THESIS FOR THE DEGREE OF LICENTIATE OF PHILOSOPHY

Automated Derivation of Random Generators for Algebraic Data Types

AGUSTÍN MISTA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2020

Automated Derivation of Random Generators for Algebraic Data Types
AGUSTÍN MISTA

Technical Report 208L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Information Security

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2020

Abstract

Many testing techniques such as generational fuzzing or random property-based testing require the existence of some sort of random generation process for the values used as test inputs. Implementing such generators is usually a task left to end-users, who do their best to come up with somewhat sensible implementations after several iterations of trial and error. This necessary effort is of no surprise, implementing *good* random data generators is a hard task. It requires deep knowledge about both the domain of the data being generated, as well as the behavior of the stochastic process generating such data. In addition, when the data we want to generate has a large number of possible variations, this process is not only intricate, but also very cumbersome.

To mitigate this issues, this thesis explores different ideas for automatically deriving random generators based on existing static information. In this light, we design and implement different derivation algorithms in Haskell for obtaining random generators of values encoded using Algebraic Data Types (ADTs). Although there exists other tools designed directly or indirectly for this very purpose, they are not without disadvantages. In particular, we aim to tackle the lack of flexibility and static guarantees in the distribution induced by derived generators. We show how automatically derived generators for ADTs can be framed using a simple yet powerful stochastic model. This models can be used to obtain analytical guarantees about the distribution of values produced by the derived generators. This, in consequence, can be used to optimize the stochastic generation parameters of the derived generators towards target distributions set by the user, providing more flexible derivation mechanisms.

Acknowledgments

There are many who deserve acknowledgment for their contributions to this thesis. First and foremost, to Alejandro Russo, not only for helping me on every step of this journey, but also for making it equally fun and challenging. To the fika crew, for making the slow days run faster. Thanks for making working at Chalmers great fun. I also want to thank Leonidas Lampropoulos, for likely being among the few willing to take time to read this thesis and to come and discuss about it with me. Saving the best for the last, my lovely partner, Carla, deserves special recognition. You gave me nothing but love and support ever since we met, and this whole adventure would not have been the same without you.

CONTENTS

Chapter 0

Introduction

Software systems are, for the most part, tested much more poorly than we like to admit. This is often not due to laziness (except when it is), neither to lack of investment (except when it is). Testing software effectively is *extremely difficult*. Even the most formal and theoretically robust techniques are sometimes seen as a futile countermeasures against the unquantifiable number of things that can go wrong in our ever-growing systems. How should we test them then? As expected, this thesis does not provide anything closer to an answer for this question. Instead, it focuses particularly on an appealing idea: testing software against unexpected inputs using randomly generated data.

1 Fuzzing

Fuzzing [32] is a technique used in penetration testing [1] that involves providing unexpected inputs to a system under test, and a program that performs fuzzing to test a program is usually known as a *fuzzer*. The intuition behind a fuzzer is rather simple: it picks an input from some inputs repository, feeds it to the system under test, and monitors it for different kinds of exceptions, e.g., crashes, memory leaks and failed code assertions. This process is repeated in a loop until something bad happens in the target system. Then, any anomaly detected in the expected behavior of the system under test is reported along with the input producing it. Figure 1 displays a simplified representation of this approach.

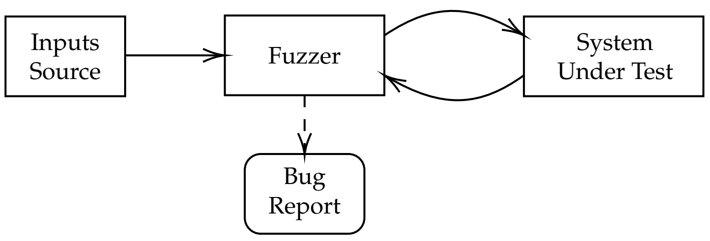

Figure 1: Simplified representation of a fuzzing environment.

