the parameter defining whether to consider only largest applicable subterms was evaluated. Due to the fact that shrinking is a greedy process, it is impossible to predict the outcome of restricting the lists of shrinking candidates, but we expected that doing it might reduce the numbers of failed shrinking attempts. At the same time, it might reduce the effectiveness of shrinking.

The experiment showed that using this restriction has no effect on the results of shrinking whatsoever, as all the terms were shrunk in the same way regardless of it. The number of successful shrinking steps was also the same in all cases.





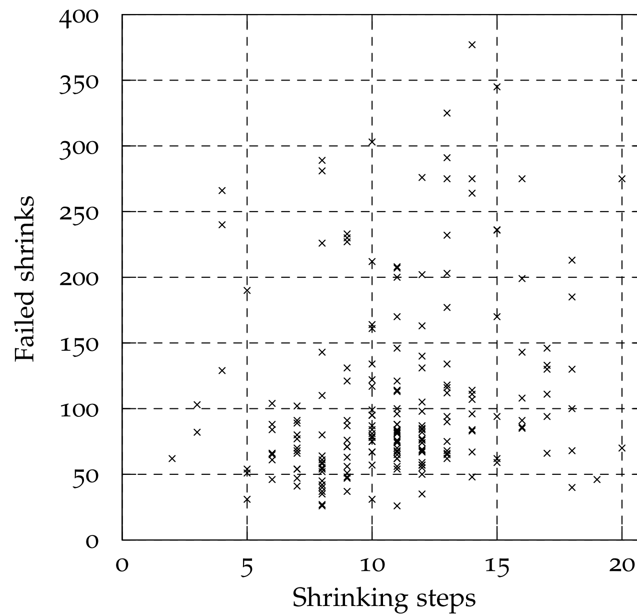
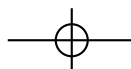


Figure 1.7: Number of shrinking steps and failed shrink attempts for different terms from a pool

However, the number of unsuccessful shrinking attempts for some terms was higher when all proper subterms were considered during shrinking. The differences were not large on average, with up to 2% more failed attempts in the first property when a specific ordering of shrinking rules was used (the other parameter of shrinking) and up to 7% more in the second one. The biggest discrepancy for any given term was 22% more failed attempts in the first property and 64% in the second one, however these were isolated cases.

There are typically 5–15 more failed shrinking attempts than successful steps for a given term. For example, Figure 1.7 shows a scatter-plot for respective numbers for the first of the considered properties where shrinking was performed with all subterms considered and the with the default ordering of shrinking rules.

It seems to be very beneficial to limit the number of failed shrink attempts since their number is often much larger than the number of



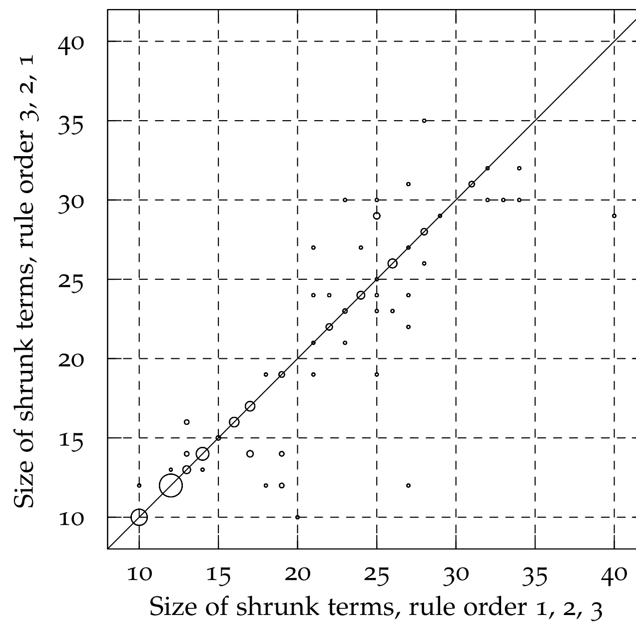


Figure 1.8: Sizes of shrunk terms for different terms from a pool; the diagrams are similar for all other pairs of rule orderings.

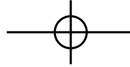
successful steps. However, the benefit seems less profound when we consider that using batch shrinking, described in Section 5.6, makes failed shrink attempts much cheaper, as in our case up to 40 shrinking candidates are tested in one batch.

This optimisation has little effect with testing the properties that we used in our experiment, but we can imagine a situation where terms contain many subterms of the same type (for example involving a binary tree data type), in which case the reduction in failed shrink attempts might be substantial.

Ordering of shrinking rules

The second parameter of the shrinking method is the order in which the three shrinking rules, mentioned in Section 5.3, are tried on a term.

We investigated the influence of the ordering of rules on the quality of



	Property 1		Property 2	
rule order.	shrinks	failed shr.	shrinks	failed shr.
1, 2, 3	10.95	108.43	7.89	61.61
1, 3, 2	10.92	137.92	8.41	119.13
2, 1, 3	12.64	107.93	11.35	52.02
2, 3, 1	7.33	194.85	5.88	155.09
3, 1, 2	6.92	267.45	5.90	310.94
3, 2, 1	6.90	257.39	5.94	290.38

Table 1.1: Shrinking steps and failed shrink attempts

shrunk terms. For any given two orderings of rules, the shrunk terms were the same in at least 61% of cases for Property 1 and in at least 76% of cases for Property 2. This makes us think that when we change the ordering of rules the results of shrinking remain quite consistent.

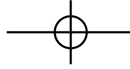
To compare different shrunk terms that come from the same original counterexample we decided to use the size of the final shrunk terms as the measure of their quality, and to naturally prefer smaller terms. Figure 1.8 shows a scatter-plot of sizes of shrunk terms that were generated by Property 1 and shrunk using two different rule orderings. The area of each circle in the plot is proportional to the number of data points that occurred in the same place.

We can infer by looking at the plot that no ordering of rules is better than the other when it comes to sizes of shrunk term. Plots for other pairs of orderings and for Property 2 look very similar to the one in Figure 1.8, which makes us conclude that the ordering of rules does not influence the quality of shrunk terms in a measurable way.

The other factor that we considered was the speed of shrinking. Instead of using wall clock time for benchmarking we used two numbers: (a) the number of shrinking steps performed during a single shrinking and (b) the number of failed shrink attempts. Table 1.1 shows the mean values of these numbers for different orderings of rules. In all cases we consider shrinking that uses the optimisation presented in Section 5.8.

Looking at Table 1.1 lets us isolate two groups of orderings. The first three generally result in more successful shrinking steps, but fewer failed attempts, whereas for the latter three it is the other way around.

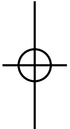
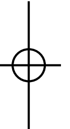
For a possible explanation of this phenomenon notice that in the first group Rule 1 (replacing a subterm with its proper subterm) always precedes Rule 3 (replacing a subterm with a constant), while in the second group the



opposite is the case. We suspect that Rule 3 generates very many shrinking candidates and many of them fail before a successful one is tested. Rule 1, on the other hand, if placed before Rule 3, might successfully reduce a term before shrinking candidates of Rule 3 are considered. Nevertheless, the lower numbers of shrinking steps in the second group might indicate that shrinking candidates generated by Rule 3 are more effective in quickly reducing the term's size.

Looking more closely at the rule orderings in the first group, we can see that ordering 1, 2, 3 seems to be better than 1, 3, 2, because the mean number of shrinks is either very similar (Property 1), or smaller in 1, 2, 3 (Property 2), and the mean number of failed attempts is always lower in 1, 2, 3. Orderings 1, 2, 3 and 2, 1, 3 are closely tied as the mean number of shrinks is lower in 1, 2, 3, but the mean number of failed attempts is better in 2, 1, 3 in Property 2 by a considerable margin.

In the second group 2, 3, 1 seems to be the winner thanks to lower numbers of failed attempts, even though its numbers of shrinking steps are a little higher.

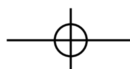



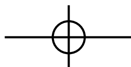
Having a choice between an ordering from the first group and one from the second one, which one should we choose? The numbers of failed shrinking attempts are usually much higher than the numbers of successful shrinks, which hints at the first group. However, given that when testing the GHC compiler we use batch shrinking, failed attempts are much cheaper than shrinking steps as up to 40 shrinking candidates are tested at the same time. With this assumption, all the orderings get much closer to each other in terms of performance. In contrast, when batch shrinking is not used, then the orderings from the first group are clearly better.

Figures 1.9 and 1.10 show scatter-plots of numbers of successful shrinks and failed shrink attempts for terms generated by Property 1, when using orderings 1, 2, 3 (first group) and 2, 3, 1 (second group). The scatter-plot of the numbers of failed shrink attempts is plotted using the logarithmic scale.

As we can see in Figure 1.9, ordering 2, 3, 1 generally results in fewer shrinking steps, the difference being more than 10 in some cases. On the other hand, ordering 1, 2, 3 brings about fewer failed shrink attempts, as shown in Figure 1.10. The discrepancies for some terms are as large as over 1000 failed shrink attempts for ordering 2, 3, 1 and less than 200 for the other ordering. Given that ordering 2, 3, 1 generates over 800 more failed shrink attempts for these terms, this corresponds to about 20 more batches used in shrinking.

Comparisons of other orderings from the two groups yield similar scatter-plots to the ones presented here, which prompts us to conclude that the first group results in fewer pathological cases when shrinking takes a





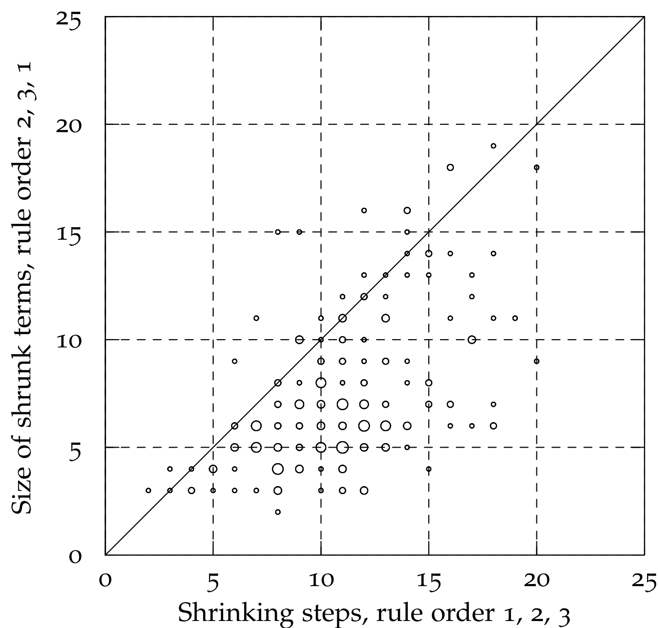


Figure 1.9: Numbers of shrinks for different terms from a pool

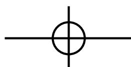
long time.

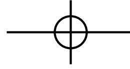
Conclusions about shrinking parameters

We conclude that restricting the shrinking candidates to terms where only largest applicable subterm can replace its superterm does not yield a sizeable speed-up. However, we also found that it has no detrimental effect on the quality of shrunk counterexamples, which is why we recommend that it is enabled by default.

The conclusions about which ordering of rules gives the best performance depends on whether we assume that batch shrinking is used. There is no measurable difference in the quality of the shrunk counterexamples when different orderings are used. What differed was the mean numbers of shrinking steps and failed shrink attempts.

The first three orderings shown in Table 1.1 performed fewer failed shrink attempts during shrinking, which made them appropriate for situa-





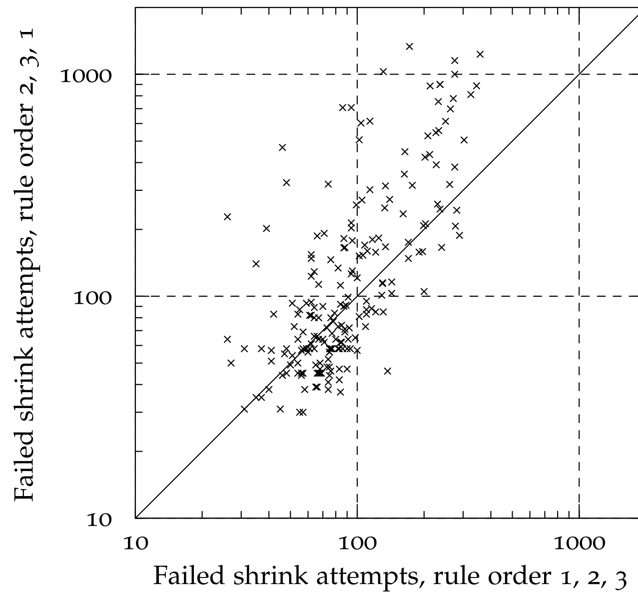
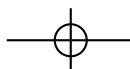


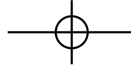
Figure 1.10: Numbers of failed shrink attempts for different terms from a pool

tions when batch shrinking is not used. Out of this group, orderings 1, 2, 3 and 2, 1, 3 performed particularly well.

When batch shrinking is used, with a maximum of 40 terms in one batch, as in our case, the differences in the average performance between the orderings is much smaller. In this case orderings 1, 2, 3 and 2, 1, 3 from the first group, and 2, 3, 1 from the second one could be considered. Based on the scatter-plot in Figure 1.10 we would still recommend the orderings from the first group to limit the amount of pathological cases. Choosing one of 1, 2, 3 and 2, 1, 3 over the other may impose a hit of 20% in the worst case, which is why we could arbitrarily choose the first one as a safe default.

Clearly, some of these recommendations are conditional and weak, but should serve well as the default values, whereas it should be possible to use alternative parameters whenever the user chooses to do so.





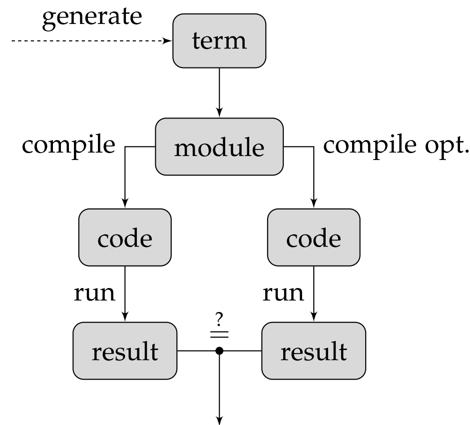


Figure 1.11: Differential testing

6 Applications

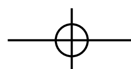
We used the generator to test the GHC compiler using *differential testing* [McKeeman, 1998] directed towards testing the compiler's middle-end. Even though the generator was capable of generating only a very limited subset of all Haskell programs, it was able to uncover interesting bugs in the compiler.

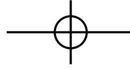
The testing resulted in finding eight interesting failures and four bugs were reported for GHC and subsequently fixed. The counterexamples obtained were concise enough to be understandable thanks to automatic shrinking.

Apart from bugs detected in the GHC compiler, the testing allowed us to understand better the effects of optimisation performed by it.

6.1 Strictness changed by optimisation

We investigate whether optimisation performed by GHC influences strictness of the compiled programs. Optimisation is performed on a program in order to turn it into a more 'optimal' one without changing its observable behaviour, where more 'optimal' might mean, for example, one that runs quicker. However, erroneous transformations might change the semantics of a program. In particular, changes in strictness, while subtle, can influence the program's observable behaviour—for example, the program may crash or consume more memory than it should.






```
module Main where
import Control.Monad (forM_)
import qualified Control.Exception as E
import System.IO (hSetBuffering, stdout,
  BufferMode (NoBuffering))

handler (E.ErrorCall s) = putStrLn $ "*** Exception: "

inputs :: [[Int]]
inputs = ...

code :: [Int] -> [Int]
code = ...

main = do
  hSetBuffering stdout NoBuffering
  forM_ inputs $ \x -> do
    E.catch (print $ code x) handler
```

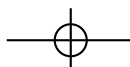
Figure 1.12: Module skeleton

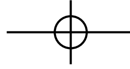
To detect whether the GHC's optimiser modifies strictness of some expressions we used differential testing where observed output of an optimised program is compared with observed output of the same program compiled with no optimisation, as shown in Figure 1.11. Changes in strictness of expressions might result in less or more output to be printed by tested functions before crashing.

A generated term whose strictness is to be analysed is compiled as part of a module whose skeleton is shown in Figure 1.12. The generated term is bound to the `code` variable and is a function of type `[Int] -> [Int]`.

The `main` function of the module is devised to print the results of the `code` function applied to a small number of partially-defined lists of integers defined by the constant `inputs`, which we elide here. Expression `print $ code x` prints the result of the function, and thus forces its evaluation.

When an error is encountered during the evaluation, it is caught as an exception by `E.catch`. The exception handler prints a message indicating an exception, but disregards its exact kind as we do not consider changing the exception kind as a change in program's semantics following the Haskell Report [Marlow, 2010]. The program is then allowed to continue to evaluate the function for all remaining inputs.





To provide accurate information on partial values potentially returned by the observed function, it is necessary to turn off output buffering, which is done by calling `hSetBuffering` in the `main` function. Had buffering been active, the printing code would try to print a whole line of characters at once and if an exception were triggered before the buffer is flushed, the output that had already been generated would be discarded. Therefore, we would not be able to distinguish between different partially-defined results. On the other hand, when buffering is turned off, characters are printed one by one and the exception handler will trigger only after the last defined character is printed. Even though a more accurate technique is available for discovering the semantics of partial values [Danielsson and Jansson, 2004], we chose this method as fast and accurate enough.

Each module created from the skeleton is compiled in two variants. The unoptimised variant is compiled without any optimisation options, which results in the default `-O0` 'zero' optimisation level. The optimised variant is compiled with `-O -fno-full-laziness`, which turns on typical optimisation options, but turns off the *full laziness* optimisation, which is also known as *let-floating*. The reason for leaving out full laziness is that it is known to change the strictness of compiled code and that examples involving it are less interesting.

The initial environment for terms contains simple integer constants and functions, like `0` and `+`, as well as common functions operating on lists from the Haskell prelude. The initial environment that is used in this subsection is presented in Figure 1.13.

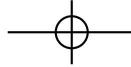
Given that we want to test the strictness of compiled functions, we must make sure that their arguments are not inlined, otherwise each function would be compiled and optimised many times together with each of its arguments. Fortunately, iterating through inputs using the `forM_` function is enough to stop GHC from inlining the arguments.

Due to high fixed cost associated with compilation and linking a module, 1000 terms are included in a single 'batch' module during testing, which requires a slightly different module skeleton and a suitable output comparison procedure.

Testing the property for a given generated term is done by (1) creating a module using the skeleton from Figure 1.12, (2) compiling it twice with different optimisation settings, (3) running the compiled modules and (4) comparing their output for equality. If the outputs are different then we have found a counterexample that has a different observable behaviour depending on the optimisation options. We can define the property in a more formal way as follows, where t is the term, `module` is a function that creates a module out of a term, and $\mathtt{c}_*$ and `run` are functions that,

```
seq    :: a -> b -> b
id     :: a -> a
[]     :: [a]
0      :: Int
1      :: Int
2      :: Int
(+)    :: Int -> Int -> Int
(+1)   :: Int -> Int
(-)    :: Int -> Int -> Int
(:)    :: a -> [a] -> [a]
enumFromTo :: Int -> Int -> [Int]
enumFromTo' :: Int -> Int -> [Int]
head   :: [a] -> a
tail   :: [a] -> [a]
take   :: Int -> [a] -> [a]
(!!)   :: [a] -> Int -> a
length :: [a] -> Int
filter :: (a -> Bool) -> [a] -> [a]
map    :: (a -> b) -> [a] -> [b]
null   :: [a] -> Bool
(++)   :: [a] -> [a] -> [a]
odd    :: Int -> Bool
even   :: Int -> Bool
(&&)   :: Bool -> Bool -> Bool
(||)   :: Bool -> Bool -> Bool
not    :: Bool -> Bool
True   :: Bool
False  :: Bool
foldr  :: (a -> b -> b) -> b -> [a] -> b
(==)   :: Int -> Int -> Bool
(==)   :: Bool -> Bool -> Bool
(==)   :: [Int] -> [Int] -> Bool
case1  :: (a -> [a] -> b) -> b -> [a] -> b
undefined :: a
```

Figure 1.13: Initial environment. Many instances of `(==)` are needed because our generator does not support Haskell type classes. The constant `enumFromTo'` is our own definition where the second argument is the length of the enumeration. The constant `case1` is another definition that performs case analysis on lists.



respectively, compile and run the module yielding its output:

$$\forall t.\mathsf{run}(\mathsf{c}_{\mathsf{opt}}(\mathsf{module}(t))) = \mathsf{run}(\mathsf{c}_{\mathsf{noopt}}(\mathsf{module}(t)))$$

Results

The property described above led to observing a discrepancy between optimised and unoptimised code for about one in 10000 terms, which takes about 3 minutes of CPU time when terms are tested in batches.

One failure for each 10000 tests is an unusually low number for a QuickCheck property, and we might speculate why this number is so low. One likely reason is the very high number of all expressions even among terms of small size. It is possible that failing test cases comprise a small number of all expressions. Another possible reason is an imperfect distribution of our generator, which may increase the probability of generating the same terms many times.

All counterexamples that we present here have been shrunk by the shrinking process, which means that any further shrinking results in a test case that does not trigger a failure.

Failure 1 The following snippet is an example expression that violated the property.

```
foldr (\a -> seq) id ((:) 0 (undefined::[Int]))
```

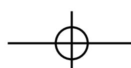
For clarity, it can be rewritten as follows by η-expanding one of the subexpressions, while keeping the same behaviour:

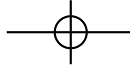
```
foldr (\a b c -> seq b c) id (0 : undefined :: [Int])
```

The expression exhibits different observable behaviour depending on compilation options, which indicates that at least one of the versions is compiled wrongly. However, to find out which one we must determine by hand what is its correct semantics.

The intended semantics of the higher-order function `foldr` is to return its first argument applied to the first element of the list (variable `a` gets bound to `0`) and to the result of `foldr`'s recursive call (`b` gets bound to the result of the recursive call). Function `foldr` is defined as a two-argument function, but is applied to three arguments in its context.

The whole expression should reduce to `\c -> seq (foldr ...) c`, which acts as the identity function except when expression `foldr ...` crashes, in which case the whole expression should also crash when applied to an argument. The expression should in fact crash as the recursive `foldr` is applied to the `undefined` list.





We were able to construct a simple program that demonstrates the incorrect compilation, shown below, which is simpler than the original module and more suitable for submitting a bug report.

```
main = print $
  wrap
   (foldr (\a b c -> seq b c) id (0 : undefined::[Int]))
   [0]
```

The tested expression is passed as argument to `wrap`, which acts as the identity function and an example list `[0]` is applied to the resulting expression. The purpose of `wrap` is to prevent the expression from being simplified together with its argument, which is achieved by implementing `wrap` in such way that GHC cannot reduce it[6].

```
wrap :: a -> a
wrap x = [x]!!0
```

When the program is compiled using GHC without optimisation, it prints

```
program: Prelude.undefined
```

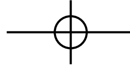
which indicates that the expected exception has been raised. However, with optimisation turned on, the expression gets incorrectly compiled into the identity function and `[0]` is printed instead. A likely explanation for this is that `seq` is somehow omitted from the generated code.

The demonstrated change in observable behaviour caused by optimisation violates the semantics defined by the Haskell Report. To assess the seriousness of this bug we will analyse its possible consequences.

The bug might cause a crashing function to successfully return a result. This does not seem dangerous, but can have two implications. First, it is a surprise factor for the programmer, which may hinder the understanding of the code. And furthermore it may invalidate a partial correctness arguments about a program, which state that *Program has property P if it terminates*. If the program's overall correctness relies on such partial correctness argument, the programmer might be mistakenly convinced about its correctness.

A more common manifestation of the bug might be, however, a space leak caused by an omitted `seq` application. Common wisdom about Haskell programs compiled using GHC is that optimisation occasionally reduces their performance, which often happens by introducing space leaks. This and similar bugs might have a role in causing the performance regressions.

[6]Using the builtin function of GHC named `GHC.Exts.lazy` has the same effect.



Failure 2 Another counterexample yielded by the same property is this expression.

```
seq (seq (head []) (\a -> undefined))
```

Identifying incorrect compilation again requires analysing the expression's semantics. The nested expression containing `seq` should evaluate to an exception, since its first argument is `head` of an empty list. Therefore the whole expression should also be a function that crashes. To demonstrate the error we can use the same skeleton program as with the previous term.

```
main = print $
  wrap (seq (seq (head []) (\a -> undefined))) [0]
```

Compiling the program with no optimisation yields the correct result.

```
program: Prelude.head: empty list
```

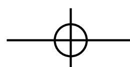
However, when compiled with optimisation, the expression behaves as the identity function and yields `[0]`. Thus, the semantics in the optimised version is changed to more lazy, which is a similar to the problem triggered by the expression in Failure 1.

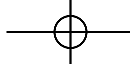
The fact that the counterexample that we are considering has been shrunk allows us to draw some conclusions about the bug that has been triggered. From the way the shrinking is performed, we know that neither of the shrinking candidates generated from the reported term failed the property. This means that if we simplify the counterexample in a structural way it will no longer be a counterexample.

For example, if we replace the expression `head []` with `undefined` the term will be compiled correctly, or at least testing will not detect any error. Note that these two terms ought to have the same semantics, nevertheless using `head []` is required to trigger a bug. We might speculate that `undefined` is correctly identified by the compiler as a crashing expression, while `head []` is not, which makes the compiler perform different transformations each time.

Another term that would look like a good candidate for replacement is `\a -> undefined` and again replacing it with `undefined` makes the failure go away. If treated as functions, both terms behave as undefined functions, but it is possible to differentiate between them by executing `seq` on them. One hypothesis might be that the compiler at some point assumes that `\a -> undefined` is equal to `undefined` 'for all practical purposes', but the distinction between them turns out to be relevant in this context.

This speculation might be further reinforced by looking at some other counterexamples for the same property, presented below.






```
seq (seq ((!!) ([]::[] Bool) 0) (\a -> (undefined::Int)))
seq (seq ((!!) ([]::[] Bool) 0) (\a -> (undefined::Int)))
seq (seq ((+1) (undefined::Int)) (\a -> (undefined::Int)))
seq (seq (even (undefined::Int)) (\a -> (undefined::Int)))
seq (seq ((+) (undefined::Int) 0) (\a -> (undefined::Bool)))
```

All of them are structured similarly to the original one, having a crashing expression (but not `error ...` or `undefined`) as the argument of the nested `seq`, and `\a -> undefined` as the second argument of the other `seq`. This may suggest which combination of features is needed to trigger a failure and which details are not relevant in an expression.

Apart from the expressions shown above, several other groups of expressions could be distinguished among the ones reported by the property. It is impossible to say, based on testing, if each of the groups is caused by a distinct bug, or whether there is one bug that causes all the failures. By looking at bug fixes that were added to the GHC we deduced that, for example, this failure was probably caused by a different bug than Failure 1.

Failure 3 Counterexamples presented in previous examples were terms whose behaviour was always more lazy in the optimised version of the program, and indeed all found terms that were scrutinised by us during testing had this characteristic. We decided to check if it is possible to find a term that is more strict when it is compiled with optimisation.

For this we modified the function that compares outputs of two variants of a program compiled with different optimisation levels to signal failure only when the optimised version prints *less* output, indicating that it is more strict. As expected, terms like this were much harder to come across, but still possible to find at a rate about 100 times lower than the previous kind.

The following term was found by the modified property.

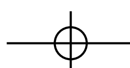
```
(\a -> seq a (seq (a []) id)) (\a -> seq undefined (+1))
```

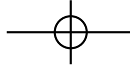
For clarity we might rewrite it as follows:

```
let a = \x -> seq undefined (+1) in a ‘seq‘ a [] ‘seq‘ id
```

If we consider its semantics: variable `a` is bound to a function whose semantics is equivalent to that of `\x -> undefined`, which means that `seq a` x is defined, but `seq (a` y`)` x is undefined for any defined x and y.

Given that the considered expression performs `seq` both on `a` and on `a []` its result should be undefined and the correct outcome of a program that evaluates this expression should be a raised exception. Strangely





enough, compiling the expression with no optimisation gives a program that prints `[0]` (when the expression is applied to `[0]`) instead of crashing and with optimisation turned on the result is correct.

It seems, thus, that our working hypothesis that unoptimised programs are correct and should be treated as reference for testing is not always true as we have just found a program that gets fixed by compiling with optimisation.

This very unexpected bug was reported as ticket 5625[7] in the GHC bug tracker and was subsequently fixed.

As previously, we may look at other reported counterexamples to determine what features of the expression are relevant to causing failures. The counterexamples are shown below, each spanning two lines.

