Thesis for The Degree of Doctor of Philosophy

Language-Based Techniques and Stochastic Models for Automated Testing

Agustín Mista

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg
Göteborg, Sweden, 2023

Language-Based Techniques and Stochastic Models for Automated Testing

Agustín Mista

ISBN 978-91-7905-851-7
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 5317
ISSN 0346-718X

Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology | University of Gothenburg
SE-412 96 Göteborg
Sweden
Phone: +46(0)31 772 1000

Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2023.

"Sometimes I'll start a sentence, and I don't even know where it's going. I just hope I find it along the way."
- Michael Scott

Language-Based Techniques and Stochastic Models for Automated Testing

Agustín Mista

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

As software systems become bigger and scarier, automating their testing is crucial to ensure that our confidence in them can keep up with their growth. In this setting, Generational Fuzzing and Random Property-Based Testing are two sides of the same testing technique that can help us find bugs effectively without having to spend countless hours writing unit tests by hand. They both rely on generating large amounts of random (possibly broken) test cases to be used as inputs to the system. Test cases that trigger issues such as crashes, memory leaks, or failed assertions are reported back to the developer for further investigation. Despite being fairly automatable, the Achilles heel of this technique lies in the quality of the randomly generated test cases, often requiring substantial manual work to tune the random generation process when the system under test expects inputs satisfying complex invariants.

This thesis tackles this problem from the Programming Languages perspective, taking advantage of the richness of functional, statically-typed languages like Haskell to develop automated techniques for generating good-quality random test cases, as well as for automatically tuning the testing process in our favor. To this purpose, we rely on well-established ideas such as coverage-guided fuzzing, meta-programming, type-level programming, as well as novel interpretations of centuries-old statistical tools designed to study the evolution of populations such as branching processes. All these ideas are empirically validated using an extensive array of case studies and supported by a substantial number of real-world bugs discovered along the way.

Keywords

Automated Software Testing, Property-Based Testing, Functional Programming Languages, Meta-Programming, Stochastic Models, Haskell

List of Publications

Appended publications

This thesis is based on the following publications, listed in chronological order:

[**Paper I**] *Gustavo Grieco, Martín Ceresa, **Agustín Mista** and Pablo Buiras*, *QuickFuzz testing for fun and profit*, *Journal of Systems and Software (Volume 134), 340-354 (2017).*

[**Paper II**] ***Agustín Mista*** *and Alejandro Russo*, *Branching Processes for QuickCheck Generators*, *ACM SIGPLAN Haskell Symposium (2018).*

[**Paper III**] ***Agustín Mista*** *and Alejandro Russo*, *Generating Random Structurally Rich Algebraic Data Type Values*, *14th IEEE/ACM International Workshop on Automation of Software Test (2019).*

[**Paper IV**] ***Agustín Mista*** *and Alejandro Russo*, *Deriving Compositional Random Generators*, *31st Symposium on Implementation and Application of Functional Languages (2019).*

[**Paper V**] ***Agustín Mista*** *and Alejandro Russo*, *BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs*, *21st International Symposium on Trends in Functional Programming (2020).*

[**Paper VI**] *Matthías Páll Gissurarson and **Agustín Mista***, *Short Paper: Weak Runtime-Irrelevant Typing for Security*, *ACM SIGSAC 15th Workshop on Programming Languages and Analysis for Security (2020).*

[**Paper VII**] ***Agustín Mista*** *and Alejandro Russo*, *MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations*, *16th IEEE International Conference on Software Testing, Verification and Validation (2023).*

Acknowledgments

To my advisor, Alejandro Russo, may someone finally beat you at Street Fighter.

To my friends and colleagues, my rants about Reviewer #2 are over.

To Reviewer #2, it's not you, it's me.

To my parents, for their constant support and love despite the distance.

To the love of my life, Carla, and to the love of her life, Archie.

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.

Contents

I Overview 1

1 Introduction 1
- 1 Fuzzing 4
- 2 Property-Based Testing and QuickCheck 6
- 3 Automated Derivation of Generators 10
- 4 Coverage Guided, Property-Based Testing 11
- 5 Domain-Specific Programming Language Tools 13

2 Statement of contributions 17

Bibliography 23

II Appended Papers 35

Paper I - QuickFuzz testing for fun and profit
- 1 Introduction . . . . . 1 (I)
- 2 Background . . . . . 3 (I)
- 3 A Quick Tour of QuickFuzz . . . . . 8 (I)
- 4 Automatically Deriving Random Generators . . . . . 11 (I)
- 5 Detecting Unexpected Termination of Programs . . . . . 18 (I)
- 6 Evaluation . . . . . 20 (I)
- 7 Related Work . . . . . 27 (I)
- 8 Conclusions and Future Work . . . . . 28 (I)

Paper II - Branching Processes for QuickCheck Generators
- 1 Introduction . . . . . 1 (II)
- 2 Background . . . . . 2 (II)
- 3 Simple-Type Branching Processes . . . . . 5 (II)
- 4 Multi-Type Branching Processes . . . . . 9 (II)
- 5 Terminal Constructors . . . . . 12 (II)
- 6 Mutually-Recursive and Composite ADTs . . . . . 14 (II)
- 7 Implementation . . . . . 18 (II)
- 8 Case Studies . . . . . 20 (II)
- 9 Related Work . . . . . 22 (II)
- 10 Final Remarks . . . . . 24 (II)

1 Demonstrations . . . . 25 (II)
2 Additional Information . . . . 31 (II)

Paper III - Generating Random Structurally Rich Algebraic Data Type Values
1 Introduction . . . . 1 (III)
2 Background . . . . 2 (III)
3 Sources of Structural Information . . . . 3 (III)
4 Capturing ADTs Structure . . . . 6 (III)
5 Predicting Distributions . . . . 7 (III)
6 Case Studies . . . . 10 (III)
7 Related Work . . . . 13 (III)
8 Final Remarks . . . . 13 (III)

Paper IV - Deriving Compositional Random Generators
1 Introduction . . . . 1 (IV)
2 Random Generators in Haskell . . . . 4 (IV)
3 Modular Random Constructions . . . . 7 (IV)
4 Generating Random Constructions . . . . 13 (IV)
5 Type-Level Generation Specifications . . . . 20 (IV)
6 Benchmarks and Optimizations . . . . 22 (IV)
7 Related Work . . . . 24 (IV)
8 Conclusions . . . . 26 (IV)

Paper V - BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs
1 Introduction . . . . 1 (V)
2 Generating Source Annotations Using Source Plugins . . . . . . 4 (V)
3 Consuming Source Annotations . . . . 7 (V)
4 Extensions . . . . 10 (V)
5 Case Study: Theorem Proving EDSL . . . . 12 (V)
6 Discussion . . . . 16 (V)
7 Conclusions . . . . 19 (V)

Paper VI - Short Paper: Weak Runtime-Irrelevant Typing for Security
1 Programming with Type Constraints . . . . 1 (VI)
2 Weakening Runtime-Irrelevant Typing . . . . 2 (VI)
3 Implementation . . . . 6 (VI)
4 Conclusions and Future Work . . . . 7 (VI)

Paper VII - MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations
1 Introduction . . . . 1 (VII)
2 Background . . . . 3 (VII)
3 Mutagen . . . . 5 (VII)
4 Mutagen Heuristics . . . . 10 (VII)
5 Case studies . . . . 11 (VII)

6 Evaluation . . . . 16 (VII)
7 Threats to Validity . . . . 21 (VII)
8 Related work . . . . 22 (VII)
9 Conclusions . . . . 23 (VII)

Part I

Overview

Chapter 1

Introduction

Testing software is critical to ensure its correctness, robustness and reliability. So much so, that it is hard to imagine that any non-trivial system would be released without having first thoroughly validated it meets its functional and security requirements. Despite this, most of the software we use nowadays is frustratingly buggy in one way or another. For the most part, the bugs we experience daily are no more than little annoyances left there to remind us how much of a can of worms computers can be, e.g., Slack randomly crashing *only* when VirtualBox is running in the background.[1] However, every so often, some bugs are so influential that they make it all the way to the news. Most of these are yet another buffer overflow in an open-source library that was exploited to run arbitrary code and steal sensitive data from millions of users — news outlets have slow days, too. But not all of them are caused by using deprecated string functions that no one cared enough to update. The ones that stick are those that evoke empathy for the humans behind them. Let us take CVE-2014-1266 (a.k.a. Apple's *goto fail*) as an example:

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
... // Other checks
err = sslRawVerify(...);
...
fail:
    ... // Cleanup code
    return err;
```

[1] A real bug that some software gurús claim is most likely caused by having a screensaver enabled in the guest virtual machine. A different school of thought, however, believes the issue comes from using VirtualBox's bidirectional clipboard. None of these diagnoses have led to a permanent solution, and the bug is still at large at the time of writing. Intrigued readers can find the entrance of the rabbit hole at `https://www.virtualbox.org/ticket/20022`.

A code fragment similar to the one above was supposed to validate the authenticity of a secure SSL/TLS connection, but the repeated `goto fail`; causes it to skip a vital check `sslRawVerify` and return a non-error code. This, in turn, opened the door for potential eavesdropping by a malicious attacker, affecting anyone running iOS or macOS up until early 2014.

Interestingly, this critical security bug was not caused by a subtle race condition, a memory leak, or anything else making it hard to reason about. It was likely just a simple copy-and-double-paste mistake that put Apple's SSL/TLS system and thus the trust in the security of their products in shambles.

"Who is responsible for all this havoc?" One might ask.

One could start by claiming that C allowing **if** statements without curly braces in their **then** block is the main offender here. But C is more than 50 years old, and by now we should all be used to dealing with its quirks and features. Changing them would take more effort than it is worth spending, and still, we risk making matters worse, so we quickly establish that whatever the C standard accepts as a valid program is *the law*, and we must abide by it.

We want names here, so let us try to pin the crime on the development process instead. The first suspect is the team who wrote the buggy code. They seem to have done so at 16:45 on a Friday, though. We should give them some leeway to make an honest mistake. The second suspect in the police lineup is the team in charge of writing tests for that code, but it turns out their alibi was having a "case of the Mondays" at the time of the crime, so they deserve to get some slack too. The last suspect we gather is the project manager responsible for that code, under the presumption that they could have allocated more budget into testing it before it was deployed. Easier said than done, the defendant claimed. "Budget (either time or financial) is finite, and the project was already over it anyways." After conducting this thorough fictitious investigation, the case remains open.

Hopefully clear by now, we cannot expect us humans[2] to deliver perfectly reliable solutions at any step of the development process of a critical software system. One way to improve the quality of software is, perhaps unsurprisingly, to put as much effort as possible into testing and/or formally verifying it. Even more so when a system is both critical and widely used, as it becomes an attractive target for attackers looking for a challenge. However, finding programmers who enjoy and are good at testing (or verifying) software is hard. To tackle this problem in the supply chain, the software industry tends to complement any existing testing effort with raw computing power and automation. Instead of carefully crafting test code that covers all possible corners of a system, the system can be fed with *many* random, arbitrary inputs until something inevitably breaks — signaling the existence of a bug or even a security vulnerability. This simple idea is the essence of *software fuzzing* and it is remarkably powerful at finding bugs that uncover corner cases that humans cannot easily spot in advance.

Fuzzing is quite popular in part because it is very automatable: it can run 24/7 with minimal intervention, and changes in the codebase do not necessarily

[2]Assuming the reader is not a bot scraping this thesis to train an AI model.

require any change in the testing harness. In some occasions, however, fuzzing can be *too coarse* to be effective. This is because fuzzers normally handle full systems at once, which either break or crash with a given input or do not. The problem is that even if a system does not crash with a given input, there might still be plenty of opportunities for something to go wrong inside of it. With this in mind, a notable approach for testing systems is known as *Random Property-Based Testing* (RPBT), where we split the initial assertion "does the system crash?" into several *testing properties*, each one specialized on validating a particular aspect of the expected behavior of the system under test, e.g., the correctness of a concrete optimization pass in a compiler; the consistency of a database after a trigger is executed; the parser and pretty-printer of a programming language being "somewhat" inverse with each other, and so on. Running these specialized properties in tandem not only gives us a better understanding of the correctness of a system but also simplifies finding the origin of the bugs they find, as individual testing properties tend to encompass smaller portions of the codebase.

Although RPBT is very useful to validate the properties a system must satisfy, as these properties drift away from the relatively simple initial assertion "does the system crash?" they become more challenging to test effectively. This is in part because using test inputs resembling random noise to test complex properties can render the whole process ineffective, since the code they intend to test might rely on non-trivial internal assumptions, e.g., their inputs having already been validated against some specification. This is often referred to as having properties with "sparse" preconditions, and it is accepted that testing them effectively requires substantial manual intervention. In this scenario, programmers need to tailor the generation process that produces random testing inputs to satisfy these sparse preconditions on a reasonable basis so they can penetrate the surface layers of the system under test. This increases the chances of finding real bugs (because the inputs remain mostly within their specification) but, at the same time, increases costs and opens the door for human bias.

As mentioned earlier, being automatable is a key feature for these testing approaches to be used in real-world systems. Thus, the need for manual intervention when requirements become less trivial makes them much less appealing, and it seems as though we are also back to where we started from: having to trust humans to manually do the right thing, even on Mondays. Fortunately, there are many of us, some being lucky enough to have the opportunity to dive into this timely problem a little deeper, so the goals of this thesis are to:

1. Recognize the limitations of existing automated testing approaches, focusing on the aspects that deter effectiveness in the face of testing systems with sparse preconditions.

2. Develop both theoretical and empirical techniques to tackle these limitations. To this purpose, we rely on well-established ideas such as coverage-guided fuzzing, meta-programming, type-level programming, as well as novel interpretations of centuries-old statistical tools designed to study the evolution of populations such as branching processes.

3. Develop software tools implementing these ideas to enable the end-user to test their systems as automatically as possible. These tools are released as publicly available open-source software.

4. Demonstrate how these ideas are robust enough to improve the state-of-the-art of automated PBT, collecting and presenting enough empirical data for the reader to make informed decisions in their future testing endeavors.

Why Haskell? Strongly-typed programming languages like Haskell are prime tools for developing automated testing techniques. This is because programmers can statically encode much of the structure of their programs using the language's type system, e.g., by defining custom algebraic data types that precisely represent the program's inputs. This enables the compiler to collect useful information and pass it onto our automated testing framework, which can later use it to fine-tune the testing process.

Although we use Haskell as the lingua franca of this thesis, the ideas presented here should be reproducible in other programming languages with similar features up to a reasonable extent.

Thesis structure This thesis includes seven peer-reviewed articles accepted to journals, conferences, symposia and workshops spread out across different academic communities, with Software Testing (Papers I, III, and VII) and Programming Languages (Papers II, IV, V and VI) being the main categories. Naturally, these boundaries are rather fuzzy so, to make some justice to this taxonomy, Figure 1 groups these articles by their areas of contribution.

The rest of this chapter briefly outlines the main ideas covered by this thesis, indicating when progress has been made in some of the peer-reviewed papers included in it.

1 Fuzzing

Fuzzing [1] is a technique used in software testing and security analysis (e.g., penetration testing [2]) which involves providing unexpected inputs to a system under test. A program that performs fuzzing campaigns to test a program is colloquially known as a *fuzzer*. The intuition behind a fuzzer is simple: it picks an input from some inputs repository, feeds it to the system under test, and monitors its behavior to signal different kinds of results, e.g., normal executions, crashes, memory leaks and failed code assertions. This process is repeated in a loop until something bad happens in the target system or, alternatively, until a stopping condition is reached (e.g., number of tests or total time). Then, any anomaly detected in the expected behavior of the system under test is reported along with the input producing it. Figure 2 shows a simplified representation of this approach.

Real-world fuzzers typically implement certain tweaks to boost the chances of finding different kinds of vulnerabilities with remarkable success [3]–[14].

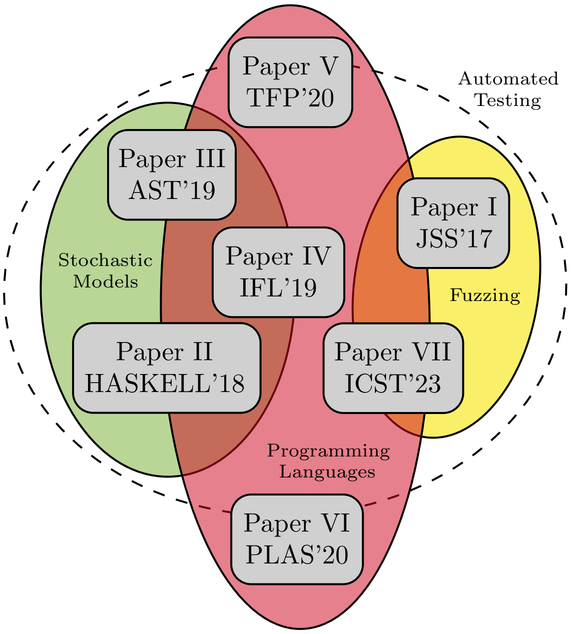


Figure 1: Areas of contribution of each paper included in this thesis.

One of the biggest differentiators between fuzzers is the nature of the inputs used to test the system under test, covering the full spectrum from completely random noise to completely semantically valid ones. Moreover, the origin of these inputs denotes an important distinction used to classify different kinds of fuzzing models [15], as described below.

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

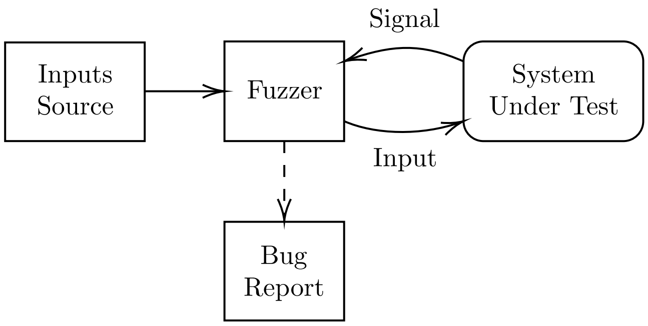


Figure 2: Simplified representation of a fuzzing environment.

input 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 test inputs from scratch using a model that describes the format of the inputs expected by the system under test. These models do not necessarily need to fully match the specification of the system under test, and deliberately producing almost valid but yet broken inputs can still be useful to uncover certain kinds of bugs. 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 deep domain knowledge, which can be tedious and expensive to achieve.

In this work, we focus particularly on the generational approach, although Paper VII demonstrates a hybrid technique that could partially fit both the generational and mutational categories — more on this later. We aim to develop automated techniques for the random generation of unexpected inputs based on statically-known information. This information can be extracted either directly from the system under test or from other external sources. In particular, Paper I is focused on automatically leveraging existing file-format manipulating libraries to derive random input generators used to test massively used programs. This led us to discover dozens of bugs and security vulnerabilities.

2 Property-Based Testing and 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 a more formal specification than just "does the system crash?" This specification can be defined, for instance, as a set of testing properties that our code must fulfill for every possible input. These properties do not necessarily involve only the input format of the system under test, but can also be specified in terms of intermediate or specialized data formats, e.g., a parsed abstract syntax tree, a serialized value, a set of command-line flags, etc. Then, these properties can be individually validated using a large number of randomly generated inputs. As mentioned earlier, this is the basis for the technique known as *Random Property-Based Testing* (RPBT).

In the Haskell realm, QuickCheck [16] is the de facto tool of this sort. Originally conceived by Koen Claessen and John Hughes, 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 [17]–[25].

Using QuickCheck requires the programmer to interact with two main components: *executable testing properties* and *random data generators.* Akin to a fuzzer, testing properties encapsulate the system under test (or a portion of it) into an executable predicate that signals whether an input produces an unexpected result. Moreover, concrete input sources such as corpora are instead replaced with random data generators that produce random inputs on-the-fly. Figure 3 shows a simplified representation of this approach. Although this thesis focuses strictly on improving random data generators, automating the process of deriving testing specifications is also a non-trivial task that comprises a research field of its own [16], [26]. For completeness, the following subsections briefly introduce the reader to the usage of both components.

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`] → [`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 list. This property of our function can be written in QuickCheck simply as a Haskell predicate parameterized over its input, which we can think of as being universally quantified:

```
prop_reverse_ok :: [Int] → Bool
prop_reverse_ok xs =
  reverse (reverse xs) == xs
```

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 (`xs`) of our property using a large number of randomly generated lists of integers, asserting that `prop_reverse_ok` returns `True` for all of them.

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 a list of integers once will not always return the original list:

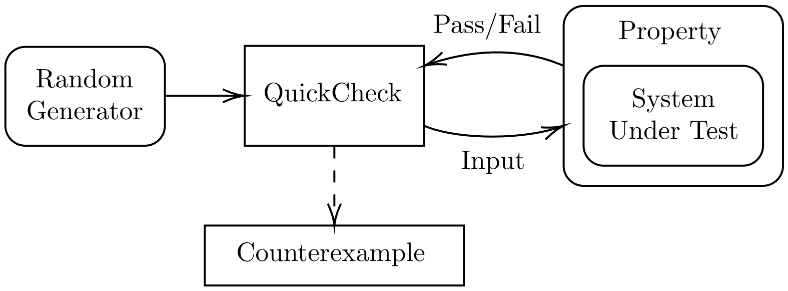


Figure 3: Random Property-Based with QuickCheck.

```
prop_reverse_bad :: [Int] → Bool
prop_reverse_bad xs =
  reverse xs == xs
```

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 of 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's *type classes* [27]. In particular, QuickCheck defines the `Arbitrary` type class for the types that can be randomly generated:

```
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a → [a]
```

The interface of this type class encodes two basic primitives. Firstly, `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, `shrink :: a → [a]` specifies how a given counterexample (of type `a`) can be minimized into different smaller ones. This function is used to report a 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. This way, 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
```

```
instance Arbitrary Html where
  arbitrary = oneof
    [Text  <$> arbitrary
    ,Sing  <$> arbitrary
    ,Tag   <$> arbitrary <*> arbitrary
    ,(:+:) <$> arbitrary <*> arbitrary]
```

Figure 4: Naïve type-driven random generator of `Html` values.

This type allows building 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`. When generating random values, 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 these 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 4 (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” its fields recursively. This type-driven 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

Although simple, writing the random generator defined in Figure 4 can be quite tedious, so it is of no surprise that automated derivation mechanisms [28], [29] have emerged to relieve the programmer of the burden of this task—something especially valuable for large data types! Most of these tools use Template Haskell [30], the Haskell meta-programming framework, which allows one to examine the user code and synthesize new code based on it.

However, a suitable mechanism for deriving random generators cannot be as simple as just producing code like the one shown in Figure 4. This naïve generator is ridden with flaws, and QuickCheck users are often aware of them when implementing random generators — even an unfamiliar but attentive reader might have recognized them too. 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 4 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 a simple mechanism to overcome this issue—this is addressed by papers I-IV presented in this thesis.

Generation parameters: The generator from Figure 4 picks the next random constructor on a uniform basis. This is the simplest approach we can mechanically follow but 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 we later discovered to be a science on its own. Both problems are addressed in detail in Paper II, where we use a stochastic model known as a *branching process* to model, predict and optimize the generation process on demand.

Abstraction level: The generation process encoded in the generator shown in Figure 4 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 constructors that could be required to test the corner cases of our code, leaving the door open for subtle bugs that might never get triggered within the testing budget.

On the other hand, the implementation of our code under test could rely on internal invariants that are necessary to make it work properly — consider 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. In this case, our testing properties will likely require providing somewhat well-formed inputs as a precondition. Thus, testing this kind of software becomes much more complicated using the approach described above, as constructing random values one data constructor at a time will very rarely produce values satisfying such preconditions. This issue is addressed in detail in Paper III, where we first show how extra static information present in the codebase can be used to generate better random data automatically by including abstract interfaces and functions' branching patterns in the mix. Later, in Paper IV we show how this enhancement can be implemented with modularity in mind in an elegant way using type-level programming. Using this idea, different static sources of structural information can be combined to generate data showing different lightweight invariants on a per-property basis.

4 Coverage Guided, Property-Based Testing

As described above, Papers I-IV contribute to the state-of-the-art of automatic derivation of random data generators based on static information. These generators are initially intended to be used with a black-box testing framework such as QuickCheck, where they show a noticeable improvement with respect to other existing automated techniques in certain real-world testing scenarios.

Being intentionally designed to follow a black-box approach, the only signal QuickCheck gets back from running a test is whether it passes, fails or gets discarded due to not passing the property's precondition. This design choice is in part what makes QuickCheck fast and easy to use. However, it also entails that if our random generators (automatically derived or otherwise) cannot generate data satisfying the sparse precondition of a testing property, then QuickCheck has no other choice but to give up early. This led us to believe that, regardless of the improvements we developed in the previous papers, automatically derived generators can still be remarkably ineffective when used to test properties with sparse preconditions if we limit ourselves to black-box RPBT. Concretely, the main limitation of the black-box approach is that neither the testing loop, the property nor the random generator can tune their behavior dynamically based on feedback taken from the execution of the system under test — a missed opportunity that the fuzzing community has taken advantage of many times in the past.

If we accept moving away from QuickCheck, we can enhance its testing loop with two key features to make it more flexible:

- **Target code instrumentation:** to capture execution information from each test case. For this, we instrument each branching point of the code in the system under test with a small wrapper that logs its execution in a global execution trace. This way, the testing loop can retrieve the path in the code of the system under test taken by each test case.

- **High-level, type-preserving mutations:** to produce syntactically valid test cases by altering existing ones at the data constructor level. Similar to automatically derived generators, these mutations can also be automatically computed by inspecting the data types used to represent input test cases using meta-programming.

Relying on code instrumentation in tandem with mutations is a well-established testing technique used outside of RPBT. Existing fuzzing tools use execution traces to recognize interesting test cases, e.g, those that exercise previously undiscovered parts of the target code [3], [4], [31]–[33]. Moreover, given a valid input test case, high-level, type-preserving mutations are a useful tool for producing new valid input test cases from existing ones.

These two features are tied together using a mutation pool that stores previously executed test cases. On one end, whenever a new test case discovers a new portion of the code in the system under test, its corresponding value is saved in the mutation pool. On the other end, values are drawn from this pool and mutated to create a new test case similar to the original one. If the mutation pool is empty, the testing loop simply generates a new random test case and repeats the process. Figure 5 outlines a simplified representation of this approach.

This technique was originally conceived by Lampropoulos *et al.* under the name of *Coverage-Guided, Property-Based Testing* (CGPT). In their original work, this technique leaves considerable room for improvement. Concretely, we observed a large reliance on randomness, which in conjunction with its simple, sub-par scheduling can prevent bugs from being found on a timely basis. To tackle these issues, Paper VII introduces MUTAGEN, a novel CGPT

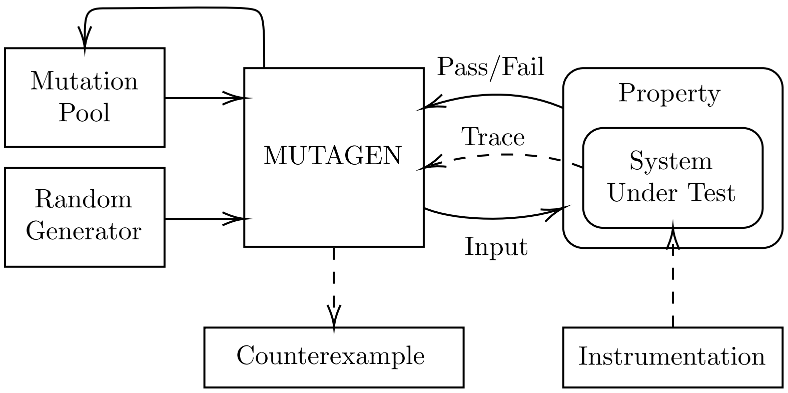


Figure 5: Coverage-Guided, Property-Based Testing with MUTAGEN.

tool that addresses the main limitations of the original CGPT approach using exhaustively computed mutations along with dynamic heuristics to improve its scalability.

5 Domain-Specific Programming Language Tools

In addition to automated software testing, this thesis covers two contributions to the state-of-the-art of domain-specific programming languages. This section discusses how the ideas behind these contributions could be used to solve some of the main problems this thesis aims to tackle.

5.1 Enhancing Embedded Domain-Specific Languages

Embedded Domain Specific Languages (EDSLs) are a useful approach to developing custom programming languages tailored to specific requirements without reinventing the wheel. Instead of having to manually implement lexers, parsers, type-checkers, optimizers or code-generators, to name a few, EDSLs can be designed to reuse some (or all) of these components from a host language. In this setting, Haskell excels at hosting EDSLs given its powerful type system and relatively extensible syntax, where its monadic **do** notation is extremely powerful to implement complex EDSLs.

Despite their evident appeal, EDSLs are not without limitations. Perhaps one of the most common ones is the lack of source metadata collected by earlier stages of the compilation pipeline. For instance, parsers often collect useful metadata that later compilation passes rely on to emit code or to generate error messages. Understandably, most compilers would not make this metadata available to the compiled program, GHC being no exception. This, in turn, limits how expressive our Haskell EDSLs can be, as they cannot access useful source-level details of the embedded programs such as source locations or variable names.

To alleviate this problem, Paper V describes BinderAnn, a flexible plugin for the GHC compiler that automatically inserts source code annotations into the user's monadic EDSL code. These annotations capture both the location (i.e., line number and file name) of every EDSL statement, as well as any name that the statement might be bound to (using the $\leftarrow$ operator). Enabling our EDSLs to use this (otherwise lost) information makes it possible to implement better code-generating tools, as well as improve the quality of domain-specific error messages.

We demonstrate using several examples how BinderAnn can be used to enhance existing EDSLs with source annotations, e.g, improving code-generation (where variables names in the generated code reflect those used in the EDSL code), as well as improving error messages (where EDSLs can guide the user to domain-specific errors using source-level locations). Moreover, we showed how EDSLs enhanced with source annotations can open the door for new programming patterns. In particular, we demonstrate how BinderAnn can be an instrumental tool to implement a simple interactive proof assistant.

Using the BinderAnn approach, it could be possible automatically insert source annotations into the system under test to guide the testing process towards new kinds of goals. In this setting, there exist several fuzzing techniques that take advantage of external information to direct their efforts towards exercising specific parts of the code, e.g., to maximize the time spent on fuzzing recent changes in the codebase or to target code that uses specific functions or that has potential vulnerabilities [35]–[37]. In our case, we could instruct a RPBT tool like MUTAGEN to use the extra source information added at compile time to optimize the effort put into testing certain EDSL code more effectively and at a higher level of abstraction — an interesting challenge to tackle in the future.

5.2 Weakening the Type System

Continuing with the topic of embedded languages, we will now focus on a specific kind: EDSLs that use Haskell's type-level capabilities to enforce domain-specific constraints. Perhaps the most remarkable example of this technique is the existence of security *libraries*, where programmers can specify Information-Flow Control policies in their programs by using type-level constraints to denote the sensitivity of the data handled by them. These constraints are checked along with the rest of the code, preventing the program to compile in the presence of forbidden information flows.

Despite providing strong domain-specific guarantees, EDSLs relying on type-level constraints often suffer from a lack of adoption. For instance, a reverse dependency search for the most (academically) popular security libraries (e.g., *MAC*, *LIO*, *HLIO*, etc.) in Hackage returns fewer than five results combined. We hypothesize that the extra friction added by using domain-specific type-level constraints takes a toll on the usability of these EDSLs, where users must fully abide by the language's domain-specific quirks before an otherwise functionally correct program can be even compiled. We argue that this all-or-nothing paradigm can be improved by letting the programmer start by developing functionally correct code, and then gradually tweak it until the domain-specific constraints are met. This can be particularly useful in the case of security libraries, where the programmer can progressively adapt their existing code to satisfy the desired security policy, but checking that the functional correctness of the program is also satisfied after each refactoring step — an approach akin to that popularized by gradually typed languages.

To tackle this issue, Paper VI describes WRIT, a type-checking plugin for the GHC compiler that *weakens* the type system to allow for certain ill-typed programs to compile in a controlled way. The kind of programs that WRIT allows GHC to compile are those that would otherwise compile just fine should their type-level constraints were removed. In other words, WRIT allows GHC to ignore type errors caused by unsolved *runtime irrelevant* constraints. We found that, oftentimes, domain-specific constraints are irrelevant at runtime, having no runtime representation, hence the compiler erases them completely during compilation. This is commonly seen when using type-level tools like phantom types and empty type classes. In such cases, WRIT allows the programmer to transform runtime irrelevant type errors into warnings to be addressed later. Additionally, our plugin enables the EDSL designer to replace the generic error messages GHC produces when there are unsolved type-level constraints with specialized ones that refer to the concrete domain-specific constraint the client code violates.

We showed how this approach can be used to write programs matching a static IFC policy using the *MAC* library in a gradual manner and with more descriptive error messages when something goes wrong. In addition, we discussed how our approach can be extended to consider other scenarios when unsolved constraints are not runtime-irrelevant, but there exists a mechanical way to solve them by simulating dynamic typing as seen in other languages like Python or Erlang.

Weakening the type system in our favor could be used to further automate the testing process by dividing the complexity of generating test cases into different steps. For instance, if we want to test a security library, we could try to randomly generate programs using its abstract interface and verify that its security guarantees are preserved. This might not be an easy task, though. The type-level security constraints used by such a library might be too hard to satisfy using an automatically derived (or even a manually written) random generator [38], [39]. In turn, we could start by generating weaker programs that compile albeit with potential security flaws. These programs could be automatically lifted to use the library's interface via the WRIT plugin, generating security warnings in the process. Then, a subsequent step could use the information encoded into these security warnings to try to automatically "patch" their corresponding random programs into ones that still use the library's interface but compile without warnings. We believe this multi-level technique for generating random test cases could be useful to complement MUTAGEN's mutation approach to overcome the complexity of testing properties with complex security preconditions.

Chapter 2

Statement of contributions

This chapter lists the abstracts of the individual paper chapters and outlines the personal contributions for each.