As expected, the previous description is extremely oversimplified. Fuzzers typically use many different approaches to boost the chances of finding different kinds of vulnerabilities with remarkable success [5,8,9,12,14–17,23,26,28,29]. Notably, the unexpected inputs used by this technique can be of a very varied nature, covering the full spectrum between completely valid values to completely random noise ones. Moreover, the origin of these inputs denotes an important distinction used to classify different kinds of fuzzing models [25]:

– **Mutational Fuzzers:** they use an existing set of (usually valid) inputs that are combined in different ways through randomization. In practice, they usually rely on an external set of input files provided by the user, known as a *corpus*. A mutational fuzzer takes one or more files from this corpus and produces a mutated version that is as a test case for the system under test.

  While this approach has shown to be quite powerful for finding bugs, its inherent disadvantage is that the user has to collect and maintain a carefully curated corpus manually for each kind of input that wants to test, e.g., for each input file format.
– **Generational Fuzzers:** they generate inputs from scratch using an specification or model for the different kinds of inputs they are used for. In general, generational fuzzers avoid the problem of having to maintain an external corpus of inputs. However, users must then develop and maintain models of the input types they want to generate. As expected, creating such models requires a deep domain knowledge, which can be tedious and expensive to achieve.

In this work, we focus particularly on the generational model. Our aim is to develop automated techniques for random generation of unexpected inputs based on statically available information. This information can be extracted either directly from system under test or from external sources. In particular, Paper 1 is focused on automatically leveraging on existing file-format manipulating libraries to derive random input generators used for fuzzing massively used programs.

Fuzzers are seen in practice as black-box tools acting over complete programs. In consequence, they are often applied to finished systems to find vulnerabilities that might have not been discovered during the early development stages. However, testing smaller pieces of our systems using randomly generated inputs during development is also a popular technique. As opposed to unit testing, where programmers are forced to write and maintain a set of individual test inputs (unit tests), this technique lets us test each part of our system using randomly generated inputs. The next section introduces an attractive variant of this idea.

2 QuickCheck

Instead of just feeding our software with random inputs and waiting for unexpected behavior, it is also possible to test our programs using randomly generated inputs in a more controlled way. The idea behind this is to verify our code against some sort of specification. This specification can be defined, for instance, as a set of properties that our code must fulfill for every possible input. Then, these properties can be validated using a large number of randomly generated inputs. This technique is known as *Random Property-Based Testing* (RPBT).

In the Haskell realm, QuickCheck [10] is the de facto tool of this sort. Originally conceived by Koen Claessen and John Hughes twenty years ago, this tool counts with many success stories, and inspired the ideas behind it to be replicated in other programming languages and systems with remarkable success [2–4, 7, 19–21, 24, 24, 30].

Essentially, using this tool can be seen as composed of two main parts: *testing properties* and *random generators*. This thesis focuses strictly on the latter, as automating the process of deriving testing specifications can be seen as a field in its own right [6, 11]. Nonetheless, the following subsections briefly introduce the reader to both for the sake of completeness.

2.1 Testing Properties

One of the attractive aspects of QuickCheck is its simplicity. To illustrate this, suppose we write a Haskell function $reverse :: [Int] \to [Int]$ for reversing lists of integers. While specifying the expected behavior of this function, we might want to assert that our implementation is its own inverse , i.e., reversing a list twice always yields the original value.[1] This desired property of our function can be written in QuickCheck simply as a Haskell predicate parameterized over its input, which we can think as being universally quantified:

$$
\begin{array}{l}
prop\_reverse\_ok :: [Int] \to Bool \\
prop\_reverse\_ok\ xs = \\
\quad reverse\ (reverse\ xs) \equiv xs
\end{array}
$$

Then, verifying that our function holds this property becomes simply running QuickCheck over it:

```
ghci> quickCheck prop_reverse_ok
++++ OK, passed 100 tests
```

What happens under the hood is that QuickCheck will instantiate every input of our property using a large number of randomly generated values (lists of integers in our example above), asserting that it holds (returns $True$) for all of them.

[1]In mathematical jargon, we could say that $reverse$ must be *involutive*.