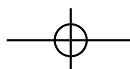
```
(\a -> seq (a (seq a (undefined::[] Int))) id)
    (\a -> seq (undefined::[] Int) (+1))
(\a -> seq a (seq (a (undefined::[] Int)) tail))
    (\a -> seq (undefined::[] Int) (+1))
(\a -> seq a (seq (a ([]::[] (Int -> Int))) (:)))
    (\a -> seq (undefined::[] Bool) (+1)) 0
(\a -> seq (a (seq a (undefined::[] Int))) id)
    (\a -> seq (undefined::Bool) (\b -> head))
(\a -> seq (a ([]::[] Int) (undefined::Int)) (seq a))
    (\a -> \b -> seq (undefined::Int) (+1)
```

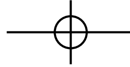
The common features visible in these expressions are (a) subexpression `\x -> seq undefined` *e* (slightly modified in the last example) that is bound to the variable `a` of the first function, and (b) that variable `a` is used twice in the function. Given the expected semantics of the subexpression, it may also be important that `a` is applied to an argument at least once in each counterexample.

It is worth noting that it is actually possible to reduce this counterexample further by hand. It appears that the bug is triggered only if `a` is not inlined and the reported term contains two occurrences of `a`, which prevent it from being inlined. But if we move out `a` to become a top-level definition and export it from the module then only one occurrence of `a` is needed. The module demonstrating the bug becomes then:

```
module Main (a, main) where

a = \x -> seq undefined (+1)
```

[7]Available at `http://hackage.haskell.org/trac/ghc/ticket/5625`. All GHC tickets can be accessed in this manner by altering the ticket number.






```
main = do
  print $ (a [] ‘seq‘ id) [0]
```

Were the bugs fixed?

In the three examples presented above, we demonstrated that GHC may subtly change the semantics of expressions by changing their strictness. The compiled expressions were too lazy, which in two cases was caused by optimisation, while in one case optimisation unexpectedly fixed the error.

The counterexamples reduced by the shrinking process were concise enough for us to initially analyse and understand the failures. Even though we have no expertise about the internals of the GHC compiler, we were able to make educated guesses about the failures based on the counterexamples. Furthermore, the found test cases were of high enough quality that they could be used in bug reports. Out of the three presented expressions, one was used to submit a bug report for GHC.

The bugs causing all three failures have been fixed. The term in Failure 1 is no longer miscompiled by GHC 7.3.20111127 when option `-fpedantic-bottoms` is used. The option was introduced as a fix for another bug that we reported (ticket 5587). Therefore, we conclude that the fix affects also this failure.

Failure 2 has been fixed before GHC version 7.3.20111022. However, we cannot identify a specific bug that is relevant. The bug concerning Failure 3 has been fixed before GHC version 7.3.20111127 through ticket 5625.

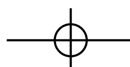
Failure 4 Out of the hundreds of reported counterexamples, those that were scrutinised by us always contained `seq`s. This is not surprising as `seq` is a construction that is a difficult case for the compiler. However, `grep`ping through the counterexamples revealed that there are several that do not involve `seq`, all rather similar.

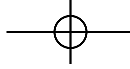
Here is the simplest of such terms that has been found:

```
\a -> foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0
```

The correct semantics of this function applied to a defined list is to concatenate the list with itself and then execute a fold on it. The fold ignores the elements of the list, but threads through the initial value of the fold. The initial value, which is `undefined ()`, is returned as the result of the fold and is applied to `0`, so the whole expression should crash with an exception.

Compiling the code with no optimisation yields correct results, but the following program was found to demonstrate incorrect compilation:






```
f :: [Int] -> [Int]
f a = foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0

main = print $ f []
```

When compiled with no optimisation it correctly prints

```
program: Prelude.undefined
```

However, when optimisation is turned on, the program does not crash, and does not print any output. This is surprising, as we would expect that the program would either crash or print a list of integers, which is the type of values returned by `f`.

To investigate how this happens, we looked at the intermediate Core representation of the program, which revealed that the code that is supposed to print the resulting list is missing. The only sensible reason why a compiler would omit this code is that it expects the function to crash before returning the value. This seems a likely guess, as GHC strictness analysis marks `f` as a function returning bottom. To confirm our findings, we decided to fool the GHC's strictness analyser using the `wrap` function.

```
f :: [Int] -> [Int]
f a = foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0

main = print $ wrap $ f []
```

When the result of `f` is passed through `wrap`, the printing code is included and a (somewhat less) incorrect result gets printed.

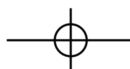
```
[]
```

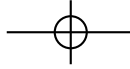
But unlike previous examples, where expressions were lazier as a result of some `seq`s not being executed, here they do not occur and the returned value seems arbitrary. Indeed, it turned out that the body of `f` is polymorphic in the result type and we can make it return an integer, as follows:

```
f :: [Int] -> Int
f a = foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0

main = print $ wrap $ f []
```

This program, when optimised, prints:

```
1099511628032
```





What is more, when we choose `(Int, Int)` as the result type, the program dies with a segmentation fault.

Earlier in this section, we discussed the situation where a partial correctness argument might be invalidated if some expression in a program is evaluated successfully by mistake instead of crashing. This example suggests that GHC itself can fall into this trap. If GHC's analysis concludes that `f []` will crash, then the compiler is 'careless' about handling its result, because it believes that the result will never be returned. Due to an error in optimisation, the result is nevertheless returned, which may lead to the effects that we demonstrated. Thus, failure to throw an exception may lead to worse consequences than space leaks or too lazy expressions.

The bug was reported as ticket 5626 and fixed as a result of another bug report.

6.2 Optimisation influencing the evaluation order

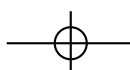
We decided to investigate whether optimisation changes the order of evaluation of expressions, which is interesting for two reasons. First, it is interesting to see which changes actually take place to get an understanding of optimisations performed by GHC; and secondly, to relate that to the changes that GHC is allowed to make, as it is possible that the evaluation order is changed in an invalid way.

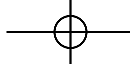
The Haskell programming language is designed with the intention that the evaluation order should not matter for programs. Specifically, the evaluation order is not observable, unless the program uses unsafe programming constructs, and the compiler is free to use any evaluation order as long as the non-strict semantics is preserved [Marlow, 2010].

Still, the evaluation order is important for reasons of efficiency, which is why GHC gives certain guarantees about it to make it possible to write efficient programs. Essentially, the evaluation order used by GHC is determined by the call-by-need evaluation strategy, with the exception that in some cases the order might be chosen freely by the compiler [Peyton Jones et al., 1999].

A Haskell program may exhibit different levels of space usage depending on what is the exact order of evaluation [Gustavsson and Sands, 2001], but efficient programs rely on the additional guarantees provided by GHC for predictable space behaviour.

Determining the evaluation order is impossible in general, since it is not observable as far as pure computations are concerned, but we can use a trick to observe it. To detect which of two subexpressions of an expression is evaluated first, we can make use of catching exceptions. Not unexpectedly,





catching exceptions is an impure operation, which is why it lets us observe the evaluation order.

Consider expression e that contains two subexpressions a and b.

$$e = \ldots a \ldots b \ldots$$

We can replace the two subexpressions with error terms as follows:

$$e' = \ldots\texttt{error "aaa"}\ldots\texttt{error "bbb"}\ldots$$

If evaluation of the term requires both subterms to be evaluated, one of the exceptions will be thrown and precisely which one gets thrown depends on their relative evaluation order. It is reasonable to expect that the error term that is evaluated first will yield an exception.

Replacing a subexpression with the `error` x expressions carries the risk that the presence of undefined subterms will affect the optimisations performed by the compiler on the expression. Thus, instead of placing the `error` x terms inside of the expression it is better to create a function that is later applied to suitable error terms and make sure its arguments are not inlined.

$$e'' = \lambda ab. \ldots a \ldots b \ldots$$

$$e'' \; (\texttt{error "aa"}) \; (\texttt{error "bb"})$$

Testing was performed using the same approach as with the previous property, that is by comparing outputs of the same module compiled with different optimisation options. The module skeleton was slightly different as this time we were interested in discriminating between different exception kinds. Therefore, the exception handler also prints the exception string:

```
handler (E.ErrorCall s) = putStrLn $ "*** Exception: " ++ s
```

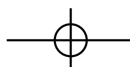
The modules were again compiled with no optimisation and with `-O -fno-full-laziness` options. The criterion for selecting offending terms was that a different exception from the two passed as arguments is thrown by each variant of the module.

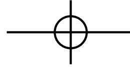
Results

Failure 5 Terms that exhibit the above behaviour turned out to be quite common. One of the simplest terms found is the following:

```
\aa bb -> seq aa (seq bb aa)
```

We can rewrite it using the infix notation for `seq` for clarity.






```
\aa bb -> aa ‘seq‘ bb ‘seq‘ aa
```

The term represents a function that forces the evaluation of both of its arguments and returns the first one. When applied to two error terms the unoptimised and optimised versions produced different results. The unoptimised one printed the exception thrown by the first error term:

```
*** Exception: aa
```

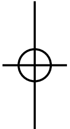
The optimised threw the other exception, indicating that `bb` is evaluated first.

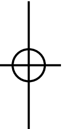
```
*** Exception: bb
```

Inspecting the GHC's internal Core representation of the optimised program reveals that the code of the function was transformed by the compiler into `\aa bb -> bb ‘seq‘ aa`, omitting the `seq` operation on `aa` altogether.

This result surprised us at first, as `seq` is often used to eliminate space leaks by making sure that its first argument is evaluated before the second one. For example, the definition of the standard library function `foldl'` relies on performing `seq` to avoid a space leak.

To answer whether omitting this `seq` is acceptable, we consulted the Haskell Report [Marlow, 2010], which provides the following definition of `seq`:



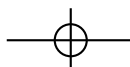


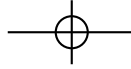
```
seq bot b = bot
seq a   b = b if a /= bot
```

This definition is a semantic one and ensures that `seq` is strict in its first argument, however it says nothing about the evaluation order. A naïve implementation of `seq` would simply force the first argument before returning the second one, and this would result in the behaviour which programmers seem to rely on. However, if evaluating the second argument of `seq` forces the evaluation of the first one by itself, the compiler might omit the `seq` operation without violating the semantics. This is indeed the case in the discussed example, as returning `aa` means that it will be forced by the caller of the function, so forcing it beforehand is not necessary.

The lesson from this example is that `seq`, which is implemented correctly by the compiler, *might still not guarantee that its first argument will be evaluated before the second!* Unfortunately, many of the Haskell programs and libraries rely on this guarantee to avoid space leaks, such as the above mentioned `foldl'` function.

This fact, while neglected by many, has been known before. For example, as outlined in [Marlow et al., 2009], it is apparent that an operation that





forces a specific evaluation order is needed for implementing efficient parallel computations in Haskell. However, restricting `seq` to allow only a specific evaluation order might eliminate some optimisation opportunities, which is why another variant of it called `pseq` was introduced. The new construction has the same semantics as `seq`, but is guaranteed to evaluate its first argument before its second.

Thus, consistently with [Marlow et al., 2009] and the Haskell Report, we would make the following recommendation.

- Whenever strict ordering of evaluation of expressions is needed, `pseq` should be used.
- The `seq` operator should be used to change the strictness of expressions, but not to enforce the order of evaluation.

Failure 6 Here is another term reported using the property.

```
\aa bb -> (+) (length (take bb ([]::[] Bool))) aa
```

The term performs integer addition of an expression that depends on arguments `bb` and `aa`. At first it seems that `bb` should always be evaluated first, since `+` should evaluate its first argument before the second one. However, the `+` operation is one of the cases where GHC is allowed to alter the evaluation order for reasons of efficiency [Peyton Jones et al., 1999]. Thus, the change of evaluation order in this case is also legitimate.

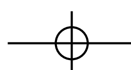
Many more terms involving `seq`, `+` and functions involving `+` were reported by the property. Unfortunately, only manual inspection of them was able to establish whether the changes in the order of evaluation were allowed, which means that we checked only a small number.

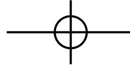
To perform more effective testing of this issue, a more accurate property would have to be constructed. However, it would require precise modelling of the Haskell semantics, which is an ambitious task by itself. It is not clear if a truly 'differential' technique that does not rely on a complex oracle could be applied here.

6.3 Equivalence of inlined and non-inlined expressions

Another property that we constructed compares the observable behaviour of a `let` expression and its reduced form, which we can portray with the following equation.

$$\texttt{let x} = e \ \texttt{in} \ C[\texttt{x}] \ \approx_{\textsf{obs}} \ C[e]$$





Notation $C[t]$ denotes an expression with zero or more gaps that are filled with occurrences of expression t. The two expressions should behave in the same way if no impure language constructs are used by them.

The initial goal of this effort was to reveal impure behaviour of some library functions. We failed to reach it. However, an interesting compiler bug was found in the process.

The property is implemented in a similar way as previous ones. However, only one module is created that contains both variants of expressions that are compared. For each test case two terms are generated, one representing the expression e and one representing the context $C[\bullet]$. The second term is then used to create expressions $C[\mathtt{x}]$ and $C[e]$.

To disregard the effects of possibly changed evaluation order the property treats all thrown exception types as equal, as was the case in our first property.

Failure 7 The following expression was found to behave differently than its reduced form when compiled with GHC:

```
let x = error "aaa" in seq (seq (tail ([]::[Int])) (\a -> x))
```

When we run this expression as part of the program below, it yields the correct result, that is it crashes and prints the following exception: `Prelude.tail: empty list`.

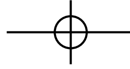
```
print $
  wrap
    (let x = error "aaa" in
       seq (seq (tail ([]::[Int])) (\a -> x)))
    [0]
```

However, if we replace the generated expression in this program with its second variant, shown below, the program prints `0` instead.

```
seq (seq (tail ([]::[Int])) (\a -> error "aaa"))
```

It is interesting that the program gets miscompiled regardless of the optimisation level used and even unoptimised compilation yields the same erroneous results. Thus, this failure could not be detected using our previous approach of using differential testing with different optimisation levels.

The approach of differential testing using pairs of expressions like the one in this example is inspired by traditional property-based testing where simple logical properties, such as equality laws, are tested. This gives



us additional bug-finding power, which is not possible with traditional differential testing when only one compiler, the one under test, is available.

And as in any programming language, in Haskell there are many possible schemes of generating equivalent expressions. The failure that we found suggests one more such scheme, apart from a `let` expression and its reduced form. As it turns out, if we replace `error "aaa"` with `undefined` in the offending expression, as shown below, it is again compiled correctly.

```
seq (seq (tail ([]::[Int])) (\a -> undefined))
```

Thus, one possible scheme is pairs of expressions where occurrences of `error "aaa"` are replaced with `undefined`.

The failure has been reported as GHC ticket 5557 and fixed before version 7.3.20111022.

6.4 Equivalence of different crashing expressions

The bug that we found using the property described above led us to investigate the following property.

$$C[\texttt{error "aaa"}] \approx_{\mathsf{obs}} C[\texttt{undefined}]$$

This property is implemented similarly to the previous one, but requires generating only one term that represents $C[\bullet]$.

As expected, testing this property yielded the same, and similar, failures as the previous one, and unfortunately no fresh failures.

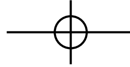
Thus, we decided to modify the property and tested the following one instead.

$$C[\texttt{hiddenError}] \approx_{\mathsf{obs}} C[\texttt{undefined}]$$

Here `hiddenError` is an expression that crashes, but is defined in such a way that makes it impossible for GHC to determine its semantics at compile time. We hoped that using an expression with a concealed crash might make GHC assume that it will not crash and use a transformation that is only valid for non-crashing expressions on it.

Failure 8 Indeed, testing found interesting terms that were miscompiled, one of which is presented below.

```
hiddenError = error "hidden error"
main = do
  print $ seq
          (head (map (\a -> \b -> hiddenError)
                     (hiddenError::[] Bool)))
```




```
id
[1]
```

When this program is compiled with optimisation using GHCwith the -O -fno-full-laziness options, it prints [1] instead of crashing. As visible in the program, it is actually enough that hiddenError is just a plain definition using error ..., without further obfuscation. However, using error ... directly does not trigger the bug.

The failure has been reported as GHC ticket 5587, which now contains a comprehensive explanation of its causes. The offending expression, which contains a head function applied to map, undergoes complex transformations using rewrite rules [Peyton Jones et al., 2001] thanks to list fusion [Gill et al., 1993]. This process leads to a subexpression that is a case expression whose one branch crashes, while the other one is a function that expects one more argument. For reasons of performance, this subexpression gets η-expanded, so that it crashes only when the extra argument is applied, changing its behaviour in some contexts.

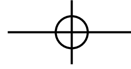
This particular counterexample has an interesting feature, in that the code that triggered the bug was created indirectly by GHC by transforming the original expression using inlining and rewrite rules. The resulting expression had elements that were not possible to generate using our generator, but were needed to cause the failure, such as case-expressions. Thus, given the multi-pass operation of the GHC optimiser, it is possible to subject the compiler to a wider variety of expressions than the ones that are possible to generate directly.

It is also worth noting that even though the shrinking process only tries to reduce terms structurally, it shrank this counterexample very well. The sequence of rewrite steps performed during the optimisation process reveals that the counterexample is in fact minimal.

The bug has been fixed in GHCby introducing a new compiler option -fpedantic-bottoms. Using this option causes the compiler to omit the erroneous transformation. However, the default is still to perform it. The motivation for this is that increased performance might be attained at a price of changes in the semantics. However, it is not known at this point what is the performance penalty of -fpedantic-bottoms.

6.5 Summary

Testing the GHC compiler using randomly generated simply-typed lambda terms and differential testing proved to be effective, even when only a small fragment of the Haskell language is covered. We found many interesting failures, eight of which we presented here. Four of these counterexamples



resulted in high-quality bug reports, which were acknowledged by the GHC developers as relevant.

We managed to obtain succinct counterexamples without understanding the inner workings of GHC, only by using shrinking, which automatically reduces a failing test case to a smaller one. Shrinking usually resulted in test cases that were not possible to be shrunk further by hand. Shrunk counterexamples for a single property are often similar and looking at their similarities allows for making educated guesses about the cause of the failures.

The fact that the test cases have been reduced by shrinking proved to be of enormous help. In one case it was possible to find a well-hidden error in one of the GHC's phases of compilation in a matter of minutes, something that would not be possible if the reported test case was large.

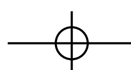
All the reported bugs concerned strictness of expressions being incorrectly changed in the process of optimisation. We also investigated whether optimisation might result in changes in the evaluation order of expressions. However, we were not able to detect any bugs there. What we discovered, instead, was that in many cases the order of evaluation is unspecified, such as when the operator `seq` is used, which is permitted by the Haskell Report. On the other hand, programmers often rely on `seq` having a determined evaluation order, which may cause their programs to exhibit space leaks if an alternative order is used. Thus, in addition to finding bugs in the compiler, our testing method can be used to support or disprove hypotheses about the compiled programs, which may help in understanding of the compiler.

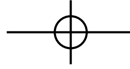
We used differential testing using one compiler implementation, but comparing behaviour of programs compiled using different optimisation levels. In addition to that, we used another form of differential testing where two programs are compared, which are different but have equivalent semantics. The second approach led to discovering failures that were not possible to discover using the first one.

7 Related Work

7.1 Compiler test tools

CSmith [Yang et al., 2011] is a random C program generator aimed at testing compilers. It attempts to generate C programs that avoid undefined or unspecified behaviour [ISO, 1999] without compromising the expressiveness of the generated programs. To achieve this goal CSmith employs a relatively complex program generator that uses different techniques for producing safe programs. Firstly, it avoids some unsafe behaviours simply by intro-

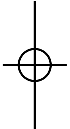


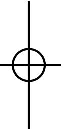


ducing structural constraints. And secondly, for cases where this would be too restrictive, the generator resorts to performing static analysis on already generated code fragments to determine whether a given operation is safe, or by inserting runtime safety checks in the generated code.

Evaluation of test results is done using differential testing with different compilers, or different options to the same compiler. The comparison of effects of two executions is performed by comparing checksums of non-pointer global variables sampled at the end of each execution. A variety of compilers were tested, including GCC, LLVM, CompCert and commercial C compilers. CSmith was able to uncover as many as 325 previously unreported bugs in all compilers altogether, most of them in GCC and LLVM. Even CompCert, which has a formally verified core, exhibited a number of bugs.

CSmith has no means of reducing the size of a failing test case, as it would be difficult to ensure that a shrunk test case is also free from undefined or unspecified behaviour. Programs containing 8k–16k tokens gave the highest rate of triggering bugs, and reducing them by hand was employed to obtain understandable test cases.

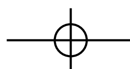


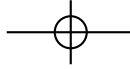


Lindig [2005] created a simple tool called Quest for testing the C function calling convention of C compilers. This tool randomly generates programs containing C functions that execute consistency checks to verify that their arguments have been passed correctly. Program generation is *type-driven*, that is the type of a function is first picked at random and a suitable body is generated algorithmically. Although Lindig claims that his method does not require a language specification, he relies on a partial specification stipulating that the consistency checks should succeed. The scheme was able to detect bugs related to passing function arguments in 5 different compilers. Bugs found by Quest were triggered by surprisingly simple code, which is explained by the fact that the static test suites used to by compiler writers contain very few kinds of argument and result types of functions.

McKeeman [1998] presents a case of differential testing of C compilers using inputs of various *quality levels*. Starting with sequences of any ASCII characters, which have the lowest quality level, the inputs range through valid sequences of tokens and syntactically correct programs to reach programs with well-defined semantics. This led to successful finding of errors in different stages of the compilers tested. Additionally, starting with a test case from any level, 'nearby' test cases are created by introducing small changes to the original test case, which often causes a tested compiler to crash, uncovering a bug.

The test case generator was implemented as a Tcl script, which is based





on a context-free-grammar-based generator enhanced to support context-sensitive features, like tracking defined variables. Grammar rules are weighted and termination is ensured by assigning small enough weights to recursive rules.

If a failing test case is found, a shrinking process is applied to reduce its size. Failing test cases can be as big as 600 lines of code and can often be shrunk to just several lines of code. However, this might require about 10000 compilations.

Instead of avoiding illegal operations at higher quality levels, whenever there is a discrepancy in the behaviour of two compiled versions, the program is rerun with all potentially problematic operations replaced by their error-checking variants. If an error is detected, the test case is discarded.

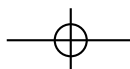
The highest level of *quality* that can be generated comprises of programs with meaningful semantics. Programs of this level are generated from specific templates that define their high-level structure, which guarantees certain semantic properties. Of course the diversity of the generated programs is traded here for semantic correctness, as the programs are much more specific than those from lower quality levels.

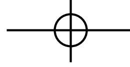
Our work does not have the breadth of CSmith or the McKeeman's tool, as we cover a much smaller part of the language that we generate. However, in that part we are able to generate very interesting programs, thanks to using a formal calculus that ensures well-typedness. We also had to solve problems that are absent while generating C programs, such as generating higher-order and curried functions and parametric polymorphism. Like McKeeman's work, our testing tool shrinks counterexamples, but does it in a type-safe way that guarantees to preserve typing and makes it much more efficient by using batch testing.

Hanford [1970] presented an early example of a recursive, grammar-based random program generator used for testing compilers. The generator is based on context-free grammars, which are dynamically modified during generation to accommodate some context-sensitive behaviour, for example when a new variable is introduced. The generator has a limited support for backtracking, which occurs when it is not possible to rewrite some non-terminal. The tool has been used to test compilers for simple properties, such as using programs that are syntactically-correct, or containing syntax errors, for example integer expressions in place of boolean ones.

7.2 Shrinking

Shrinking proved to be a very effective technique in property-based testing and is now standard in Haskell QuickCheck [Claessen and Hughes, 2000] and Erlang QuickCheck [Hughes, 2007]. Shrinking allows for defining



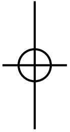


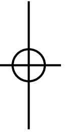
generic shrinking methods for polymorphic data types, which can be composed with shrinking methods for their element types. For instance, the default shrinking method for lists of integers uses the shrinking method for lists and also that for integers to reduce individual elements.

A similar technique has been invented concurrently, called *delta debugging* [Zeller and Hildebrandt, 2002], which is broader, but when applied to test input it bears resemblance to shrinking. For example, the standard method for reducing strings using delta debugging is very similar to the default shrinking method for lists. Delta debugging has been applied successfully to obtain small failing test cases for large and complex software.

Like shrinking, delta debugging finds a failing test case that is locally minimal. However, delta debugging assumes a different model for reducing test cases. A test case is first decomposed into a number of independent changes that represent transformation of an empty test case into the original test case. Then, a locally-minimal set of changes is determined, that results in a failing test case. Shrinking in QuickCheck, on the other hand, places very loose requirements on each specific shrinking method.

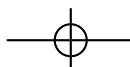
7.3 Library test tools

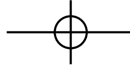




Klein et al. [2010b] created a testing tool that generates random programs to test an object-oriented library. Their generator is capable of producing higher-order object-oriented programs (which override methods) and supports monitoring of pre- and post-conditions, which are used to establish the validity and result of the test. Their generation method uses generation rules similar to ours, with random rule selection, size bound, and backtracking. Rather than our Indir rule, which generates calls of functions in the environment only when their result type matches the target type, they use a rule that can generate a call of *any* function in the environment at any time, binding its result to a fresh local variable, which can then in turn be used in another attempt to generate a term of the target type. The advantage of their approach is that it is easier to generate calls of functions in the environment; the disadvantage is that many of the local variables they create are never used, because their types do not match the target type. Klein et al. do not consider polymorphic types, nor do they shrink failing test cases to minimal examples as we do.

Wrangler, a refactoring tool for Erlang has also been tested using random program generation [Drienyovszky et al., 2010]. A rich program generator has been created, which is capable of generating full modules. Even though Erlang is an untyped language, the generator takes types into consideration in order to avoid argument mismatches when calling functions. Similarly,





Daniel et al. [2007] exhaustively generate Java programs (up a to certain size) in order to test the refactoring engines in Eclipse and NetBeans. Different from our approach, some of the generated programs are not valid inputs for the Java compiler.

Generating random sequential programs is practised in testing monadic code with QuickCheck [Claessen and Hughes, 2002] and in testing stateful programs with Erlang QuickCheck [Hughes, 2007]. Such a program is usually a sequence of actions, which might contain variables, but all variable handling is outsourced to the programmer using QuickCheck. Often such generated programs are not parametrised, which makes it possible to ensure that preconditions of all actions are satisfied.

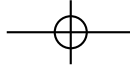
7.4 Testing of formal models

Redex [Klein et al., 2012] is a tool for lightweight verification of programming language formal models. Formal models are randomly tested whether they satisfy the stated properties using randomly generated expressions of the 'object' language. Generation of size-bounded terms is based on grammars (syntax of the object language is defined using a context-free grammar), and malformed expressions, for example containing free variables, are filtered out. Naïve generation and filtering is, of course, not enough to test more complex models, as many reduction rules are never exercised. To raise the likelihood of generating expressions that could be reduced with these reduction rules, their left-hand sides are used to guide the generation. Redex has been successfully used to formalise and test a nine existing formal models and find mistakes in all of them.

7.5 Typed term generators

Djinn [Augustsson, 2005] solves the *type inhabitation* problem for simply-typed λ-calculus, that is, it returns *any* term instead of a random one for a given type. It is based on a terminating proof procedure for intuitionistic propositional logic [Dyckhoff, 1992], which makes it find a term of a given type reliably when one exists. It is limited, however, in that it does not perform *polymorphic instantiation*, which means that it cannot generate some terms involving polymorphic constants.

Vytiniotis and Kennedy [2010] present encoding of data types into streams of bits, which can be used for their random generation. In their approach to generating simply-typed λ-terms, the target type is *never* fixed, and thus the generation never fails, eliminating the need for backtracking. This way of generating well-typed terms can also be extended to simply-typed lambda calculus with polymorphic constants. Reducing the problem

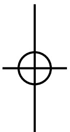


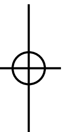
of random data generation to encoding in bit-streams has the consequence that improving the distribution of generated data corresponds to inventing efficient compression schemes, such as Huffman coding.

The λ-term *enumerator* developed by Rodriguez Yakushev and Jeuring [2010] creates function applications in the same way as our method, by generating a candidate type for the argument, and trying to generate the argument afterwards.

7.6 Untyped term generators

Statistical properties of random untyped λ-terms have been explored in [Bodini et al., 2011], which also explores a method of generating them using Boltzmann sampling. Generation of random untyped λ-terms is tackled in [Wang, 2005], which employs counting of possible subterms to achieve uniform generation distribution. Correspondingly, the work in [Moczurad et al., 2000] examines the proportion of simple types that are inhabited, that is, for which it is possible to create a term of that type.