Paper I - QuickFuzz testing for fun and profit

Gustavo Grieco, Martín Ceresa, **Agustín Mista** and Pablo Buiras

Abstract

Fuzzing is a popular technique to find flaws in programs using invalid or erroneous inputs but not without its drawbacks. On one hand, mutational fuzzers require a set of valid inputs as a starting point, in which modifications are then introduced. On the other hand, generational fuzzing allows synthesizing somehow valid inputs according to a specification. Unfortunately, this requires to have a deep knowledge of the file formats under test to write specifications of them to guide the test case generation process.

In this paper, we introduce an extended and improved version of QuickFuzz, a tool written in Haskell designed for testing unexpected inputs of common file formats on third-party software, taking advantage of off-the-self well known fuzzers.

Unlike other generational fuzzers, QuickFuzz does not require to write specifications for the files formats in question since it relies on existing file-format-handling libraries available on the Haskell code repository. It supports almost 40 different complex file types including images, documents, source code and digital certificates.

In particular, we found QuickFuzz useful enough to discover many previously unknown vulnerabilities in real-world implementations of web browsers and image processing libraries among others.

Contributions

This project was a collaboration between people from CIFASIS-Conicet and Chalmers University of Technology. Agustín contributed to this project by i) developing an extension to the existing generators' derivation mechanism, which contemplates the common case of existing libraries written using shallow embeddings of the target file format; and ii) carrying out a complete rewrite of the testing harness from scratch, maximizing the use of meta-programming to ease the task of adding support for new file-format targets.

Moreover, Agustín collaborated with the writing of the journal paper resulting from this project.

Paper II - Branching Processes for QuickCheck Generators

Agustín Mista, Alejandro Russo and John Hughes

Abstract

In *QuickCheck* (or, more generally, random testing), it is challenging to control random data generators' distributions—especially when it comes to *user-defined algebraic data types* (ADT). In this paper, we adapt results from an area of mathematics known as *branching processes*, and show how they help to analytically predict (at compile-time) the expected number of generated constructors, even in the presence of mutually recursive or composite ADTs. Using our probabilistic formulas, we design heuristics capable of automatically adjusting probabilities in order to synthesize generators whose distributions are aligned with users' demands. We provide a Haskell implementation of our mechanism in a tool called *DRAGEN* and perform case studies with real-world applications. When generating random values, our synthesized *QuickCheck* generators show improvements in code coverage when compared with those automatically derived by state-of-the-art tools.

Contributions

This project was a collaboration with Alejandro Russo. Agustín was responsible for i) developing a generic meta-programming mechanism for deriving random generators using the stochastic model based on branching processes (the first version of DRAGEN), and ii) designing and carrying out the evaluation of these ideas, comparing the results of different generator derivation techniques in terms of the code coverage observed when feeding real-world applications with randomly generated inputs.

The technical writing of this paper was initially done in equal parts between Alejandro and Agustín. John Hughes joined at a later stage with invaluable feedback.

Paper III - Generating Random Structurally Rich Algebraic Data Type Values

Agustín Mista and Alejandro Russo

Abstract

Automatic generation of random values described by algebraic data types (ADTs) is often a hard task. State-of-the-art random testing tools can automatically synthesize random data generators based on ADTs definitions. In that manner, generated values comply with the structure described by ADTs, something that proves useful when testing software that expects complex inputs. However, it sometimes becomes necessary to generate structurally richer ADTs values in order to test deeper software layers. In this work, we propose to leverage static information found in the codebase as a manner to improve the generation process. Namely, our generators are capable of considering how programs branch on input data as well as how ADTs values are built via interfaces. We implement a tool, responsible for synthesizing generators for ADTs values while providing compile-time guarantees about their distributions. Using compile-time predictions, we provide a heuristic that tries to adjust the distribution of generators to what developers might want. We report on preliminary experiments where our approach shows encouraging results.

Contributions

This project was a collaboration with Alejandro Russo. Agustín contributed to this project by i) extending the previous derivation tool and its underlying stochastic model with support for extracting and generating function patterns and API calls automatically (this extension is called DRAGEN2); and ii) designing and carrying out the evaluation of these ideas, comparing the effects of including more static information when deriving random data generators versus using a simple type-directed derivation approach.

The technical writing of this paper was done in equal parts between Alejandro and Agustín.

Paper IV - Deriving Compositional Random Generators

Agustín Mista and Alejandro Russo

Abstract

Generating good random values described by algebraic data types is often quite intricate. State-of-the-art tools for synthesizing random generators serve the valuable purpose of helping with this task, while providing different levels of

invariants imposed over the generated values. However, they are often not built for composability nor extensibility, a useful feature when the shape of our random data needs to be adapted while testing different properties or subsystems.

In this work, we develop an extensible framework for deriving compositional generators, which can be easily combined in different ways in order to fit developers' demands using a simple type-level description language. Our framework relies on familiar ideas from the à la Carte technique for writing composable interpreters in Haskell. In particular, we adapt this technique with the machinery required in the scope of random generation, showing how concepts like generation frequency or terminal constructions can also be expressed in the same type-level fashion. We provide an implementation of our ideas, and evaluate its performance using real-world examples.

Contributions

This project was a collaboration with Alejandro Russo. Agustín was responsible for i) carrying out the technical development, using meta-programming and type-level features available in Haskell to derive composable random data generators; and ii) designing and evaluating these ideas, which focus on the runtime overhead induced by the usage of composable random data generators.

The majority of the technical writing was done by Agustín. Alejandro provided invaluable feedback throughout the process.

Paper V - BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs

Agustín Mista and Alejandro Russo

Abstract

Embedded Domain-Specific Languages (EDSLs) are an alternative to quickly implement specialized languages without the need to write compilers or interpreters from scratch. In this territory, Haskell is a prime choice as the host language. EDSLs in Haskell, however, are often incapable of reifying useful static information from the source code, namely variable binding names and source locations. Not having access to variable names directly affects EDSLs designed to generate low-level code, where the variables names in the generated code do not match those found in the source code—thus broadening the semantic gap between source and target code. Similarly, many existing EDSLs produce poor error messages due to the lack of knowledge of source locations where errors are generated.

In this work, we propose a simple technique for enhancing monadic EDSLs expressed using **do** notation. This technique employs *source-to-source plugins*, a relatively new feature of GHC, to annotate every **do** statement of our EDSLs with relevant information extracted from the source code at compile time. We show how these annotations can be incorporated into EDSL designs either

directly inside values or as monadic effects. We provide *BinderAnn*, a GHC source plugin implementing our ideas, and evaluate it by enhancing existing real-world EDSLs with relatively minor modification efforts to contemplate the source-level static information related to variables names and source locations.

Contributions

This project was a collaboration with Alejandro Russo. Agustín contributed to this project by i) designing and implementing BinderAnn with support for multiple annotation styles based on valuable input from Alejandro, Koen Claessen and John Hughes; and ii) evaluating these ideas, showing how BinderAnn could solve existing real-world problems, as well as allowing for new programming patterns to emerge.

The majority of the technical writing was done by Agustín. Alejandro Russo provided invaluable feedback throughout the process.

Paper VI - Short Paper: Weak Runtime-Irrelevant Typing for Security

Matthías Páll Gissurarson and **Agustín Mista**

Abstract

Types indexed with extra type-level information are a powerful tool for statically enforcing domain-specific security properties. In many cases, this extra information is runtime-irrelevant, and so it can be completely erased at compile-time without degrading the performance of the compiled code. In practice, however, the added bureaucracy often disrupts the development process, as programmers must completely adhere to new complex constraints in order to even compile their code.

In this work we present WRIT, a plugin for the GHC Haskell compiler that relaxes the type-checking process in the presence of runtime-irrelevant constraints. In particular, WRIT can automatically coerce between runtime equivalent types, allowing users to run programs even in the presence of some classes of type errors. This allows us to gradually secure our code while still being able to compile at each step, separating security concerns from functional correctness.

Moreover, we present a novel way to specify which types should be considered equivalent for the purpose of allowing the program to run, how ambiguity at the type level should be resolved and which constraints can be safely ignored and turned into warnings.

Contributions

This project was a collaboration with Matthías Páll Gissurarson, and the result of a joint project for the course Language-Based Security led by Andrei

Sabelfeld. Agustín contributed to this project by helping with the design of the WRIT GHC plugin and its case studies.

The technical writing of this paper was done in equal parts between Matthías and Agustín.

Paper VII - MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations

Agustín Mista and Alejandro Russo

Abstract

Automatically-synthesized random data generators are an appealing option when using property-based testing. There exists a variety of techniques that extract static information from the codebase to produce random test cases. Unfortunately, such techniques cannot enforce the complex invariants often needed to test properties with sparse preconditions.

Coverage-guided, property-based testing (CGPT) tackles this limitation by enhancing synthesized generators with structure-preserving mutations guided by execution traces. Albeit effective, CGPT relies largely on randomness and exhibits poor scheduling, which can prevent bugs from being found.

We present MUTAGEN, a CGPT framework that tackles such limitations by generating mutants *exhaustively*. Our tool incorporates heuristics that help to minimize scalability issues as well as cover the search space in a principled manner. Our evaluation shows that MUTAGEN not only outperforms existing CGPT tools but also finds previously unknown bugs in real-world software.

Contributions

This project was a collaboration with Alejandro Russo. Agustín contributed to this project by i) carrying out most of the technical development, with several rounds of helpful feedback from Alejandro, Koen Claessen and John Hughes; and ii) designing and performing the evaluation of these ideas, adapting existing case studies with the assistance of Leonidas Lampropoulos.

The technical writing of this paper was done in equal parts between Alejandro and Agustín, with invaluable feedback from John, Koen and Robert Feldt.

Bibliography

[1] M. Sutton, A. Greene and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, 2007.

[2] B. Arkin, S. Stender and G. McGraw, 'Software penetration testing', *IEEE Security Privacy*, 2005.

[3] M. Zalewski, *American Fuzzy Lop: a security-oriented fuzzer*, `http://lcamtuf.coredump.cx/afl/`, 2010.

[4] R. Swiecki, *Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options*, `https://honggfuzz.dev`, 2017.

[5] Oulu University Secure Programming Group, *A Crash Course to Radamsa*, `https://github.com/aoh/radamsa`, 2010.

[6] Deja vu Security, *Peach: a smartfuzzer capable of performing both generation and mutation based fuzzing*, `http://peachfuzzer.com/`, 2007.

[7] CACA Labs, *zzuf - multi-purpose fuzzer*, `http://caca.zoy.org/wiki/zzuf`, 2010.

[8] Mozilla, *Dharma: a generation-based, context-free grammar fuzzer*, `https://github.com/MozillaSecurity/dharma`, 2015.

[9] J. Wang, B. Chen, L. Wei and Y. Liu, 'Superion: Grammar-aware greybox fuzzing', in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[10] G. Grieco, M. Ceresa and P. Buiras, 'QuickFuzz: An automatic random fuzzer for common file formats', in *Proceedings of the 9th International Symposium on Haskell*, 2016.

[11] M. Böhme, V.-T. Pham and A. Roychoudhury, 'Coverage-based greybox fuzzing as markov chain', in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016.

[12] P. Godefroid, A. Kiezun and M. Y. Levin, 'Grammar-based whitebox fuzzing', in *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, 2008.

[13] S. K. Cha, M. Woo and D. Brumley, 'Program-Adaptive Mutational Fuzzing', in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[14] P. Godefroid, M. Y. Levin and D. Molnar, ‘Sage: Whitebox fuzzing for security testing’, *Communications of the ACM*, vol. 55, 2012.

[15] C. Miller and Z. N. Peterson, ‘Analysis of mutation and generation-based fuzzing’, *Independent Security Evaluators, Techincal Report*, 2007.

[16] K. Claessen and J. Hughes, ‘QuickCheck: A lightweight tool for random testing of Haskell programs’, in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.

[17] 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*, no. ICFP, 2017.

[18] T. Arts, L. M. Castro and J. Hughes, ‘Testing Erlang data types with Quviq Quickcheck’, in *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, 2008.

[19] T. Arts, J. Hughes, U. Norell and H. Svensson, ‘Testing AUTOSAR software with QuickCheck’, in *IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015.

[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] J. Hughes, ‘QuickCheck testing for fun and profit’, in *International Symposium on Practical Aspects of Declarative Languages*, Springer, 2007.

[22] L. Bulwahn, ‘The new QuickCheck for isabelle’, in *International Conference on Certified Programs and Proofs*, Springer, 2012.

[23] T. Arts, J. Hughes, J. Johansson and U. Wiger, ‘Testing telecoms software with Quviq QuickCheck’, in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, 2006.

[24] P. Holser, *Junit-quickcheck: Property-based testing, JUnit-style*, 2019.

[25] L. Pike, ‘Smartcheck: Automatic and efficient counterexample reduction and generalization’, in *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, 2014.

[26] R. Braquehais and C. Runciman, ‘Fitspec: Refining property sets for functional testing’, in *Proceedings of the 9th International Symposium on Haskell, 2016*, 2016.

[27] 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*, 1989.

[28] Neil Mitchell, *Data.Derive is a library and a tool for deriving instances for Haskell programs*, `http://hackage.haskell.org/package/derive`, 2006.

[29] J. Duregård, P. Jansson and M. Wang, ‘Feat: Functional enumeration of algebraic types’, in *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2012.

[30] T. Sheard and S. L. P. Jones, ‘Template meta-programming for Haskell’, *SIGPLAN Notices*, vol. 37, 2002.

[31] *LibFuzzer: A library for coverage-guided fuzz testing.* `http://llvm.org/docs/LibFuzzer.html`, 2019.

[32] S. Dolan and M. Preston, ‘Testing with crowbar’, in *OCaml Workshop*, 2017.

[33] R. Kersten, K. Luckow and C. S. Păsăreanu, ‘Poster: Afl-based fuzzing for java with Kelinci’, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[34] L. Lampropoulos, M. Hicks and B. C. Pierce, ‘Coverage guided, property based testing’, *Proceedings of the ACM on Programming Languages, (OOPSLA)*, 2019.

[35] M. Böhme, V.-T. Pham, M.-D. Nguyen and A. Roychoudhury, ‘Directed greybox fuzzing’, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[36] X. Zhu and M. Böhme, ‘Regression greybox fuzzing’, in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[37] S. Österlund, K. Razavi, H. Bos and C. Giuffrida, ‘Parmesan: Sanitizer-guided greybox fuzzing’, in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020.

[38] C. Hritcu, J. Hughes, B. C. Pierce *et al.*, ‘Testing noninterference, quickly’, *ACM SIGPLAN Notices*, vol. 48, 2013.

[39] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky *et al.*, ‘Testing noninterference, quickly’, *Journal of Functional Programming*, vol. 26, 2016.

[40] S. K. Cha, T. Avgerinos, A. Rebert and D. Brumley, ‘Unleashing Mayhem on Binary Code’, in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[41] B. P. Miller, L. Fredriksen and B. So, ‘An Empirical Study of the Reliability of UNIX Utilities’, *ACM Communications*, vol. 33, 1990.

[42] K. Claessen and J. Hughes, ‘QuickCheck: a lightweight tool for random testing of Haskell programs’, *ACM SIGPLAN Notices*, vol. 46, 2011.

[43] Hackage, *The Haskell community’s central package archive of open source software*, `http://hackage.haskell.org/`, 2010.

[44] Pedram Amini and Aaron Portnoy, *sulley: a pure-python fully automated and unattended fuzzing framework*, `https://github.com/OpenRCE/sulley`, 2012.

[45] M. Höschele and A. Zeller, 'Mining input grammars with AUTOGRAM', in *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.

[46] O. Bastani, R. Sharma, A. Aiken and P. Liang, 'Synthesizing program input grammars', in *Programming Language Design and Implementation (PLDI)*, 2017.

[47] S. Marlow, *Haskell 2010 language report*, 2010.

[48] C. McBride and R. Paterson, 'Applicative programming with effects', *Journal of Functional Programming*, vol. 18, 2008.

[49] V. Berthoux, *Juicy.Pixels: Haskell library to load and save pictures*, `https://hackage.haskell.org/package/JuicyPixels`, 2012.

[50] Eric S. Raymond, *GIFLIB: A library and utilities for processing GIFs*, `http://giflib.sourceforge.net`, 1989.

[51] N. Nethercote and J. Seward, 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation', *SIGPLAN Notices*, vol. 42, 2007.

[52] K. Serebryany, D. Bruening, A. Potapenko and D. Vyukov, 'AddressSanitizer: A fast address sanity checker', in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012.

[53] P. Hudak, 'Modular domain specific languages and tools', in *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, 1998.

[54] J. Svenningsson and E. Axelsson, 'Combining deep and shallow embedding of domain-specific languages', *Computer Languages, Systems & Structures*, vol. 44, 2015.

[55] Jasper Van der Jeugt, *blaze-html: a blazingly fast HTML combinator library for Haskell*, `https://hackage.haskell.org/package/blaze-html`, 2010.

[56] Anton Kholomiov, *language-css: a library for building and pretty printing CSS 2.1 code*, `https://hackage.haskell.org/package/language-css`, 2010.

[57] K. Claessen and J. Hughes, 'Testing monadic code with QuickCheck', *SIGPLAN Notices*, vol. 37, 2002.