Shall any of our properties not hold for some input, QuickCheck will try to find a minimal counterexample for us to further analyze. For instance, reversing any list once will not return the original input:

$$\begin{array}{l} \mathit{prop\_reverse\_bad} :: [\mathit{Int}] \rightarrow \mathit{Bool} \\ \mathit{prop\_reverse\_bad}\ \mathit{xs} = \\ \quad \mathit{reverse}\ \mathit{xs} \equiv \mathit{xs} \end{array}$$

This property can be easily refuted using QuickCheck as before:

```
ghci> quickCheck prop_reverse_bad
*** Failed! Falsifiable (after 3 tests and 1 shrink):
[0,1]
```

And after a handful random tests, we obtain a minimal counterexample (`[0,1]`) which falsifies *prop_reverse_bad* when used as an input.

This way, running a large number of random tests gives us statistical confidence about the correctness of our code against its specification.

2.2 Random Generators

One of the reasons behind the simplicity of the previous examples is that the random generation of test cases is transparently handled for us by QuickCheck. This is achieved by using Haskell *type classes* [34]. In particular, QuickCheck defines the *Arbitrary* type class for the types that can be randomly generated:

$$\begin{array}{l} \textbf{class}\ \mathit{Arbitrary}\ a\ \textbf{where} \\ \quad \mathit{arbitrary} :: \mathit{Gen}\ a \\ \quad \mathit{shrink} :: a \rightarrow [a] \end{array}$$

The interface of this type class encodes two basic primitives. In first place, *arbitrary* specifies a monadic random generator of values of type a. Such generators are defined in terms of the *Gen* monad which provides random generation primitives. Moreover, $\mathit{shrink} :: a \rightarrow [a]$ specifies how a given counterexample (of type a) can be reduced in different smaller ones. This function is used while reporting minimal counterexample after a bug is found.

QuickCheck comes equipped with *Arbitrary* instances for most basic data types in the Haskell prelude. In particular, our previous testing examples simply use the default *Arbitrary* instances for integers and lists. In this light, it is quite easy to test properties defined in terms of basic data types using QuickCheck. However, things get more complex when we start defining our own custom data types.

Algebraic Data Types Haskell has a powerful type system that can be extended with custom data types defined by the user. For instance, suppose we want to represent simple HTML pages as Haskell values. For this purpose, we can define the following custom algebraic data type:

```
data Html =
    Text String
  | Sing String
  | Tag String Html
  | Html :+: Html
```

This type allows to build pages via four possible constructions: *Text* represents plain text values, *Sing* and *Tag* represent singular and paired HTML tags, respectively, and (:+:) concatenates two HTML pages one after another. These four constructions are known as data constructors (or constructors for short) and are used to distinguish which variant of the ADT we are constructing. Each data constructor is defined as a product of zero or more types known as fields. For instance, *Text* has a field of type *String*, whereas the infix constructor (:+:) has two recursive fields of type *Html*. In general, we will say that a data constructor with no recursive fields is *terminal*, and *non-terminal* or *recursive* otherwise. Then, the example page:

```
<html>hello<hr>bye</html>
```

can be encoded using our freshly defined *Html* data type as:

```
Tag "html" (Text "hello" :+: Sing "hr" :+: Text "bye")
```

Later, suppose we implement two functions over *Html* values for simplifying and measuring the size of an HTML page:

```
simplify :: Html → Html
size :: Html → Int
```

The concrete implementation of these functions is not relevant here. What is important, though, is that with this functions in place, we might be interested in asserting that simplifying an HTML page never returns a bigger one. This can be encoded with the following QuickCheck property:

```
prop_simplify :: Html → Bool
prop_simplify html =
  size (simplify html) ⩽ size html
```

However, testing this property using random inputs is not possible yet. The reason behind this is simple: QuickCheck does not know how to generate random *Html*s to instantiate this property's input parameter. To solve this issue, we can provide a user defined *Arbitrary* instance for *Html* as shown in Figure 2 (avoiding for simplicity the definition of *shrink*). To generate a random *Html* value, this generator picks a random *Html* data constructor with uniform probability and proceeds to "fill"

```
instance Arbitrary Html where
  arbitrary = oneof
    [ Text  ⟨$⟩ arbitrary
    , Sing  ⟨$⟩ arbitrary
    , Tag   ⟨$⟩ arbitrary ⟨⋆⟩ arbitrary
    , (:+:) ⟨$⟩ arbitrary ⟨⋆⟩ arbitrary ]
```

Figure 2: Naive random generator of *Html* values.

its fields recursively. This definition implements the simplest generation procedure for *Html* that is theoretically capable of generating any possible *Html* value.