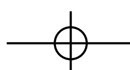
TGGS [Guilmette, 1995] is a random test data generation system based on context-free grammars enhanced with context-sensitive constructs, like imperative actions conditional clauses and stacks, which serve a similar rôle to attributes in attribute grammars. The system generates data by expanding non-terminal symbols by choosing grammar rules at random and backtracks whenever that is not possible, rolling back all relevant imperative actions. It is possible to affect the distribution of the generated data using weights, which influence how often different grammar rules are chosen.

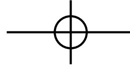
8 Future work

The potential for finding bugs in GHC using the presented method has not at all been exhausted during the testing that we performed. Although most of the presented bugs have been fixed, and our properties find counterexamples at a much lower rate, we are still able to find new interesting error cases using them. Many more new properties and variations of existing ones are likely to yield even more new counterexamples.

The same method can be applied to other Haskell compilers. However, it might be less effective for compilers that do not perform as sophisticated optimisation as GHC. When more Haskell compilers are available, it would be interesting to perform the standard variant of differential testing i.e. cross-testing of two different implementations.

Given that the subset of Haskell that is randomly generated is very limited, there is room for improvement by adding support for more language





constructions that can occur in the generated terms. For example, `let` and `where` clauses could be generated in a similar way to function applications and `case` expressions could be generated by having polymorphic constants that are required to be fully-applied. Polymorphic `let` bindings could also be supported by means of introducing polymorphic constants in their bodies and dummy type constants (simulating 'rigid' type variables) in the type of bound expressions.

The biggest technical challenge to solve in the current generator is to improve on the generation of terms involving polymorphic constants like `map` or monadic `bind`, which suffers from problems described in Section 4.2. The problem can be alleviated, for example, by allowing the generator to generate terms with partially-specified types. One argument of `map` would be generated with a partially-specified type while the other argument would be generated with a type that agrees with that of the already generated subterm.

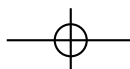
More drastic redesign is also possible in order to solve this problem. Theorem proving techniques might be used to track unresolved type variables and propagate the changes whenever they are refined in one of the subterms. However, we have been trying to avoid using theorem proving techniques as this approach would change the scope of this project too radically. We nevertheless think that using such an approach would be legitimate in well-typed term generation.

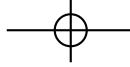
Even more radical would be creating a generator that approximates the uniform distribution of terms (of a given size) by counting all possible terms that can be generated, as it is done in AGATA [Almström Duregård, 2009]. However, it is not clear whether this approach is feasible computationally as the data type of well-typed terms is very complex, and quite large terms are needed to perform useful testing.

Another approach that could possibly be used is to adapt the technique proposed by Vytiniotis and Kennedy [2010]. The advantage of this approach is its simplicity and elegance, but a naïve generator seems to be very skewed towards creating terms with partially-applied constants, which suggests that much effort might be required to correct the distribution.

9 Conclusions

We applied property-based testing and random program generation for testing a sophisticated optimising Haskell compiler. Even though we generated a limited subset of Haskell, we were able to find interesting bugs in the compiler. Found counterexamples were reduced structurally using shrinking, which made them understandable and well-suited for bug reports.

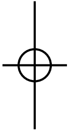
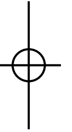




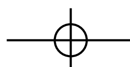
The properties used for finding the bugs employed differential testing by comparing the behaviour of the same program compiled with different optimisation levels. Also, we used an alternative form of differential testing where the behaviour of two equivalent programs is compared, which was used to find more bugs. In addition to bug reports, we learned about valid interesting behaviour of GHC, and in particular about changes to the default order of evaluation that it performs.

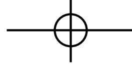
We have two positive observations about structural shrinking of counterexamples. First, even though the process is oblivious to the complex internal workings of the tested compiler the shrunk counterexamples could not be reduced by us further by hand in most cases, which suggests that the results were close to optimal. And secondly, looking at shrunk counterexamples allowed us to make educated guesses about the cause of the failures, even without referring to or understanding the compiler's code.

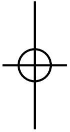
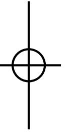
Unfortunately, testing a compiler using a random program generator is hardly a *fully automatic* technique, which we hoped it to be in the beginning. In contrast, we found that effective testing requires spending effort on creatively devising properties. In particular, some unexpected bugs were found by properties that were created for another purpose. However, the technique brings some automation to finding compiler bugs.

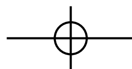



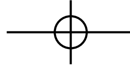
We were satisfied with the relevance and quality of counterexamples that we found for GHC with reasonable effort. Based on this experience we think that random compiler testing is an attractive technique for finding compiler bugs, which could be scaled up to perform much more comprehensive testing than we performed.





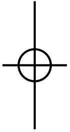


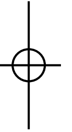


Paper II

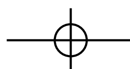
Generating Constrained Random Data with Uniform Distribution

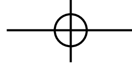
Koen Claessen **Jonas Duregård** **Michał H. Pałka**

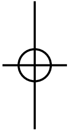
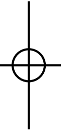


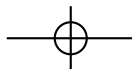


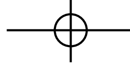
This in a revised version of a paper to be presented at the International Symposium on Functional and Logic Programming (FLOPS), 2014.









Paper II: Generating Constrained Random Data with Uniform Distribution

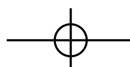
Abstract

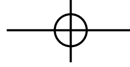
We present a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. The distribution of these generators is uniform over values of a given size. To make the generation efficient we rely on laziness of the predicate, allowing us to prune the space of values quickly. In contrast, implementing test data generators by hand is labour intensive and error prone. Moreover, handwritten generators often have an unpredictable distribution of values, risking that some values are arbitrarily underrepresented. We also present a variation of the technique where the distribution is skewed in a limited and predictable way, potentially increasing the performance. Experimental evaluation of the techniques shows that the uniform derived generators are much easier to define than handwritten ones, and their performance, while lower, is adequate for some realistic applications.

1 Introduction

Random property-based testing has proven to be an effective method for finding bugs in programs [Arts et al., 2006, Claessen and Hughes, 2000]. Two ingredients are required for property-based testing: a *test data generator* and a *property* (sometimes called oracle). For each test, the test data generator generates input to the program under test, and the property checks whether or not the observed behaviour is acceptable. This paper focuses on the test data generators.

The popular random testing tool QuickCheck [Claessen and Hughes, 2000] provides a library for defining random generators for data types.





$$\textbf{data}\ Expr = Ap\ Expr\ Expr\ Type \mid Vr\ Int \mid Lm\ Expr$$
$$\textbf{data}\ Type = A \mid B \mid C \mid Type :\!\rightarrow Type$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{check :: [Type] \rightarrow Expr \rightarrow Type \rightarrow Bool} \\
check\ env\ (Vr\ i) & t & = env\ !!\ i \equiv t \\
check\ env\ (Ap\ f\ x\ tx) & t & = \\
\quad \multicolumn{3}{l}{check\ env\ f\ (tx :\!\rightarrow t) \wedge check\ env\ x\ tx} \\
check\ env\ (Lm\ e) & (ta :\!\rightarrow tb) & = check\ (ta : env)\ e\ tb \\
check\ env\ _ & _ & = False
\end{array}
$$

Figure 2.1: Data type and type checker for simply-typed lambda calculus. The *Type* in the *Ap* nodes represents the type of the argument term.

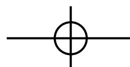
Typically, a generator is a recursive function that at every recursion level chooses a random constructor of the relevant data type. Relative frequencies for the constructors can be specified by the programmer to control the distribution. An extra resource argument that shrinks at each recursive call is used to control the size of the generated test data and ensures termination.

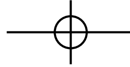
The above method for test generation works well for generating structured, well-typed data. But it becomes much harder when our objective is to generate well-typed data *that satisfies an extra condition*. A motivating example is the random generation of programs as test data for testing compilers. In order to successfully test different phases of a compiler, programs not only need to be grammatically correct, they may also need to satisfy other properties such as all variables are bound, all expressions are well-typed, certain combinations of constructs do not occur in the programs, or a combination of such properties.

In previous work by some of the authors, it was shown to be possible but very tedious to manually construct a generator that (a) could generate random well-typed programs in the polymorphic lambda-calculus, and at the same time (b) maintain a reasonable distribution such that no programs were arbitrarily excluded from generation.

The problem is that generators mix concerns that we would like to separate: (1) what is the structure of the test data, (2) which properties should it obey, and (3) what distribution do we want.

In this paper, we investigate solutions to the following problem: Given a definition of the structure of test data (a data type definition), and given one or more predicates (functions computing a boolean), can we automatically generate test data that satisfies all the predicates and at the same time has a





predictable, good distribution?

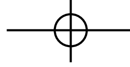
To be more concrete, let us take a look at Figure 2.1. Here, a data type for typed lambda expressions is defined, together with a function that given an environment, an expression, and a type, checks whether or not the expression has the stated type in the environment. From this input alone, we would like to be able to generate random well-typed expressions with a good distribution.

What does a 'good' distribution mean? First, we need to have a way to restrict the size of the generated test data. In any application, we are only ever going to generate a finite number of values, so we need a decision on what test data sizes to use. An easy and common way to control test data size is to control the *depth* of a term. This is for example done in SmallCheck [Runciman et al., 2008]. The problem with using depth is that the cardinality of terms of a certain depth grows extremely fast as the depth increases. Moreover, good distributions for, to give an example, the set of trees of depth d are hard to find, because there are many more almost full trees of depth d than there are sparse trees of depth d, which may lead to an overrepresentation of almost full trees in randomly generated values.

Another possibility is to work with the set of values of a given *size* n, where size is understood as the number of data constructors in the term. Previous work by one of the authors on FEAT [Duregård et al., 2012] has shown that it is possible to efficiently index in, and compute cardinalities of, sets of terms of a given size n. This is the choice we make in this paper.

The simplest useful and predictable distribution that does not arbitrarily exclude values from a set is the *uniform distribution*, which is why we chose to focus on uniform distributions in this paper. We acknowledge the need for other distributions than uniform in certain applications. However, we think that a uniform distribution is at least a useful building block in the process of crafting test data generators. We anticipate methods for controlling the distribution of our generators in multiple ways, but that remains future work.

Our first main contribution in this paper is an algorithm that, given a data type definition, a predicate, and a test data size, generates random values satisfying the predicate, with a perfectly uniform distribution. It works by first computing the cardinality of the set of all values of the given size, and then randomly picking indices in this set, computing the values that correspond to those indices, until we find a value for which the predicate is true. The key feature of the algorithm is that every time a value x is found for which the predicate is false, it is removed from the set of values, together with all other values that would have lead to the predicate returning false using the same execution path as x.



Unfortunately, even with this optimisation, uniformity turns out to be a very costly property in many practical cases. We have also developed a backtracking-based generator that is more efficient, but has no guarantees on the distribution. Our second main contribution is a hybrid generator that combines the uniform algorithm and the backtracking algorithm, and is 'almost uniform' in a precise and predictable way.

2 Generating Values of Algebraic Datatypes

In this section we explain how to generate random values of an algebraic data type (ADT) uniformly. Our approach is based on a representation of sets of values that allows efficient *indexing*, inspired by FEAT [Duregård et al., 2012], which is used to map random indices to random values. In the next section we modify this procedure to efficiently search for values that satisfy a predicate.

Algebraic Data Types (ADTs) are constructed using units (atomic values), disjoint unions of data types, products of data types, and may refer to their own definitions recursively. For instance, consider these definitions of Haskell data types for natural numbers and lists of natural numbers:

data *Nat* = *Z* | *Suc Nat*
data *ListNat* = *Nill* | *Cons Nat ListNat*

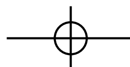
In general, ADTs may contain an infinite number of values, which is the case for both data types above. Our approach for generating random values of an ADT uniformly is to generate values of a specific *size*, understood as the number of constructors used in a value. For example, all of *Cons* (*Suc* (*Suc Z*)) (*Cons Z Nill*), *Cons* (*Suc Z*) (*Cons* (*Suc Z*) *Nill*) and *Cons Z* (*Cons Z* (*Cons Z Nill*)) are values of size 7. As there is only a finite number of values of each size, we can create a sampling procedure that generates a uniformly random value of *ListNat* of a given size.

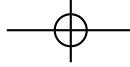
2.1 Indexing

Our method for generating random values of an ADT is based on an *indexing* function, which maps integers to corresponding data type values of a given size.

$$\text{index}_{S,k} : \{i \in \mathbb{N} \mid i < |S_k|\} \to S_k$$

Here, S is the data type, and S_k is the set of k-sized values of S. The intuitive idea behind efficient indexing is to quickly calculate *cardinalities* of subsets of the indexed set. For example, when $S = T \oplus U$ is a sum type, then





indexing is performed as follows:

$$\text{index}_{T\oplus U,k}(i) = \begin{cases} \text{index}_{T,k}(i) & \text{if } i < |T_k| \\ \text{index}_{U,k}(i - |T_k|) & \text{otherwise} \end{cases}$$

When $S = T \otimes U$ is a product type, we need to consider all ways size k can be divided between the components of the product. The cardinality of the product can be computed as follows:

$$|(T \otimes U)_k| = \sum_{k_1+k_2=k} |T_{k_1}||U_{k_2}|$$

When indexing $(T \otimes U)_k$ using index i, we first select the division of size $k_1 + k_2 = k$, such that:

$$0 \leq i' < |T_{k_1}||U_{k_2}| \quad \text{where} \quad i' = i - \sum_{\substack{l_1<k_1 \\ l_1+l_2=k}} |T_{l_1}||U_{l_2}|$$

Then, elements of T_{k_1} and U_{k_2} are selected using the remaining part of the index i'.

$$\text{index}_{T\otimes U,k}(i) = (\text{index}_{T,k}(i' \text{ div } |U_{k_2}|), \text{index}_{U,k}(i' \text{ mod } |U_{k_2}|))$$

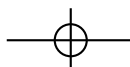
In the rest of this section, we outline how to implement indexing in Haskell.

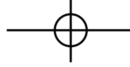
2.2 Representation of Spaces

We define a Haskell Generalized Algebraic Data Type (GADT) *Space* to represent ADTs, and allow efficient cardinality computations and indexing.

```
data Space a where
  Empty :: Space a
  Pure  :: a          → Space a
  (:+:) :: Space a    → Space a → Space a
  (:∗:) :: Space a    → Space b → Space (a, b)
  Pay   :: Space a    → Space a
  (:$:) :: (a → b)    → Space a → Space b
```

Spaces can be built using four basic operations: *Empty* for empty space, *Pure* for unit space, (:+:) for a sum of two spaces and (:∗:) for a product. Spaces also have an operator *Pay* which represents a unit cost imposed by using a constructor. The last operation (:\$:), applies a function to all values in the space. We assume that spaces are constructed in such a way that all their elements are unique. If this is not the case, a 'uniform' sampling procedure would return repeated elements more often than unique ones.





A very convenient operator on spaces is the lifted application operator, that takes a space of functions and a space of parameters and produces a space of all applications of the functions to the parameters:

$$(<\!*\!>) :: \mathit{Space}\ (a \to b) \to \mathit{Space}\ a \to \mathit{Space}\ b$$
$$s_1 <\!*\!> s_2 = (\lambda(f, a) \to f\ a)\ :\$:\ (s_1\ :\!*\!:\ s_2)$$

With the operators defined above, the definition of spaces mirror the definitions of data types. For example, spaces for the *Nat* and *ListNat* data types can be defined as follows:

$$\mathit{spaceNat} :: \mathit{Space\ Nat}$$
$$\mathit{spaceNat} = \mathit{Pay}\ (\mathit{Pure}\ Z\ :\!+\!:\ (\mathit{Suc}\ :\$:\ \mathit{spaceNat}))$$
$$\mathit{spaceListNat} :: \mathit{Space\ ListNat}$$
$$\mathit{spaceListNat} =$$
$$\quad \mathit{Pay}\ (\mathit{Pure\ Nill}\ :\!+\!:\ (\mathit{Cons}\ :\$:\ \mathit{spaceNat} <\!*\!> \mathit{spaceListNat}))$$

Unit constructors are represented with *Pure*, whereas compound constructors are mapped on the subspaces of the values they contain. In this example, *Pay* is applied each time we introduce a constructor, which makes the size of values equal to number of constructors they contain, and is the usual practice. However, the user may choose to use another way of assigning costs, which would change the sizes of individual values and, as a result,the distribution of the generated values. The only rule that must be followed when assigning costs is that all recursion is guarded by at least one *Pay* operation, otherwise the sets of values of a given size might be infinite, which would lead to non-terminating cardinality computations.

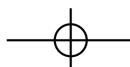
2.3 Indexing on Spaces

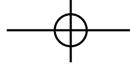
Indexing on spaces can be reduced to two subproblems: Extracting the finite set of values of a particular set, and indexing into such finite sets. Assume we have some data type for finite sets constructed by combining the empty set ($\{\}$), singleton sets ($\{a\}$), disjoint union ($\uplus$) and Cartesian product ($\times$). From the definition of such a finite set, its cardinality can be computed as follows:

$$|\{\}| = 0 \qquad |a \times b| = |a| * |b|$$
$$|\{a\}| = 1 \qquad |a \uplus b| = |a| + |b|$$

Using this function it is possible to define an indexing function on the type:

$$\mathit{indexFin}\ \{a\}\quad 0 \quad = a$$
$$\mathit{indexFin}\ (a \uplus b)\ i \mid i < |a| = \mathit{indexFin}\ a\ i$$





$$\begin{aligned}
&indexFin\ (a \uplus b)\ i \mid i \geqslant |a| = indexFin\ b\ (i - |a|) \\
&indexFin\ (a \times b)\ i = \\
&\quad (indexFin\ a\ (i \div |b|), indexFin\ b\ (i \mod |b|))
\end{aligned}$$

With these definitions at hand, all we have to do to index in spaces is to define a function *sized* which extracts the finite set of values of a given size k from a space.

$$\begin{aligned}
&sized\ Empty\ \ k = \{\} \\
&sized\ (Pure\ a)\ 0 = \{a\} \\
&sized\ (Pure\ a)\ k = \{\} \\
&sized\ (Pay\ a)\ 0 = \{\} \\
&sized\ (Pay\ a)\ k = sized\ a\ (k-1) \\
&sized\ (a\ :\!+\!:\ b)\ k = sized\ a\ k \uplus sized\ b\ k \\
&sized\ (f\ :\!\$\!:\ a)\ k = \{f\ x : x \in sized\ a\ k\}
\end{aligned}$$

We define *sized Pure* to be empty for all sizes except 0, since we want values of an exact size. For *Pay* we get the values of size $k - 1$ in the underlying space. Union and function application translate directly to union and application on sets. Selecting k-sized values of a product space requires dividing the size between its components. Thus, we can consider the set as a disjoint union of the $k + 1$ different ways of dividing size between the components:

$$sized\ (a\ :\!*\!:\ b)\ k = \biguplus_{k_1+k_2=k} sized\ a\ k_1 \times sized\ b\ k_2$$

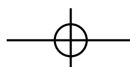
Knowing how to index in finite sets, we can implement an indexing function on spaces by composing the *sized* function with the *indexFin* function.

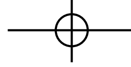
$$\begin{aligned}
&indexSized :: Space\ a \to Int \to Integer \to a \\
&indexSized\ s\ k\ i = indexFin\ (sized\ s\ k)\ i
\end{aligned}$$

Computing cardinalities and indexing requires arbitrarily large integers, which are provided by Haskell's *Integer* type. Calculating cardinalities can be computationally intensive, and to be practical requires memoising cardinalities of recursive data types, which is implemented using another constructor of the *Space a* data type not shown here.

3 Predicate-Guided Indexing

Having solved the problem of generating members of algebraic data types, we extend the problem with a predicate that all generated values must satisfy.





A first approach for uniform generation is to choose a size, generate values of that size, test them against the predicate and keep the ones for which the predicate is *True*. This works well for cases where the proportion of values that satisfy the predicate is large enough, for example larger than 1%, but is far too inefficient in many practical situations.

In order to speed up random generation of values satisfying a given predicate, we use the lazy behaviour of the predicate to know its result on sets of values, rather than individual values, similarly to [Runciman et al., 2008]. For instance, consider a predicate that tests if a list is sorted by checking the inequality of each pair of consecutive elements in turn starting from the front. Applying the predicate to $1:2:1:3:5:[\,]$ will yield *False* after the pair $(2,1)$ is encountered, before the predicate looks at the later elements, which means that it will return *False* for all lists starting with $1,2,1$. Once we have computed a set of values for which the predicate is going to return false, we remove all of these values from our original set.

To detect this we can exploit Haskell's call-by-need semantics by applying the predicate to a partially-defined value. In this case, observing that our predicate returns *False* when applied to a partially-defined list $1:2:1:\bot$, can lead us to conclude that $\bot$ can be replaced with any value without affecting the result. Thus, we could remove all lists that start with $1,2,1$ from the space. For many realistic predicates this removes a large number of values with each failed generation attempt, improving the chances of finding a value satisfying the predicate next time.

We implement this by using the function *valid*, that determines whether a given predicate needs to investigate its argument or not in order to produce its result. The function *valid* returns *Nothing* if the predicate needed its argument, and *Just b* if the predicate returns *b* regardless of its argument.

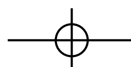
$$valid :: (a \rightarrow Bool) \rightarrow Maybe\ Bool$$

For example *valid* $(\lambda a \rightarrow True) \equiv Just\ True$, *valid* $(\lambda a \rightarrow False) \equiv Just\ False$, *valid* $(\lambda x \rightarrow x + 1 > x) \equiv Nothing$. Implementing *valid* involves applying the predicate to $\bot$ and catching the resulting exception if there is one. Catching the exception is an impure operation in Haskell, so the function *valid* is also impure (specifically, it breaks monotonicity).

The function *valid* is used to implement the indexing function, which takes the predicate, the space, the size and a random index.

$$index :: (a \rightarrow Bool) \rightarrow Space\ a \rightarrow Int \rightarrow Integer \rightarrow Space\ a$$

It returns a space of values containing at least the value at the given index, and any number of values for which the predicate yields the same result. When the returned space contains values for which the predicate is false,



the top level search procedure (not shown here) removes all these values from the original enumeration and retries with a new index in the now smaller enumeration.

The function *index* is implemented by recursion on its *Space a* argument, and composing the predicate with traversed constructor functions, until its result is independent of which value from the current space is chosen. In particular, *index* on a function application (:\$:) returns the current space if the predicate p' returns the same result regardless of its argument, which is determined by calling *valid* p'. Otherwise, it calls *index* recursively on the subspace, composing the predicate with the applied function.

```
index p (f :$: a) k i = case valid p' of
  Just _    → f :$: a
  Nothing → f :$: index p' a k i
  where p' = p ∘ f
```

3.1 Predicate-Guided Refinement Order

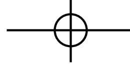
When implementing *index* for products, it is no longer possible to choose a division of size between the components, as was the case for indexing in Section 2. Determining the size of components early causes problems when generalising to sets of partial values, as the same partial value may represent values where size is divided in different ways.

We solve this problem using the algebraic nature of our spaces to eliminate products altogether. Disregarding the order of values when indexing, spaces form an algebraic semi-ring, which means that we can use the following algebraic laws to eliminate products.

$$
\begin{array}{lll}
a \otimes (b \oplus c) & \equiv (a \otimes b) \oplus (a \otimes c) & \textit{[distributivity]} \\
a \otimes (b \otimes c) & \equiv (a \otimes b) \otimes c & \textit{[associativity]} \\
a \otimes 1 & \equiv a & \textit{[identity]} \\
a \otimes 0 & \equiv 0 & \textit{[annihilation]}
\end{array}
$$

Expressing these rules on our Haskell data type is more complicated, because we need to preserve the types of the result, i.e. we only have associativity of products if we provide a function that transforms the left associative pair back to a right associative one, etc. The four rules defined on the *Space* data type expressed as a transformation operator (∗∗∗) are as follows:

```
a ∗∗∗ (b :+: c) = (a :∗: b) :+: (a :∗: c) [distributivity]
a ∗∗∗ (b :∗: c) =                         [associativity]
  (λ((x,y),z) → (x,(y,z))) :$: ((a :∗: b) :∗: c)
```



$$a *** (Pure\ x) = (\lambda y \to (y, x))\ :\$:\ a \qquad [identity]$$
$$a *** Empty = Empty \qquad [annihilation]$$

In addition to this, we need two laws for eliminating *Pay* and function application.

$$a *** (Pay\ b) = Pay\ (a\ :*:\ b) \qquad [lift\text{-}pay]$$
$$a *** (f\ :\$:\ b) = (\lambda(x, y) \to (x, f\ y))\ :\$:\ (a\ :*:\ b) \qquad [lift\text{-}fmap]$$

The first law states that paying for the component of a pair is the same as paying for the pair, the second that applying a function f to one component of a pair is the same as applying a modified (lifted) function on the pair. If recursion is always guarded by a *Pay*, we know that the transformation will terminate after a bounded number of steps.

Using these laws we could define *index* on products by applying the transformation, so $index\ p\ (a\ :*:\ b) = index\ p\ (a *** b)$. This is problematic, because $***$ is a right-first traversal, which means that for our generators the left component of a pair is never generated before the right one is fully defined. This is detrimental to generation, since the predicate may not require the right operand to be defined. To guide the refinement order by the evaluation order of the predicate, we need to 'ask' the predicate which component should be defined first. We define a function similar to *valid* that takes a predicate on pairs:

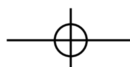
$$inspectsRight :: ((a, b) \to Bool) \to Bool$$

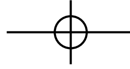
The expression *inspectsRight p* is *True* iff *p* evaluates the right component of the pair before the left. Just like *valid*, *inspectsRight* exposes some information of the Haskell runtime, which can not be observed directly.

To define indexing on products we combine *inspectsRight* with another algebraic law: commutativity of products. If the predicate 'pulls' at the left component, the operands of the product are swapped before applying the transformation for the recursive call.

```
index p (a :*: b) k i = if inspectsRight p
  then index p (a *** b)                 k i
  else index p (swap :$: (b *** a)) k i
  where swap (a, b) = (b, a)
```

The end result is an indexing algorithm that gradually refines the value it indexes to, by expanding only the part that the predicate needs in order to progress. With every refinement, the space is narrowed down until the predicate is guaranteed to be true or false for all values in the space. In the





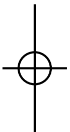
end the algorithm removes the indexed subspace from the search space, so no specialisations of the tested value are ever generated.

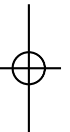
Note that the generation algorithm is still uniform because we only remove values for which the predicate is false from the original set of values. The uniformity is only concerned with the set of values for which the predicate is true.

3.2 Relaxed Uniformity Constraint

When our uniform generator finds a space for which the predicate is false, the algorithm chooses a new index and retries, which is required for uniformity. We have implemented two alternative algorithms.

The first one is to backtrack and try the alternative in the most recent choice. Such generators are no longer uniform, but potentially more efficient. Even though the algorithm start searching at a uniformly chosen index, since an arbitrary number of backtracking steps is allowed the distribution of generated values may be arbitrarily skewed. In particular, values satisfying the predicate that are 'surrounded' by many values for which it does not hold may be much more likely to be generated than other values.





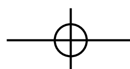
The second algorithm also performs backtracking, but imposes a bound b for how many values the backtracking search is allowed to skip over. When the bound b is reached, a new random index is generated and the search is restarted. The result is an algorithm which has an 'almost uniform' distribution in a precise way: the probabilities of generating any two values differ at most by a factor $b+1$. So, if we pick $b = 1000$, generating the most likely value is at most 1001 times more likely than the least likely value.

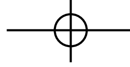
The bounded backtracking search strategy generalises both the uniform search (when the bound b is 0) and the unlimited backtracking search (when the bound b is infinite).

We expected the backtracking strategy to be more efficient in terms of time and space usage than the uniform search, and the bounded backtracking strategy to be somewhere in between, with higher bounds leading to results closer to unlimited backtracking. Our intention for developing these alternative algorithms was that trading the uniformity of the distribution for higher performance may lead to a higher rate of finding bugs. Section 4 contains experimental verification of these hypotheses.

3.3 Parallel Conjunction

It is possible to improve the generation performance by introducing the parallel conjunction operator [Runciman et al., 2008], which makes pruning the search space more efficient. Suppose we have a predicate $p\ x = q\ x \wedge r\ x$.





Given that && is left-biased, if *valid* $r \equiv$ *Just False* and *valid* $q \equiv$ *Nothing* then the result of *valid* p will be *Nothing*, even though we expect that refining q will make the conjunction return *False* regardless of what $q\ x$ returns.