[58] X. Yang, Y. Chen, E. Eide and J. Regehr, *CSmith: a tool that can generate random C programs that statically and dynamically conform to the C99 standard*, `https://embed.cs.utah.edu/csmith/`, 2011.

[59] Deja vu Security, *Peach Pits and Pit Packs*, `http://www.peachfuzzer.com/products/peach-pits/`, 2016.

[60] Vincent Berthoux, *svg-tree: SVG loader/serializer for Haskell*, `https://github.com/Twinside/svg-tree`, 2007.

[61] Willem van Schaik, *The official test-suite for PNG*, `http://www.schaik.com/pngsuite/`, 2011.

[62] PNG Development Group, *libpng: the official PNG reference library*, `http://www.libpng.org/pub/png/libpng.html`, 2000.

[63] Franco Contanstini, *language-python bug report: some stuff missing in Pretty instances*, `https://github.com/bjpop/language-python/issues/30`, 2010.

[64] K. Claessen, J. Duregård and M. H. Palka, 'Generating constrained random data with uniform distribution', in *Proceedings of the Functional and Logic Programming FLOPS*, 2014.

[65] M. Pałka, K. Claessen, A. Russo and J. Hughes, 'Testing an optimising compiler by generating random lambda terms', in *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.

[66] A. Zeller and R. Hildebrandt, 'Simplifying and isolating failure-inducing input', *IEEE Transactions on Software Engineering*, vol. 28, 2002.

[67] J. Hughes, B. C. Pierce, T. Arts and U. Norell, 'Mysteries of Dropbox: Property-based testing of a distributed synchronization service', in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[68] C. Runciman, M. Naylor and F. Lindblad, 'Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values', in *Proceedings of the ACM SIGPLAN Symposium on Haskell*, 2008.

[69] N. Mitchell, 'Deriving generic functions by example', in *Proceedings of the 1st York Doctoral Syposium*, 2007.

[70] G. Grieco, M. Ceresa and P. Buiras, 'QuickFuzz: An automatic random fuzzer for common file formats', in *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2016.

[71] G. Grieco, M. Ceresa, A. Mista and P. Buiras, 'QuickFuzz testing for fun and profit', *Journal of Systems and Software*, vol. 134, 2017.

[72] L. Lampropoulos, D. Gallois-Wong, C. Hritcu, J. Hughes, B. C. Pierce and L. Xia, 'Beginner's luck: A language for property-based generators', in *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, 2017.

[73] H. W. Watson and F. Galton, 'On the probability of the extinction of families', *The Journal of the Anthropological Institute of Great Britain and Ireland*, 1875.

[74] P. Haccou, P. Jagers and V. Vatutin, *Branching processes. Variation, growth, and extinction of populations.* Cambridge University Press, 2005, ISBN: 978-0-521-83220-5.

[75] L. Lampropoulos, Z. Paraskevopoulou and B. C. Pierce, 'Generating good generators for inductive relations', *In Proceedings ACM on Programming Languages (POPL)*, 2017.

[76] P. Duchon, P. Flajolet, G. Louchard and G. Schaeffer, 'Boltzmann samplers for the random generation of combinatorial structures', *Combinatorics, Probability and Computing*, vol. 13, 2004.

[77] M. Bendkowski, K. Grygiel and P. Tarau, 'Boltzmann samplers for closed simply-typed lambda terms', in *In Proceedings of the ACM International Symposium on Practical Aspects of Declarative Languages*, 2017.

[78] S. M. Poulding and R. Feldt, 'Automated random testing in multiple dispatch languages', *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[79] R. Feldt and S. Poulding, 'Finding test data with specific properties via metaheuristic search', in *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[80] F. Chan, T. Chen, I. Mak and Y. Yu, 'Proportional sampling strategy: Guidelines for software testing practitioners', *Information and Software Technology*, vol. 38, 1996.

[81] T. Y. Chen, H. Leung and I. K. Mak, 'Adaptive random testing', in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, 2005.

[82] A. Arcuri and L. Briand, 'Adaptive random testing: An illusion of effectiveness?', in *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, 2011.

[83] I. Ciupa, A. Leitner, M. Oriol and B. Meyer, 'ARTOO: Adaptive random testing for object-oriented software', in *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[84] N. Balakrishnan, V. Voinov and M. Nikulin, *Chi-Squared Goodness of Fit Tests with Applications. 1st Edition.* Academic Press, 2013, ISBN: 9780123971944.

[85] A. Mista, A. Russo and J. Hughes, 'Branching processes for QuickCheck generators', in *Proceedings of the ACM SIGPLAN International Symposium on Haskell*, 2018.

[86] C. Klein and R. B. Findler, 'Randomized testing in PLT Redex', in *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2009.

[87] M. Bendkowski, O. Bodini and S. Dovgal, 'Polynomial tuning of multiparametric combinatorial samplers', in *Proceedings of the 15th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, 2018.

[88] P. Godefroid, N. Klarlund and K. Sen, 'DART: Directed automated random testing', in *ACM Sigplan Notices*, vol. 40, 2005.

[89] A. Mista and A. Russo, 'Generating random structurally rich algebraic data type values', in *Proceedings of the 14th International Workshop on Automation of Software Test*, 2019.

[90] W. Swierstra, 'Data types à la carte', *Journal of Functional Programming*, vol. 18, 2008.

[91] T. Schrijvers, M. Sulzmann, S. P. Jones and M. Chakravarty, 'Towards open type functions for haskell', in *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*, 2007.

[92] R. A. Eisenberg, S. Weirich and H. G. Ahmed, 'Visible type application', in *European Symposium on Programming*, 2016.

[93] H. Kiriyama, H. Aotani and H. Masuhara, 'A lightweight optimization technique for data types a la carte', in *Companion Proceedings of the 15th International Conference on Modularity*, 2016.

[94] C. Okasaki, 'Red-black trees in a functional setting', *Journal of Functional Programming*, vol. 9, 1999.

[95] B. O'Sullivan, *Criterion: A haskell microbenchmarking library*, 2014. [Online]. Available: `http://www.serpentine.com/criterion/`.

[96] P. Wadler, *The expression problem*, 1998. [Online]. Available: `https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[97] L. E. Day and G. Hutton, 'Compilation à la carte', in *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL)*, 2013.

[98] A. Persson, E. Axelsson and J. Svenningsson, 'Generic monadic constructs for embedded languages', in *International Symposium on Implementation and Application of Functional Languages (IFL)*, 2011.

[99] N. Wu, T. Schrijvers and R. Hinze, 'Effect handlers in scope', 2014.

[100] B. Delaware, B. C. d S Oliveira and T. Schrijvers, 'Meta-theory à la carte', in *ACM SIGPLAN Notices*, vol. 48, 2013.

[101] P. Hudak *et al.*, 'Building domain-specific embedded languages', *ACM Computing Surveys*, vol. 28, 1996.

[102] P. Wadler, 'Monads for functional programming', in *International School on Advanced Functional Programming*, 1995.

[103] J. Launchbury, 'Lazy imperative programming', in *ACM Workshop on State in Programming Languages*, 1993.

[104] E. Axelsson, K. Claessen, G. Dévai *et al.*, 'Feldspar: A domain specific language for digital signal processing algorithms', in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, 2010.

[105] T. Elliott, L. Pike, S. Winwood *et al.*, 'Guilt free ivory', in *ACM SIGPLAN Notices*, vol. 50, 2015.

[106] L. Pike, A. Goodloe, R. Morisset and S. Niller, 'Copilot: A hard real-time runtime monitor', in *International Conference on Runtime Verification*, 2010.

[107] M. Pickering, N. Wu and B. Németh, 'Working with source plugins', in *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, 2019.

[108] A. Gill, *Dotgen: A simple interface for building .dot graph files.* 2008. [Online]. Available: `https://hackage.haskell.org/package/dotgen`.

[109] L. Erkok, *Sbv: SMT based verification: Symbolic Haskell theorem prover using SMT solving.* 2010. [Online]. Available: `https://hackage.haskell.org/package/sbv`.

[110] E. Axelsson, 'Compilation as a typed EDSL-to-EDSL transformation', *arXiv preprint arXiv:1603.08865*, 2016.

[111] A. Ekblad, *Shellmate: Simple interface for shell scripting in Haskell.* 2014. [Online]. Available: `https://hackage.haskell.org/package/shellmate`.

[112] S. Marlow, S. P. Jones *et al.*, *The Glasgow Haskell compiler*, 2004.

[113] S. L. P. Jones and A. M. Santos, 'A transformation-based optimiser for Haskell', *Science of Computer Programming*, vol. 32, 1998.

[114] C. V. Hall, K. Hammond, S. L. Peyton Jones and P. L. Wadler, 'Type classes in Haskell', *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, 1996.

[115] S. P. Jones, M. Jones and E. Meijer, 'Type classes: An exploration of the design space', in *Haskell Workshop*, 1997.

[116] M. Algehed, P. Jansson, S. H. Einarsdóttir and A. Gerdes, 'Saint: An API-generic type-safe interpreter', in *Trends in Functional Programming (TFP)*, 2019.

[117] B. Barras, S. Boutin, C. Cornes *et al.*, 'The Coq proof assistant reference manual: Version 6.1', 1997.

[118] G. Dévai, D. Leskó and M. Tejfel, 'The EDSL's struggle for their sources', in *Central European Functional Programming School*, 2013.

[119] M. Snoyman, *Developing web applications with Haskell and Yesod.* O'Reilly Media, Inc., 2012, ISBN: 978-1449316976.

[120] G. Giorgidze and H. Nilsson, 'Embedding a functional hybrid modelling language in Haskell', in *Symposium on Implementation and Application of Functional Languages (IFL)*, 2008.

[121] G. Mainland and G. Morrisett, 'Nikola: Embedding compiled GPU functions in Haskell', in *ACM SIGPLAN Notices*, vol. 45, 2010.

[122] G. Giorgidze, T. Grust, T. Schreiber and J. Weijers, 'Haskell boards the ferry', in *Symposium on Implementation and Application of Functional Languages (IFL)*, 2010.

[123] G. Mainland, 'Why it's nice to be quoted: Quasiquoting for Haskell', in *Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[124] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones and S. Weirich, 'Closed type families with overlapping equations', *ACM SIGPLAN Notices*, vol. 49, 2014.

[125] A. Russo, ‘Functional pearl: Two can keep a secret, if one of them uses haskell’, in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, 2015.

[126] D. Leijen and E. Meijer, ‘Domain specific embedded compilers’, in *Proceedings of the 2nd Conference on Domain-Specific Languages*, 2000.

[127] J. Cheney and R. Hinze, ‘Phantom types’, Cornell University, Tech. Rep., 2003.

[128] R. Pucella and J. A. Tov, ‘Haskell session types with (almost) no class’, in *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, 2008.

[129] D. Stefan, D. Mazières, J. C. Mitchell and A. Russo, ‘Flexible dynamic information flow control in the presence of exceptions’, *Journal of Functional Programming*, vol. 27, 2016.

[130] J. Bracker and A. Gill, ‘Sunroof: A monadic dsl for generating javascript’, in *Proceedings of the 16th International Symposium on Practical Aspects of Declarative Languages*, 2014.

[131] M. P. Gissurarson, ‘Suggesting valid hole fits for typed-holes (experience report)’, in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, 2018.

[132] D. Vytiniotis, S. Peyton jones, T. Schrijvers and M. Sulzmann, ‘Outsidein(x) modular type inference with local assumptions’, *Journal of Functional Programming*, vol. 21, 2011.

[133] G. Team. ‘The ghc-8.10.1 library Constraint module’. (2020), [Online]. Available: `https://hackage.haskell.org/package/ghc-8.10.1/docs/src/Constraint.html`.

[134] I. S. Diatchki, ‘Improving haskell types with smt’, in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, 2015.

[135] D. Otwani and R. A. Eisenberg, ‘The thoralf plugin: For your fancy type needs’, in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, 2018.

[136] T. Schrijvers, S. Peyton Jones, M. Chakravarty and M. Sulzmann, ‘Type checking with open type functions’, in *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

[137] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis and J. P. Magalhães, ‘Giving haskell a promotion’, in *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2012.

[138] J. Breitner, R. A. Eisenberg, S. Peyton Jones and S. Weirich, ‘Safe zero-cost coercions for haskell’, in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2014.

[139] A. Serrano and J. Hage, ‘Type error customization in ghc: Controlling expression-level type errors by type-level programming’, in *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages (IFL)*, 2017.

[140] J. Peterson, ‘Dynamic typing in haskell’, Technical Report YALEU/DCS/RR-1022, Yale University, Department of Computer Science, Tech. Rep., 1993.

[141] M. Toro, R. Garcia and E. Tanter, ‘Type-driven gradual security with references’, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 40, 2018.

[142] J. Hughes, ‘Erlang/quickcheck’, in *Ninth International Erlang/OTP User Conference, Älvsjö, Sweden. November 2003*, 2003.

[143] M. Papadakis and K. Sagonas, ‘A proper integration of types and function specifications with property-based testing’, in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, 2011.

[144] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou and B. C. Pierce, ‘Quickchick: Property-based testing for coq’, in *The Coq Workshop*, 2014.

[145] J. Chen, J. Patra, M. Pradel *et al.*, ‘A survey of compiler testing’, *ACM Computing Surveys*, 2020.

[146] Á. Perényi and J. Midtgaard, ‘Stack-driven program generation of webassembly’, in *Asian Symposium on Programming Languages and Systems*, 2020.

[147] X. Yang, Y. Chen, E. Eide and J. Regehr, ‘Finding and understanding bugs in c compilers’, in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[148] C. Holler, K. Herzig and A. Zeller, ‘Fuzzing with code fragments’, in *21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012.

[149] W. G. Hatch, P. Darragh and E. Eide, *Xsmith software repository.* `https://www.flux.utah.edu/project/xsmith`, 2020.

[150] C. Boyapati, S. Khurshid and D. Marinov, ‘Korat: Automated testing based on java predicates’, *ACM SIGSOFT Software Engineering Notes*, vol. 27, 2002.

[151] L. Meertens, ‘First steps towards the theory of rose trees’, *CWI, Amsterdam*, 1988.

[152] J. A. Goguen and J. Meseguer, ‘Security policies and security models’, in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[153] A. Sabelfeld and A. Myers, ‘Language-based information-flow security’, *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[154] A. Azevedo de Amorim, N. Collins, A. DeHon *et al.*, ‘A verified information-flow architecture’, *SIGPLAN Notices*, vol. 49, 2014.

[155] I. Rezvov, *wasm: WebAssembly Language Toolkit and Interpreter*, `https://hackage.haskell.org/package/wasm`, 2018.

[156] A. Haas, A. Rossberg, D. L. Schuff *et al.*, ‘Bringing the web up to speed with webassembly’, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[157] W. M. McKeeman, ‘Differential testing for software’, *Digital Technical Journal*, vol. 10, 1998.

[158] A. Chou, J. Yang, B. Chelf, S. Hallem and D. Engler, ‘An empirical study of operating systems errors’, in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[159] A. Mista and A. Russo, *MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Structure-Preserving Mutations (Replication Package)*, version 0.3, 2022. DOI: `10.5281/zenodo.7197927`. [Online]. Available: `https://doi.org/10.5281/zenodo.7197927`.

[160] M. Böhme, V.-T. Pham and A. Roychoudhury, ‘Coverage-based greybox fuzzing as markov chain’, *IEEE Transactions on Software Engineering*, vol. 45, 2017.

[161] S. Gan, C. Zhang, X. Qin *et al.*, ‘Collafl: Path sensitive fuzzing’, in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[162] N. Havrikov and A. Zeller, ‘Systematically covering input structure’, in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[163] H. L. Nguyen and L. Grunske, ‘BeDivFuzz: Integrating behavioral diversity into generator-based fuzzing’, in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[164] R. Padhye, C. Lemieux, K. Sen, M. Papadakis and Y. Le Traon, ‘Semantic fuzzing with zest’, in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

Part II

Appended Papers

Paper I

QuickFuzz testing for fun and profit

Gustavo Grieco, Martín Ceresa, **Agustín Mista** and Pablo Buiras

Journal of Systems and Software (Volume 134), 340-354 (2017)

Abstract

Fuzzing is a popular technique to find flaws in programs using invalid or erroneous inputs but not without its drawbacks. On one hand, mutational fuzzers require a set of valid inputs as a starting point, in which modifications are then introduced. On the other hand, generational fuzzing allows synthesizing somehow valid inputs according to a specification. Unfortunately, this requires to have a deep knowledge of the file formats under test to write specifications of them to guide the test case generation process.

In this paper, we introduce an extended and improved version of QuickFuzz, a tool written in Haskell designed for testing unexpected inputs of common file formats on third-party software, taking advantage of off-the-self well known fuzzers.

Unlike other generational fuzzers, QuickFuzz does not require to write specifications for the files formats in question since it relies on existing file-format-handling libraries available on the Haskell code repository. It supports almost 40 different complex file types including images, documents, source code and digital certificates.

In particular, we found QuickFuzz useful enough to discover many previously unknown vulnerabilities in real-world implementations of web browsers and image processing libraries among others.