After providing this concrete *Arbitrary* instance, QuickCheck can now proceed to test properties involving *Html* values.

3 Automated Derivation of Generators

The random generator defined above can be written quite mechanically, so it is of no surprise that automated derivation mechanisms [13, 27] have emerged to relieve the programmer of the burden of this task—something specially valuable for large data types! Most of these tools use Template Haskell [31], the Haskell meta-programming framework, as a way of introspecting the user code and synthesizing new code upon it.

However, a suitable mechanism for deriving random generators cannot be as simple as just producing code like the one shown in Figure 2. Sadly, this naive generator is ridden with flaws.

In practice, QuickCheck users are often aware of some of them, and an attentive reader might have already recognized some by just inspecting the definition above carefully. Concretely, to implement a suitable random generator we need to consider (at least) the following challenges:

Unbounded recursion: Every time a recursive subterm is needed, the generator shown in Figure 2 simply calls itself recursively. This is a common mistake that can lead to infinite generation loops due to recursive calls producing (on average) one or more subsequent recursive calls. This problem can be more or less severe depending mostly on the shape of the data type our generator produces values of, being a practical limitation nonetheless. Fortunately, QuickCheck already provides mechanisms to overcome this issue—this is addressed by all four papers presented in this thesis.

Generation parameters: The generator from Figure 2 simply picks the next random constructor in a uniform basis. This is the simplest approach we can mechanically follow. However, this is hardly the best choice in practice. In particular, generating values of any data type with more terminal than recursive data constructors using uniform choices will be biased towards generating very small values. QuickCheck provides mechanisms for adjusting the generation probability of each random choice it performs. However, doing so carries a second problem: it becomes quite tricky to assign these probabilities without knowing how they will affect the overall distribution of generated values—something that can be seen as a science to its own. Both problems are addressed in detail in Paper 2.

Abstraction level: The generation process encoded in the generator shown in Figure 2 constructs values using the smallest possible level of granularity: one data constructor at a time. In practice, this technique is often too weak to generate (with a non-negligible probability) values containing the complex patterns of values that could be required in order to test the corner cases of our code, leaving the door open for subtle bugs that might be never get triggered during the testing phase.

In the other hand, the implementation of our code under test could rely on internal invariants that are necessary to make it work properly—think for instance the case of the implementation of data structures like balanced trees, where its abstract interface must preserve the internal invariants used by their implementation. Testing this kind of software becomes much more complicated using the approach described above, as constructing random values at the abstraction level of data constructors will be very unlike to generate values satisfying such invariants—this issue is addressed in details in Paper 3 and Paper 4.

Clearly, all these issues and challenges need to be carefully considered in order for our generators to be effective at generating useful values for penetration or random property-based testing. It is the purpose of this thesis to tackle them in the most automated way possible.

4 Contributions

In this section, I give a more detailed overview of the thesis, which is based on four papers, published individually in the proceedings of peer-reviewed international conferences, symposiums and workshops—see Figure 3 for a simplified roadmap of this work.

4.1 Paper 1: QuickFuzz Testing For Fun And Profit

This paper explores the ideas behind the development of QuickFuzz, a generational fuzzer using Haskell data types as lightweight grammars. Unlike other generational fuzzers, where the generation of random inputs depends on user-provided specifications or grammars for the random inputs they can generate, QuickFuzz leverages on existing data-handling libraries written in Haskell.