We can define a new operator &&& for parallel conjunction with different behaviour when the first operand is undefined: $\bot$ &&& *False* $\equiv$ *False*. This may make the indexing algorithm terminate earlier when the second operand of a conjunction is false, without needing to perform refinements needed by the first operand at all. Similarly, we can define parallel disjunction that is *True* when either operand is *True*.

4 Experimental Evaluation

We evaluated our approach in four benchmarks. Three of them involved measuring the time and memory needed to generate 2000 random values of a given size satisfying a predicate. The fourth benchmark compared a derived simply-typed lambda term generator against a hand-written one in triggering strictness bugs in the GHC compiler. Some benchmarks were also run with a naïve generator that generates random values from a space, as in Section 2, and filters out those that do not satisfy a predicate.

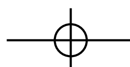
4.1 Trees

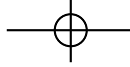
Our first example is binary search trees (BSTs) with Peano-encoded natural numbers as their elements, defined as follows.

```
data Tree a = L                          instance Ord Nat where
    | N a (Tree a) (Tree a)                _     < Z     = False
isBST :: Ord a => Tree a -> Bool           Z     < Suc _ = True
data Nat = Z | Suc Nat                     Suc x < Suc y = x < y
```

The *isBST* predicate (omitted) decides if the tree is a BST, and uses a supplied lazy comparison function for type *Nat* for increased laziness.

We measured the time and space needed to generate 2000 trees for each size from a range of sizes, allowing at most 300 s of CPU time and 4 GiB of memory to be used. Derived generators based on three different search strategies (see Section 3.2) were used: One performing uniform sampling (*uniform*), one bounded backtracking allowed to skip at most 10k values (*backtracking 10k*), and one performing unbounded backtracking (*backtracking*). A naïve generate-and-filter generator was also used for comparison.





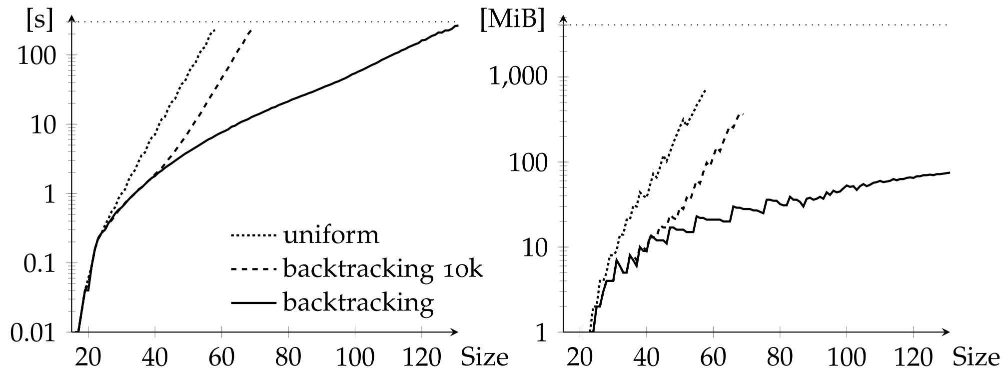


Figure 2.2: Run times in (left) and memory consumption (right) of derived generators generating 2000 BSTs depending on the size of generated values.

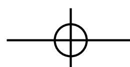
Both *backtracking 10k* and *backtracking* generators produce non-uniform distributions of values. The skew of the *backtracking 10k* generator is limited, as the least likely values are generated at most 10k times less likely than the most common ones, as mentioned in Section 3.2.

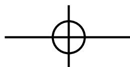
Figure 2.2 shows the time and memory consumed the runs with resource limits marked by dotted lines in the plots. Run times for all derived generators rise sharply with the increased size of generated values and seem to approach exponential growth for larger sizes. The backtracking generator performs best of all, and has a slower exponential growth rate for large sizes than the other derived generators. The *backtracking 10k* generator achieved similar performance as the *uniform* one when generating values that are about 11 size units larger. The generate-and-filter generator was not able to complete any of the runs in time, and is omitted from the graphs.

4.2 Simply-typed Lambda Terms

Generating random simply-typed lambda terms was our motivating application. Simply-typed lambda terms can be turned into well-typed Haskell programs and used for testing compilers. Developing a hand-written recursive generator for them requires the use of backtracking, because of the inability of predicting whether a given local choice can lead to a successful generation, and because typing constraints from two distant parts of a term can cause conflict. Achieving satisfactory distribution and performance requires careful tuning, and it is difficult to assess if any important values are severely underrepresented, as noted in Chapter 1 of this thesis..

On the other hand, obtaining a generator that is based on our framework





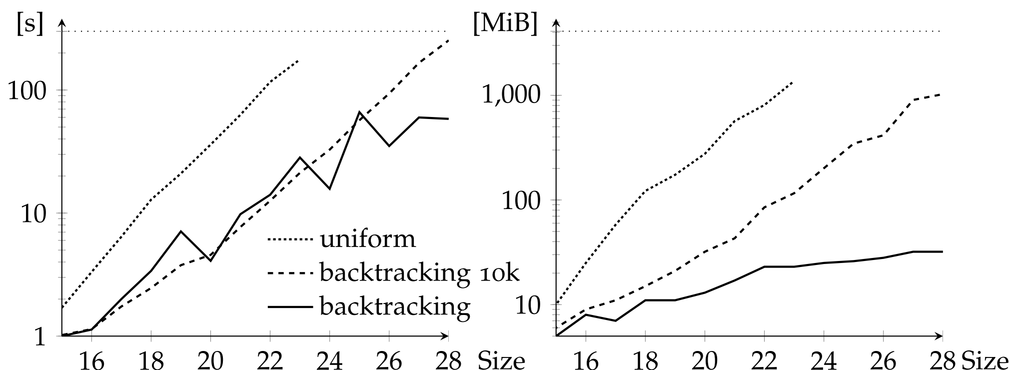


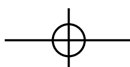
Figure 2.3: Run times (left) and memory consumption (right) of derived generators generating 2000 simply-typed lambda terms depending on the size of generated terms.

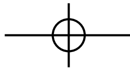
requires only the definitions from Figure 2.1, and a relatively simple space definition, which we omit here. The code for the type checker is standard and uses a type stored in each application node (*tx* in *Ap f x tx*) to denote the type of the argument term for simplicity.

To evaluate the generators, we generated 2000 terms with a simple initial environment of 6 constants. The derived generator with three search strategies and one based on generate-and-filter were used. Figure 2.3 shows the results. The uniform search strategy is capable of generating terms of size up to 23. For larger sizes, the generator exceeded the resource limits (300 s and 4 GiB, marked with dotted lines). The generator that used limited backtracking allowed generating terms up to size 28, using 9 times less CPU time and over 11 times less memory than the uniform one at size 23. Unlimited backtracking improved memory consumption dramatically, up to 30-fold, compared to limited backtracking. The run time is improved only slightly with unlimited backtracking. Finally, the generator based on generate-and-filter exceeded the run times for all sizes, and is not included in the plots.

4.3 Testing GHC

Discovering strictness bugs in the GHC optimising Haskell compiler was our prime reason for generating random simply-typed lambda terms. To evaluate our approach, we compared its bug finding power to a hand-written generator that is presented in Paper I of this thesis using the same test property that had been used there.





Generator	Hand-written	Derived (size 30)
Terms per ctr ex. (k)	18.6	52.5
Gen. CPU time per ctr ex.	1.7	14.0
Test CPU time per ctr ex.	1.8	10.4
Tot. CPU time per ctr ex.	3.5	24.4

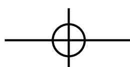
Table 2.1: Performance of the reference hand-written term generator compared to a derived generator using backtracking with size 30. We compare the average number of terms that have to be generated before a counterexample (ctr ex.) is found, and how much CPU time (in min.) the generation and testing consumes per found counterexample.

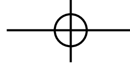
Random simply-typed lambda terms were used for testing GHC by first generating type-correct Haskell modules containing the terms, and then using them as test data. In this case, we generated modules containing expressions of type $[Int] \rightarrow [Int]$ and compiled them with two different optimisation levels. Then, we tested their observable behaviour and compared them against each other, looking for discrepancies.

We implemented the generator using a similar data type as in Figure 2.1 extended with polymorphic constants and type constructors. For efficiency reasons we avoided having types in term application constructors, and used a type checker based on type inference, which is more complex but still easily implementable. It allows generators to scale up to larger effective term sizes because not having types in the term representation increases the density of well-typed terms.

A generator based on this data type was capable of generating terms containing 30 term constructors, and was able to trigger GHC failures. Table 2.1 shows the results of testing GHC both with the hand-written simply-typed lambda term generator and our derived generator. The hand-written generator used for comparison generated terms of sizes from 0 to about 90, with most terms falling in the range of 20–50. It needed the least total CPU time to find a counterexample, and the lowest number of generated terms. The derived generator needs almost 7 times more CPU time per failure than the hand-written one.

The above results show that a generator derived from a predicate can be used to effectively find bugs in GHC. The derived generator is less effective than a hand-written one, but is significantly easier to develop. Developing an efficient type-checking predicate required for the derived generator took a few days, whereas the development and tuning of the hand-written generator took an order of months.





Predicates	Backtracking	Backtracking c/o
1, 2, 3, 4, 5	13	15
1, 3, 4, 5	13	30
1, 3, 5	31	30

Table 2.2: Maximum practical sizes of values generated by derived program generators that use unlimited backtracking and backtracking with cut-off of 10k.

4.4 Programs

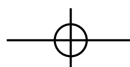
The *Program* benchmark is meant to simulate testing of a simple compiler by generating random programs, represented by the following data type.

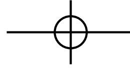
type *Name* = *String*
data *Program* = *New Name Program* | *Name* := *Expr* | *Skip*
| *Program* :>> *Program*
| *If Expr Program Program*
| *While Expr Program*
data *Expr* = *Var Name* | *Add Expr Expr*

The programs contain some common imperative constructs and declarations of new variables using *New*, which creates a new scope.

A compiler may perform a number of compilation passes, which would typically transform the program into some kind of normal form that may be required by the following pass. Our goal is to generate test data that satisfy the precondition in order to test the code of each pass separately. We considered 5 predicates on the program data type that model simple conditions that may be required by some compilation phases: (1) *boundProgram* saying that the program is well-scoped, (2) *usedProgram* saying that all bound variables are used, (3) *noLocalDecls* requiring all variables to be bound on the top level, (4) *noSkips* forbidding the redundant use of :>> and *Skip*, and (5) *noNestedIfs* forbidding nested *if* expressions.

Table 2.2 shows maximum value sizes that can be practically reached by the derived generators for the program data type with different combinations of predicates. All runs were generating 2000 random programs with resource limits (300 s and 4 GiB). When all predicates were used, the generators performed poorly being able to reach at most size 15. When the *usedProgram* predicate was omitted, the generator that uses limited backtracking improved considerably, whereas the one using unlimited backtracking remained at size 13. Removing the *noSkips* predicate turns the





tables on the two generators improving the performance of the unlimited backtracking generator dramatically.

A generator based on generate-and-filter was also benchmarked, but did not terminate within the time limit for the sizes we tried.

4.5 Summary

All derived generators performed much better than ones based on generate-and-filter in three out of four benchmarks. In the fourth one, testing GHC, using a generator based on generate-and-filter was comparable to using our uniform or near-uniform derived generators, and slower than a derived generator using backtracking. In that benchmark the backtracking generator was the only that was able to find counterexamples, and yet it was less effective than a hand-written generator. However, as creating the derived generators was much quicker, we believe that they are still an attractive alternative to a hand-written generator.

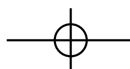
The time and space overhead of the derived generators appeared to rise exponentially, or almost exponentially with the size of generated values in most cases we looked at, similarly to what can be seen in Figures 2.2 and 2.3.

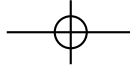
In most cases the backtracking generator provided the best performance, which means that sometimes we may have to sacrifice our goal of having a predictable distribution. However, we found the backtracking generator to be very sensitive to the choice of the predicate. For example, some combinations of predicates in Section 4.4 destroyed its performance, while having less influence on the uniform and near-uniform generators. We hypothesise that this behaviour may be caused by regions of search space where the predicates evaluate values to a large extent before returning *False*. The backtracking search remain in such regions for a long time, in contrast to the other search that gives up and restarts after a number of values have been skipped.

Overall, the performance of the derived generators is practical for some applications, but reaching higher sizes of generated data might be needed for effective bug finding. In particular, being able to generate larger terms may improve the bug-finding performance when testing for GHC strictness bugs.

5 Related Work

Feat Our representation of spaces and efficient indexing is based on FEAT (Functional Enumeration of Algebraic Types) [Duregård et al., 2012]. The practicalities of computing cardinalities and the deterministic indexing





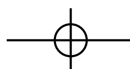
functions are described there. The inability to deal with complex data type invariants is the major concern for FEAT, which is addressed by this paper.

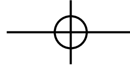
Lazy SmallCheck and Korat. Lazy SmallCheck [Runciman et al., 2008] uses laziness of predicates to get faster progress in an exhaustive depth-limited search. Our goal was to reach larger, potentially more useful values than Lazy SmallCheck by improving on it in two directions: using size instead of depth and allowing random search in sets that are too large to search exhaustively. Korat is a framework for testing Java programs [Boyapati et al., 2002]. It uses similar techniques to exhaustively generate size-bounded values that satisfy the precondition of a method, and then automatically check the result of the method for those values against a postcondition.

EasyCheck: Test Data For Free EasyCheck is a library for generating random test data written in the Curry functional logic programming language [Christiansen and Fischer, 2008]. Its generators define search spaces, which are enumerated using diagonalisation and randomising local choices. In this way values of larger sizes have a chance of appearing early in the enumeration, which is not the case when breadth-first search is used. The Curry language supports narrowing, which can be used by EasyCheck to generate values that satisfy a given predicate. The examples that are given in the paper suggest that, nonetheless, micro-management of the search space is needed to get a reasonable distribution. The authors point out that their enumeration technique has the problem of many very similar values being enumerated in the same run.

Metaheuristic Search In the GödelTest [Feldt and Poulding, 2013] system, so-called metaheuristic search is used to find test cases that exhibit certain properties referred to as *bias objectives*. The objectives are expressed as fitness metrics for the search such as the mean height and width of trees, and requirements on several such metrics can be combined for a single search. It may be possible to write a GödelTest generator by hand for well typed lambda terms and then use bias objectives to tweak the distribution of values in a desired direction, which could then be compared to our work.

Lazy Nondeterminism There is some recent work on embedding nondeterminism in functional languages [Fischer et al., 2011]. As a motivating example an *isSorted* predicate is used to derive a sorting function, a process which is quite similar to generating sorted lists from a predicate. The framework defined in [Fischer et al., 2011] is very general and could potentially be



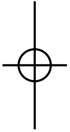


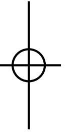
used both for implementing SmallCheck style enumeration and for random generation.

Generating Lambda Terms There are several other attempts at enumerating or generating well typed lambda terms. One such attempt uses generic programming to exhaustively enumerate lambda terms by size [Rodriguez Yakushev and Jeuring, 2010]. The description focuses mainly on the generic programming aspect, and the actual enumeration appears to be mainly proof of concept with very little discussion of the performance of the algorithm. There has been some work on counting lambda terms and generating them uniformly [Grygiel and Lescanne, 2013]. This includes generating well typed terms by a simple generate-and-filter approach.

6 Discussion

Performance of Limiting Backtracking The performance of our generators depends on the strictness and evaluation order of the used predicate. The generator that performs unlimited backtracking was especially sensitive to the choice of predicate, as shown in Section 4.4. Similar effects have been observed in Korat [Boyapati et al., 2002], which also performs backtracking.

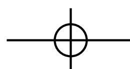


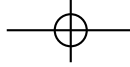


We found that for most predicates unbounded backtracking is the fastest. But unexpectedly, for some predicates imposing a bound on backtracking improves the run time of the generator. This also makes the distribution more predictable, at the cost of increased memory consumption. We found tweaking the degree of backtracking to be a useful tool for improving the performance of the generators, and possibly trading it for distribution guarantees.

In-place Refinement We experimented with a more efficient mechanism for observing the evaluation order of predicates, which avoids repeated evaluation of the predicate. For that we use an indexing function that attaches a Haskell IO-action to each subcomponent of the generated value. When the predicate is applied to the value, the IO-actions will fire only for the parts that the property needs to inspect to determine the outcome. Whenever the indexing function is required to make a choice, the corresponding IO-action records the option it did not take, so after the predicate has finished executing the refined search space can be reconstructed. Guiding the evaluation order is handled automatically by the Haskell run time system, which has call-by-need built into it.

In-place refinement is somewhat more complicated than the procedure described in Section 3. Also, defining parallel conjunction for this type





of refinement is difficult, because inspecting the result of a predicate irreversibly makes the choices required to compute the result. For this reason our implementation of in-place refinement remains a separate branch of development and a topic of future work.

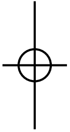
Conclusion Our method aims at preserving the simplicity of generate-and-filter type generators, but supporting more realistic predicates that accept only a small fraction of all values. This approach works well provided the predicates are lazy enough.

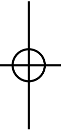
Our approach reduces the risk of having incorrect generators, as coming up with a correct predicate is usually much easier than writing a correct dedicated generator. Creating a predicate which leads to an efficient derived generator, on the other hand, is more difficult.

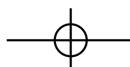
Even though performance remains an issue when generating large test cases, experimental results show that our approach is a viable option for generating test data in many realistic cases.

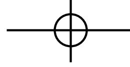
Acknowledgements

This research has been supported by the Resource-Aware Functional Programming (RAW FP) grant awarded by the Swedish Foundation for Strategic Research.





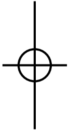


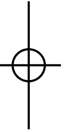


Paper III

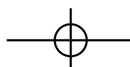
Splittable Pseudorandom Number Generators using Cryptographic Hashing

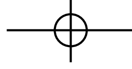
Koen Claessen **Michał H. Pałka**

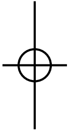
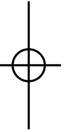


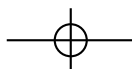


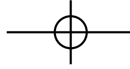
This in a revised version of a paper that appeared in the Haskell Symposium, 2013.









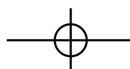
Paper III: Splittable Pseudorandom Number Generators using Cryptographic Hashing

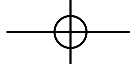
Abstract

We propose a new splittable pseudorandom number generator (PRNG) based on a cryptographic hash function. Splittable PRNGs, in contrast to linear PRNGs, allow the creation of two (seemingly) independent generators from a given random number generator. Splittable PRNGs are very useful for structuring purely functional programs, as they avoid the need for threading around state. We show that the currently known and used splittable PRNGs are either not efficient enough, have inherent flaws, or lack formal arguments about their randomness. In contrast, our proposed generator can be implemented efficiently, and comes with a formal statements and proofs that quantify how 'random' the results are that are generated. The provided proofs give strong randomness guarantees under assumptions commonly made in cryptography.

1 Introduction

Splittable pseudorandom number generators (PRNGs) are very useful for structuring purely functional programs that deal with randomness. They allow different parts of the program to independently (without interaction) generate random values, thus avoiding the threading of a random seed through the whole program [Burton and Page, 1992]. Moreover, splittable PRNGs are essential when generating random infinite values, such as random infinite lists in a lazy language, or random functions. In addition, deterministic distribution of parallel random number streams, which is of





interest to the High-Performance Computing community [Hill et al., 2012, Leiserson et al., 2012], can be realised using splitting.

In Haskell, the standard module `System.Random` provides a default implementation of a splittable generator `StdGen`, with the following API:

```
split :: StdGen -> (StdGen, StdGen)
next  :: StdGen -> (Int, StdGen)
```

The function `split` creates two new, independent generators from a given generator. The function `next` can be used to create one random value. A user of this API is not supposed to use both `next` and `split` on the same argument; doing so voids all warranties about promised randomness.

The property-based testing tool QuickCheck [Claessen and Hughes, 2000] makes heavy use of splitting. Let us see it in action. Consider the following simple (but somewhat contrived) property:

```
newtype Int14 = Int14 Int
  deriving Show

instance Arbitrary Int14 where
  arbitrary = Int14 `fmap` choose (0, 13)

prop_shouldFail (_, Int14 a) (Int14 b) = a /= b
```

We define a new type `Int14` for representing integers from 0 to 13. Next, we create a random generator for it that randomly picks a number from 0 to 13. Finally, we define a property, which states that two randomly picked `Int14` numbers, one of which is a component of a randomly picked pair, are always unequal.

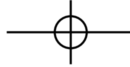
Testing the property yields the following result:

```
*Main> quickCheckWith
  stdArgs { maxSuccess = 10000 } prop_shouldFail
+++ OK, passed 10000 tests.
```

Even though the property is false (we would expect one of every 14 tests to fail), all 10000 tests succeed!

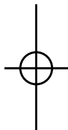
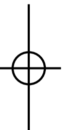
The reason for this surprising behaviour is a previously unknown flaw in the standard Haskell pseudorandom number generator used by QuickCheck during testing. The PRNG should pick all combinations of numbers 0–13 for `a` and `b`, but in fact combinations where `a` and `b` are the same number are never picked.

It turns out that the `StdGen` standard generator used in current Haskell compilers contains an *ad hoc* implementation of splitting. The current



implementation is the source of the randomness flaw[1] demonstrated above. The flaw requires a particular pattern of split operations to manifest and results in very strong correlation of generated numbers. In fact, when 13 in the `Int14` generator is replaced by other numbers from range 1–500, the problem arises for 465 of them! Unfortunately, this pattern of splits is simple and likely to arise often in typical usage of QuickCheck. Because of this, we cannot be sure that QuickCheck properties that pass a large number of tests are true with high probability.

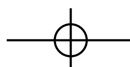
Unfortunately, research devoted to pseudorandom generation has mainly concentrated on linear generators, which do not support on-demand splitting. Several attempts have been made at extending linear PRNGs into splittable ones [Burton and Page, 1992, Frederickson et al., 1984, Mascagni et al., 1993]. Most proposed constructions were incompatible with unlimited on-demand splitting. Yet the ones that supported it did not assure the independence of derived generators. In fact, the current implementation of splitting in `System.Random` contains a comment *'no statistical foundation for this!'*. Indeed.

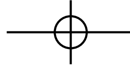



Attempting to generalise an existing traditional linear PRNG into a splittable one is problematic for two reasons. First, it is not clear how randomness properties of a linear generator carry over to a splittable generator. For example, demanding that every path through a splitting tree is a good linear PRNG is not at all enough to ensure that the two subgenerators created by a split are independent. And second, which may even be more problematic, the formal requirements satisfied by traditional linear PRNGs are not even sufficient to guarantee good randomness. Instead, their randomness is assessed using suites of statistical tests [L'Ecuyer and Simard, 2007, Salmon et al., 2011, Yao, 1982]. However, even if a linear generator passes these tests, this may not guarantee that it will work well in particular situations, as some linear PRNGs have been found to fail in intricate ways [McCullough, 2009]. So, because of the lack of strong formal properties satisfied by linear PRNGs, there is no reliable way of extending a linear PRNG into a splittable one that guarantees good statistical properties.

In contrast, cryptographic research has resulted in methods for generating high-quality pseudorandom numbers, which are provably random, using pseudorandom functions (PRFs) [Bellare et al., 1996, Goldreich et al., 1986, Håstad et al., 1999, Micali and Schnorr, 1991]. The proofs depend on unproven but commonly accepted assumptions, such as the computational difficulty of some problems, or on the existence of secure block ciphers. Despite that, they provide a much higher degree of confidence than statistical test suites. Even more importantly, the proofs serve as guidance for

[1] `http://hackage.haskell.org/trac/ghc/ticket/3575` and `.../3620`





deciding which constructions of PRNGs are sound.

Cryptographic methods have been proposed for implementing splittable PRNGs recently. In a Haskell-Cafe mailing list discussion [Peyton-Jones et al., 2010] Burton Smith et al. propose basing such a generator on a cryptographic block cipher. Another example are random number generators specified in NIST SP 800-90A [Barker and Kelsey, 2012], which are based on PRNGs (called DRBGs[2] there), whose one instance can be initialised using pseudorandom output from another instance. This mechanism has been used for implementing splitting in the crypto-api Hackage package[3].

The idea behind both of these designs is similar and appears to give good results. However, only an informal justification is provided for the first of them, and none for the second. Furthermore, splitting is costly in both of them, as every `split` operation requires one (or more) run of the underlying cryptographic primitive, such as a block cipher.

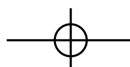
An efficient PRNG that supports splitting under the hood has been proposed by Leiserson et al. [2012]. To generate random numbers, the generator hashes a path, which identifies the current program location, using a hash function. The hash function is constructed in a way that minimises the probability of collisions and contains a final mixing step to make the output unpredictable. However, low probability of collisions is the only property that is proven for the hash function, whereas randomness of its results is only evaluated using statistical test.

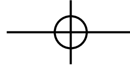
Micali and Schnorr [1991] present a PRNG based on the RSA cryptosystem, which is provably random and supports n-way splitting. However, the results are asymptotic, which means that they do not indicate what parameters to choose to achieve a particular level of randomness, or whether the generator is practical at all [Chatterjee et al., 2012, Fischlin and Schnorr, 1997, Sidorenko and Schoenmakers, 2005].

Thus, until now all splittable PRNGs were either *ad hoc* extensions of linear PRNGs, informally justified solutions based on cryptographic primitives, or cryptographic constructions covered by asymptotic arguments, which do not say anything about their practical quality. In this paper, we propose an efficient, provably-random splittable PRNG and give concrete security bounds for it. Our generator is based on an existing cryptographic keyed hash function [Bellare et al., 1996], which uses a block cipher as the cryptographic primitive. The generator is provably-random, which makes use of previously established concrete security proof of the hash function. Our contributions are as follows:

[2] Deterministic Random Bit Generators

[3] http://hackage.haskell.org/packages/archive/crypto-api/0.12.2.1/doc/html/Crypto-Random.html





- We propose a splittable prng that provides provable randomness guarantees under the assumption that the underlying block cipher is secure. The construction is conceptually very simple, and relies on a known keyed hash function design. (Section 3)
- We provide proofs of randomness of the proposed generator, which rely on previously known results about the hash function. The randomness results are concrete, not asymptotic, and degrade gracefully if the assumptions about the block cipher are relaxed. (Section 4)
- We present benchmark results indicating that QuickCheck executes typical properties about 8% slower than with `StdGen`, and the performance of linear random number generation is good. (Section 6)
- We show that the problem solved by a splittable prng is essentially the same as the one solved by a keyed hash function. (Section 7)
- The obtained randomness bounds are weaker than those possible for a linear prng. So, we quantify the price in randomness we have to pay for the increased flexibility that splitting provides. (Section 7)

2 Splittable prngs

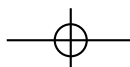
The traditional way of using random numbers is to give a program access to a process that generates a linear sequence of random numbers. This, however, is not a modular approach. Consider that we have function `f` that calls two other functions `g` and `h` that perform subcomputations.

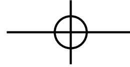
```
f x = ... (g y) ... (h z)
```

Normally, when using a linear pseudorandom generator, `f` would pass a reference to the generator to `g`, which would consume some number of random numbers and change the state of the generator, and then to `h`, which would do the same.

There are two problems with this approach. First, it would create an unnecessary data dependency between `g` and `h` which may otherwise have been independent computations, destroying opportunities for possible laziness or parallelism. Secondly, any change to `g` that would influence the amount of random numbers that it consumes would also influence the computation of `h`. Thus, repeating the computation with the same random seed and a changed program would also introduce disturbances in unchanged places.

Addressing these problems is the goal of splittable prngs [Burton and Page, 1992]. Consider the following interface for a prng whose state is





represented by the type Rand. This API is simpler than the one presented in the introduction in that the linear next operation is replaced by rand that only returns one random number. We will come back to the original API in Section 5.

```
split :: Rand -> (Rand, Rand)
rand  :: Rand -> Word32
```

Operation split takes a generator state and returns two independent generator states, which are derived from it. Operation rand generates a single random number from a generator state. Both split and rand are *pure*, which means that given the same arguments they yield the same result. Calling rand many times with the same generator state will yield the same result each time. The intended way of using such a generator is to start with an initial value of Rand and then split it to generate many random numbers.

Similar to the original API, an additional requirement is placed on the program that it is not allowed to call both split and rand on a given generator state, since the parent generator state is not guaranteed to be independent from generator states derived from it.

When using this API, function f can simply call split and pass two independent derived generator states to its subcomputations, which may use them without any data dependencies. This allows for generating random lazy infinite data structures in an efficient way, as independent subcomputations can be launched at any time.