1 Introduction

Modern software is able to manipulate complex file formats that encode richly-structured data such as images, audio, video, HTML documents, PDF documents or archive files. These entities are usually represented either as binary files or as text files with a specific structure that must be correctly interpreted by programs and libraries that work with such data. Dealing with the low-level nature of such formats involves complex, error-prone artifacts such as parsers and decoders that must check invariants and handle a significant number of corner cases. At the same time, bugs and vulnerabilities in programs that handle complex file formats often have serious consequences that pave the way for security exploits [40].

How can we test this software? As a complement to the usual testing process, and considering that the space of possible inputs is quite large, we might want to test how these programs handle *unexpected* input.

Fuzzing [1], [12], [41] has emerged as a promising tool for finding bugs in software with complex inputs, and consists in random testing of programs using potentially invalid or erroneous inputs. There are two ways of producing invalid inputs: *mutational* fuzzing involves taking valid inputs and altering them through randomization, producing erroneous or invalid inputs that are fed into the program; and *generational* fuzzing (sometimes also known as grammar-based fuzzing) involves generating invalid inputs from a specification or model of a file format. A program that performs fuzzing to test a target program is known as a *fuzzer*.

While fuzzers are powerful tools with impressive bug-finding ability [3]–[5], they are not without disadvantages. Mutational fuzzers usually rely on an external set of *input* files which they use as a starting point. The fuzzer then takes each file and introduces mutations in them before using them as test cases for the program in question. The user has to collect and maintain this set of input files manually for each file format she might want to test. By contrast, generational fuzzers avoid this problem, but the user must then develop and maintain models of the file format types she wants to generate. As expected, creating such models requires deep domain knowledge of the desired file format and can be very expensive to formulate.

In this paper, we introduce QuickFuzz, a tool that leverages Haskell's QuickCheck [42], the well-known property-based random testing library and Hackage [43], the community Haskell software repository in conjunction with off-the-shelf mutational fuzzers to provide automatic fuzz-ing for several common file formats, without the need of an external set of input files and without having to develop models for the file types involved. QuickFuzz generates invalid inputs using a mix of generational and mutational fuzzing to try to discover unexpected behavior in a target application.

Hackage already contains Haskell libraries that handle well-known image, document, archive and media formats. We selected libraries that have two important features: (a) they provide a *data type* `T` that serves as a lightweight specification and can be used to represent individual files of these formats, and (b) they provide a function to *serialize* elements of type `T` to write into files. In general, we call this function `encode` that takes a value of type `T` and returns a `ByteString`. Using ready-made Hackage libraries as models save programmers from having to write these by hand.

The key insight behind QuickFuzz is that we can make random values of type `T` using QuickCheck's *generators*, the specialized machinery for type-driven random values generation. Then we serialize the test cases and pass them to an off-the-shelf fuzzer to randomize. Such a mutation is likely to produce a corrupted version of the file. Then, the target application is executed with the corrupted file as input.

The missing piece of the puzzle is a mechanism to automatically derive the QuickCheck generators from the definitions of the data types in the libraries, which we call *MegaDeTH*.

Finally, if an abnormal termination is detected (for instance, a segmentation fault), the tool will report the input producing the crash.

Thanks to Haskell implementations of file-format-handling libraries found on Hackage, QuickFuzz currently generates and mutates a large set of different file types out of the box. However, it is also possible for the user to add file types by providing a data type `T` and the suitable serializing functions. Our framework can derive random generators fully automatically, to be used by QuickFuzz to discover bugs in new applications.

Although QuickFuzz is written in Haskell, we remark that it treats its target program as a black box, giving it randomly generated, invalid files as arguments. Therefore, **QuickFuzz can be used to test programs written in any language**.
Our contributions can be summarized as follows:

- We present QuickFuzz, a tool for automatically generating inputs and fuzzing programs parsing several common types of files. QuickFuzz uses QuickCheck behind the scenes to generate test cases, and is integrated with fuzzers like *Radamsa*, *Honggfuzz* and other bug-finding tools such as *Valgrind* and *Address Sanitizer*.

- We release QuickFuzz[1]as **open-source** and **free of charge**. As far as we know, QuickFuzz is the first fuzzer to offer the generation and mutation of almost forty complex file types without requiring the user to develop the models: just install, select a target program and wait for crashes!

- We introduce *MegaDeTH*, a library to derive random generators for Haskell data types. *MegaDeTH* is fully automatic and capable of handling mutually recursive types and deriving instances from external modules. This library can be used to extend QuickFuzz with new data types. Additionally, we describe the strategy adopted to improve the automated derivation of random generators by using not only the information found on a data type definition, but the one on its abstract interface as well. Moreover, we detail and exemplify the technique used to enforce some

[1]The tool is available at `https://github.com/CIFASIS/QuickFuzz`.

semantic properties in the generation of source code. This is implemented in our tool for widely used programming languages like JavaScript, Python and Lua among others.

- We evaluate the practical feasibility of QuickFuzz and show an extensive list of security-related bugs discovered using QuickFuzz in complex real-world applications like browsers, image-processing utilities and file archivers among others.

This paper is a revised and extended version of [10] which appeared in the Haskell Symposium 2016. This new version brings many theoretical and experimental contributions.

First, we extended our tool with the improved random generators using the information obtained from the abstract interface available for every library used.

Second, in the case of source code generation, we presented a technique to enforce semantic properties immediately after the generation. We implemented this approach using meta-programming, in order to improve the random code generation of some widely used programming languages.

Third, we added three sets of experiments to explore how our tool generates and mutates files. The related work section and the experiments comparing to other fuzzers were also expanded to cover the latest developments in the field.

Finally, QuickFuzz now supports a greater number of file formats, including complex file formats found in public key infrastructure such as ASN.1, X509 and CRT certificates. Using all the proposed extensions, we have found more security-related bugs, updating our results and conclusion sections accordingly.

The rest of the paper is organized as follows. Section 2 introduces fuzzing and the functional programming concepts useful to perform value generation. Section 3 provides an overview of how QuickFuzz works using an example. Section 4 discusses how to automatically derive random generators using *MegaDeTH*. In Section 5 we highlight some of the key principles in the design and implementation of our tool using the QuickCheck framework. Later, in Section 6, we perform an evaluation of its applicability. Section 7 presents related work and Section 8 concludes.

2 Background

2.1 Fuzzers

Fuzzers are popular tools to test how a program handles *unexpected* input. There are two approaches for fuzzing [15]: *mutational* and *generational.*

Mutational fuzzers These tools produce inputs for testing programs taking valid inputs and altering them through randomization, producing erroneous or invalid inputs that are fed into the program. Typically they work by producing a random mutation at the bit or byte level.

Nowadays, there are plenty of robust and fast mutational fuzzers. For instance, zzuf [7] is a fuzzer developed by Caca Labs that produces mutations in the program input automatically hooking the functions to read from files or network interfaces before a program is started. When the program reads an input, zzuf randomly flips a small percentage of bits, corrupting the data. Another popular mutational fuzzer is radamsa [5]. It was developed by the Oulu university secure programming group and works at the byte level randomly adding, removing or changing complete sequences of bytes of the program input. It features a large amount of useful mutations to detect bugs and vulnerabilities.

Both radamsa and zzuf are dumb mutation fuzzers since they do not use any feedback provided by the actual execution of the program to test. In the last few years, feedback-driven mutational fuzzers such as american fuzzy lop [3] and honggfuzz [4] were developed. These fuzzers use lightweight program instrumentation to collect information of every execution and use it to guide the fuzzing procedure.

While mutational fuzzers are one of the simpler and more popular types of fuzzers to test programs, they still require a good initial corpus to mutate in order to be effective.

Generational fuzzers These tools produce inputs for testing programs generating invalid or unexpected inputs from a specification or model of a file format.

This type of fuzzers are also popular in testing. For instance, one of the most mature and commercially supported generational fuzzers is Peach [6]. It was originally written in Python in 2007, and later re-written in C# for the latest release. It provides a wide set of features for generation and mutation, as well as monitoring remote processes. However, in order to start fuzzing, it requires the specification of two main components to generate and mutate program inputs:

- Data Models: a formal description of how data is composed in order to be able to generate fuzzed data.
- Target: a formal description of how data can be mutated and how to detect unexpected behavior in monitored software.

As expected, the main issue with Peach is that the user has to write these configuration files, which requires very specific domain knowledge. Another option is Sulley [44], a fuzzing engine and framework in Python. It is frequently presented as a simpler alternative to Peach since the model specification can be written using Python code. A more recent alternative open-sourced by Mozilla in 2015 is Dharma [8], a generation-based, context-free grammar fuzzer also in Python. It also requires the specification of the data to generate, but it uses a context-free grammar in a simple plain text format.

In recent years, tools like AUTOGRAM [45] and GLADE [46] helped to learn and synthesize inputs grammars to test programs. These tools start from valid input files and using the analyzed program itself, they approximate the input grammar. AUTOGRAM uses dynamic taint analysis to synthesize the input grammar while GLADE executes the program as an oracle to answer membership queries (i.e., whether a given input is valid). Later such grammars can be used as models in generational fuzzers [46].

2.2 Haskell

Haskell is a general-purpose purely-functional programming language [47]. It provides a powerful type system with highly-expressive user-defined algebraic data types. With the power to precisely constrain the values allowed in a program, types in Haskell can serve as adequate lightweight specifications.

Data Types Data types in Haskell are defined using one or more *constructors*. A constructor is a tag that represents a way of creating a data structure and it can have zero or more arguments of any other type.

For instance, we can define the `List a` data type representing lists of values of type `a` by using two constructors: `Nil` represents the empty list, while `Cons` represents a non-empty list formed by combining a value of type `a` and a list (possibly empty) as a tail. Note that this is a recursive type definition:

```haskell
data List a = Nil | Cons a (List a)
```

As an example, we define a few functions that we are going to use in the rest of this work:

```haskell
length :: List a → Int
length Nil = 0
length (Cons x xs) = 1 + length xs
snoc :: a → List a → List a
snoc x Nil = Cons x Nil
snoc x (Cons y ys) = Cons y (snoc x ys)
reverse :: List a → List a
reverse Nil = Nil
reverse (Cons x xs) = snoc x (reverse xs)
```

The function `length` computes the length of a given list, `snoc` adds an element at the end of the list and finally `reverse` reverses the entire list. Their definitions are straightforward applications of pattern-matching and recursion. Free type variables in types, such as `a` above, are implicitly universally quantified.

Type Classes Haskell provides a powerful overloading system based on the notion of a *type class*. Broadly speaking, a type class is a set of types with a common abstract interface. The functions defined in the interface are said to be overloaded since they can be used on values of any member of the type class. In practice, membership in a type class is defined by means of an *instance*, i.e. a concrete definition of the functions in the interface specialized to the chosen type. For example, Haskell includes a built-in type class called `Eq` which defines the equality relation (==) for a given type. Assuming that `a` is in the `Eq` type class, we can define an instance of `Eq` for `List a`:

```
instance Eq a ⇒ Eq (List a) where
  Nil         == Nil         = True
  (Cons x xs) == (Cons y ys) = (x == y) ∧ (xs == ys)
  _           == _           = False
```

Note that the (==) operator is used on two different types: in the expression `x == y` it uses the definition given in the instance for `Eq a` (equality on `a`), while in the expression `xs == ys` it is a recursive call to the (==) operator being defined (equality on `List a`). Haskell uses the type system to dispatch and resolve this overloading.

Applicative Functors In this work, we use a well-known abstraction for structuring side-effects in Haskell, namely *applicative functors* [48]. Haskell being a pure language means that all function results are fully and uniquely determined by the function's arguments, in principle leaving no room for effects such as random-number generation or exceptions, among others. However, such effects can be encoded in a pure language by enriching the output types of functions, e.g. pseudo-random numbers could be achieved by explicitly threading a seed over the whole program. Applicative functors are one of the ways in which we can hide this necessary boilerplate to implement effects.

Applicative functors in GHC are implemented as a type class. In order to define an applicative functor one has to provide definitions of two functions, `pure` and (⟨∗⟩), with the types given below.

```
class Applicative p where
  pure :: a → p a
  (⟨∗⟩) :: p (a → b) → p a → p b
```

The function `pure` inserts pure values into the applicative structure (the boilerplate), and (⟨∗⟩) gives us a way to "apply" a function inside the structure to an argument. Due to overloading, computations written using this interface can be used with any applicative effect.

For example, assume that we have a function (+) :: `Int → Int → Int` that adds two numbers, and that we have a type `RNG` with an instance `Applicative RNG` that represents random-number generation, and moreover, that there is a value `gen :: RNG Int` that produces a random `Int`. We can express a computation that adds two random numbers using the applicative interface as follows: `pure (+) ⟨∗⟩ gen ⟨∗⟩ gen`. This expression has type `RNG Int` (which can be read as "an `Int` produced potentially from random data"), and it can be further used in other applicative computations as needed.

Hackage This work draws on packages found in *Hackage*. Hackage is the Haskell community's central package archive. As we will explain, we take from this archive the data types used to generate different file formats. For instance, the JuicyPixels library is available in Hackage [49], and it has support for reading and writing different image formats.

Hackage is a fundamental part of QuickFuzz, since it provides lightweight specifications for free and we carefully designed QuickFuzz to easily include new formats as they appear in this code repository.

2.3 QuickCheck

QuickCheck is a tool that aids the programmer in formulating and testing properties of programs, first introduced as a Haskell library by Koen Claessen and John Hughes [42]. QuickCheck presents mechanisms to generate random values of a given type, as well as a simple language to build new generators and specify properties in a modular fashion. Once the generators have been defined, the properties are tested by generating a large amount of random values.

Properties To use this tool, a programmer should define suitable properties that the code under test must satisfy. QuickCheck defines a property basically as a predicate, i.e. a function that returns a boolean value. For instance, we can check if the size of a list is preserved when we reverse it:

```
prop_reverseSize :: List a → Bool
prop_reverseSize xs = length xs == length (reverse xs)
```

QuickCheck will try to falsify the property by generating random values of type `List a` until a counterexample is found.

Generators QuickCheck requires the programmer to implement a generator for `List a` in order to test properties involving such data type, like `prop_reverseSize` above. The tool defines an applicative functor `Gen` and a new type class called `Arbitrary` for the data types whose values can be generated. Its abstract interface consists solely of a function that returns a generator for the data type `a` being instantiated. The applicative functor `Gen` provides the required mechanisms to generate random values. As seen in the previous subsection, effectful behavior requires an applicative structure:

```
class Arbitrary a where
  arbitrary :: Gen a
```

Then it is up to the programmer to define a proper instance of `Arbitrary` for `List a` using the tools provided by `QuickCheck`:

```
instance Arbitrary a ⇒ Arbitrary (List a) where
  arbitrary = genList
    where
      genList = oneof [genNil, genCons]
      genNil = pure Nil
      genCons = pure Cons
                  ⟨∗⟩(arbitrary :: Gen a)
                  ⟨∗⟩(arbitrary :: Gen (List a))
```

The function `oneof` chooses with the same probability between a `Nil` value generator or a `Cons` value generator. Note that `genCons` calls to `arbitrary` recursively in order to get a generated `List a` for its inner list parameter.

However, the previous implementation has a problem; it is possible for `oneof` to always choose a `genCons`, getting the computation in an endless loop. To solve this, QuickCheck provides tools to limit the maximum value generation size. An improved implementation uses the size-dependent functions `sized` and `resize`, which take care of the maximum generation size, decreasing it after every recursive step. When the size reaches zero, the generation always returns `Nil`, ensuring that the value construction process never gets stuck in an infinite loop. The generation size is controlled externally and is represented in this case by the `n` parameter.

```
instance Arbitrary a ⇒ Arbitrary (List a) where
  arbitrary = sized genList
    where
      genList n = oneof [genNil, genCons n]
      genNil = pure Nil
      genCons 0 = genNil
      genCons n = pure Cons
                    ⟨∗⟩(resize (n − 1) arbitrary :: Gen a)
                    ⟨∗⟩(resize (n − 1) arbitrary :: Gen (List a))
```

Using this instance, QuickCheck can properly generate *arbitrary* values of `List a` and test properties using them:

```
quickCheck prop_reverseSize
```

And if the test passed for all the randomly generated values, QuickCheck will answer:

```
++++ OK, passed 100 tests
```

3 A Quick Tour of QuickFuzz

In this section, we will show QuickFuzz in action with a simple example. More specifically, how to discover bugs in *giffix*, a small command line utility from *giflib* [50] that attempts to fix broken Gif images. Our tool has built-in support for the generation of Gif files using the JuicyPixels library [49].

In order to find test cases to trigger bugs in a target program, our tool only requires from the user:

- A file format name to generate fuzzed inputs
- A command line to run the target program

It is worth mentioning that no instrumentation is required to run the target program. For instance, to launch a fuzzing campaign on *giffix*, we simply execute:

```
$ QuickFuzz Gif 'giffix @@' -a radamsa -s 10
```

Our tool replaces `@@` by a random filename that will represent the fuzzed Gif file before executing the corresponding command line. The next parameter specifies the mutational fuzzer it uses (`radamsa` in this example) and the last one is the abstract maximum size in the Gif value generation. Such limitation will effectively bound the memory and the CPU time used during the file generation.