Haskell ecosystem [18] has a large number of existing libraries for interacting with most kinds of structured data we use nowadays, e.g., common file formats, network packets, public key infrastructure certificates, etc. The fact that these libraries often define complex data types encoding

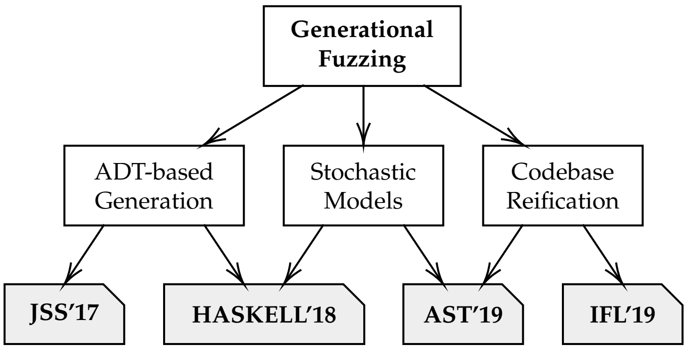

Figure 3: Roadmap of this thesis.

such data is what makes QuickFuzz particularly appealing: these data types can be a good approximation of the grammar of our data, and we can obtain random generators for them for free! For this to be effective, however, we need an automatic mechanism for extracting random generators from data type definitions, solely based on introspecting the existing code, and with minimal interaction required by the programmer.

With more than 40 file formats supported for random generation, and combined with the ability of introducing off-the-shelf mutational fuzzers into the testing pipeline, QuickFuzz has been shown to be remarkably useful for discovering bugs on real-world code with minimal effort. During the development of this tool, dozens of security vulnerabilities were discovered on massively used open-source programs and libraries.

4.2 Paper 2: Branching Processes for QuickCheck Generators

Despite that automatically deriving random generators from data type definitions can be seen as a mechanical task, doing so too naively can degrade the performance of the derived generators quite substantially. The main problem with the derivation mechanism used in QuickFuzz is that, whenever an automatically derived generator needs to randomly choose which random construction to generate next, it does so with uniform probability across all the possible choices. While this approach is (theoretically) able to generate the full space of values of any algebraic data type, empirical results show that it often introduces strong biases towards generating very small and rather uninteresting values. This limitation is mostly dependent on the shape of the data type being generated, and cannot be improved nor adjusted once the random generator is derived at compile time.

While there exists other approaches for solving this problem, we consider that none of them effectively achieves a good trade-off between automation level and flexibility.

In this paper, we propose modeling the generation process encoded into automatically derived generators using *branching processes*. This statistical model lets us predict the distribution of values produced by our generators. This distribution depends on two main factors. First, their particular data type definition takes an important role on how the data gets generated each time a recursive sub-term is needed. This is a fixed part of our model, and represent the invariants introduced by the data types we generate. Second, the frequency in which our generators pick each random construction whenever they generate a value also makes a large difference in the distribution of generated values. Interestingly, these frequencies are a tunable parameter of our derived generators, and thus we can optimize them towards a configuration that fits user's demands by using the prediction model based on branching processes in a feedback optimization loop. This way, the optimization of parameters depends on a model that can be predicted analytically and, in consequence, is much cheaper to compute than sampling a large number of random values every time we evaluate a possible optimization candidate.

The ideas presented in this paper are implemented in an automated tool for deriving optimized generators called DRAGEN.

Using this approach, we found that the performance of automatically derived generators can be considerably improved by tuning their generation parameters at compile time using our stochastic model. In practice, we found that this can be used to increase the code coverage triggered by the random values they generate over real-world applications quite substantially.

4.3 Paper 3: Generating Random Structurally Rich Algebraic Data Type Values

The previous paper proposes using a stochastic model for automatically deriving optimized random generators. In principle, this model only contemplates the information encoded into data type definitions. However, in practice, much of the *structural* information of our data is often encoded aside of its corresponding types.

In first place, a common limitation of random testing arises whenever we try to generate random values to test functions or procedures branching differently on very specific patterns of inputs. The reason behind this is simple: whenever a function input pattern grows linearly in the number of matched constructors, the probability of generating a value satisfying such pattern decreases *multiplicatively* if we follow the standard approach, i.e., building random values using one atomic piece of data (constructor) at a time.

On the other hand, it is common that data type definitions simply do not encode enough structural information of the actual data they represent in order for the derivation process to derive useful random generators. This is particularly the case in the presence of shallow embedded domain-specific languages, where data types are often too generic, and invariants are preserved mostly via their abstract interfaces.