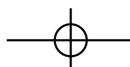
2.1 Naïve approach

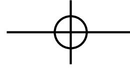
A naïve way of implementing a referentially-transparent splittable PRNG is to start with a linear PRNG and make split divide the sequence of random numbers in some way [Burton and Page, 1992]. For example, split may return the even-indexed elements as the left sequence and odd-indexed elements as the right sequence, whereas rand may return the first number of the sequence.

$$\texttt{split}\ (\langle x_0, x_1, \ldots \rangle) = (\langle x_0, x_2, \ldots \rangle, \langle x_1, x_3, \ldots \rangle)$$

Similarly, split may divide the whole sequence period into two halves and return them as subsequences. However, both of these approaches allow for a very short sequence of split operations, as after n splits the resulting sequence would be of length one, assuming the state size of n bits. Moreover, the amount of memory required to store a generator grows at least linearly in the number of splits.

Furthermore, there are no guarantees that the resulting numbers have good randomness properties. Traditionally, linear PRNGs are evaluated in





terms of their periods, which is how long a sequence they can generate without experiencing a state collision. Making statements about periods is meaningless for a splittable generator; for any generator that uses a constant amount of n bits of memory, there will be two identical generator states at most n splits away from each other, given that a full binary tree of depth n has 2^n leaves. The precise overall randomness properties have not been formalised for traditional linear prngs, and thus cannot be carried over to splittable generators at all.

An improved idea for a splittable prng is that the `split` operation jumps to a *random* place in the linear sequence [Burton and Page, 1992]. The design is compelling, as it does not suffer from the obvious limitation that concerns regular ways splitting. However, the quality of returned random numbers would again depend on the suitability of the original sequence, which is hard to determine.

Another approach for distributing unique generator states to subcomputations is to use an impure operation under the hood that returns numbers from a pseudorandom sequence, similarly to this solution for generating unique names [Augustsson et al., 1994]. However, such generator would no longer give deterministic results if the order of evaluation changes.

2.2 Pseudorandomness

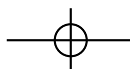
We found that a formal definition of pseudorandomness is essential for designing and evaluating a splittable prng. In particular, we found that the concept of pseudorandomness used in cryptography matches closely our goal of creating random-looking numbers, which we explain in this section.

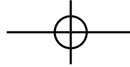
For simplicity, we first consider linear pseudorandom number generation. Consider non-deterministic program D that can perform a number m of coin flips and use their results in computations. We can model such a program with a deterministic function that has access to a sequence of m bits that has been chosen uniformly at random (we will use bits instead of numbers for simplicity). We assume that all 'deterministic' inputs are already baked into the program, and that it returns '0' or '1'.

We quantify the behaviour of the non-deterministic program D by the probability that it returns '1' when run with a sequence of bits chosen uniformly at random.

$$\Pr_{r \leftarrow \{0,1\}^m}[D(r) = 1]$$

The goal of pseudorandom number generation is to replace the randomly chosen sequence of bits by one generated by deterministic function p (the generator) from a small seed ($s \in S = \{0,1\}^n$) chosen at random,



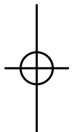


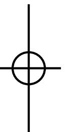
so that the program gives almost the same results with both random sequences. Thus, the probabilities of returning '1' by program D should be close to each other when run with a fully random sequence and with a pseudorandom sequence generated by function p. The absolute difference in these probabilities is D's *advantage* in distinguishing the output of p from random bits.

$$\mathrm{Adv}_D(p) = \left| \Pr_{r \leftarrow \{0,1\}^m}[D(r) = 1] - \Pr_{s \leftarrow S}[D(p(s)) = 1] \right|$$

Unfortunately, when $m > n$, it is always possible to construct a 'distinguishing' program that will make the difference in results large, since the image $p[S]$ is only a small part of all possible sequences. If program D checks whether the sequence of bits belongs to $p[S]$ outputting '1' in that case and '0' otherwise, then its results would always be '1' with p, but only half of the time or less with fully random bits.

Thus, any pseudorandom number generator p can be distinguished from a fully random choice of bits in the sense of information theory, that is when the distinguishing program has unlimited resources. However, it is widely believed that it is possible to construct effectively computable p, whose output can be distinguished from a true random sequence by a non-negligible margin only by programs that perform an enormous amount of computation [Blum and Micali, 1984, Goldreich et al., 1986, Håstad et al., 1999, Yao, 1982].

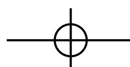


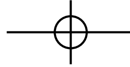


It has been shown that such deterministic functions whose output appears to be random (in an asymptotic sense) to any polynomial-time program can be constructed from one-way functions [Goldreich et al., 1986, Håstad et al., 1999]. At the same time, the existence of one-way functions is considered to be a reasonable complexity-theoretic assumption. For example, some number-theoretic functions, such as the RSA-function, are believed to be one-way [Bellare and Rogaway, 2005].

However, PRNG constructions based on one-way functions are inefficient. Cryptographic *block ciphers*, on the other hand, offer a more efficient alternative for implementing PRNGs [Bellare et al., 1996, Salmon et al., 2011]. Pseudorandomness of such generators depends on the security of block ciphers they are based on.

Block ciphers are common and widely used cryptographic primitives. Unlike number-theoretic constructions, the security of block ciphers does not follow from complexity-theoretic assumptions. Instead, their security is *asserted* based on careful design and unsuccessful cryptanalytic attempts on them [Bellare and Rogaway, 2005, Rogaway, 2011]. Even though there are many block ciphers that had once been considered safe and were subsequently broken, there exists a selection of them that successfully





underwent a thorough peer-review process without being broken, and are generally considered to be trusted primitives.

Asserting the security of a block cipher appears to be an unnecessarily strong assumption, given that constructions based on one-way functions could be used. However, proving their concrete (non-asymptotic) pseudorandomness requires making additional concrete assumptions about the one-way functions, which are much less obvious.

We propose to use the 256-bit variant of the ThreeFish [Ferguson et al., 2010] block cipher for the construction of our PRNG, which is one such peer-reviewed primitive. It has been proposed as the basis for the Skein hash function, which was one of the finalists of the National Institute of Standards and Technology (NIST) SHA-3 secure hash standard competition[4]. The hash function survived three rounds of public peer-review, withstanding the best attacks with a reasonable security margin, and its security was considered acceptable by NIST [Chang et al., 2012].

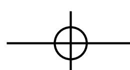
3 Proposed construction

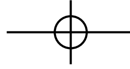
We propose a simple construction that gives a splittable PRNG, whose output appears to be fully random to any program that performs a reasonably bounded amount of computation. A program that has access to such a generator, using the API presented in Section 2, can use splitting to create a number of generator states over the course of its execution, and can also query some of these states for random numbers using `rand`. Observe that we can identify each generator state with the path that was used to derive it from the initial generator state. The main idea behind our construction is to map these paths to numbers using a keyed hash function, whose results for different arguments appear to be unpredictably random.

First, we injectively encode each path as a sequence of bits. Starting from the initial generator state we take '0' each time we go to the left and '1' to the right. Then we use a cryptographic keyed hash function to map that sequence of bits to a hash value. The initial seed of the generator is used as the key for the hash function. The result of the hash function is our random number.

Figure 3.1 shows a tree of generator states created by splitting the initial generator state a. Random numbers are generated by computing the hash function of the encoded paths leading to the respective generator states. For example, generator state c has been obtained from the initial state using 3 `split` operations: first the right generator state has been chosen, second also the right one, and third the left one. Hence, the path leading to c is

[4] http://csrc.nist.gov/groups/ST/hash/sha-3/index.html





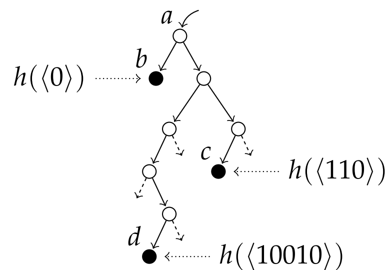


Figure 3.1: Splittable PRNG through hashing of paths.

encoded as bit sequence $\langle 110 \rangle$, and the hash of that is the random number returned by `rand` c.

The following code shows a simple, but inefficient Haskell implementation of this generator.

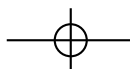
```
type Rand = (Seed, [Bit])

split :: Rand ->  (Rand, Rand)
split (s, p) = ((s, p ++ [0]), (s, p ++ [1]))

rand :: Rand -> Word32
rand  (s, p) = extract $ hash s p
```

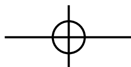
The concern of this simple construction is to uniquely encode each derived generator state. The entire task of creating random-looking output is outsourced to the keyed hash function, which we chose to implement with a tried and tested construction.

3.1 Efficient hashing

Using a cryptographic hash function to process the paths leading to every random number may seem to be very slow. However, cryptographic hash functions often have an iterated construction, which allows them to be computed incrementally. By doing so we can implement both `split` and `rand` in $O(1)$ time, and make `split` a very cheap operation most of the time. In this way we achieved performance that almost matches the current standard splittable generator for Haskell.

We use the Merkle-Damgård construction [Bellare et al., 1996, Coron et al., 2005, Damgård, 1990] to implement hashing, which is a common pattern for iterated hash functions. In this construction, shown in Figure 3.2, the input consists of a number of fixed-width data blocks m_0 to m_n that are





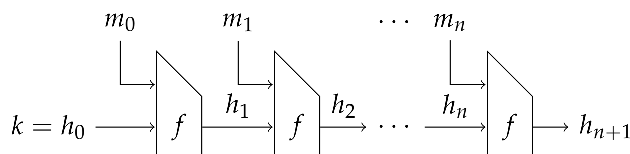


Figure 3.2: Merkle-Damgård construction. f represents the compression function.

iteratively fed into function f (the *compression* function), which takes both a data block and intermediate state h_i and computes new intermediate state h_{i+1}.

$$h_{i+1} = f(h_i, m_i)$$

The hash key is used as the initial state of the iteration h_0, and the final state h_{i+1} is the value of the hash. The function is essentially a *fold* of the compression function f, and can be computed incrementally.

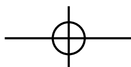
$$\texttt{hash}\ k\ [m_0, \ldots, m_n]\ \texttt{=}\ \texttt{foldl'}\ f\ k\ [m_0, \ldots, m_n]$$

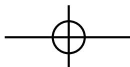
The purpose of the compression function is to mix a block of input data into an intermediate state in an unpredictable way.

The unpredictability can only be achieved when the initial state and all intermediate states are unknown to the program receiving the outputs of the hash function. For this reason, none of the inputs for the hash function with a given key should be a prefix of another one. Otherwise, the result of the hash function for the shorter input would be the same as one of the intermediate states for the longer one, which would mean that one function result could be predicted from another. In other words, the construction requires that the set of queried inputs is *prefix-free*.

The construction relies on the compression function for unpredictability, merely extending its domain from single blocks to their sequences. The compression function can be implemented using a block cipher by feeding the state h_i as the key of the cipher and the data blocks as the data blocks of the cipher, as shown in Figure 3.3.

The requirements of the hash function that the input is a sequence of blocks of a size determined by the block cipher, and that the inputs used during one program run form a prefix-free set, need to be addressed by the encoding used to create inputs. Below we present such an encoding.





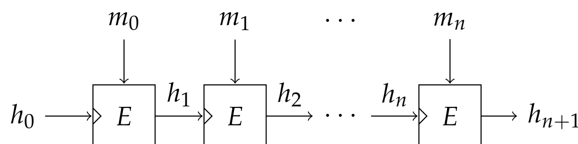


Figure 3.3: Merkle-Damgård construction using a block cipher. Input indicated with a triangle is the key of the block cipher.

3.2 Encoding

We encode the paths leading to generator states as sequences of blocks. First, let's represent a path as a sequence of bits as previously, by walking the path from the initial generator state and taking '0' for going left and '1' for going right. Then, we divide this sequence into blocks of the required size and if the last block is incomplete, pad it with '0's.

Observe that the paths to the generator states that are queried in a single run of a program must form a prefix-free set, due to the requirement that `split` and `rand` cannot be called on the same generator state.

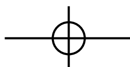
The fact that the paths form a prefix-free set is required for the encoding to be injective. To show the injectivity, let's assume the opposite. Suppose that we have two different paths that are mapped to the same sequence of k blocks. Given that the initial $k-1$ blocks are equal, the paths must have a common prefix that was encoded in these blocks. Since the last block is also equal, the two paths must have different lengths and the suffix of the longer one must be encoded as all '0's to match the padding in the encoding of the shorter one. Furthermore, the part that precedes that suffix must be equal to the last part of the shorter path. From this we can see that the shorter path must be a prefix of the longer one, which contradicts our assumption.

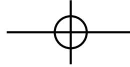
We can similarly show that no sequence of blocks is a prefix of another one if both are encodings of paths that are not each other's prefixes. Thus, the encodings of paths that form a prefix-free set will themselves form a prefix-free set, which is required by the hash function.

The encoding is efficient in the sense that to hash a given path a block cipher is invoked only once every b splits where b is the block size in bits.

3.3 Incremental computation

To obtain hashing and encoding presented above in an efficient and incremental way we implemented both of these stages together in the `split` and `rand` operations. Operation `split` takes a partially-computed hash value and returns two partially-computed hash values. Operation `rand` completes the computation of the hash and returns its value. Implementing





the encoding and hashing together has the added benefit that the hash computation of the initial common part of the path can be shared between different generator states.

To allow incremental computation the generator state must contain both the last intermediate state block of the hash function and the last part of the path that has not been hashed yet. Operation `split` adds a bit ('0' in the right generator state, '1' in the left one) to the unhashed sequence and if the unhashed sequence has reached the length of a block, it runs one iteration of the hash function that consumes the sequence and updates the intermediate state. Operation `rand` runs the final iteration of the hash function with whatever unhashed sequence there is (it might be empty), pads it with '0's and extracts a random number from the hash.

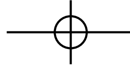
```
type Rand = (Block, [Bit])

hashStep :: Rand -> Rand
hashStep s@(h_i, unhashed)
  | length unhashed < blockLength = s
  | otherwise = (compress h_i (pad unhashed), [])

split :: Rand -> (Rand, Rand)
split (h_i, unhashed) =
  (hashStep (h_i, unhashed ++ [0]),
   hashStep (h_i, unhashed ++ [1]))

rand :: Rand -> Word32
rand (h_i, unhashed) =
  extract $ compress h_i (pad unhashed)
```

Since the length of `unhashed` is at most equal to `blockLength`, the run time of both `rand` and `split` operations is bounded. Also, `split` is usually inexpensive, as most of the time it only adds a single bit to the state.

4 Correct hashing

The Merkle-Damgård hashing construction shown by us in the previous section relies on a block cipher, or another compression function with similar properties, to provide pseudorandomness. However, the block cipher invocations are chained, making its use rather non-trivial. Thus, it is important to consider the correctness of the construction.

One way of analysing a cryptographic hash function is to use a security reduction proof to bound the advantage of any program in distinguishing



the results of the hash function from completely random results, depending on the amount of computation the program can perform. Proving the hash function's correctness amounts to showing that it gives only negligible advantage to programs that perform reasonable amount of computation.

The M-D construction has previously been analysed from this perspective. Below, we present a reduction proof of its pseudorandomness, based on [Bellare and Rogaway, 2005, Bellare et al., 1996]. the proof allows us to derive concrete bounds on the maximum discrepancy that a program using the generator can observe.

Using security proofs turned out to be very helpful in designing the PRNG, as they helped to identify incorrect designs as well as their elements, which were not bringing any benefits. In addition, the bounds that they provide made the limitations of the generator explicit.

The reduction proof assumes that the used block cipher is secure in the sense that it can be effectively broken only by key enumeration, which is the main assumption of our PRNG (see Section 2.2). Moreover, even if the cipher's known security is reduced, the bound provided by the proof degrades gracefully.

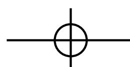
However, perhaps the biggest value of the proof is the guarantee that the block cipher is used in a correct way in the construction, as otherwise it would not have been possible to achieve a low bound on the discrepancy.

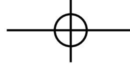
4.1 Pseudorandom functions

In order to formally model block ciphers and keyed hash functions, we need to introduce pseudorandom functions (PRFs) [Bellare and Rogaway, 2005, Bellare et al., 1996, Goldreich et al., 1986], which generalise pseudorandom number generators.

Let $F = \{f_k : k \in K\}$ be a finite family of functions $f_k \in B^A$ indexed by elements of finite set K. We write $f \leftarrow F$ to mean that f has been chosen from F by choosing its index uniformly at random from K. Next, consider $F' = B^A$ to be the set of functions from A to B, where B is a finite set, and A is possibly infinite.

This introduces a slight complication, as F' may be infinite, and the uniform distribution does not exist for coutably infinite sets. To address that, we use the *lazy sampling* principle [Bellare and Rogaway, 2004]. That is, instead of choosing a random member of B^A, we create a lazy non-deterministic procedure that generates the function's random results on demand. The procedure picks a random value of B each time the function is called with a new argument. Consequently, the results of the function appear exactly as if they were returned by a 'randomly chosen' function. We use the same notation $f' \leftarrow F'$ to mean that f' has been chosen from F'





using lazy sampling. For function families that are finite, when both A and B are finite, lazy sampling gives the same observed distribution as choosing their representant uniformly at random.

We generalise the definition of advantage from Section 2.2. Let F_1 and F_2 be two function families of functions B^A, each of which may be an indexed function family, or the set of all functions from A to B. The *advantage* of D in distinguishing between F_1 and F_2 is defined as follows:

$$\mathrm{Adv}_D(F_1, F_2) = \left| \Pr_{f_1 \leftarrow F_1}[D(f_1) = 1] - \Pr_{f_2 \leftarrow F_2}[D(f_2) = 1] \right|$$

where $D(f)$ is program D instantiated with a particular function $f : B^A$, which it can query as a black-box oracle.

Our objective now will be to have a small and easily computable indexed function family $F \subseteq B^A$, whose random member appears to programs as a random function B^A. In other words, the advantage $\mathrm{Adv}_D(F, B^A)$ should be as small as possible for any (reasonably bounded) D, making F and B^A difficult to distinguish (*indistinguishable*). Such an indexed function family is said to be *pseudorandom*.

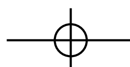
The original goal of bounding the advantage of any program with a certain run time limit has been to defend against an adversary who could create a program that breaks the pseudorandom number generator, or another cryptographic entity. Even though we do not assume any adversarial behaviour, we would like that the user of a PRNG can write any program without risking that a flaw in the generator will compromise his results.

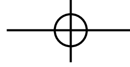
4.2 Iterative hash construction

We will now formalise the construction of the hash function presented in Section 3.1. The hash function uses a compression function f to process each of the data blocks in turn. Let $F = \{f_k : k \in B\}$ be an indexed function family representing the compression function, so that $f_k(x) = f(k, x)$ and $B = \{0, 1\}^n$ represent the set of data blocks, as well as the set of keys. We define $F^* = \{f_k^* : k \in B\}$ with $f_k^* \in B^{B^*}$ to be a finite function family that represents the iterative hash construction based on f, as follows by induction.

$$\begin{aligned} f_k^*(\langle\rangle) &= k \\ f_k^*(\langle B_0, \ldots, B_{n-1}, B_n \rangle) &= f_{f_k^*(\langle B_0, \ldots, B_{n-1} \rangle)}(B_n) \end{aligned}$$

To show that this iterative hash construction is a good pseudorandom function we show a bound on the advantage program D can get in distin-





guishing it from a random function.

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leqslant \epsilon$$

The above bound can only be shown assuming that is that the advantage in distinguishing the compression function from a random function ($\mathrm{Adv}_{D'}(F, B^B)$) is also bounded. The proof also captures the crucial assumption that the sequences used by D for making queries form a prefix-free set.

To show the bound we refer to a result from [Bellare et al., 1996]. The following theorem assumes the Random-Access Machine (RAM) computational model, which we also assume for the rest of this section.

Theorem 4.1. *Let F be a finite function family $F = \{f_k : k \in B\}$ with $f_k : B \to B$. Suppose that there exists program D, which has access to an oracle of type $B^* \to B$, and makes at most q prefix-free queries to this oracle at most $l \geqslant 1$ blocks long, consuming at most t units of time[5]. Then, there exists program E, which has access to an oracle of type $B \to B$, as well as to program D as another oracle, such that:*

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leqslant ql\, \mathrm{Adv}_{E(D)}(F, B^B)$$

Program $E(D)$ makes at most q queries and runs in time at most $t + cq(l + k + b)(\mathrm{Time}(F) + \log q)$, where $\mathrm{Time}(F)$ is the time required to execute the implementation of F and c is a small constant.

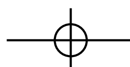
Proof. The theorem follows from Theorem 3.1 in [Bellare et al., 1996]. □

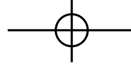
The theorem states that if there exists program D that distinguishes between F^* from a random function with probability ϵ, then it is possible to construct program $E(D)$, which uses D as its subprogram, and that is able to distinguish the compression function F from a random function with probability $ql\epsilon$, with additional constraints on the run time of both programs.

Conversely, if we assert that the advantage of any program in distinguishing F^* from a random function is not greater than ϵ, then we can bound the advantage of any program for distinguishing F from a random function by $ql\epsilon$.

It is worth noting that we only specified the computational model for the reduction as the RAM computational model, and did not precise what additional operations programs may execute, such as the decryption function of a block cipher. This may seem suspicious as using a computational model that is weaker may appear to 'leave out' some ways to attack the hash.

[5] Time to read the program text is also counted in t.





However, in fact using a weaker model can only result in $\mathrm{Adv}_{E(D)}(F, K^B)$ being smaller and providing a more conservative bound on $\mathrm{Adv}_D(F^*, K^{B^*})$.

To use Theorem 4.1 to bound the advantage on the hash function, we need to supply the bound on the compression function.

4.3 Compression function

In order to obtain an unpredictable compression function, we employ a cryptographic *block cipher*. A block cipher is a pair of effectively computable functions $\mathsf{enc} : K \times B \to B$ and $\mathsf{enc}^{-1} : K \times B \to B$, where K is the set of keys and B is the set of blocks. Both keys and blocks are fixed-length sequences of bits, and for our purposes we assume that $K = B = \{0,1\}^n$. Each $\mathsf{enc}_k : B \to B$ is a permutation on B and also $(\mathsf{enc}_k)^{-1} = \mathsf{enc}_k^{-1}$.

The theoretical model for block ciphers that we will use is pseudorandom permutations (PRPs) [Bellare et al., 1998, Rogaway, 2011]. A PRP is an indexed function family that is indistinguishable from a random permutation in the same way as a pseudorandom function is indistinguishable from a random function.

To show the security bound of the hash function we need the compression function to be a PRF, but a block cipher that we have is a PRP. However, using a PRP in place of a PRF does not impose a large penalty on observed randomness, which resulted in many analyses treating block ciphers as PRFs [Bellare and Rogaway, 2004].

The 'similarity' of PRPs and PRFs is stated by the PRP/PRF Switching Lemma [Bellare and Rogaway, 2004, Hall et al., 1998], which gives a bound on the ability to differentiate between a PRF and a PRP by a computationally-unbounded program (Perm(B) denotes the set of permutations of B).

$$\mathrm{Adv}_D(B^B, \mathrm{Perm}(B)) \leqslant \frac{q(q-1)}{2^{b+1}}$$

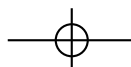
where $B = \{0,1\}^b$ and q is the number of queries D can make to the oracle. Thus, the best way of distinguishing a PRP from a PRF is to query it a large number of times and look for collisions.

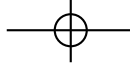
Given that, we can obtain a bound on expression $\mathrm{Adv}_E^D(F, B^B)$ from Theorem 4.1. From the definition of Adv we have the inequality:

$$\mathrm{Adv}_{E^D}(F, B^B) \leqslant \mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B)) + \mathrm{Adv}_{E^D}(\mathrm{Perm}(B), B^B)$$

Now we can bound $\mathrm{Adv}_{E^D}(\mathrm{Perm}(B), B^B)$ using the Switching Lemma:

$$\mathrm{Adv}_{E^D}(B^B, F) \leqslant \mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B)) + \frac{q(q-1)}{2^{b+1}}$$





To bound $\mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B))$, we have to invoke our assertion that F is a good block cipher. It is reasonable to assume that the best attacks on F are by key search [Bellare and Rogaway, 2005].

$$\mathrm{Adv}_{E^D}(F, \mathrm{Perm}(B)) \leqslant 2^{-b}\left(q + \frac{t'}{\mathrm{Time}(F)}\right)$$

Finally

$$\mathrm{Adv}_{E^D}(F, B^B) \leqslant 2^{-b}\left(q + \frac{t'}{\mathrm{Time}(F)} + \frac{q(q-1)}{2}\right)$$

Now invoking Theorem 4.1, we get:

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leqslant \\ 2^{-b} ql\left(q + \frac{t + cq(l+k+b)(\mathrm{Time}(F) + \log q)}{\mathrm{Time}(F)} + \frac{q(q-1)}{2}\right)$$

After simplification and omitting insignificant terms we obtain the final bound:

$$\mathrm{Adv}_D(F^*, B^{B^*}) \leqslant 2^{-b}\left(\frac{q^3 l}{2} + c_1 q^2 l^2 (1 + \log q) + \frac{c_2 tql}{\mathrm{Time}(F)}\right)$$

For example for a 256-bit block cipher, if program performs at most 2^{50} queries, each of at most 2^{30} blocks, then the discrepancy can be bounded by 2^{-70}, provided that the run time is bounded by 2^{90}. We assume that the constants are small ($c_2 / \mathrm{Time}(F) < 2^{10}$).

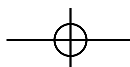
5 Linear generation

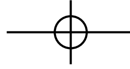
We presented only a simplified version of splittable pseudorandom number generators with `split` and `rand` operations. Splittable generators in Haskell also support linear generation, which is possible with a modified API:

```
split :: Rand -> (Rand, Rand)
next  :: Rand -> (Word32, Rand)
```

Operation `next` replaces `rand` and, in addition to returning a random number, also returns a new generator state. Calling `next` repeatedly allows generating a sequence of random numbers. The API does not allow calling both `split` and `next` on the same generator state, similarly as in the original API presented earlier.

Replacing `rand` with `next` does not give any additional expressiveness to the API, as `next` can be implemented on top of the old API in the following way:





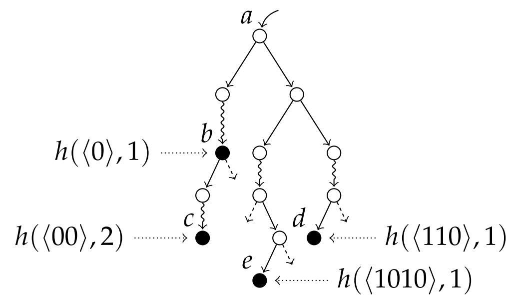


Figure 3.4: Splittable PRNG with `next` operation. Snake arrows (⇝) lead to states derived using `next` operation.

```
next g = (rand g1, g2)
  where
  (g1, g2) = split g
```

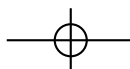
However, having `next` allows implementing linear random number generation more efficiently by counting the number of `next` operations on the path (which takes only $O(\log n)$ bits), in addition to keeping the sequence of `right` and `left` operations. Figure 3.4 presents encodings of paths from the initial state a to several derived generator states (marked in black). For example, path to state d is encoded as $(\langle 110\rangle, 1)$, because it contains one occurrence of `next` and the remaining operations on the path form the sequence `right`, `right`, `left`. Observe that it is possible for another path to be encoded as $(\langle 110\rangle, 1)$. However, reaching both that other state and d would require calling both `split` and `next` on some state, and thus violating the API requirement.

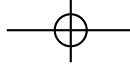
The used encoding (`e1`) is defined below. We formally show its uniqueness, together with a stronger property, which is of interest for creating more advanced encodings.

Definition 5.1. Encoding `e1` is a function that maps a path of three operations: `right`, `left` and `next` to a pair of a sequence of bits and a number. The number is equal to the number of `next` operations and the sequence of bits represents the sequence of `right` and `left`.

The following definition captures the requirement that must be satisfied by programs using the splittable PRNG API.

Definition 5.2. A program run that uses the API is *valid* iff, for any two distinct generator states queried in a single program run, the following holds. Let paths c_1 and c_2 represent the two states. Let c_p be their longest





common prefix, and c'_1 and c'_2 be the respective remainders of c_1 and c_2. The following condition must be satisfied

$$\begin{aligned} & c'_1 = \langle\rangle && \wedge c'_2 = \langle \texttt{next}, \ldots \rangle \\ \vee\ & c'_1 = \langle \texttt{next}, \ldots \rangle && \wedge c'_2 = \langle\rangle \\ \vee\ & c'_1 = \langle \texttt{right}, \ldots \rangle && \wedge c'_2 = \langle \texttt{left}, \ldots \rangle \\ \vee\ & c'_1 = \langle \texttt{left}, \ldots \rangle && \wedge c'_2 = \langle \texttt{right}, \ldots \rangle \end{aligned} \tag{5.1}$$

where $\langle\rangle$ denotes the empty path, and '...' any remainder of a path.