After a few seconds, QuickFuzz stops since it finds an execution that fails with a segmentation fault. At this point, we can examine the output directory (*outdir* by default) to see the Gif file produced by our tool that caused *giffix* to fail.

Figure 1 shows the QuickFuzz pipeline and architecture. An execution of QuickFuzz consists of three phases: high-level fuzzing, low-level fuzzing and execution. The diagram also shows the interaction between the compile-time and the run-time of QuickFuzz. Let us take a look at what happens in each phase in the *giffix* example.

3.1 High-Level Fuzzing

During this phase, QuickFuzz generates values of the data type `T` that represents the file format of the input to the target program. It relies on the tools provided by QuickCheck. More specifically, the random number generation tools that can be used to construct randomized structured data in a compositional manner. In our example this representation type `T` (borrowed from JuicyPixels) is called `GifFile`.

```
data GifFile = GifFile Header Images Looping
data Looping
   = LoopingNever
   | LoopingForever
   | LoopingRepeat Int
```

A `GifFile` contains a header (of type `Header`), the raw bitmap images (of type `Images`), and a looping behavior (of type `Looping`), specified by three *type*

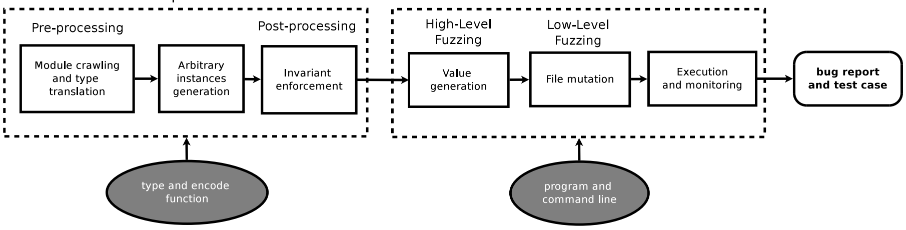


Figure 1: Summary of the random generators deriving using *MegaDeTH* at compile-time and the test case generation using QuickFuzz at run-time where gray nodes represent inputs provided by a user and bold nodes represent outputs.

constructors denoting the possible behaviors. We left `Header` and `Images` data types unspecified for the sake of the example. Note that randomly generated elements of type `GifFile` might not be valid Gif files, since the type system is unable to encode all invariants that should hold among the parts of the value. For example, the header might specify a width and height that doesn't match the bitmap data. For this reason, we consider that this step corresponds to generational fuzzing, where the data type definition serves as a lightweight approximate model of the Gif file format which generates potentially invalid instances of it.

After generating a value of type `GifFile` with QuickCheck, we use the `encode` function for this file type to serialize the `GifFile` into a sequence of bytes, which is written into the output directory for further inspection by the user. Finally, the result of this phase is a Gif image, most likely corrupted.

3.2 Low-Level Fuzzing

Usually the use of high-level fuzzing produced by the values generated by QuickCheck is not enough to trigger some interesting bugs. Therefore, this phase relies on an off-the-shelf mutation fuzzer to introduce errors and mutations at the bit level on the `ByteString` produced by the previous step. In particular, the current version supports the following fuzzers:

- Zzuf: a transparent application input fuzzer by Caca Labs [7].
- Radamsa: a general-purpose fuzzer developed by the Oulu University Secure Programming Group [5].
- Honggfuzz: a general purpose fuzzer developed by Google [4].

One of the key principles of the design of QuickFuzz was to require no parameter tuning in the use of third-party fuzzers and bug-detection tools. Usually, the use of mutational fuzzers requires fine-tuning some critical parameters. Instead, we decided to incorporate default values to perform an effective fuzzing campaign even without fine-tuning values like mutation rates.

After this phase, the result will be a very corrupted Gif file thanks to the combination of high-level and low-level fuzzing.

3.3 Execution

The final phase involves running the target program with the mutated file as input and check if it produces an abnormal termination. For each test case file producing a runtime failure, we can also find in the output directory the intermediate values for each step of the process:

- A text file with the printed value generated by QuickCheck
- The test case file before the mutation by the mutational fuzzer
- The actual mutated test case file which was passed as input to the target program and resulted in failure

Using this information, developers can examine how the test case file was corrupted in order to understand why their program failed and how it can be fixed.

After corrupting a few Gif files, QuickFuzz finds a test case to reproduce a heap-based overflow in *giffix* (CVE-2015-7555). This issue is caused by the lack of validation of the size of the logical screen and the size of the actual Gif frames. In fact, if we run the tool for no more than 5 minutes in a single core, we will obtain dozens of test cases triggering failed executions (crashes and aborts). Crash de-duplication is currently outside the scope of our tool, so we manually checked the backtraces using a debugger and determined that *giffix* was failing in 3 distinctive ways.

The root cause of such crashes can be the same, for instance, if the program is performing a read out-of-bounds. Nevertheless, QuickFuzz can still obtain valuable information by finding different crashes associated with the same issue: they can be very useful to determine if the original issue is exploitable or not.

Additionally, QuickFuzz can use Valgrind [51] and Address Sanitizer [52] to detect more subtle bugs like a read out-of-bounds that would not cause a segmentation fault or the use of uninitialized memory.

4 Automatically Deriving Random Generators

In this section we explain the compilation-time stage of QuickFuzz, that can be separated into three methodologies depending on *how* the file format was implemented, and *which* file format is in order to enforce information not coded in the library:

- Automatically deriving `Arbitrary` instances for target file formats data types. Explained in subsection 4.1.
- Crawling libraries interfaces related to the generation of the target file formats, and then, generating a higher level structure that represents manipulations of values using those interfaces. Explained in subsection 4.2.
- Post-processing the arbitrarily generated values to enforce specific semantic properties. In particular, we use such a technique to improve source code generation. Explained in subsection 4.3.

The last two stages are not required for every file format generation and fuzzing, however, they improve the variety of generated values as discussed on their respective subsections.

4.1 *MegaDeTH*

Mega Derivation TH (*MegaDeTH*) is a tool that gives the user the ability to provide class instances for a given type, taking care to provide suitable class instances automatically. As an example, we will analyze the `GifFile` data type:

```
data GifFile = GifFile Header Images Looping

data Looping
  = LoopingNever
  | LoopingForever
  | LoopingRepeat Int
```

In order to define an Arbitrary instance for GifFile, the programmer has to define such instances for Header, Images and Looping as well. We will refer to GifFile as our *target* data type, since it is the top-level data type we are looking to generate. Also, we will refer to Header, Images and Looping as the *nested* data types of GifFile. If any of these data types define further nested data types, this process has to be repeated until every data type involved in the construction of GifFile is a member of the Arbitrary type class.

Since Haskell benefits the practice of defining custom data type in an algebraic way, a data type definition can be seen as a hierarchical structure. Hence, deriving Arbitrary instances for every data type present at the hierarchy can be a repetitive task. *MegaDeTH* offers a solution to this problem: it gives the user a way to *thoroughly* derive instances for all the intermediate data types that are needed to make the desired data type instance work.

MegaDeTH was implemented using Template Haskell [30], a meta-programming mechanism built into GHC that is extremely useful to process the syntax tree of Haskell programs and to insert new declarations at compilation time. We use the power of Template Haskell to extract all the nested types for a given type and derive a class instance for each one of them, finally instantiating the top-level data type. Since Haskell gives the user the possibility of writing mutually recursive types, *MegaDeTH* implements a *topological sort* to find a suitable order in which to instantiate each data type satisfying their type dependencies.

We can simply derive all the required instances using *MegaDeTH*'s function devArbitrary that automatically generates the following instances (among others), simplified for the sake of understanding:

```
instance Arbitrary Looping where
  arbitrary = sized gen
    where
      gen n = oneof
        [pure LoopingNever
        ,pure LoopingForever
        ,pure LoopingRepeat
          ⟨∗⟩(resize (n − 1) arbitrary :: Gen Int)
        ]
instance Arbitrary GifFile where
  arbitrary = sized gen
    where
      gen n = pure GifFile
                ⟨∗⟩(resize (n − 1) arbitrary :: Gen Header)
                ⟨∗⟩(resize (n − 1) arbitrary :: Gen Images)
                ⟨∗⟩(resize (n − 1) arbitrary :: Gen Looping)
```

As we can see, the derived code reduces the size whenever a type constructor is used and select which one is to be used with QuickCheck's `oneof` function. These automatically generated random generators follow directly the ideas presented in Section 4, that is to choose between all the available constructors and generate the required arguments of it.

However, it is not always the case that we can choose between available constructors in order to generate rich structured values. We explore the limitations of this approach in further detail. The next example introduces a different manner to define a data type which exploits the limitations of *MegaDeTH*, and serves as an introduction to the solution.

Designing an Html manipulating library One of the main decisions involved when designing a domain-specific language [53] (DSL) manipulation library is the level of *embedding* this DSL will have. The most common approaches are *deep embedding* and *shallow embedding* [54]. Deeply-embedded DSLs usually define an internal intermediate representation of the terms this language can state, along with functions to transform this intermediate representation forth and/or back to the target representation. In this kind of embedding, the domain-specific invariants are mainly preserved by the internal representation. The previously presented `GifFile` data type is an example of this technique. On the other hand, shallow-embedded DSLs often use a simpler internal representation, leading the task of preserving the domain-specific invariants to the functions at the library abstract interface.

Since HTML is a markup language, it is essentially comprised of plain text. Hence, instead of defining a complex data type using a different type constructor for each HTML tag, the library designer could be tempted to use a shallow embedding representation, employing the same plain-text representation for the library's internal implementation:

```
module Html where
type Html = String
head :: Html → Html
body :: Html → Html
div :: Html → Html
hruler :: Html
(⟨+⟩) :: Html → Html → Html
toHtml :: String → Html
renderHtml :: Html → ByteString
```

In the definition above, the `Html` data type is a synonym to the `String` data type. Thus, the functions on its abstract interface are basically `String` manipulating functions with the implicit assumption that if they take a correct HTML, they will return a correct HTML, for instance:

```
head :: Html → Html
head hd = "<head>" ++ hd ++ "</head>"
```

```
hruler :: Html
hruler = "</hr>"

(⟨+⟩) :: Html → Html → Html
h1 ⟨+⟩ h2 = h1 ++ h2
```

Given that our guide in the derivation of random generators is the data type, *MegaDeTH* needs it to be structurally complex in order to generate complex data, remember that we based our generators on the assumption that we can choose with the same probability between different constructors in order to generate random values. If we derive a random generator for the given `Html` data type, its type definition does not provide enough structure to generate useful random values. Instead, the generated `Arbitrary` instance delegates this task to such an instance of the `String` data type:

```
instance Arbitrary Html where
  arbitrary = (arbitrary :: Gen String)
```

The resulting `Html` values generated by this `Arbitrary` instance are just random strings, which rarely represent a valid `Html` value. Therefore, these kinds of generators are useless for our purpose of discovering bugs on complex software parsing markup languages such as HTML.

This approach to defining libraries is common to find in the wild, being *blaze-html* [55] or *language-css* [56] some examples of this. Instead of discarding them, the next subsection introduces a different approach we took to derive powerful `Arbitrary` instances for these kinds of libraries.

4.2 Encoding functions information into actions

Haskell's expressive power allows the library programmer to define a file format representation as a custom data type in several ways. As we have seen previously, *MegaDeTH* derive useful `Arbitrary` instances when the programmer had encoded invariants directly in the data type. On the other hand, as we have seen in the previous subsection, those invariants can be forced in the operations declared in the data type abstract interface. These operations manipulate the values of the data type, transforming well-formed values into well-formed results.

Since we need data type constructors to be able to use *MegaDeTH*, we use the concept of *Actions* [57]. Given a type `T` we can look up all the functions that return a `T` value and think of them as a way to create a new `T` value and call these functions actions. Henceforth, we can define a new data type where each function that creates a `T` value defines a constructor in this new type. In general, for a given data type we will refer to its actions-oriented data type by simply as *its actions data type.*

In order to illustrate this technique, we will reuse the Html manipulating library example defined in the previous subsection:

```
module Html where

type Html = String
```

To build a complex Html document, the programmer should use the functions defined in the abstract interface of this module. For example, a simple Html document could be represented as follows:

```
myPage :: Html
myPage =
  head (toHtml "my head")
  ⟨+⟩ body (div (toHtml "text")
          ⟨+⟩ hruler
          ⟨+⟩ div (toHtml "more text"))
```

The Html actions data type can be automatically generated, where each constructor represents a possible action over the original data type, whose type parameters corresponds to the ones at the original function this action intends to express. Note that, if an action has a parameter that comprises the original data type, it is replaced by its actions-oriented one, making this a recursively defined data type.

```
data HtmlAction
  = Action_head   HtmlAction
  | Action_body   HtmlAction
  | Action_div    HtmlAction
  | Action_hruler
  | Action_toHtml String
  | Action_+      HtmlAction HtmlAction
```

Note that renderHtml will play the role of the encoding function in our representation, since it gives us a way to serialize Html values. Also, is worth mentioning that it is not included as an action since it does not return an Html value.

The previous value could be encoded using actions as follows:

```
myPageActions :: HtmlAction
myPageActions =
  (Action_head (Action_toHtml "my head"))
   `Action_+`
     (Action_body
       ((Action_div (Action_toHtml "text")
         `Action_+`
           Action_hruler)
             `Action_+`
               Action_div (Action_toHtml "more text")))
```

Once an actions data type is derived for a given data type, a value of its type describes a particular composition of functions that returns a value of the original data type. Hence, we need a function performHtml that *performs* an action using the underlying implementation of the interface functions, returning corresponding values of the original type.

```
performHtml :: HtmlAction → Html
performHtml (Action_head v) = head (performHtml v)
performHtml (Action_body v) = body (performHtml v)
performHtml (Action_div v) = div (performHtml v)
performHtml Action_hruler = hruler
performHtml (Action_toHtml v) = toHtml v
performHtml (Action_+  v1 v2) =
  (performHtml v1) ⟨+⟩(performHtml v2)
```

Writing the action data type for common target data types is usually an straightforward task. A similar approach was taken in [18] in order to manually derive random generators for a particular data type of interest. However, this task also becomes repetitive, especially when the target data type contains several functions on its abstract interface. That is the reason why we automate this process by using Template Haskell. The function `devActions` is responsible for this, generating at compile time the actions data type and the performing function for a target data type. This process can be described as follows:

Step 1. Crawl the modules where the target data type is present, extracting all type constructors and function declarations.

Step 2. Find any declarations that return a value of the target data type. Each one will become a type constructor at the actions data type.

Step 3. Generate the actions data type and the performing function for the target data type by using the previously obtained actions.

Once the actions data type and performing function have been generated for a given target data type, it is possible to use *MegaDeTH* to obtain an `Arbitrary` instance for the actions data type, and then, we can obtain such instance for the target data type by simply performing an arbitrary value of the first one:

```
instance Arbitrary Html where
  arbitrary = pure performHtml
                ⟨∗⟩(arbitrary :: Gen HtmlAction)
```

We found this actions-oriented approach to be a convenient way to deal with Haskell libraries with no restrictive type definitions, wrapping their interfaces with a higher level structure and deriving suitable `Arbitrary` instances for them. Given that, it is possible to define useful `Arbitrary` instances for a variety of target data types based on the abstractions defined by the library writer, regardless of *how* the library was implemented.

There are limitations related to the generation of the actions data type. One of them involves definitions using complex types wrapping the target data type. For instance, suppose we extend the `Html` module by adding a function for splitting Html values:

```
split :: Html → (Html, Html)
```

The result type for split does not match the target data type. However, we would like to translate it into an action as well, since the target data type (Html) is somehow wrapped by its result type ((Html, Html)). In order to translate split into an action, we need to know beforehand how to extract the target data type values from the wrapped value.

Another limitation is related to the special treatment required by polymorphic function definitions. Remember the definition of the polymorphic data type List a which represents a list of elements of type a, where a could be any data type:

```
data List a = Nil | Cons a (List a)
```

We can define the following polymorphic functions for all a.

```
append :: a → List a → List a
concat :: List a → List a → List a
```

Our current approach can only handle non-polymorphic functions. We use a naive workaround to solve this consisting of instantiating every polymorphic function at the abstract interface of a module into non-polymorphic ones. This instantiation process is driven by the user, who decides which data types are interesting enough to be replaced. For instance, if the user decides to instantiate the previous list-handling functions with Int and String data types, our tool generates the following functions:

```
append_1 :: Int → List Int → List Int
append_2 :: String → List String → List String
concat_1 :: List Int → List Int → List Int
concat_2 :: List String → List String → List String
```

Then, these instantiated functions are treated like any other non-polymorphic ones at the stage of deciding which ones will be used as actions.

4.3 Enforcing Variable Coherence

Using the previously explained machinery, our tool can randomly generate source code from various programming languages such as Python, JavaScript, Lua and Bash. The generation process relies on the type representing the abstract syntax tree (AST) of the code of each language.