In this paper, we identify two extra sources of structural information that can be statically extracted and taken advantage of during the generator derivation process. In first place, every input pattern matching of a function of interest can be automatically extracted from the user codebase, and included into the generation process. This lets us generate complex compositions of data constructors at once, ensuring that our random data will satisfy the input patterns of our code under test, and hence will be used to test code branches that otherwise could remain untested using naive generators. Secondly, the abstract interface of our data types of interest can be analyzed and extracted from the codebase. Each combinator of this interface can be used to generate random data as well, somewhat replicating the behavior a real programmer would follow to interact with the user code in a real-world scenario. This ability also lets us generate random data preserving the invariants introduced by this interface, and that are not encoded directly in the data type definition.

The ideas presented in this paper are implemented as an extension of DRAGEN, called DRAGEN2.

Using this approach, it becomes possible to generate random values by interleaving data constructors, input patterns and abstract interface function calls. This can effectively improve the performance of our derived generators, which are able to use more domain-specific information extracted from the source code in order to generate structured data.

One of the key contributions of this work is to show how the stochastic model of branching processes used previously can be extended to contemplate these two new sources of structural information. Using this extended model, we can automatically derive random generators optimized towards producing complex distributions of values, parameterized by higher-level random constructions other than just data constructors, like input patterns and abstract interface function calls. For instance, it becomes possible to reason about random distributions of values where certain patterns of constructors appear (on average) in a given ratio within every generated value. In the same manner, we can use this model to derive generators which produce random values following a particular distribution of high-level combinators (from abstract interfaces) used to build them, which can specified by the programmer.

4.4 Paper 4: Deriving Compositional Random Generators

The previous paper provides an extension to the automated derivation mechanism of random generators proposed originally. This extension enables us to consider additional sources of structural information when deriving random generators apart from just data type definitions. Each source of structural information introduces a new set of random constructions that can be used by our generators when producing random values, i.e., one random construction per each data constructor, function input pattern and abstract interface combinator.

In principle, we could combine every random construction extracted from the codebase into a single random generator. However, as our codebase grows, this practice can become unmanageable. The reason for this is that different parts of our system could expect different kinds of inputs, and therefore, they should be tested using random values resembling such expected inputs. For instance, if we consider a code compiler, a type checker phase should be tested using both valid and invalid random input programs. On the other hand, any subsequent phase would be implemented under the assumption that they work over syntactically and/or semantically valid inputs. In this case, testing such phases effectively would require having a random generator that *only* produces somewhat valid inputs—or more generically, inputs satisfying certain invariants. In this light, our testing framework would benefit from having not one, but many specialized random generators depending on the concrete subsystem to be tested. This is, sadly, not compatible with most automated generator derivation approaches, where a unique (and rigid) random generator is synthesized.

In this paper we demonstrate how it is possible to implement a fully compositional generators' derivation mechanism. Instead of deriving a single random generator encompassing every possible random construction, our approach works by deriving a small specialized generator for each one. Later, these generators can be combined in different ways using a simple yet powerful type-level domain-specific specification language. This domain-specific language lets the programmer specify which random constructions are of interest while generating values in a simple manner, abstracting much of the cumbersome details of writing random generators by hand. Notably, specifying different random generators using this approach doesn't require synthesizing their implementation every time. In turn, the user simply specifies each generator variant by referring to the components of the same common underlying machinery, which is automatically derived once and for all.

To achieve this compositionality, we use the familiar functor coproduct pattern in Haskell, popularized by Swierstra with the name of *Data Types à la Carte* [33]. We extended this programming pattern with the functionality required in the scope of random generation of values, and shown how the performance limitations [22] commonly associated to this pattern can be alleviated by using a self-optimizing representation.

4.5 Statement of Contributions

Paper 1: QuickFuzz Testing For Fun And Profit My contributions to this project include: i) a generators derivation extension, which contemplates the common case of existing libraries written using shallow embeddings of the target file format. Before this extension, such libraries simply could not be used due to the lack of domain-specific structure encoded into data types, which are the main source of information used by QuickFuzz to derive useful random generators for free, and ii) a complete rewrite of the testing harness from scratch, using as much meta-programming as possible in order to ease the task of adding support for new file-format targets.