To be able to state the main property of encoding `e1` we define relation $\mathrel{\substack{\prec\\\succ}}$. Let p_1 and p_2 be sequences; then $p_1 \mathrel{\substack{\prec\\\succ}} p_2$ means that one of p_1 and p_2 is a prefix of the other one. That is,

$$p_1 \mathrel{\substack{\prec\\\succ}} p_2 \quad \text{iff} \quad p_1 \preceq p_2 \vee p_2 \preceq p_1.$$

Note that $\preceq$ is reflexive, hence $\mathrel{\substack{\prec\\\succ}}$ is also reflexive. We will use $\mathrel{\substack{\prec\\\succ}}$ as a relation on both, sequences of bits and sequences of blocks.

The uniqueness of encoding `e1` for all states reachable from a single program run is implied by the following, stronger property.

Proposition 5.3. *If a program run is valid by Definition 5.2, then, for any two distinct generator states queried in the run, the following holds. Let paths c_1 and c_2 represent the two states, and (p_1, n_1) and (p_2, n_2) be their encodings using encoding `e1`. Then*

$$p_1 \not\mathrel{\substack{\prec\\\succ}} p_2 \vee n_1 \neq n_2 \tag{5.2}$$

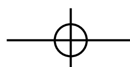
Proof. Consider a program run that is valid according to Definition 5.2. Then, if c_1 and c_2 are paths to two queried states, then c'_1 and c'_2 defined as in Definition 5.2 satisfy Condition 5.1. We perform case analysis on the alternatives of that condition.

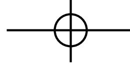
If any of the two initial alternatives is satisfied, then there is at least one more occurrence of `next` in one of c_1 and c_2 compared to the other one, hence $n_1 \neq n_2$, which means that the second alternative of Condition 5.2 is satisfied.

If any of the two final alternatives is satisfied, then $p_1 \not\mathrel{\substack{\prec\\\succ}} p_2$. □

5.1 Concrete encoding

We use the following scheme (`e2`) to encode the bit sequence and the counter, obtained from encoding `e1`, into a sequence of blocks. Zero or more initial blocks are devoted to encoding the bit sequence, while the last block encodes the counter. The following diagram shows the data layout of a sequence of blocks that encodes a path.





The bit sequence runs through the bseq_i regions from front to back and is padded with '0's in the last region if it does not occupy the whole of it. The region in the last block marked ctr contains a binary number representing the counter. The last bit of the last block is set to '1' and to '0' for all other blocks.

We ignore the situation where the value of the counter is too large to be represented by the ctr field, and leave the encoding undefined in that case. We deal with overflow in a more complex encoding that is presented in Appendix A.

Note that linear random number generation using `next` requires only one iteration of the hash function as only the value of the counter is changing, which requires changing only the last block. This way, when `next` is called repeatedly, the block cipher used as the compression function is effectively run in counter mode for generating random numbers. Splitting, on the other hand, requires rehashing two blocks at the end. The more efficient encoding that is presented in Appendix A usually requires updating only the last block on during splitting.

5.2 Compatibility with hash function

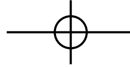
To be compatible with the hash function presented in Section 3.1, the encoding must transform any set of paths, which are reachable in a single program run that satisfies the restriction imposed by the API, into a prefix-free set of blocks. Below, we show that this property holds for paths for which the encoding is defined.

Recall that, from Proposition 5.3, if (p_1, n_1) and (p_2, n_2) are two different paths that are inspected in the same program run encoded with e1, then the following holds:

$$p_1 \npreceq p_2 \vee n_1 \neq n_2$$

We must show that if (p_1, n_1) and (p_2, n_2) satisfy the above condition, then their encodings with e2 are not each other's prefixes. We prove this fact by contrapositive. Without loss of generality, we assume that $s_1 \preceq s_2$, where $s_1 = \mathsf{e2}\ (p_1, n_1)$ and $s_2 = \mathsf{e2}\ (p_2, n_2)$. The last bit of each block of s_1 and s_2 is set to '0', except for the last block, where it is set to '1'. Therefore, the last block of s_1 cannot be equal to a non-terminal block of s_2, hence $s_1 = s_2$. The counter is uniquely determined by the ctr region, therefore $n_1 = n_2$. The sequence of bits encoded with e2 is also uniquely determined, except for a number of possible trailing '0', which are indistinguishable

thesis March 27, 2014 10:04 Page 122



from padding. Thus, p_1 and p_2 differ only by the amount of trailing zeroes, hence $p_1 \Bumpeq p_2$, which gives us $\neg(p_1 \not\Bumpeq p_2 \vee n_1 \neq n_2)$.

5.3 n-way split

The design of the generator suggested one more primitive operation `splitn`:

```
splitn :: Rand -> Word32 -> Rand
```

Calling `splitn g` yields 2^{32} new generator states derived from `g`, which can be accessed by applying the resulting function to numbers $0 \ldots 2^{32} - 1$. Consistently with the original API, we require that only one of the operations `next`, `split` and `splitn` can be called on a given generator state.

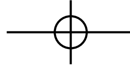
The n-way split operation can be used to efficiently create many derived generator states when their number is known in advance. For example, an array of random numbers can be efficiently generated using `splitn`. Another scenario when `splitn` might be useful is generating random functions in QuickCheck, which uses a sequence of `split` operations to derive a generator state, which is uniquely determined by an argument that was applied to the random function.

Semantically, `splitn` can be expressed in terms of a sequence of 32 `split` operations, and selecting one of the 2^{32} possible derived states, determined by the `splitn` argument. An efficient implementation, however, would instead add all 32 bits at once to the bit sequence that encodes the path to the generator state.

6 Performance

Cryptographic techniques have long been known for producing high-quality random numbers. However, their perceived low performance has been a barrier for their adoption. The original goal for this work has been to assess whether a splittable PRNG based on a block cipher can give acceptable performance so that it can be proposed to be the default PRNG in Haskell.

The proposed PRNG (Section 3), whose implementation we will refer to as `TFGen` in this section, uses a cryptographic block cipher to generate random numbers. We chose the 256-bit ThreeFish [Ferguson et al., 2010] block cipher, which is efficiently implementable in software. Despite the large block size, the encryption of a single block takes less time than for 128-bit AES, which is a standard contemporary block cipher. In addition, ThreeFish does not require an additional costly key setup phase, which is required by AES when a new encryption key is used. The actual implementation of the cipher used is a simplified version of the one from the



Skein C reference implementation[6], which is accessed from Haskell using the Foreign Function Interface.

The encoding used in TFGen is the one presented in Appendix A. We chose to use 64 bits for the bit sequence in each block (bseq_i) and another 64 bits for the counter (ctr_i), leaving 128 bits unused. Choosing 64 bits for the bit sequence means that rehash is needed every 64 splits, which brings the cost of doing that to below 2%[7] in split-intensive benchmarks. Similarly, overflow of a 64 bit counter will happen very rarely and have a negligible impact on performance. Using more bits for representing the bit sequence or the counter would, on the other hand, likely cause more overhead than give benefits.

One difference between the benchmarked code and the encoding from Appendix A is that if `next` is called repeatedly, it returns subsequent words from a single generated block, generating a new block every 8 calls. Implementing this particular feature requires only small changes to the encoding.

6.1 QuickCheck

The primary application that we considered for the proposed PRNG is random testing tool QuickCheck. Currently, QuickCheck uses the default splittable PRNG in Haskell, `StdGen`, for generating random test data. QuickCheck's random data generators make heavy use of splitting, in order to avoid generating parts of values that are never inspected.

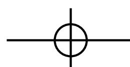
Figure 3.5 shows the relative performance of some typical properties that test implementations of an ordered set (`S.*`) and heap (`H.*`) data structures, taken from the `examples` folder in the QuickCheck distribution. In addition, two last properties used randomly generated functions, whose generators are also provided in QuickCheck. The set and heap properties execute 3–11% slower (8% slower on average) with `TFGen`, compared to `StdGen`. The two last properties, on the other hand, execute about 9% quicker.

The likely explanation for this is that QuickCheck's random function generators perform a large number of splits. As micro benchmarks presented in Figure 3.6 suggest, `split` is executed 2.3x faster by `TFGen` than by `StdGen`.

On the other hand, operation `next` is over 30% slower with `TFGen` (`micro 0`) when it is called for isolated states, and never called twice in a sequence. Such a situation occurs often in QuickCheck generators, which can explain the slow down in the properties. However, executing subsequent `next` operations is much less expensive in `TFGen` as it only requires reading the

[6] `http://www.schneier.com/skein.html`

[7] All measurements were performed using GHC 7.6.2 targeting x86-64 architecture on Intel XEON E5620 processor (Westmere-EP) clocked at 2.4 GHz.



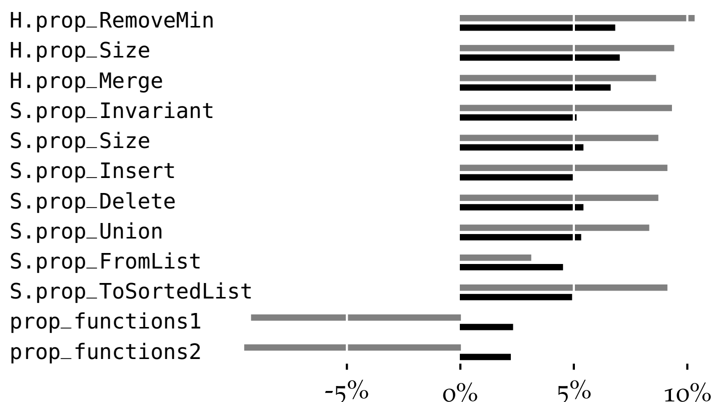


Figure 3.5: Relative slowdown (speedup) of example QuickCheck properties using `TFGen` compared to baseline runs with `StdGen` (▬). Black bars (▬) indicate how much time has been spent in the ThreeFish cipher by `TFGen`.

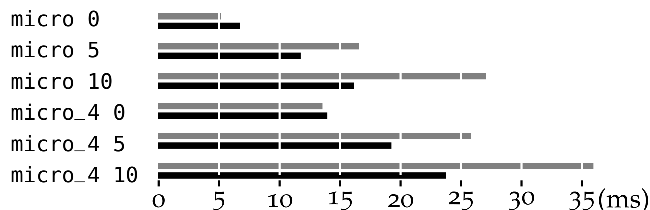
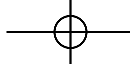


Figure 3.6: Run times of micro benchmarks for `StdGen` (▬) and `TFGen` (▬). Benchmark `micro` n executes 20k `next` operations and $(n+1)$20k `split` operations. Benchmark `micro_4` is the same, except that it runs 4 `next` operations in sequence instead of one in each case.

next word from the block, and regenerating the block once every 8 words, which is confirmed by the `micro_4` benchmark.

Benchmarks using QuickCheck (Figure 3.5) also contain an estimation of the percentage of time consumed by computing the ThreeFish block cipher. The estimation has been obtained by running the properties with a modified version of `TFGen`, which runs the block cipher four times, instead of one, and discards three of the results. As can be seen from the figure, the cost of running ThreeFish is very low for all of the properties, which indicates that the run time cannot be improved much by speeding up the cipher.



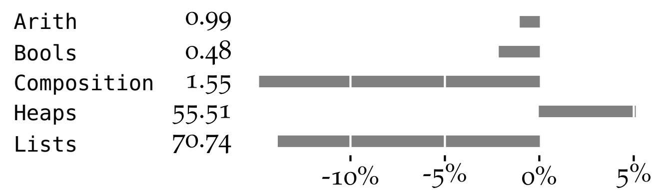


Figure 3.7: Run times of QuickSpec (in s) on its example problems with `StdGen` and relative `TFGen` performance (▬).

6.2 QuickSpec

QuickSpec [Claessen et al., 2010] is a tool that discovers an equational specification of Haskell code based on the behaviour it observes through random testing. Testing is performed with the use of QuickCheck's random data generators, and usually consumes a significant portion of total run time of QuickSpec.

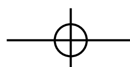
Figure 3.7 shows run times of QuickSpec version 0.9 on examples from its `examples` folder. As seen in the figure, `Arith` and `Bools` perform the same with both generators, `Lists` and `Composition` are about 13–15% faster with `TFGen`, and `Heaps` is are about 5% slower with it. QuickSpec relies on generating random valuation functions for testing, which require executing many `split` operations. This is likely to have equalised the advantage `StdGen` has over `TFGen`. In addition, `Composition` and `Lists` examples contain higher-order functions in their signatures, necessitating random generation of even more functions.

6.3 Linear generation

The standard `StdGen` generator is considered to be very slow for linear random number generation. Its slowness to a large extent comes from using standard `Random` instances, which serve as high-level primitives for generating random values of different types using the numbers returned by the random generator. On top of that, code using `StdGen` usually uses the Haskell lazy list as the intermediate data structure, which adds additional overhead.

To benchmark linear generation we decided to sidestep the `Random` instances and generate numbers directly using `next`. To remove the overhead caused by using lists we used the `vector`[8] package, which allows for efficient generation of unboxed vectors.

[8] `http://hackage.haskell.org/package/vector`



thesis March 27, 2014 10:04 Page 126

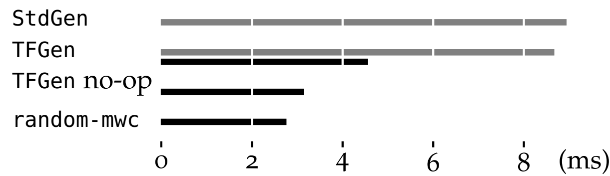


Figure 3.8: Run time of generating a vector of 100k 32-bit integers (in ms) using `next` (▬), and functions native for each generator (▬).

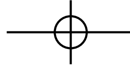
Figure 3.8 shows run times for generating an unboxed vector of 100k 32-bit random numbers. The grey (▬) bars show results for code that used the `next` method from `RandomGen` class, implemented by both generators. `StdGen` returns `Int` values from range 0–2147483562. `TFGen`, on the other hand, generates the full range of `Word32` values, which are then transformed into `Int` values from `StdGen`'s range, in order to be able to benchmark it with `Random` instances, which were written for `StdGen`. Results shown in black (▬) were obtained using code that directly generated `Word32` values.

As shown in the figure, `TFGen` is marginally faster than `StdGen` at generating numbers using `next`. However, `TFGen` performs almost twice as fast when directly generating `Word32` values. The observed difference is most likely due to the code that transforms the results into the smaller range, which is simple, and yet appears to prevent some optimisations to be performed by the compiler, such as *fusion* [Leshchinskiy, 2009]. We did not see the need to improve that code, as its only function was benchmarking that also used the `Random` instances, which themselves impose considerable overhead. However, it suggests that much performance is to be gained by giving that code, and the `Random` instances, some attention.

We chose the `random-mwc` package, which is considered to be the fastest linear PRNG for Haskell, as the baseline for comparing the 'raw' random generation speed. As shown in the figure, `TFGen` is slower by 65% than `random-mwc` in generating 32-bit random numbers. The fastest reported performance for `random-mwc` is still higher than the one measured by us, at 16.7 ns per 32-bit number (with an unreported CPU)[9], which would correspond to 1.67 ms in the figure.

The entry marked '`TFGen` no-op' in the figure is the run time of the `TFGen` generator with the code running the actual cipher replaced by a very cheap operation (XOR). This version of `TFGen` is only slower by 1.41 ms, suggesting that the generator's performance is heavily affected by book-keeping computations.

[9] `http://www.serpentine.com/blog/2011/03/18/a-little-care`



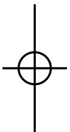
6.4 Conclusion

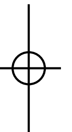
We found that an implementation of a high-quality splittable PRNG based on a cryptographic block cipher can have competitive performance with respect to traditional PRNGs. We found our implementation (`TFGen`) to be, on average, 8% slower than `StdGen` on typical QuickCheck properties and about 9% faster on properties involving random functions, and we observed a similar speedup with QuickSpec. We found that linear random number generation with `TFGen` is slower by 65% than with `random-mwc`, a state-of-the-art linear random number generator for Haskell.

Measurements performed by us suggest that it will be possible to improve the performance of `TFGen` in the future by optimising the `Random` instances and making sure that the generator's code gets properly optimised together with the code that uses it.

7 Discussion and future work

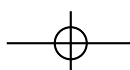
Splittable prngs are keyed hash functions We showed that a splittable PRNG can be constructed using a keyed hash function in a straightforward way. The crucial observation that allowed this was that the property that is expected from keyed hash functions, that they are indistinguishable from random mappings to computationally-bounded programs, is exactly the property we need for splittable PRNGs. Similarly, a keyed hash function can be constructed based on a splittable PRNG with a reasonable efficiency by mapping each possible input to the hash function to a unique sequence of PRNG operations. In that case, the security of the keyed hash function would depend on the pseudorandomness of the PRNG. Based on this observation, both these constructions appear to *solve the same problem*.

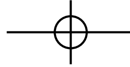




Bounds As shown in Section 4, the presented splittable PRNG construction has a bound on the order of $2^{-b}(q^3l + q^2l^2 + tql)$. In contrast, the bound for a linear PRNG based on a block cipher running in the CTR mode is $2^{-b}(q^2 + tq)$ [Rogaway, 2011], which can be shown using the Switching Lemma, mentioned in Section 4.3. Thus, by using a splittable PRNG we have to trade a worse bound for the flexibility that splitting provides. The bound on the splittable PRNG used as a linear generator trivialises to $2^{-b}(q^3 + tq)$, which is a worse result than the bound derived directly. It may be possible to improve our analysis and get a better bound for the generator.

Analysis performed in [Bellare et al., 1996] for the Merkle-Damgård construction shows that the best bound that can be achieved with this construction is on the order of $2^{-b}lq^2$, if the compression function is modelled





as a pseudorandom function. However, other similar constructions can provide better bounds (see below).

Alternative hashing constructions Thanks to basing a splittable PRNG on hashing, any keyed hash function can be used for its construction. Keyed hash functions are commonly used as Message Authentication Codes (MACs), and there is a large variety of constructions available.

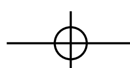
One construction that we considered is CBC-MAC [Bellare et al., 2005], which is a standard way of creating a MAC on top of a block cipher. Its mode of operation is very similar to the construction used by us, and it also requires a prefix-free set of inputs. The main difference from our construction is that the intermediate state of CBC-MAC contains both a block with the result of previous block cipher run, and the key, which needs to be kept during the entire computation. In practical terms, this would mean that the state of the generator would have to keep one more pointer to the key that would be shared by all states. On the other hand, a better bound has been proved for CBC-MAC than for our construction, namely on the order of $2^{-b}lq^2$ [Bellare et al., 2005] (when $l < 2^{b/3}$). Therefore, the trade-off between slightly slower performance and better bound should be explored.

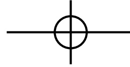
A variation of this construction, called ECBC-MAC, has even better bound $2^{-b}l^{o(1)}q^2$, where $l < 2^{b/4}$ and $o(1)$ is a diminishing function. However, ECBC-MAC requires an extra encryption step at the end of hashing, which would impose large overhead in a splittable PRNG.

Prefix-freedom Another design choice that we considered was whether to use a hash function, which requires the set of its inputs queried in one run to be prefix-free, or a hash function that does not have this requirement. Hash functions that do not require this are, in general, less efficient (for example ECBC-MAC), and usually need to perform more work at the end of hashing, which makes them less suitable for our application.

A related issue concerns the API requirement that `next` and `split` cannot be called on the same state. This requirement results in the set of paths to states queried in one program run to be prefix-free. Relaxing this requirement would be possible, but would require another, possibly less efficient encoding. However, we found that this API requirement is reasonable for the reason of compositionality, not performance. Consider a composable random data generator [Claessen and Hughes, 2000] that generates random lists given a generator for their elements, and another generator for integers:

```
myList :: Gen a -> Gen [a]
```



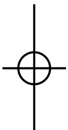


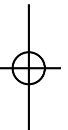

```
myInt  :: Gen Int
```

Generator `myList` will internally perform some number of `split` operations, which may depend on random numbers that it had consumed itself. Then it will use its argument generator to generate elements of the list, giving a different generator state to it each time. Now consider that the API requirement has been dropped, and that of two different states given to the argument generator, one may be derived from the other. If the argument generator also performs the `split` operation, as would be the case when we call `myList (myList myInt)`, then two equal states may be used in different places of the program, leading to the same numbers being generated.

Thus, to be safe `myList` must make sure that it does not call the argument generator with states that may have been derived from each other. The easiest way to ensure that is to never call `split` on a state which is passed to a subcomputation, which is essentially the strategy used for satisfying the original API requirement. It is thus likely that compositional use of the API was the reason for this requirement to be created.

8 Related work





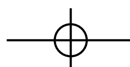
Splittable PRNGs based on a Linear Congruential Generator (LCG) proposed by [Frederickson et al., 1984] ensured that a number of right-sequences of bounded length starting from a single left-sequence are disjoint. Mascagni et al. discuss a number of traditional PRNGs (LCGs and others) that can be used as parallel generators. However, none of these two works supports unlimited on-demand splitting.

[Burton and Page, 1992] discuss a number of splittable PRNGs for Haskell, based on an LCG. The ideas include (1) distributing halves of the random sequence, which would exhaust the sequence very fast, (2) using a non-deterministic solution, and (3) randomly jumping in the sequence on split. The last solution looks promising, but the only statistical argument presented in favour of it is the measured lack of local state collisions. We implemented the last solution and found it to be slower than our generator.

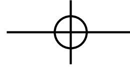
The idea about using a block cipher to implement a splittable PRNG has appeared on the Haskell-Cafe mailing list [Peyton-Jones et al., 2010]. The proposed design keeps a block cipher's key and a counter and implements `split` as follows, by regenerating the key in the right derived state at `split` operation.

$$\texttt{split}\ (k, n) = ((k, n+1), (\mathsf{enc}_k\ (n), 0))$$

The rationale behind it is that the randomness of block cipher encryption will carry over to the whole construction. However, it is not formally



thesis March 27, 2014 10:04 Page 130



justified why it is the case, or how many splits can be executed without compromising the randomness. In our view the scheme *is* correct, since in fact it is an instance of the generator proposed by us, using a suitable encoding, and thus is covered by our correctness argument. Its disadvantage is the high cost of `split` due to having to run the block cipher for each right derived generator state.

Random number generators specified in NIST SP 800-90A [Barker and Kelsey, 2012] are designed to provide unpredictable random input for cryptographic applications. The generators have pseudorandom cores (DRBGS[10]), each of which is based on a different cryptographic primitive, such as a block cipher, a keyed hash function (HMAC), a non-keyed hash function or Eliptic Curves (EC).

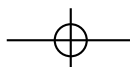
The generators can be seeded using other generators' pseudorandom output, which been used for implementing splitting in the crypto-api Hackage package.[11] Unfortunately, while all the generators appear to be correct, the NIST publication does not formalise or prove any aspects of any of them. The pseudorandom parts of the HMAC and EC generators have been analysed elsewhere [Brown and Gjøsteen, 2007, Hirose, 2009], however the proofs do not cover seeding one generator using another generator's output. Furthermore, purely deterministic (up to initial seeding) functionality is likely not to be the main focus of these generators, as cryptographic applications require frequent reseeding with external entropy [Barker and Kelsey, 2012].

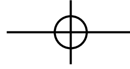
Micali and Schnorr [1991] present a PRNG based on the rsa cryptosystem, which is provably random and supports n-way splitting. However, the randomness proofs are asymptotic, which means that they do not indicate what parameters to choose to achieve a particular level of indistinguishability, or whether the generator can practically achieve reasonable randomness [Chatterjee et al., 2012, Sidorenko and Schoenmakers, 2005].

Leiserson et al. [2012] propose a PRNG for deterministic parallel random number generation. Their generator is integrated with the MIT Cilk platform, which tracks the call path of the current program location, which is then used to generate random numbers in a way that is independent of thread scheduling. To generate random numbers, the generator hashes the call path using a specially-constructed hash function. The hash function first compresses the path minimising the likelihood of collisions, and then applies a mixing operation. The compressing function ensures that the probability of collisions is low, while the mixing function provides randomisation. It is only the former of these properties that is formally stated and

[10] Deterministic Random Bit Generators

[11] `http://hackage.haskell.org/packages/archive/crypto-api/0.12.2.1/doc/html/Crypto-Random.html`





proved. The quality of the generated random numbers obviously depends on the quality of the mixing function, but it is hard to say what level of randomness it provides, especially that the presented results of statistical tests include failures.

The generator supports paths of length up to 100 (or similar value), due to the fact that the whole path must hashed when a random number is requested, and that a vector of random 'seed' values of the same length as the path is required. Thus, the general construction could not be considered to use bounded space, although the paper considers adapting it into an incremental one, as well as using alternative methods for path compression.

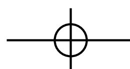
Salmon et al. [Salmon et al., 2011] present a high-performance PRNG based on a block cipher running in CTR mode. Their generator is parallelisable, but does not support splitting on demand. Their proposed generator solves two problems of traditional PRNGs, namely that they are difficult to parallelise and that their quality is unproven, and often low. The proposed generator is correct, but the randomness claims are only stated informally. The authors consider a number of block ciphers, such as ThreeFish and AES, but also their weakened versions, which offer higher performance. Finally, they propose a parallel random number generation API, which separates keeping track of counters from random number generation. The generator does not provide functionality equivalent to on-demand splitting, as it is the application that is responsible for 'distributing' the independent random streams.

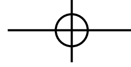
9 Conclusion

In this paper, we show that cryptographic keyed hash functions are attractive means for implementing splittable PRNGs. While the general construction of a splittable PRNG shown by us can be based on any keyed hash function, we propose using a well-known and efficient keyed hash function based on a cryptographic block cipher.

The hash function itself is based on a provably secure construction, which is guaranteed to yield high-quality randomness under the assumption that a secure block cipher is used. Our Haskell implementation is only marginally slower than Haskell's default splittable PRNG, which makes it a promising drop-in replacement.

The proposed design also suggests a new operation `splitn`, which would speed up some of the split-intensive code, and could be added to the API of splittable PRNGs in Haskell.





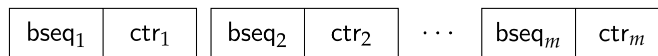
Acknowledgements

This research has been supported by the Resource Aware Functional Programming (RAW FP) grant awarded by the Swedish Foundation for Strategic Research.

A Appendix. Definitions and Proofs

We present an efficient encoding of paths containing the `next` operation. The encoding uses the general idea of keeping the bit sequence and counter separate, as does e1.

We use the following scheme (e4) to encode the bit sequence and the counter in a sequence of blocks. Each block has a fixed region for encoding part of the bit sequence and another one for encoding the counter. The following diagram shows the data layout of a sequence of blocks that encodes a path.



Regions marked ctr_i contain binary numbers ranging from 0 to N. Regions marked bseq_i contain B-bit segments of a bit sequence, except the last segment, which may be shorter. The main part of the encoding is defined inductively, as a function e3 mapping a path into a sequence of pairs of numbers and sequence of bits, each pair representing a block. The following definition assumes that $\mathsf{e3}\ (c) = \langle \ldots, (b, n) \rangle$ for recursive cases.

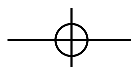
$$\mathsf{e3}\ (\langle\rangle) = (\langle\rangle, 0) \tag{A.3a}$$

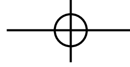
$$\mathsf{e3}\ (c \,\|\, \langle \mathtt{left} \rangle) = \begin{cases} \langle \ldots, (b \,\|\, \langle 0 \rangle, n) \rangle & \text{if } |b| < B, \\ \langle \ldots, (b, n), (\langle 0 \rangle, 0) \rangle & \text{otherwise.} \end{cases} \tag{A.3b}$$

$$\mathsf{e3}\ (c \,\|\, \langle \mathtt{right} \rangle) = \begin{cases} \langle \ldots, (b \,\|\, \langle 1 \rangle, n) \rangle & \text{if } |b| < B, \\ \langle \ldots, (b, n), (\langle 1 \rangle, 0) \rangle & \text{otherwise.} \end{cases} \tag{A.3c}$$

$$\mathsf{e3}\ (c \,\|\, \langle \mathtt{next} \rangle) = \begin{cases} \langle \ldots, (b, n+1) \rangle & \text{if } n < N', \\ \langle \ldots, (b \,\|\, \langle 1 \rangle, 0) \rangle & \text{if } n = N' \text{ and } |b| < B, \\ \langle \ldots, (b, N), (\langle\rangle, 0) \rangle & \text{otherwise.} \end{cases} \tag{A.3d}$$

Adding an operation at the end of the path changes only the last block of the resulting sequence and possibly adds another one after it. Operations `left` and `right` add one bit to the segment in the last block. If the segment



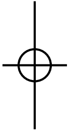


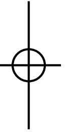
is full, a new block is started. Operation next increments the counter in the last block. Valid values of the counter are $0 \ldots N' = N - 1$. In the event of an overflow, bit '1' is added to the last segment. If the segment is full, special value N is used as the counter and a new block is added.