Unfortunately, we found that automatically derived generators for such languages are not always effective at the generation of complex test cases, since they cannot account for all the invariants required for source code files to be semantically correct. In particular, one of the things that random code cannot account for is variable coherence, i.e., when we use a variable, it has to be defined (or declared).

We can see in the example below that QuickFuzz generates a complete program with variables and assignments but without any sense or coherence between them. For example, the following program is rejected by any compiler within one of the first passes.

```
rpa = kk
meg = −18.3 == p
ize = le
```

In order to tackle this issue, we developed a generic technique to enforce properties in the resulting generated values (in this case, Python code). In particular, our goal is to correct generated source code as a first step to use QuickFuzz to test compilers and interpreters in deep stages of the parsing and executing process.

While there are some tools to test compilers, for instance, CSmith [58] for stressing C compilers, they are specific tools developed for certain languages. Our approach is different, since we aim to develop a general technique that works in different complex languages provided some general guidance.

In this work, we decided to enforce variable coherence by making some corrections in the freshly generated test case. QuickFuzz goes through its AST collecting declared variables in a pool of variables identifications and changing unknown variables for previously declared ones arbitrarily taken from that pool. The special case when the pool is empty and a variable is required is sorted by generating an arbitrary constant expression.

As a result, we get programs where every variable used is already defined before it is used.

```
rpa = 4
meg = −18.3 == rpa
ize = meg
```

As we have seen in this section, it is possible to enforce user knowledge not encoded in either the type nor the library of a desired source code. It is also worth noting that this approach is as general as it can be. Therefore, we can implement complex invariants based on how we want to post-process the AST with all the information this structure provides.

5 Detecting Unexpected Termination of Programs

This section details how we defined suitable properties in QuickCheck to perform the different phases of the fuzzing process and detect unexpected termination of programs.

Detecting Unexpected Termination in Programs In Haskell, a program execution using certain arguments can be summarized using this type:

```
type Cmd = (FilePath, [String])
```

First, we defined the notion of a *failed execution*. In our tool a program execution *fails* if we detect an abnormal termination. According to the POSIX.1-1990 standard, a program can be abnormally terminated after receiving the following signals:

- A *SIGILL* when it tries to execute an illegal instruction
- A *SIGABRT* when it called `abort`
- A *SIGFPE* when it raised a floating point exception
- A *SIGSEGV* when it accessed an invalid memory reference
- A *SIGKILL* at any time (usually when the operating system detects it is consuming too many resources)

After a process finishes, it is possible to detect signals associated with failed executions by examining its exit status code. Traditionally in GNU/Linux systems a process that exits with a zero exit status has succeeded, while a non-zero exit status indicates failure. When a process terminates with a signal number n, a shell sets the exit status to a value greater than 128. Most of the shells use $128 + n$. We capture such a condition in the Haskell function `has_failed`, in order to catch when a program finished abnormally:

```
has_failed :: ExitCode → Bool
has_failed (ExitFailure n) = (n<0 ∨ n > 128) ∧ n ≢ 143
has_failed ExitSuccess = False
```

We only excluded *SIGTERM* (with exit status of 143) since we want to be able to use a timeout in order to catch long executions without considering them *failed*.

High-Level Fuzzing Properties In order to use QuickCheck to uncover failed executions in programs, we need to define a property to check. Given an executable program and some arguments, QuickFuzz tries to verify that there is no failed execution as we defined above for arbitrary inputs. We call this property `prop_NoFail`. It serializes inputs to files and executes a given program. Its definition is very straightforward:

```
prop_NoFail :: Cmd → (a → ByteString) → FilePath → a → Property
prop_NoFail pcmd encode filename x = do
  run (write filename (encode x))
  ret ← run (execute pcmd)
  assert (¬ (has_failed ret))
```

After that, we can *QuickCheck* the property of no-failed executions instantiating `prop_NoFail` with suitable values. For instance, let us assume we want to test the conversion from Gif to Png images using ImageMagick. The usual command to achieve this would be:

```
$ convert src.gif dest.png
```

In terms of `prop_NoFail`, to test the command above we call the *QuickCheck* function using the following property:

```
let cmd = ("convert", ["src.gif", "dest.png"])
in prop_NoFail cmd encodeGif "src.gif"
```

where encodeGif is a function to serialize GifFiles. Finally, QuickCheck will take care of the GifFile generation, reporting any value that produces a failed assert in prop_NoFail.

Low-Level Fuzzing Properties In the next phase of the fuzzing process, we enhance the value generation of QuickCheck with the systematic file corruption produced by off-the-shelf fuzzers. Intuitively, we augment prop_NoFail with a low-level fuzzing procedure abstracted as a call to the fuzz function.

```
fuzz :: Cmd → FilePath → IO ()
```

After calling fuzz, the content of a file will be changed somehow. Using this new function, we define a new property called prop_NoFailFuzzed which mutates the serialized file before the execution takes place:

```
prop_NoFailFuzzed :: Cmd → Cmd → (a → ByteString)
                  → FilePath → a → Property
prop_NoFailFuzzed pcmd fcmd encode filename x = do
  run (write filename (encode x))
  run (fuzz fcmd filename)
  ret ← run (execute pcmd)
  assert (¬ (has_failed ret))
```

Finally, is up to QuickCheck to find a counterexample of prop_NoFailFuzzed. This counter-example is a witness which causes the target program to fail execution.

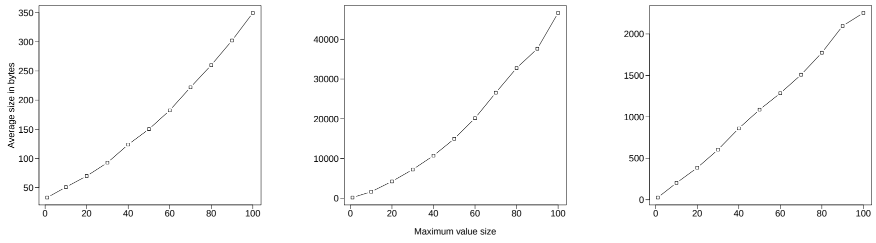
As a result of this process, we can test any compiled program, written in any language, with a plethora of low-level fuzzers with prop_NoFailFuzzed.

6 Evaluation

In this section we will describe different experiments to understand how QuickFuzz is generating and mutating input files. From the extensive list of file formats supported by QuickFuzz, shown in Figure 6a, we have selected five of them to perform our experiments: Zip, Png, Jpeg, Xml and Svg. We have selected these because they are binary and human-readable markup formats in different applications. We aim to observe how QuickFuzz behaves in the generation and fuzzing among those. Since the generation and fuzzing are intrinsically a random procedure, each experimental measure detailed in this section was repeated 10 times in a dedicated core of an Intel i7 running at 3.40GHz.

6.1 Generation Size

An important parameter for generational fuzzers is the maximum size of the resulting file. Such value should be carefully controlled, allowing the user to set



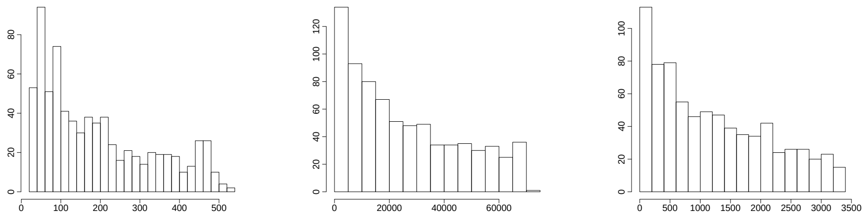
(a) Png (b) Svg (c) Zip

Figure 2: Average size in bytes of the generated files per file format

it, according to the resources available for the fuzzing campaign. Otherwise, if the file generation results in a very large number of tiny input files or extremely large ones, it will not be effective to detect bugs. The resulting fuzzing campaign will be either useless to trigger bugs in the target program or will consume a huge amount of memory and abort.

To avoid this pitfall, our instances of `Arbitrary` are carefully crafted to keep the size generated value under control using the `resize` function provided by QuickCheck. Figures 2a, 2b and 2c show how the average size of bytes behaves when the maximum QuickCheck *size* is increased. The size of the resulting files grows linearly according to the maximum size allowed to generate by the QuickCheck framework.

It is also important to take a deeper look in the sizes of the generated files to understand how they are distributed, considering that a bias toward the generation of small files is useful in the context of the bug finding task. In fact, the benefit is twofold since 1) it keeps the amount of time spent in program executions low and 2) it prefers to generate small test cases. The resulting files triggering bugs or vulnerabilities tend to be quite small and therefore are easier to understand for developers looking to patch the faulty code.



(a) Png (b) Svg (c) Zip

Figure 3: Frequency of generated file sizes in bytes

In our experiments, we analyzed the size of the files of generated by QuickFuzz bucketing them in Figures 3a, 3b and 3c. In such figures, we can observe a bias for the generation of small input files.

6.2 Generation Effectiveness

Ideally, a fuzzer should generate or mutate inputs to produce a large number of distinctive executions to exercise different lines of code. Hopefully, this process should trigger conditions to discover unexpected behaviors in programs.

In order to explore the effectiveness of the generation of fuzzed files in QuickFuzz, we evaluate how many different executions we can obtain in the parsing and processing of the generated files. For the purposes of our experiments, we use the coverage measure know as *path* employed by American Fuzzy Lop [3], a well-known fuzzer, because:

- It was designed to be useful in the fuzzing campaigns: finding more *paths* is highly correlated with the discovery of more bugs [11].
- It was built using a modular approach: we can easily re-use the corresponding command line program to only extract *paths* and count them.
- It has a very fast instrumentation: it allows to extract *paths* at a nearly native speed.

Note that the AFL coverage metric might map different executions to the same *path*.

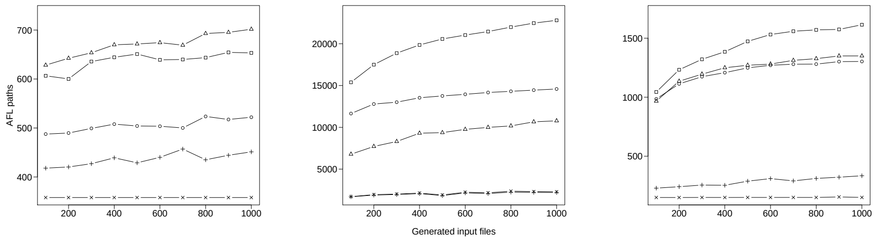
In our experiments, we use QuickFuzz to generate and fuzz Png, Jpeg and Xml files. Then, we run each fuzzed file as input to widely deployed open source libraries to parse and process them: we compiled instrumented libraries to parse Png files using *libpng 1.2.50*, Xml files using *libxml 2.9.1*, and Jpeg files using *libjpeg-turbo 1.3.0*. Figures 4a, 4b and 4c show how many *paths* can be extracted from each instrumented implementation either using low-level mutators (zzuf and radamsa) or directly executing the generated file.

We also included two baseline measures to compare how the file structure created by our tool improves the *path* discovery. The first one generating files of random bytes and the second one using the corresponding magic numbers followed by a random bytes.

In the case of random generation, the image parsers *libjpeg-turbo* and *libpng* will try to find a valid image since they work with arbitrary binary data. The *libxml 2.9.1* rejects the random file very early in the parsing process even if it starts like a valid Xml file.

QuickFuzz discovers consistently more paths that these two baselines using random file generation.

Also, as expected, if the user generates more files using QuickFuzz, it is more likely to discover more *paths*. Additionally, the number of discovered *paths* will grow very slowly after a few thousands files generated. This is understandable, since QuickFuzz works as *blind* fuzzer: it does not receive any feedback on the executions.



(a) Png/libpng 1.2.50 (b) Xml/libxml 2.9.1 (c) Jpeg/libjpeg-turbo 1.3.0

Figure 4: Average of *paths* discovered per file formats given the number of generated files. In this plots, circles (◯) represent execution of unaltered files, while triangles (△) are executions using files mutated by zzuf and squares (□) are executions using files mutated by radamsa. Pluses (+) and crosses (×) represent generation of random files with and without magic numbers respectively.

In some file formats the effect of low-level fuzzing becomes relevant. For instance, in the case of parsing fuzzed Xml files with libxml2, using radamsa as a low level fuzzers noticeable improves the number of discovered *paths*, compared to the executions of unaltered files.

Interestingly enough, mutating the files using zzuf produces quite the opposite effect: the number of *paths* is significantly reduced when this fuzzer is used. This behavior might be caused by the bit flipping of this fuzzer, causing the files to become too corrupted to be read, rejecting the files at the early stages of parsing.

6.3 Generation, Mutation and Execution Overhead

A good performance is critical in any fuzzer: we want to spend as little time as possible in the generation and mutation. For the overhead evaluation of QuickFuzz in the different stages of the fuzzing process, we measured the time required for high-level fuzzing with and without execution (noted as `gen+exec` and `gen` respectively) as well as high and low-level fuzzing using zzuf and radamsa (noted as `gen+exec+zzuf` and `gen+exec+rad` respectively).

To strictly quantify the overhead in execution, we used `/bin/echo` which does not read any file. Therefore, it should always take the same amount of time to execute.

Figure 5 shows a comparison of the time that QuickFuzz took to perform each step of the fuzzing process for three different file types. Our experiments suggest that the performance of the code generated by *MegaDeTH* is not limiting the other components of the tool. Additionally, as expected, there is a noticeable overhead in the execution. It is possible that most of the extra time executing is used for calling fork and exec primitives: this why is one the reasons some fuzzers implement a fork server [3].

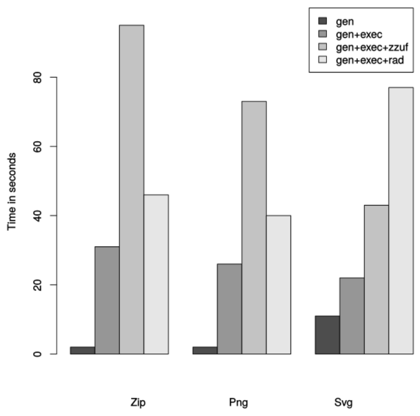


Figure 5: Overhead of QuickFuzz performing the fuzzing process.

We expected that the overhead introduced by the use of a fuzzer to be consistent regardless of the data to mutate. For instance, in the case of `zzuf`, a fuzzer which only XORs bits from the input files without reading them, it should be a constant overhead. However, the case of `Radamsa` is different. It is a fuzzer which looks at the structure of the data and performs some mutations according to it. In fact, it was specially designed to detect and fuzz markup languages: this can explain the higher overhead in the mutation of Svg files using it.

6.4 Real-World Vulnerabilities Detection

Thanks to Haskell implementations of file-format-handling libraries found on Hackage, QuickFuzz currently generates and mutates a large set of different file types out of the box. Table 6a shows a list of supported file types to generate and corrupt using our tool.

We tested QuickFuzz using complex real-world applications like browsers, image processing utilities and file archivers among others. All the security vulnerabilities presented in this work were previously unknown (also known as zero-days). The results are summarized in Table 6b. An exhaustive list is available at the official website of QuickFuzz, including frequent updates on the latest bugs discovered using the tool.

Additionally, we reported some ordinary bugs. For instance, the use of variable coherence enforcement allowed us to find a bug that stalls the compilation in Python, and more than a twenty memory issues in GNU Bash and Busybox.

6.5 Comparison with Other Fuzzers

Making a fair comparison between fuzzers is a challenge. First, it only makes sense to compare fuzzers using similar techniques. Second, in the case of generative ones, the model to produce files in all the compared fuzzers should be similar or somehow equivalent; otherwise, generating a complex input will

Code	Image	Document	Media	Archive	PKI
Javascript	Bmp	Pdf	Ogg	Zip	asn.1
Python	Gif	Ps	ID3	GZip	x509
HTML	Png	Docx	Midi	Tar	CRT
Lua	Jpeg	Odt	TTF	CPIO	
Json	Svg	Rtf	Wav		
Xml	Eps	ICal			
Css	Ico				
Sh	Tga				
GLSL	Tiff				
Dot	Pnm				
Regex					

(a) File-types supported for fuzzing

Program	File-Type	Reference	Program	File-Type	Reference
Firefox	Gif	CVE-2016-1933	Cairo	Svg	CVE-2016-9082
Firefox	Zip	CVE-2015-7194	libgd	Tga	CVE-2016-6132
Firefox	Svg	1297206	libgd	Tga	CVE-2016-6214
Firefox	Gif	1210745	GraphicsMagick	Svg	CVE-2016-2317
mujs	Js	CVE-2016-9109	GraphicsMagick	Svg	CVE-2016-2318
Webkit	Js	CVE-2016-9642	Mini-XML	Xml	CVE-2016-4570
Webkit	Regex	CVE-2016-9643	libical	Ical	CVE-2016-9584
gif2webp	Gif	CVE-2016-9085	Mini-Xml	Xml	CVE-2016-4571
VLC	Wav	CVE-2016-3941	GDK-pixbuf	Bmp	CVE-2015-7552
Jasper	Jpeg	CVE-2015-5203	GDK-pixbuf	Gif	CVE-2015-7674
libXML	Xml	CVE-2016-4483	GDK-pixbuf	Tga	CVE-2015-7673
libXML	Xml	CVE-2016-3627	GDK-pixbuf	Ico	CVE-2016-6352
Jq	Json	CVE-2016-4074	mplayer	Wav	CVE-2016-5115
Jasson	Json	CVE-2016-4425	mplayer	Gif	CVE-2016-4352
cpio	CPIO	CVE-2016-2037	libTIFF	Tiff	CVE-2015-7313

(b) Some of the security issues found by QuickFuzz

Figure 6: Implementation and results

most likely take varying amounts of time and could result in some fuzzers being unfairly flagged as *slow and inefficient*.