Moreover, I actively participated in the technical writing of the journal paper resulting from this project.

Paper 2: Branching Processes for QuickCheck Generators I developed a generic meta-programming mechanism for deriving random generators using this model based on branching processes (the first version of DRAGEN). This includes developing an adjustable optimization process for the stochastic parameters, based on different statistical goodness-of-fit measures. This is hidden behind a simple generators specification interface.

The technical writing of this paper was initially done in equal parts between Alejandro and I, with John Hughes joining us at later stages with invaluable feedback.

Paper 3: Generating Random Structurally Rich Algebraic Data Type Values I extended our previous derivation tool and its underlying stochastic model with support for extracting and generating such patterns automatically. I later extended this mechanism to also contemplate extracting abstract interfaces from the user codebase, which greatly improved the performance of the derived generators.

The technical writing of this paper was done by both authors jointly.

Paper 4: Deriving Compositional Random Generators I carried out most of the technical development of this idea, using both meta-programming and type-level features available in Haskell.

The majority of the writing was initially done by me.

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet.

References

1. B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security Privacy*, 2005.
2. T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*, 2015.
3. Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with Quviq Quickcheck. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 1–8, New York, NY, USA, 2008. ACM.
4. Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, 2006.
5. Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.
6. Rudy Braquehais and Colin Runciman. Fitspec: refining property sets for functional testing. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 1–12, 2016.
7. Lukas Bulwahn. The new QuickCheck for isabelle. In *International Conference on Certified Programs and Proofs*, pages 92–108. Springer, 2012.
8. CACA Labs. zzuf - multi-purpose fuzzer. `http://caca.zoy.org/wiki/zzuf`, 2010.
9. Sang Kil Cha, Maverick Woo, and David Brumley. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 725–741, 2015.
10. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.
11. Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs, TAP 2010, Málaga, Spain, July 1-2, 2010.*, pages 6–21, 2010.
12. Deja vu Security. Peach: a smartfuzzer capable of performing both generation and mutation based fuzzing. `http://peachfuzzer.com/`, 2007.
13. J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2012.
14. Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. *SIGPLAN Not.*, 2008.
15. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 2012.
16. Google. honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options. `https://github.com/aoh/radamsa`, 2010.
17. Gustavo Grieco, Martín Ceresa, and Pablo Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 13–20, New York, NY, USA, 2016. ACM.
18. Hackage. The Haskell community's central package archive of open source software. `http://hackage.haskell.org/`, 2010.

19. Paul Holser. junit-quickcheck: Property-based testing, JUnit-style, 2019.
20. J. Hughes, U. Norell, N. Smallbone, and T. Arts. Find more bugs with QuickCheck! In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2016.
21. John Hughes. QuickCheck testing for fun and profit. In *International Symposium on Practical Aspects of Declarative Languages*, pages 1–32. Springer, 2007.
22. H. Kiriyama, H. Aotani, and H. Masuhara. A lightweight optimization technique for data types a la carte. In *Companion Proceedings of the 15th Int. Conference on Modularity*, MODULARITY 2016, New York, NY, USA, 2016. ACM.
23. M. Zalewski. American Fuzzy Lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2010.
24. J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson. Effect-driven QuickChecking of compilers. *In Proceedings of the ACM on Programming Languages, Volume 1*, (ICFP), 2017.
25. Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 2007.
26. Mozilla. Dharma: a generation-based, context-free grammar fuzzer. `https://github.com/MozillaSecurity/dharma`, 2015.
27. Neil Mitchell . Data.Derive is a library and a tool for deriving instances for Haskell programs. `http://hackage.haskell.org/package/derive`, 2006.
28. Oulu University Secure Programming Group. A Crash Course to Radamsa. `https://github.com/aoh/radamsa`, 2010.
29. Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.
30. Lee Pike. Smartcheck: automatic and efficient counterexample reduction and generalization. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*, pages 53–64, 2014.
31. T. Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.
32. Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
33. Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.
34. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’89, pages 60–76, 1989.