The result of the function is transformed into actual sequence of blocks by encoding the numbers in binary and putting the segments verbatim. The last incomplete segment is zero-padded. Let to_block be the function that turns a pair into a block. We omit it's definition, but we note the important property that we require from it.

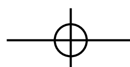
Proposition A.1. *Let (b_1, n_1) and (b_2, n_2) be two inputs to to_block function. If $n_1 \neq n_2$ or $b_1 \prec b_2$ and $b_2 \neq b_1 \parallel \langle 0, \ldots, 0 \rangle$[12] or $b_1 \not\preceq b_2$ then to_block $(b_1, n_1) \neq$ to_block (b_2, n_2).*

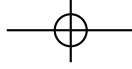
Note that linear random number generation using next requires only one iteration of the hash function for most random numbers as only the counter in the last block is changing. This way, when next is called repeatedly, the block cipher used as the compression function is effectively run in counter mode for generating random numbers. Similarly, splitting usually requires updating only the last block.

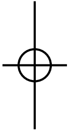
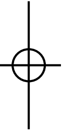


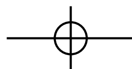


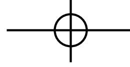
[12]We use the shorthand $b_2 \neq b_1 \parallel \langle 0, \ldots, 0 \rangle$ to denote that b_2 is not b_1 extended with some number of zeroes.





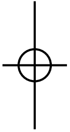
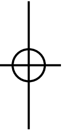




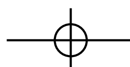
Paper IV

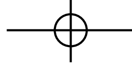
Ranking Programs using Black Box Testing

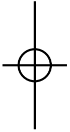
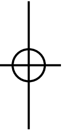
Koen Claessen **John Hughes** **Michał H. Pałka**
Nick Smallbone **Hans Svensson**

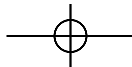



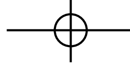
This in a revised version of a paper that appeared in the International Workshop on Automation of Software Test (AST), 2010.









Paper IV: Ranking Programs using Black Box Testing

Abstract

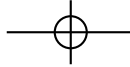
We present an unbiased method for measuring the relative quality of different solutions to a programming problem. Our method is based on identifying possible bugs from program behaviour through black-box testing. The main motivation for such a method is its use in experimental evaluation of software development methods. We report on the use of our method in a small-scale such experiment, which was aimed at evaluating the effectiveness of property-based testing vs. unit testing in software development.

1 Introduction

Property-based testing is an approach to testing software against a formal specification, consisting of universally quantified *properties* which supply both test data generators and test oracles. QuickCheck is a property-based testing tool first developed for Haskell [Claessen and Hughes, 2000], and which forms the basis for a commercial tool developed by Quviq [Arts et al., 2006]. As a simple example, using QuickCheck, a programmer could specify that list reversal is its own inverse like this,

```
prop_reverse (xs :: [Integer]) =
  reverse (reverse xs) == xs
```

which defines a property called `prop_reverse` which is universally quantified over all lists of integers `xs`. Given such a property, QuickCheck generates random values for `xs` as test data, and uses the body of the property as an oracle to decide whether each test has passed. When a test fails, QuickCheck *shrinks* the failing test case, searching systematically for a minimal failing example, in a way similar to delta-debugging [Zeller



and Hildebrandt, 2002]. The resulting minimal failing case usually makes diagnosing a fault easy. For example, if the programmer erroneously wrote

```
prop_reverse (xs :: [Integer]) =
  reverse xs == xs
```

then QuickCheck would report the minimal counterexample `[0,1]`, since at least two different elements are needed to violate the property, and the two smallest different integers are `0` and `1`.

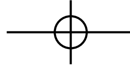
The idea of testing code against general properties, rather than specific test cases, is an appealing one which also underlies Tillmann and Schulte's *parameterized unit tests* [Tillmann and Schulte, 2005] and the Pex tool [Tillmann and de Halleux, 2008] (although the test case generation works differently). We believe firmly that it brings a multitude of benefits to the developer, improving quality and speeding development by revealing problems faster and earlier. Yet claims such as this are easy to make, but hard to prove. And it is not *obvious* that property-based testing must be superior to traditional test automation. Among the possible *disadvantages* of QuickCheck testing are:

- it is often necessary to write *test data generators* for problem-specific data structures—code which is not needed at all in traditional testing.
- the developer must *formulate a formal specification*, which is conceptually more difficult than just predicting the correct output in specific examples.
- randomly generated test cases might potentially be less effective at revealing errors than carefully chosen ones.

Thus an empirical comparison of property-based testing against other methods is warranted.

Our overall goal is to evaluate property-based testing as a development tool, by comparing programs developed by students using QuickCheck for testing, against programs developed for the same problem using HUnit [Herington, 2010]—a unit testing framework for Haskell similar to the popular JUnit tool for Java programmers [JUnit.org, 2010]. We have not reached this goal yet—we have carried out a small-scale experiment, but we need more participants to draw statistically significant conclusions. However, we have identified an important problem to solve along the way: how should we *rank* student solutions against each other, without introducing experimental bias?

Our intention is to rank solutions by testing them: those that pass the most tests will be ranked the highest. But the choice of *test suite* is critical.



It is tempting to use QuickCheck to test student solutions against our own properties, using the proportion of tests passed as a measure of quality—but this would risk experimental bias in two different ways:

- By using one of the tools in the comparison to grade solutions, we might unfairly bias the experiment to favour that tool,
- The ranking of solutions could depend critically on the distribution of random tests, which is rather arbitrary.

Unfortunately, a manually constructed set of test cases could also introduce experimental bias. If we were to include many similar tests of a particular kind, for example, then handling that kind of test successfully would carry more weight in our assessment of solutions than handling other kinds of test.

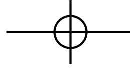
Our goal in this paper, thus, is to develop a way of ranking student solutions by testing that leaves no room for the experimenter's bias to affect the result. We will do so by generating a set of test cases *from the submissions themselves*, based on a simple 'bug model' presented in section 3, such that each test case tests for one bug. We then rank solutions by the number of bugs they contain. QuickCheck is used to help find this set of test cases, but in such a way that the distribution of random tests is of almost no importance.

Contribution The main contribution of this paper is the ranking method we developed. As evidence that the ranking is reasonable, we also present the results of our small-scale experiment, in which solutions to three different problems are compared in this way.

The remainder of the paper is structured as follows. In the next section we briefly describe the experiment we carried out. In section 3 we explain and motivate our ranking method. Section 4 analyses the results obtained. In section 5 we discuss related work, and we conclude in section 6.

2 The Experiment

We designed an experiment to test the hypothesis that *"Property-based testing is more effective than unit testing, as a tool during software development"*, using QuickCheck as the property-based testing tool, and HUnit as the unit testing tool. We used a *replicated project study*[Basili et al., 1986], where in a controlled experiment a group of student participants individually solved three different programming tasks. We planned the experiment in accordance to best practice for such experiments; trying not to exclude



participants, assigning the participants randomly to tools, using a variety of programming tasks, and trying our best not to influence the outcome unnecessarily. We are only evaluating the final product, thus we are not interested in process aspects in this study.

In the rest of this section we describe in more detail how we planned and executed the experiment, we also motivate the choice of programming assignments given to the participants.

2.1 Experiment overview

We planned an experiment to be conducted during one day. Since we expected participants to be unfamiliar with at least one of the tools in the comparison, we devoted the morning to a training session in which the tools were introduced to the participants. The main issue in the design of the experiment was the programming task (or tasks) to be given to the participants. Using several different tasks would yield more data points, while using one single (bigger) task would give us data points of higher quality. We decided to give three separate tasks to the participants, mostly because by doing this, and selecting three different types of problems, we could reduce the risk of choosing a task particularly suited to one tool or the other. All tasks were rather small, and require only 20-50 lines of Haskell code to implement correctly.

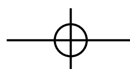
To maximize the number of data points we decided to assign the tasks to individuals instead of forming groups. Repeating the experiments as a pair-programming assignment would also be interesting.

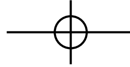
2.2 Programming assignments

We constructed three programming assignments. We tried to choose problems from three separate categories; one data-structure implementation problem, one search/algorithmic problem, and one slightly tedious string manipulation task.

Problem 1: E-mail anonymizer In this task the participants were asked to write a sanitizing function `anonymize` which blanks out E-mail addresses in a string. For example,

```
anonymize "pelle@foretag.se" ==
  "p____@f______.s_"

anonymize "Hi johnny.cash@music.org!" ==
  "Hi j_____.c___@m____.o__!"
```





The function should identify all e-mail addresses in the input, change them, but leave all other text untouched. This is a simple problem, but with a lot of tedious cases.

Problem 2: Interval sets In this task the participants were asked to implement a compact representation of sets of integers based on lists of intervals, represented by the type `IntervalSet = [(Int,Int)]`, where for example the set `{1, 2, 3, 7, 8, 9, 10}` would be represented by the list `[(1,3),(7,10)]`. The participants were instructed to implement a family 7of functions for this data type (`empty`, `member`, `insert`, `delete`, `merge`). There are many special cases to consider—for example, inserting an element between two intervals may cause them to merge into one.

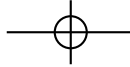
Problem 3: Cryptarithm In this task the students were asked to write a program that solves puzzles like this one:

```
  SEND
  MORE
-----+
 MONEY
```

The task is to assign a mapping from letters to (unique) digits, such that the calculation makes sense. (In the example `M = 1, O = 0, S = 9, R = 8, E = 5, N = 6, Y = 2, D = 7`). Solving the puzzle is complicated by the fact that there might be more than one solution and that there are problems for which there is no solution. This is a search problem, which requires an algorithm with some level of sophistication to be computationally feasible.

2.3 The participants

Since the university (Chalmers University of Technology, Gothenburg, Sweden) teaches Haskell, this was the language we used in the experiment. We tried to recruit students with (at least) a fair understanding of functional programming. This we did because we believed that too inexperienced programmers would not be able to benefit from either QuickCheck or HUnit. The participants were recruited by advertising on campus, email-messages sent to students from the previous Haskell-course and announcements in different ongoing courses. Unfortunately the only available date collided with exams at the university, which lowered the number of potential participants. In the end we got only 13 participants. This is too few to draw statistically significant conclusions, but on the other hand it is a rather manageable number of submissions to analyze in a greater detail. Most of



the participants were at a level where they had passed (often with honor) a 10-week programming course in Haskell.

2.4 Assigning the participants into groups

We assigned the participants randomly (by lot) into two groups, one group using QuickCheck and one group using HUnit.

2.5 Training the participants

The experiment started with a training session for the participants. The training was divided into two parts, one joint session, and one session for the specific tool. In the first session, we explained the purpose and the underlying hypothesis for the experiment. We also clearly explained that we were interested in software quality rather than development time. The participants were encouraged to use all of the allocated time to produce the best software possible.

In the second session the groups were introduced to their respective testing tools, by a lecture and practical session. Both sessions lasted around 60 minutes.

2.6 Programming environment

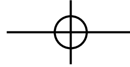
Finally, with everything set up, the participants were given the three different tasks with a time limit of 50 minutes for each of the tasks. The participants were each given a computer equipped with GHC (the Haskell compiler) [The GHC Team, 2010], both the testing tools, and documentation. The computers were connected to the Internet, but since the participants were aware of the purpose of the study and encouraged not to use other tools than the assigned testing tool it is our belief this did not affect the outcome of the experiment.[1]

2.7 Data collection and reduction

From the experiments we collected the implementations as well as the testing code written by each participant.

Manual grading of implementations Each of the three tasks were graded by an experienced Haskell programmer. We graded each implementation on a scale 0-10, just as we would have graded an exam-question. Since the

[1] Why not simply disconnect the computers from the Internet? Because we used an on-line submission system, as well as documentation and software from network file systems.



tasks were reasonably small, and the number of participants manageable, this was feasible. To prevent any possible bias, the grader was not allowed to see the testing code and thus he could not know whether each student was using QuickCheck or HUnit.

Automatic ranking The implementations of each problem were subjected to an analysis that we present in section 3.

We had several students submit uncompileable code.[2] In those cases, we made the code compile by for example removing any ill-formed program fragments. This was because such a program might be partly-working, and deserve a reasonable score; we thought it would be unfair if it got a score of zero simply because it (say) had a syntax error.

Grading of test suites We also graded participants' testing code. Each submission was graded by hand by judging the completeness of the test suite—and penalised for missing cases (for HUnit) or incomplete specifications (for QuickCheck). As we did not instruct the students to use TDD, there was no penalty for not testing a function if that function was not implemented.

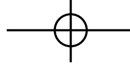
Cross-comparison of tests We naturally wanted to automatically grade students' test code too—not least, because a human grader may be biased towards QuickCheck or HUnit tests. Our approach was simply to take each student's test suite, and run it against all of the submissions we had; for every submission the test suite found a bug in, it scored one point.

We applied this method successfully to the interval sets problem. However, for the anonymizer and cryptarithm problems, many students performed white box testing, testing functions that were internal to their implementation; therefore we were not able to transfer test suites from one implementation to another, and we had to abandon the idea for these problems.

3 Evaluation Method

We assume we have a number of student *answers* to evaluate, $A_1 \ldots A_n$, and a perfect solution A_0, each answer being a program mapping a test case to output. We assume that we have a test oracle which can determine whether or not the output produced by an answer is correct, for any possible test case. Such an oracle can be expressed as a QuickCheck property—if the

[2]Since we asked students to submit their code at a fixed time, some students submitted in the middle of making changes.



correct output is unique, then it is enough to compare with A_0's output, otherwise something more complex is required. Raising an exception, or falling into a loop[3], is never correct behaviour. We can thus determine, for an arbitrary test case, which of the student answers pass the test.

We recall that the purpose of our automatic evaluation method is to find a set of test cases that is as unbiased as possible. In particular, we want to avoid counting multiple test cases that are equivalent, in the sense that they trigger the same bug.

Thus, we aim to 'count the bugs' in each answer, *using black-box testing alone*. How, then, should we define a 'bug'? We cannot refer to errors at specific places in the source code, since we use black-box testing only—we must define a 'bug' in terms of the program behaviour. We take the following as our bug model:

- A *bug* causes a program to fail for a set of test cases. Given a bug b, we write the set of test cases that it causes to fail as $BugTests(b)$. (Note that it is possible that the same bug b occurs in several different programs.)

- A *program* p will contain a set of bugs, $Bugs(p)$. The set of test cases that p fails for will be

$$FailingTests(p) = \bigcup_{b \in Bugs(p)} BugTests(b)$$

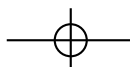
It is quite possible, of course, that two different errors in the source code might manifest themselves in the same way, causing the same set of tests to fail. We will treat these as the *same* bug, quite simply because there is no way to distinguish them using black-box testing.

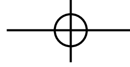
It is also possible that two different bugs in combination might 'cancel each other out' in some test cases, leading a program containing both bugs to behave correctly, despite their presence. We cannot take this possibility into account, once again because black-box testing cannot distinguish correct output produced 'by accident' from correct output produced correctly. We believe the phenomenon, though familiar to developers, is rare enough not to influence our results strongly.

Our approach is to analyze the failures of the student answers, and use them to infer the existence of possible bugs *Bugs*, and their failure sets. Then we shall rank each answer program A_i by the number of these bugs that the answer appears to contain:

$$rank(A_i) = |\{b \in Bugs \mid BugTests(b) \subseteq FailingTests(A_i)\}|$$

[3]detecting a looping program is approximated by an appropriately chosen timeout





In general, there are many ways of explaining program failures via a set of bugs. The most trivial is to take each answer's failure set $FailingTests(A_i)$ to represent a different possible bug; then the rank of each answer would be the number of other (different) answers that fail on a strictly smaller set of inputs. However, we reject this idea as too crude, because it gives no insight into the *nature* of the bugs present. We shall aim instead to find a more refined set of possible bugs, in which each bug explains a small set of 'similar' failures.

Now, let us define the failures of a *test case* to be the set of answers that it provokes to fail:

$$AnswersFailing(t) = \{A_i \mid t \in FailingTests(A_i)\}$$

We insist that if two test cases t_1 and t_2 provoke the same answers to fail, then they are equivalent with respect to the bugs we infer:

$$\begin{aligned} &AnswersFailing(t_1) = AnswersFailing(t_2) \Longrightarrow \\ &\quad \forall b \in Bugs.\ t_1 \in BugTests(b) \Leftrightarrow t_2 \in BugTests(b) \end{aligned}$$

We will not distinguish such a pair of test cases, because there is no evidence from the answers that could justify doing so. Thus we can partition the space of test cases into subsets that behave equivalently with respect to our answers. By identifying bugs with these partitions (except, if it exists, the partition which causes no answers to fail), then we obtain a maximal set of bugs that can explain the failures we observe. No other set of bugs can be more refined than this without distinguishing inputs that should not be distinguished.

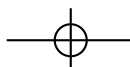
However, we regard this partition as a little *too* refined. Consider two answers A_1 and A_2, and three partitions B, B_1 and B_2, such that

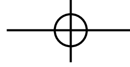
$$\begin{aligned} &\forall t \in B.\ AnswersFailing(t) = \{A_1, A_2\} \\ &\forall t \in B_1.\ AnswersFailing(t) = \{A_1\} \\ &\forall t \in B_2.\ AnswersFailing(t) = \{A_2\} \end{aligned}$$

Clearly, one possibility is that there are three separate bugs represented here, and that

$$\begin{aligned} Bugs(A_1) = \{B, B_1\} \\ Bugs(A_2) = \{B, B_2\} \end{aligned}$$

But another possibility is that there are only *two* different bugs represented, $B_1' = B \cup B_1$ and $B_2' = B \cup B_2$, and that each A_i just has one bug, B_i'. In this case, test cases in B can provoke either bug. Since test cases which can provoke several different bugs are quite familiar, then we regard the latter possibility as more plausible than the former. We choose therefore to





ignore any partitions whose failing answers are the union of those of a set of other partitions; we call these partitions *redundant*, and we consider it likely that the test cases they contain simply provoke several bugs at once. In terms of our bug model, we combine such partitions with those representing the individual bugs whose union explains their failures. Note, however, that if a *third* answer A_3 only fails for inputs in B, then we consider this evidence that B does indeed represent an independent bug (since $\{A_1, A_2, A_3\}$ is *not* the union of $\{A_1\}$ and $\{A_2\}$), and that answers A_1 and A_2 therefore contain two bugs each.

Now, to rank our answers we construct a test suite containing one test case from each of the remaining partitions, count the tests that each answer fails, and assign ranks accordingly.

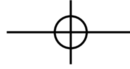
In practice, we find the partitions by running a very large number of random tests. We maintain a set of test cases *Suite*, each in a different partition. For each newly generated test case t, we test all of the answers to compute $AnswersFailing(t)$. We then test whether the testcase is redundant in the sense described above:

$$\begin{aligned}&Redundant(t, Suite) \hat{=} \\ &\quad AnswersFailing(t) = \\ &\quad \bigcup \left\{ AnswersFailing(t') \;\middle|\; \begin{array}{l} t' \in Suite, \\ AnswersFailing(t') \subseteq \\ \quad AnswersFailing(t) \end{array} \right\}\end{aligned}$$

Whenever t is not redundant, i.e. when $Redundant(t, Suite)$ evaluates to *False*, then we apply QuickCheck's shrinking to find a *minimal* t_{min} that is not redundant with respect to *Suite*—which is always possible, since if we cannot find any smaller test case which is irredundant, then we can just take t itself. Then we *add* t_{min} to *Suite*, and *remove* any $t' \in Suite$ such that $Redundant(t', (Suite - t') \cup \{t_{min}\})$. (Shrinking at this point probably helps us to find test cases that provoke a single bug rather than several—'probably' since a smaller test case is likely to provoke fewer bugs than a larger one, but of course there is no guarantee of this).

We continue this process until a large number of random tests fail to add any test cases to *Suite*. At this point, we assume that we have found one test case for each irredundant input partition, and we can use our test suite to rank answers.

Note that this method takes no account of the *sizes* of the partitions involved—we count a bug as a bug, whether it causes a failure for only one input value, or for infinitely many. Of course, the severity of bugs in practice may vary dramatically depending on precisely *which* inputs they cause failures for—but taking this into account would make our results dependent on value judgements about the importance of different kinds of



input, and these value judgements would inevitably introduce experimental bias.

In the following section, we will see how this method performs in practice.

4 Analysis

We adopted the statistical *null hypothesis* to be that there is no difference in quality between programs developed using QuickCheck and programs developed using HUnit. The aim of our analysis will be to establish whether the samples we got are different in a way which cannot be explained by coincidence.

We collected solutions to all three tasks programmed by 13 students, 7 of which were assigned to the group using QuickCheck and the remaining 6 to one using HUnit. In this section we will refer to the answers (solutions to tasks) as $A1$ to $A13$. Since the submissions have been anonymized, numbering of answers have also been altered and answers $A1$ to different problems correspond to submissions of different participants. For each task there is also a special answer $A0$ which is the model answer which we use as the testing oracle. For the anonymizer, we also added the identity function for comparison as $A14$, and for the interval sets problem we added a completely undefined solution as $A14$.

4.1 Automatic Ranking of Solutions

We ranked all solutions according to the method outlined in section 3. The ranking method produced a test-suite for each of the three tasks and assigned the number of failing tests to each answer of every task. The final score that we used for evaluation of answers was the *number of successful runs on tests from the test-suite*. The generated test suites are shown in Table 4.1. Every test in the test suite causes some answer to fail; for example `delete 0 []` is the simplest test that causes answers that did not implement the `delete` function to fail. These test cases have been shrunk by QuickCheck, which is why the only letter to appear in the anonymizer test cases is 'a', and why the strings are so short[4].

Figures 4.1 to 4.3 visualize the test results. Each node represents a set of answers which pass precisely the same tests. An arrow from one node to another means that the answers at the target of the arrow pass a subset of the tests that the answers at the source of the arrow pass. Arrows are

[4] Because Haskell encourages the use of dynamic data-structures, then none of the solutions could encounter a buffer overflow or other error caused by fixed size arrays. As a result, there is no need for tests with very long strings.

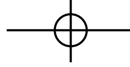
Anon	IntSet	Crypt
""	member 0 []	b+b=c
"\n"	member 0 [(-2,2)]	a+a=a
"@"	member 2 [(1,1)]	a+b=ab
"a"	member 0 [(-3,-3),(0,4)]	aa+a=bac
"&@"	insert 0 []	
".@"	insert -1 [(1,1)]	
"@@"	insert 0 [(-2,0)]	
".@@"	insert 1 [(-2,0)]	
"@_a"	insert 2 [(0,0)]	
"@a="	delete 0 []	
"_ &"	delete 0 [(0,0)]	
"a@a"	delete 0 [(0,1)]	
"#@&@"	merge [] []	
".a@#"	merge [] [(-1,0)]	
"a@_a"	merge [(0,0)] [(0,0)]	
"a@aa"	merge [(-1,0),(2,3)] [(-1,0)]	

Table 4.1: Generated test suites.

labelled with a test case that distinguishes the source and target, and the number of other such test cases in brackets. For instance, we can read from Figure 4.1 that $A2$ fails three more tests than $A7$, and that it fails on the input string "@" whereas $A7$ succeeds on it. Thus these figures visualize a 'correctness partial order' on the submitted answers.

The top node of each graph represents the entirely correct solutions, including the model answer A0. The bottom node represents incomplete solutions, in which the main functions were not defined—and which therefore fail all tests. Interestingly, our analysis *distinguishes all other answers*—no two partially correct submissions were equivalent. Moreover, there is a non-trivial partial ordering of answers in each case: some answers really are strictly better than others. We conclude that our analysis is able to classify partially correct answers in an interesting way. (We also conclude that the cryptarithm problem was too hard to solve in the time available, since more than half of the submissions failed every test).

The final score assigned to each answer is shown in figure 4.4. In order to assign better answers a higher score, we show the number of tests *passed* by each answer, rather than the number of test failures—i.e. bugs. $A0$ is the model answer in each case, and answers coming from the group assigned to using QuickCheck are marked with stars(*).



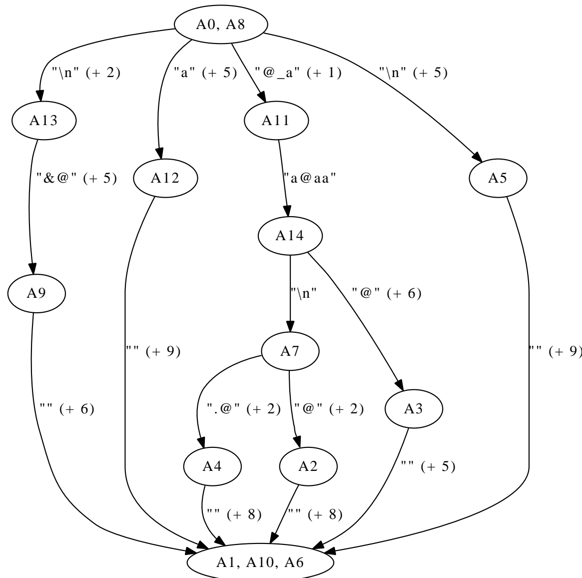


Figure 4.1: Relative correctness of anonymizer answers.

The following table shows a statistical analysis of scores from the automatic ranking. To determine whether there is a statistical difference between samples coming from the two groups we applied Welch's *t*-test (which tests whether two collections of data have the same mean) and got the values visible in the P-value row (which we shall explain below).

	Anon	IntSet	Crypto
All - Avg (Sdev)	8.15 (5.38)	9.69 (4.15)	1.15 (1.63)
QC - Avg (Sdev)	9.86 (5.01)	9.71 (4.39)	0.86 (1.57)
HU - Avg (Sdev)	6.17 (5.53)	9.67 (4.27)	1.50 (1.76)
P-value	0.2390	0.9846	0.5065

For the anonymizer example, we can see that solutions developed using QuickCheck scored higher than those developed using HUnit, for interval sets the scores were about the same, and for the cryptarithm example, then solutions developed using QuickCheck fared worse. The P-value is the probability of seeing the observed (or lower) difference in scores by sheer chance, if there is no difference in the expected score using HUnit and QuickCheck (the null hypothesis). For the anonymizer problem then the null hypothesis can be rejected with a confidence of 76%—which is

thesis March 27, 2014 10:04 Page 150

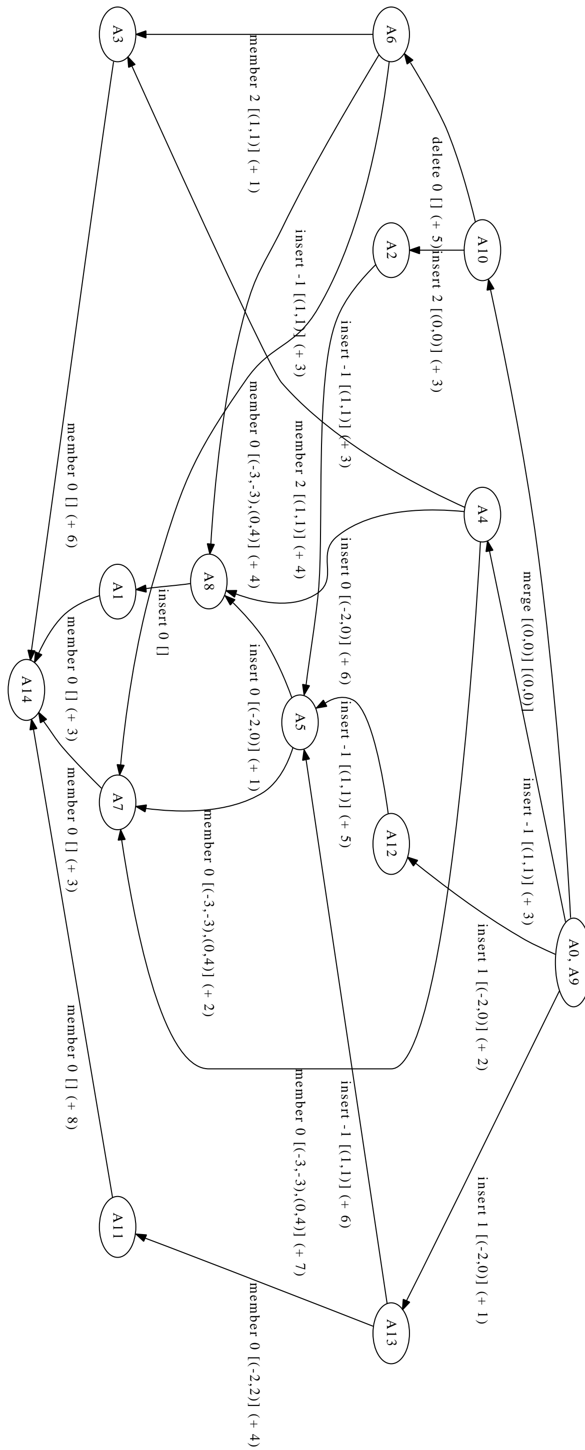
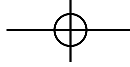


Figure 4.2: Relative correctness of interval set answers.



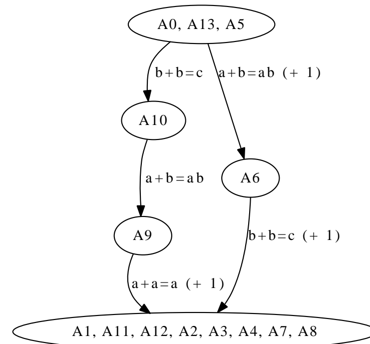


Figure 4.3: Relative correctness of cryptarithm answers.

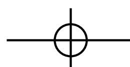
Answer	Anon	IntSet	Crypto
A0	16	16	4
A1	0*	4*	0*
A2	9*	11*	0
A3	6	7*	0*
A4	9*	12*	0*
A5	10	7	4*
A6	0	9	2*
A7	12*	4	0*
A8	16*	5*	0
A9	7	16	2
A10	0	15*	3
A11	14	9	0*
A12	10*	13	0
A13	13*	14*	4

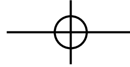
Figure 4.4: Results of automatic grading.

encouraging, but falls short of statistical significance (which would require a confidence of 95% or higher).

4.2 Stability of the automatic bug measure

Because our bug analysis does perform a random search in the space of test cases to construct its test suite, it is possible that we select a different set of tests, and thus assign a different rank to the same program in different runs. To investigate this, we ran the bug analysis ten times on the solutions to each of the three problems. We found that the partial ordering on solutions





that we inferred did not change, but the size of test suite did vary slightly. This could lead to the same answer failing a different number of tests in different runs, and thus to a different rank being assigned to it. The table below shows the results for each problem. Firstly, the number of consecutive tests we ran without refining the test suite before concluding it was stable. Secondly, the sizes of the test suites we obtained for each problem. Once a test suite was obtained, we assigned a rank to each answer, namely the number of tests it failed. These ranks did differ between runs, but no answer was assigned ranks different by more than one in different runs. The last rows show the average and maximum standard deviations of the ranks assigned to each answer.

	Anon	IntSet	Crypto
Number of tests	10000	10000	1000
Sizes of test suite	15,16	15,16	4
Avg std dev of ranks	0.08	0.06	0
Max std dev of ranks	0.14	0.14	0

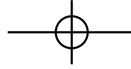
We conclude that the rank assignment is not much affected by random choices made as we construct the test suite.

4.3 Manual Grading of Solutions

In the table below we present that average scores (and their standard deviations) from the manual grading for the three problems. These numbers are not conclusive from a statistical point of view. Thus, for the manual grading we can not reject the null hypothesis. Nevertheless, there is a tendency corresponding to the results of the automatic grading in section 4.1. For example, in the E-Mail anonymizer problem the solutions that use QuickCheck are graded higher than the solutions that use HUnit.

	Anon	IntSet	Crypto
All - Avg (Sdev)	4.07 (2.78)	4.46 (2.87)	2.15 (2.91)
QC - Avg (Sdev)	4.86 (2.67)	4.43 (2.88)	1.86 (3.23)
HU - Avg (Sdev)	3.17 (2.86)	4.50 (3.13)	2.50 (2.74)

To further justify our method for automatic ranking of the solutions, we would like to see a correlation between the automatic scores and the manual scores. However, we can not expect them to be exactly the same since the automatic grading is in a sense less forgiving. (The automatic grading measure how well the program actually works, while the manual grading



measure 'how far from a correct program' the solution is.) If we look in more detail on the scores to the E-Mail anonymizer problem, presented in the table below, we can see that although the scores are not identical, they tend to rank the solutions in a very similar way. The most striking difference is for solution *A*7, which is ranked 4th by the automatic ranking and 10th by the manual ranking. This is caused by the nature of the problem. The *identity function* (the function simply returning the input, *A*14) is actually a rather good approximation of the solution functionality-wise. *A*7 is close to the identity function—it does almost nothing, getting a decent score from the automatic grading, but failing to impress a human marker.

Answer	Auto	Manual	Auto rank	Manual rank
A1	0	3	11	8
A2	9	3	7	8
A3	6	2	10	10
A4	9	5	7	4
A5	10	4	5	5
A6	0	0	11	13
A7	12	2	4	10
A8	16	9	1	1
A9	7	4	9	5
A10	0	1	11	12
A11	14	8	2	2
A12	10	4	5	5
A13	13	8	3	2

4.4 Assessment of Students' Testing

As described in section 2.7, we checked the quality of each student's test code both manually and automatically (by counting how many submissions each test suite could detect a bug in). Figure 4.5 shows the results.

The manual scores may be biased since all the authors are QuickCheck afficionados, so we would like to use them only as a 'sanity check' to make sure that the automatic scores are reasonable. We can see that, broadly speaking, the manual and automatic scores agree.

The biggest discrepancy is that student 9 got full marks according to our manual grading but only 5/11 according to the automatic grading. The main reason is that his test suite was less comprehensive than we thought: it included several interesting edge cases, such as an insert that 'fills the gap' between two intervals and causes them to become one larger interval,

thesis March 27, 2014 10:04 Page 154

	Student number						
QuickCheck	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Manual grading	0	0	0	3	9	9	12
Automatic grading	0	0	0	0	8	10	11

	Student number					
HUnit	**8**	**9**	**10**	**11**	**12**	**13**
Manual grading	3	12	6	3	6	9
Automatic grading	0	5	5	6	7	8

Figure 4.5: Manual vs automatic grading of test suite quality.

but left out some simple cases, such as `insert 2 (insert 0 empty)`. In this case, the automatic grader produced the fairer mark.

So, the automatically-produced scores look reasonable and we pay no more attention to the manual scores. Looking at the results, we see that four students from the QuickCheck group were not able to detect any bugs at all. (Three of them submitted no test code at all[5], and one of them just tested one special case of the `member` function.) This compares to just one student from the HUnit group who was unable to find any bugs.

However, of the students who submitted a useful test suite, the *worst* QuickCheck test suite got the same score as the *best* HUnit test suite! All of the HUnit test suites, as it happens, were missing some edge case or other.[6]

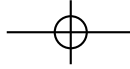
So, of the students who were using QuickCheck, half failed to submit any useful test-suite at all, and the other half's test suites were the best ones submitted. There may be several explanations for this: perhaps QuickCheck properties are harder to write but more effective than unit tests; or perhaps QuickCheck is only effective in the hands of a strong programmer; or perhaps QuickCheck properties are 'all-or-nothing', so that a property will either be ineffective or catch a wide range of bugs; or perhaps it was just a coincidence. This is something we will aim to find out in our next experiment.

5 Related Work

Much work has been devoted to finding representative test-suites that would be able to uncover all bugs even when exhaustive testing is not possible.

[5]Of course, this does not imply that these students did not test their code *at all*—just that they did not automate their tests. Haskell provides a read-eval-print loop which makes interactive testing quite easy.

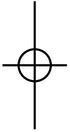
[6]Functions on interval sets have a surprising number of edge cases; with QuickCheck, there is no need to enumerate them.

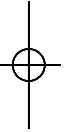


When it is possible to divide the test space into partitions and assert that any fault in the program will cause one partition to fail completely it is enough select only a single test case from each partition to provoke all bugs. The approach was pioneered by Goodenough and Gerhart[Goodenough and Gerhart, 1975] who looked both at specifications and the control structure of tested programs and came up with test suites that would exercise all possible combinations of execution conditions. Weyuker and Ostrand[Weyuker and Ostrand, 1980] attempted to obtain good test-suites by looking at execution paths that they expect to appear in an implementation based on the specification. These methods use other information to construct test partitions, whereas our approach is to find the partitions by finding faults in random testing.

Lately, test-driven development has gained in popularity, and in a controlled experiment from 2005 [Erdogmus et al., 2005] Erdogmus et. al. compare its effectiveness with a traditional test-after approach. The result was that the group using TDD wrote more test cases, and tended to be more productive. These results are inspiring, and the aim with our experiment was to show that property-based testing (using QuickCheck) is a good way of conducting tests in a development process.

In the design of the experiments we were guided by several texts on empirical research in software engineering, amongst which [Basili et al., 1986, Kitchenham et al., 2002, Wohlin et al., 2000] were the most helpful.

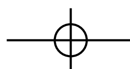


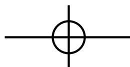


6 Conclusions

We have designed an experiment to compare property-based testing and conventional unit testing, and as part of the design we have developed an unbiased way to assess the 'bugginess' of submitted solutions. We have carried out the experiment on a small-scale, and verified that our assessment method can make fine distinctions between buggy solutions, and generates useful results. Our experiment was too small to yield a conclusive answer to the question it was designed to test. In one case, the interval sets, we observed that all the QuickCheck test suites (when they were written) were more effective at detecting errors than any of the HUnit test suites. Our automated analysis suggests, but does not prove, that in one of our examples, the code developed using QuickCheck was less buggy than code developed using HUnit. Finally, we observed that QuickCheck users are less likely to write test code than HUnit users—even in a study of automated testing—suggesting perhaps that HUnit is easier to use.

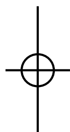
The main weakness of our experiment (apart from the small number of subjects) is that students did not have enough time to complete their

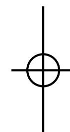


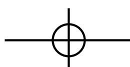


answers to their own satisfaction. We saw this especially in the cryptarithm example, where more than half the students submitted solutions that passed no tests at all. In particular, students did not have time to complete a test suite to their own satisfaction. We imposed a hard deadline on students so that development time would not be a variable. In retrospect this was probably a mistake: next time we will allow students to submit when they feel ready, and measure development time as well.

In conclusion, our results are encouraging and suggest that a larger experiment could demonstrate interesting differences in power between the two approaches to testing. We look forward to holding such an experiment in the future.







Bibliography

ISO C Standard 1999. Technical report, ISO, 1999. ISO/IEC 9899:1999 draft.
→ 1 citation on page: 68

J. Almström Duregård. AGATA: Random generation of test data. Master's thesis, Chalmers University of Technology, Dec. 2009.
→ 1 citation on page: 74

J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu. Random test run length and effectiveness. In *Proc. International Conference on Automated Software Engineering 2008*, pages 19–28, Washington, DC, USA, 2008. IEEE. ISBN 978-1-4244-2187-9.
→ 1 citation on page: 11

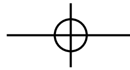
T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proc. Erlang Workshop 2006*, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1.
→ 3 citations on 3 pages: 4, 79, and 137

L. Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747, 2005.
→ 1 citation on page: 72

L. Augustsson, M. Rittri, and D. Synek. Functional pearl: On generating unique names. *J. Funct. Program.*, 4:117–123, 1 1994.
→ 1 citation on page: 107

E. Barker and J. Kelsey. NIST Special Publication 800-90a: Recommendation for random number generation using deterministic random bit generators, 2012.
→ 3 citations on 2 pages: 104 and 130

V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Trans. Softw. Eng.*, 12(7):733–743, 1986.
→ 2 citations on 2 pages: 139 and 155



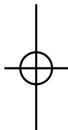
B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2nd edition, June 1990.
→ 3 citations on 2 pages: 1 and 4

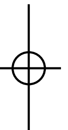
B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1st edition, 1995. ISBN 0471120944.
→ 1 citation on page: 3

M. Bellare and P. Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. http://eprint.iacr.org/2004/331.
→ 3 citations on 2 pages: 114 and 117

M. Bellare and P. Rogaway. Introduction to modern cryptography, 2005. http://www.cs.ucsd.edu/~mihir/cse207/classnotes.html.
→ 5 citations on 3 pages: 108, 114, and 118

M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: the cascade construction and its concrete security. In *Proc. Foundations of Computer Science*, pages 514–523, Los Alamitos, CA, USA, 1996. IEEE. ISBN 0-8186-7594-2.
→ 10 citations on 8 pages: 12, 103, 104, 108, 110, 114, 116, and 127





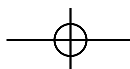
M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proc. Foundations of Computer Science, 1997*, pages 394–403, 1998.
→ 1 citation on page: 117

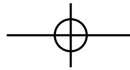
M. Bellare, K. Pietrzak, and P. Rogaway. Improved security analyses for CBC MACs. In *Proc. Advances in Cryptology — CRYPTO 2005, LNCS 3621*, pages 527–545. Springer-Verlag, 2005.
→ 2 citations on page: 128

M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, 13(4):850–864, Nov. 1984.
→ 1 citation on page: 108

O. Bodini, D. Gardy, and B. Gittenberger. Lambda terms of bounded unary height. In *Proc. 8th Workshop on Analytic Algorithmics and Combinatorics*, 2011. ISBN 978-0-898719-33-8.
→ 3 citations on 2 pages: 21 and 73

E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proc. ICSE 2013*, pages 122–131, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.
→ 1 citation on page: 11





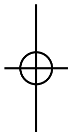
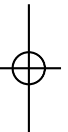
C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proc. 2002 Inl. Symp. Software Testing and Analysis (ISSTA '02)*, pages 123–133. ACM, 2002. ISBN 1-58113-562-9.
→ 2 citations on 2 pages: 96 and 97

D. R. L. Brown and K. Gjøsteen. A security analysis of the NIST SP 800-90 elliptic curve random number generator. In *Proc. Advances in Cryptology — CRYPTO '07*, pages 466–481. Springer-Verlag, 2007. ISBN 3-540-74142-9, 978-3-540-74142-8.
→ 1 citation on page: 130

F. W. Burton and R. L. Page. Distributed random number generation. *J. Funct. Program.*, 2(2):203–212, 1992.
→ 6 citations on 6 pages: 101, 103, 105, 106, 107, and 129

S.-j. Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham. *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*. NIST, 2012.
→ 1 citation on page: 109

S. Chatterjee, A. Menezes, and P. Sarkar. Another look at tightness. In *Proc. Selected Areas in Cryptography (SAC'11), LNCS. 7118*, pages 293–319, 2012.
→ 2 citations on 2 pages: 104 and 130

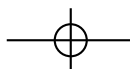
J. Christiansen and S. Fischer. Easycheck: test data for free. In *FLOPS'08*, pages 322–336. Springer, 2008.
→ 1 citation on page: 96

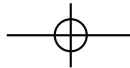
K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proc. ICFP 2000*, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.
→ 12 citations on 9 pages: 4, 5, 23, 35, 70, 79, 102, 128, and 137

K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Proc. Haskell Workshop 2002*, pages 65–77, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6.
→ 1 citation on page: 72

K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *Proc. Tests and Proofs*, TAP'10, pages 6–21. Springer-Verlag, 2010. ISBN 3-642-13976-0, 978-3-642-13976-5.
→ 1 citation on page: 125

J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. Merkle-Damgård revisited: How to construct a hash function. In *Proc. Advances in Cryptology — CRYPTO 2005, LNCS 3621*, pages 430–448. Springer-Verlag, 2005.





→ 1 citation on page: 110

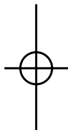
I. Damgård. A design principle for hash functions. In *Proc. Advances in Cryptology — CRYPTO '89, LNCS 435*, pages 416–427. Springer, 1990.
→ 1 citation on page: 110

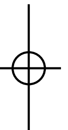
B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2007*. ACM, 2007.
→ 1 citation on page: 72

N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In *Mathematics of Program Construction*, pages 85–109. Springer, 2004.
→ 1 citation on page: 52

E. W. Dijkstra. EWD249: Notes on structured programming. Circulated privately, August 1969.
→ 1 citation on page: 1

D. Drienyovszky, D. Horpácsi, and S. Thompson. QuickChecking refactoring tools. In *Proc. Erlang Workshop 2010*. ACM, 2010. ISBN 978-1-4503-0253-1.
→ 1 citation on page: 71





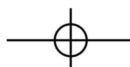
J. Duregård, P. Jansson, and M. Wang. FEAT: functional enumeration of algebraic types. In *Proc. 2012 Haskell Symposium*, pages 61–72. ACM, 2012. ISBN 978-1-4503-1574-6.
→ 4 citations on 4 pages: 5, 81, 82, and 95

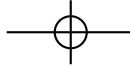
R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3), 1992.
→ 1 citation on page: 72

H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31:226–237, 2005.
→ 1 citation on page: 155

A. Faigon. Testing for zero bugs. At `http://www.yendor.com/testing/`, 2005.
→ 2 citations on page: 4

R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *Proc. International Symposium Software Reliability Engineering, ISSRE 2013*, pages 350–359. IEEE, 2013.





→ 1 citation on page: 96

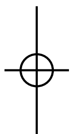
N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family, 2010. http://www.schneier.com/skein.pdf.
→ 2 citations on 2 pages: 109 and 122

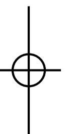
S. Fischer, O. Kiselyov, and C. chieh Shan. Purely functional lazy nondeterministic programming. *J. Funct. Program.*, 21(4-5):413–465, 2011.
→ 2 citations on page: 96

R. Fischlin and C. Schnorr. Stronger security proofs for rsa and rabin bits. In W. Fumy, editor, *Proc. Advances in Cryptology — EUROCRYPT '97, LNCS 1233*, pages 267–279. Springer, 1997. ISBN 978-3-540-62975-7.
→ 1 citation on page: 104

P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, and T. Warnock. Pseudo-random trees in monte carlo. *J. Parallel Computing*, 1(2):175–180, Dec. 1984.
→ 2 citations on 2 pages: 103 and 129

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proc. Functional Programming Languages and Computer Architecture, FPCA '93*, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.
→ 1 citation on page: 67





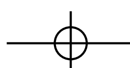
O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4):792–807, Aug. 1986.
→ 4 citations on 3 pages: 103, 108, and 114

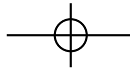
J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proc. International Conf. on Reliable Software*, pages 493–510, New York, NY, USA, 1975. ACM.
→ 1 citation on page: 155

K. Grygiel and P. Lescanne. Counting and generating lambda terms. *J. Funct. Program.*, 23:594–628, 9 2013.
→ 1 citation on page: 97

R. F. Guilmette. TGGS: A flexible system for generating efficient test case generators. Technical report, RG Consulting, 1995.
→ 1 citation on page: 73

J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proc. ICFP 2001*, pages 265–276, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0.





→ 1 citation on page: 61

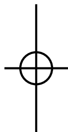
C. Hall, D. Wagner, J. Kelsey, and B. Schneier. Building PRFs from PRPs. In *Proc. Advances in Cryptology — CRYPTO '98, LNCS 1462*, pages 370–389. Springer-Verlag, 1998.
→ 1 citation on page: 117

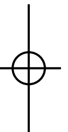
R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994. ISBN 978-0-471-54004-5.
→ 4 citations on 2 pages: 4 and 6

K. V. Hanford. Automatic generation of test cases. *IBM Syst. J.*, 9(4):242–257, Dec. 1970.
→ 1 citation on page: 70

D. Herington. HUnit: A unit testing framework for haskell. http://hackage.haskell.org/package/HUnit-1.2.2.1, January 2010.
→ 1 citation on page: 138

D. R. C. Hill, C. Mazel, J. Passerat-Palmbach, and M. K. Traore. Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, 2012.
→ 1 citation on page: 102





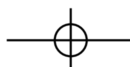
S. Hirose. Security analysis of DRBG using HMAC in NIST SP 800-90. In K.-I. Chung, K. Sohn, and M. Yung, editors, *Information Security Applications*, pages 278–291. Springer-Verlag, 2009. ISBN 978-3-642-00305-9.
→ 1 citation on page: 130

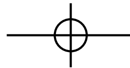
J. Hughes. QuickCheck testing for fun and profit. In M. Hanus, editor, *Proc. PADL 2007, LNCS 4354*, pages 1–32. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-69608-7.
→ 2 citations on 2 pages: 70 and 72

J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby. A pseudorandom generator from any One-way function. *SIAM Journal on Computing*, 28: 12–24, 1999.
→ 3 citations on 2 pages: 103 and 108

D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN 0262101149.
→ 2 citations on page: 10

JUnit.org. JUnit.org resources for test driven development. http://www.junit.org/, January 2010.
→ 1 citation on page: 138



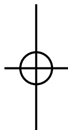


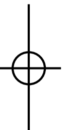
B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, Inc., New York, NY, USA, 2nd edition, 1982. ISBN 0070342075.
→ 1 citation on page: 2

B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28:721–734, 2002.
→ 1 citation on page: 155

C. Klein, M. Flatt, and R. B. Findler. The racket virtual machine and randomized testing. Available from `http://plt.eecs.northwestern.edu/racket-machine/`, 2010a.
→ 2 citations on 2 pages: 17 and 20

C. Klein, M. Flatt, and R. B. Findler. Random testing for higher-order, stateful programs. In *Proc. OOPSLA 2010*. ACM, 2010b. ISBN 978-1-4503-0203-6.
→ 2 citations on 2 pages: 20 and 71



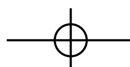


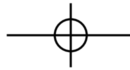
C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: On the effectiveness of lightweight mechanization. In *Proc. POPL 2012*, pages 285–296, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3.
→ 1 citation on page: 72

P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), 2007.
→ 2 citations on 2 pages: 12 and 103

Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE '05, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2482-6.
→ 1 citation on page: 11

C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc. Symp. on Principles and Practice of Parallel Programming*, pages 193–204. ACM, 2012. ISBN 978-1-4503-1160-1.
→ 3 citations on 3 pages: 102, 104, and 130





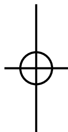
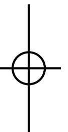
X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
→ 3 citations on 3 pages: 17, 18, and 20

R. Leshchinskiy. Recycle your arrays! In *Proc. Practical Aspects of Declarative Languages*, PADL '09, pages 209–223. Springer-Verlag, 2009. ISBN 978-3-540-92994-9.
→ 1 citation on page: 126

C. Lindig. Random testing of C calling conventions. In *Proc. 6th International Symposium on Automated Analysis-Driven Debugging*. ACM, 2005. ISBN 1-59593-050-7.
→ 3 citations on 3 pages: 4, 20, and 69

D. Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Massachusetts Institute of Technology, 2005.
→ 3 citations on 2 pages: 10 and 11

S. Marlow. Haskell 2010 language report. `http://www.haskell.org/definition/haskell2010.pdf`, 2010.
→ 3 citations on 3 pages: 51, 61, and 63

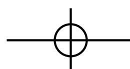
S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *Proc. ICFP 2009*, pages 65–78, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.
→ 2 citations on 2 pages: 63 and 64

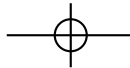
M. Mascagni, S. A. Cuccaro, D. V. Pryor, and M. L. Robinson. Recent developments in parallel pseudorandom number generation. In *SIAM Conf. on Parallel Processing for Scientific Computing*, volume 11, pages 524–529, 1993.
→ 1 citation on page: 103

B. D. McCullough. *The Accuracy of Econometric Software*, pages 55–79. John Wiley & Sons, Ltd, 2009. ISBN 9780470748916.
→ 2 citations on 2 pages: 6 and 103

W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, December 1998.
→ 8 citations on 7 pages: 4, 8, 9, 19, 20, 50, and 69

S. Micali and C. P. Schnorr. Efficient, perfect polynomial random number generators. *J. Cryptology*, 3:157–172, 1991.
→ 3 citations on 3 pages: 103, 104, and 130





B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.
→ 3 citations on 2 pages: 4 and 9

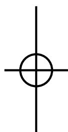
A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.
→ 1 citation on page: 2

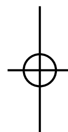
M. Moczurad, J. Tyszkiewicz, and M. Zaionc. Statistical properties of simple types. *Mathematical. Structures in Computer Science*, 10, Oct. 2000.
→ 1 citation on page: 73

G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. Wiley, 3rd edition, 2012.
→ 10 citations on 4 pages: 1, 2, 4, and 11

S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
→ 1 citation on page: 27

S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *Proc. PLDI 1999*, pages 25–36, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5.
→ 2 citations on 2 pages: 61 and 64





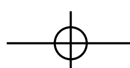
S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proc. Haskell Workshop 2001*, pages 203–233, Sept. 2001.
→ 1 citation on page: 67

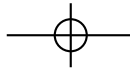
S. Peyton-Jones, B. Smith, et al. Splittable random numbers. Mailing list discussion, 2010. http://www.haskell.org/pipermail/haskell-cafe/2010-November/085959.html.
→ 2 citations on 2 pages: 104 and 129

S. L. Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *Proc. ESOP*. Springer-Verlag, 1996.
→ 2 citations on page: 18

B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.
→ 4 citations on 3 pages: 18, 23, and 39

J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *Proc. PLDI 2012*, pages 335–346, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9.





→ 2 citations on 2 pages: 9 and 11

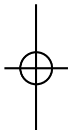
J. S. Reich, M. Naylor, and C. Runciman. Lazy generation of canonical test programs. In A. Gill and J. Hage, editors, *Proc. IFL 2011, LNCS 7257*, pages 69–84. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34406-0.
→ 1 citation on page: 11

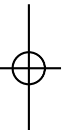
A. Rodriguez Yakushev and J. Jeuring. Enumerating well-typed terms generically. In *Approaches and Applications of Inductive Programming (AAIP 2009), LNCS 5812*. Springer Berlin / Heidelberg, 2010.
→ 3 citations on 3 pages: 21, 73, and 97

P. Rogaway. Evaluation of some blockcipher modes of operation. Unpublished manuscript, 2011.
→ 3 citations on 3 pages: 108, 117, and 127

C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Proc. Haskell Symposium 2008*, pages 37–48, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-064-7.
→ 7 citations on 6 pages: 10, 11, 81, 86, 89, and 96

J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *Proc. High Performance Computing, Networking, Storage and Analysis*, pages 1–12. ACM, 2011. ISBN 978-1-4503-0771-0.
→ 3 citations on 3 pages: 103, 108, and 131



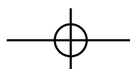


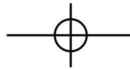
A. Sidorenko and B. Schoenmakers. Concrete security of the Blum-Blum-Shub pseudorandom generator. In *Cryptography and Coding 2005, LNCS 3796*, pages 355–375, 2005.
→ 2 citations on 2 pages: 104 and 130

J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM. ISBN 1-59593-123-6.
→ 1 citation on page: 1

I. Sommerville. *Software Engineering*. Addison-Wesley, Harlow, England, 9 edition, 2010. ISBN 978-0-13-703515-1.
→ 4 citations on 3 pages: 1, 2, and 3

G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.
→ 2 citations on 2 pages: 1 and 18





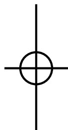
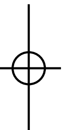
The GHC Team. The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, January 2010.
→ 1 citation on page: 142

N. Tillmann and J. de Halleux. Pex—white box test generation for .NET. In *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-79123-2.
→ 1 citation on page: 138

N. Tillmann and W. Schulte. Parameterized unit tests. *SIGSOFT Softw. Eng. Notes*, 30(5):253–262, 2005.
→ 1 citation on page: 138

D. Vytiniotis and A. J. Kennedy. Functional pearl: every bit counts. In *Proc. ICFP 2010*. ACM, 2010. ISBN 978-1-60558-794-3.
→ 3 citations on 3 pages: 21, 72, and 74

J. Wang. Generating random lambda calculus terms. Technical report, Boston University, 2005.
→ 3 citations on 2 pages: 21 and 73

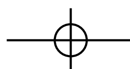
E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. IEEE *Trans. Softw. Eng.*, 6(3):236–246, 1980.
→ 1 citation on page: 155

C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.
→ 1 citation on page: 155

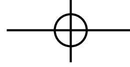
X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. PLDI 2011*, pages 283–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.
→ 9 citations on 5 pages: 4, 9, 20, 21, and 68

A. C. Yao. Theory and application of trapdoor functions. In *Proc. Symp. Foundations of Computer Science*, pages 80–91. IEEE, 1982.
→ 2 citations on 2 pages: 103 and 108

A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005. ISBN 1558608664.
→ 9 citations on 4 pages: 1, 2, 3, and 4



thesis March 27, 2014 10:04 Page 168



A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.
→ 3 citations on 3 pages: 9, 71, and 137