Moreover, some fuzzers like Peach are not useful to start discovering bugs immediately after installing them since they include almost no models to start the input generation process. Usually, if you want to have a wide support of file types or protocols to fuzz, you need to pay to access them [59] or hire an specialist to create them. In other cases like Sulley, fuzzers are developed to be more like a framework in which you can define models, mutate, and monitor the process. As a result, no file-type specifications are provided out of the box.

Recently, Mozilla released Dharma, a fuzzer to generate very specific files like Canvas2D and Node.js buffer scripts. It was designed by the Mozilla Security team to stress the API of Firefox. Nevertheless, this tool is a good candidate to compare with QuickFuzz since it includes a grammar to generate Svg files and our tool currently supports to generate this kind of files through the types and functions of *svg-tree* package [60].

A comparison of the bugs and vulnerabilities discovered by both fuzzers is not possible: we could not find any public information regarding how many issues were reported thanks to Dharma. However, we suspect that Mozilla

Security already used it extensively to improve the quality of the Firefox parsers and the render engine.

Fortunately, it is certainly possible to compare the throughput of both fuzzers: QuickFuzz has approximately 1.9 times more throughput generating files Svg files than Dharma. While this measure is far from perfect, it gives a hint at how optimized is the generation of files using our tool.

6.6 Limitations

The use of third-party modules from Hackage carries some limitations. Some of the modules we used to serialize complex file types do not implement all the features. For instance, the Bmp support in `Juicy.Pixels` cannot handle or serialize compressed files. Therefore this feature will not be effectively tested in the Bmp parsers. In this sense, types are used as incomplete specifications of file formats.

We performed some experiments to compare how good is the input generation variety of QuickFuzz against a mature and complete test suite of Png files. We used a test suite created by Willem van Schaik [61] that contains a variety of small Png files. It covers different color types (grayscale, rgb, palette, etc.), bit depths, interlacing and transparency configurations allowed by the Png standard. Also, in order to test robustness in the Png parsers, this test suite includes valid images using odd sizes (for instance, very small and very large) and corrupted images. We counted the amount of distinctive *paths* after processing all the Png files in the test suite using `pngtest` from libpng [62]. We performed the same experiment, but using QuickFuzz to generate and mutate Png files 10,000 times.

The execution of test suite uncovers 6268 different *paths*, while the generation and fuzzing of 10,000 Png files using QuickFuzz, only discovers 746 different *paths*. Therefore, our tool can only trigger $\sim 11\%$ of the *paths* we discover parsing a complex image format like Png.

There are several explanations for such low coverage compared with a complete test suite like `pngtest`. On one hand, the generation of Png files in QuickFuzz is limited by supported features in third-party libraries like *Juicy.Pixels* [49]. For instance, this library lacks the code to encode interleaved Png images. On the other hand, good test suites like this one are very expensive to create since they require a very deep knowledge of the file format to test. The use of automatic tools for test suites synthesis is still challenging.

Despite the automatic generation of a high-quality corpus of a very complex file format like Png is still unfeasible, it is a long-term goal of our research.

Another limitation related to the `encode` function is caused by the use of partial functions. There, the encoding could fail to execute correctly in a large number of randomly generated inputs. For instance, if the `encode` function requires some *hard* constrain to be present in the generated value such as some particular *magic* number to be guessed:

```
encodeHeader :: Int → ByteString
encodeHeader version
   | version == 87 = "GIF87"
   | version == 89 = "GIF89"
   | otherwise     = error "invalid version"
```

In this function, the encoding of gif format files only defines two version numbers 87 and 89: therefore, the approach to value generation defined in 4.1 is not going to be effective, since the probability of selecting a valid version number is 1 in $2,147,483,647$. Currently, these kinds of issues are avoided by manually selecting suitable libraries from Hackage to integrate into QuickFuzz.

Finally, the `encode` function used in the serialization includes its own bugs. Unsurprisingly some of them can be triggered by the generation of QuickCheck values. We reported some of these issues as bugs [63] to the upstream developers of the libraries we use in QuickFuzz. In any case, we have a simple workaround when no fix is available: if the `encode` function throws an unhandled exception, we ignore it and continue the fuzzing process using the next generated value to serialize.

7 Related Work

Automatic algebraic data type test generation Claessen et al. [64] propose a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. The derived generators are both efficient and guaranteed to produce a uniform distribution over values of a given size.

While *MegaDeTH* currently produces generators with ad-hoc distributions, it would be feasible to integrate this technique into the existing machinery to achieve more control over the test case generation process.

Testing compilers generating random programs As we stated in 4.3, we observed that `Arbitrary` instances are not always effective in the generation of source code, since it requires to carefully define variable names and functions before trying to use them. Therefore, the fuzzer-generated source code will very likely be rejected in the first parsing steps of interpreters or compilers. This is a well-known issue that has been studied extensively by Pałka et al. [65] in the context of testing a compiler.

The approach in that paper always generates valid lambda calculus terms, representing programs in Haskell. Then, they compiled the resulting terms using the Glasgow Haskell compiler in different optimization levels, to try to discover incorrectly compiled code.

In this sense, our tool also manages to generate source code and can be used to test compilers. Nevertheless, they are designed with different goals in mind; on one hand, the authors of [65] generate a program of a strongly typed language. They define suitable rules for the generation, and how to backtrack in case of failing to use them.

On the other hand, QuickFuzz generates only source code from dynamically typed programs, without using any backtracking in order to keep the generation very fast, but not always correct.

8 Conclusions and Future Work

We have presented QuickFuzz, a tool for automatically generating inputs and fuzzing programs that work on common file formats. Unlike other fuzzers, QuickFuzz does not require the user to provide a set of valid inputs to mutate, not to place the burden of writing specifications for file formats on the programmer. Our tool combines both generational and mutational fuzzing techniques by bringing together Haskell's QuickCheck library and off-the-shelf robust mutational fuzzers. In addition, we introduce *MegaDeTH*, a library that can be used to generate instances of the `Arbitrary` type classes. *MegaDeTH* works in tandem with QuickFuzz, allowing us to crowd-source the specifications for well-known file formats that are already present in Hackage libraries. We tried QuickFuzz in the wild and found that the approach is effective in discovering interesting bugs in real-world implementations. Moreover, to the best of our knowledge QuickFuzz is the only fuzzing tool that provides out-of-the-box generation and mutation of almost forty complex common file formats, without requiring users to write models or configuration files.

As future work, we intend to introduce mutations at different levels of the QuickFuzz pipeline rather than just at the level of the serialized `ByteString`. In particular, we aim to explore code analysis of the serialization functions to detect and selectively break invariants and to perform mutations on such functions to corrupt files.

Another interesting feature to add to our tool is the input simplification procedure [66]. This procedure can be used just after a crash is detected and is very important for the developers looking to fix the issue, since the minimized test case should only trigger the code that is required to reproduce the unexpected behavior.

Our goal is to implement a general way to automatically derive specialized input simplification strategies for algebraic data types encoding different file formats. Moreover, by using the actions-based approach we would like to work in a higher level of abstraction, reducing a test case to the minimal sequence of actions needed to trigger an error on target programs.

Additionally, we observed that in general Haskell programmers implement their libraries in the more general way they can abusing of the expressive power of Haskell data-type ecosystem. Therefore the action-based approach is a good starting point to derive Generalized Algebraic Data-types that can provide us with more information based on the functions found in the library, and we might capture effectful behaviors with this idea.

Finally, we would like to extend our approach to the generation and fuzzing of network protocols, since most of the vulnerabilities there can be remotely exploitable.

Paper II

Branching Processes for QuickCheck Generators

Agustín Mista and Alejandro Russo

ACM SIGPLAN Haskell Symposium (2018)

Abstract

In *QuickCheck* (or, more generally, random testing), it is challenging to control random data generators' distributions—especially when it comes to *user-defined algebraic data types* (ADT). In this paper, we adapt results from an area of mathematics known as *branching processes*, and show how they help to analytically predict (at compile-time) the expected number of generated constructors, even in the presence of mutually recursive or composite ADTs. Using our probabilistic formulas, we design heuristics capable of automatically adjusting probabilities in order to synthesize generators whose distributions are aligned with users' demands. We provide a Haskell implementation of our mechanism in a tool called *DRAGEN* and perform case studies with real-world applications. When generating random values, our synthesized *QuickCheck* generators show improvements in code coverage when compared with those automatically derived by state-of-the-art tools.

1 Introduction

Random property-based testing is an increasingly popular approach to finding bugs [19], [20], [67]. In the Haskell community, *QuickCheck* [16] is the dominant tool of this sort. *QuickCheck* requires developers to specify *testing properties* describing the expected software behavior. Then, it generates a large number of random *test cases* and reports those violating the testing properties. *QuickCheck* generates random data by employing *random test data generators* or *QuickCheck* generators for short. The generation of test cases is guided by the *types* involved in the testing properties. It defines default generators for many built-in types like booleans, integers, and lists. However, when it comes to user-defined ADTs, developers are usually required to specify the generation process. The difficulty is, however, that it might become intricate to define generators so that they result in a suitable distribution or enforce data invariants.

The state-of-the-art tools to derive generators for user-defined ADTs can be classified based on the automation level as well as the sort of invariants enforced at the data generation phase. *QuickCheck* and *SmallCheck* [68] (a tool for writing generators that synthesize small test cases) use type-driven generators written by developers. As a result, generated random values are well-typed and preserve the structure described by the ADT. Rather than manually writing generators, libraries *derive* [69] and *MegaDeTH* [70], [71] automatically synthesize generators for a given user-defined ADT. The library *derive* provides no guarantees that the generation process terminates, while *MegaDeTH* pays almost no attention to the distribution of values. In contrast, *Feat* [29] provides a mechanism to uniformly sample values from a given ADT. It enumerates all the possible values of a given ADT so that sampling uniformly from ADTs becomes sampling uniformly from the set of natural numbers. *Feat*'s authors subsequently extend their approach to *uniformly* generate values constrained by user-defined predicates [64]. Lastly, *Luck* is a domain-specific language for manually writing *QuickCheck* properties in tandem with generators so that it becomes possible to finely control the distribution of generated values [72].

In this work, we consider the scenario where developers are not fully aware of the properties and invariants that input data must fulfill. This constitutes a valid assumption for *penetration testing* [2], where testers often apply fuzzers in an attempt to make programs crash—an anomaly that might lead to a vulnerability. We believe that, in contrast, if users can recognize specific properties of their systems then it is preferable to spend time writing specialized generators for that purpose (e.g., by using *Luck*) instead of considering automatically derived ones.

Our realization is that *branching processes* [73], a relatively simple stochastic model conceived to study the evolution of populations, can be applied to predict the generation distribution of ADTs' constructors in a simple and automatable manner. To the best of our knowledge, this stochastic model has not yet been applied to this field, and we believe it may be a promising foundation to develop future extensions. The contributions of this paper can be outlined as follows:

- We provide a mathematical foundation that helps to analytically characterize the distribution of constructors in derived *QuickCheck* generators for ADTs.
- We show how to use type reification to simplify our prediction process and extend our model to mutually recursive and composite types.
- We design (compile-time) heuristics that automatically search for probability parameters so that distributions of constructors can be adjusted to what developers might want.
- We provide an implementation of our ideas in the form of a Haskell library[2] called *DRAGEN* (the Danish word for *dragon*, here standing for *Derivation of RAndom GENerators*).
- We evaluate our tool by generating inputs for real-world programs, where it manages to obtain significantly more code coverage than those random inputs generated by *MegaDeTH*'s generators.

Overall, our work addresses a timely problem with a neat mathematical insight that is backed by a complete implementation and experience with third-party examples.

2 Background

In this section, we briefly illustrate how *QuickCheck* random generators work. We consider the following implementation of binary trees:

```
data Tree = LeafA | LeafB | LeafC | Node Tree Tree
```

In order to help developers write generators, *QuickCheck* defines the `Arbitrary` type-class with the overloaded symbol `arbitrary :: Gen a`, which denotes a monadic generator for values of type `a`. Then, to generate random trees, we need to provide an instance of the `Arbitrary` type-class for the type `Tree`. Figure 1 shows a possible implementation. At the top level, this generator simply uses *QuickCheck*'s primitive `oneof :: [Gen a] → Gen a` to pick a generator from a list of generators with uniform probability. This list consists of a random

[2]Available at https://github.com/OctopiChalmers/dragen

```
instance Arbitrary Tree where
  arbitrary = oneof
    [pure LeafA, pure LeafB, pure LeafC
    , Node ⟨$⟩ arbitrary ⟨∗⟩ arbitrary]
```

Figure 1: Random generator for Tree.

```
instance Arbitrary Tree where
  arbitrary = sized gen
    where
      gen 0 = oneof
        [pure LeafA, pure LeafB, pure LeafC]
      gen n = oneof
        [pure LeafA, pure LeafB, pure LeafC
        , Node ⟨$⟩ gen (div n 2) ⟨*⟩ gen (div n 2)]
```

Figure 2: *MegaDeTH* generator for `Tree`.

generator for each possible choice of data constructor of `Tree`. We use *applicative style* [48] to describe each one of them idiomatically. So, `pure LeafA` is a generator that always generates `LeafA`s, while `Node ⟨$⟩ arbitrary ⟨*⟩ arbitrary` is a generator that always generates `Node` constructors, "filling" its arguments by calling `arbitrary` recursively on each of them.

Although it might seem easy, writing random generators becomes cumbersome very quickly. Particularly, if we want to write a random generator for a user-defined ADT `T`, it is also necessary to provide random generators for every user-defined ADT inside of `T` as well! What remains of this section is focused on explaining the state-of-the-art techniques used to *automatically* derive generators for user-defined ADTs via type-driven approaches.

2.1 Library *derive*

The simplest way to automatically derive a generator for a given ADT is the one implemented by the Haskell library *derive* [69]. This library uses Template Haskell [30] to automatically synthesize a generator for the data type `Tree` semantically equivalent to the one presented in Figure 1.

While the library *derive* is a big improvement for the testing process, its implementation has a serious shortcoming when dealing with recursively defined data types: in many cases, there is a non-zero probability of generating a recursive type constructor every time a recursive type constructor gets generated, which can lead to infinite generation loops. A detailed example of this phenomenon is given in Appendix 2.1. In this work, we only focus on derivation tools that accomplish terminating behavior, since we consider this an essential component of well-behaved generators.

2.2 *MegaDeTH*

The second approach we will discuss is the one taken by *MegaDeTH*, a metaprogramming tool used intensively by *QuickFuzz* [70], [71]. In the first place, *MegaDeTH* derives random generators for ADTs as well as all of its nested types—a useful feature not supported by *derive*. Secondly, *MegaDeTH* avoids potentially infinite generation loops by setting an upper bound to the random generation recursive depth.

Figure 2 shows a simplified (but semantically equivalent) version of the random generator for `Tree` derived by *MegaDeTH*. This generator uses *QuickCheck*'s function `sized` :: (`Int` → `Gen a`) → `Gen a` to build a random generator based on a function (of type `Int` → `Gen a`) that limits the possible recursive calls performed when creating random values. The integer passed to `sized`'s argument is called the *generation size*. When the generation size is zero (see definition `gen 0`), the generator only chooses between the `Tree`'s terminal constructors—thus ending the generation process. If the generation size is strictly positive, it is free to randomly generate any `Tree` constructor (see definition `gen n`). When it chooses to generate a recursive constructor, it reduces the generation size for its subsequent recursive calls by a factor that depends on the number of recursive arguments this constructor has (`div n 2`). In this way, *MegaDeTH* ensures that all generated values are finite.

Although *MegaDeTH* generators always terminate, they have a major practical drawback: in our example, the use of `oneof` to uniformly decide the next constructor to be generated produces a generator that generates leaves approximately three quarters of the time (note this also applies to the generator obtained with *derive* from Figure 1). This entails a distribution of constructors heavily concentrated on leaves, with a very small number of complex values with nested nodes, regardless of how large the chosen generation size is—see Figure 3 (left).

2.3 *Feat*

The last approach we discuss is *Feat* [29]. This tool determines the distribution of generated values in a completely different way: it uses uniform generation based on an *exhaustive enumeration of all the possible values of the ADTs being considered*. *Feat* automatically establishes a bijection between all the possible values of a given type `T`, and a finite prefix of the natural numbers. Then, it guarantees a *uniform generation over the complete space of values of a given data type* `T` up to a certain size.[3] However, the distribution of size, given by the number of constructors in the generated values, is highly dependent on the structure of the data type being considered.

Figure 3 (right) shows the overall distribution shape of a *QuickCheck* generator derived using *Feat* for `Tree` using a generation size of 400, i.e., generating values of up to 400 constructors.[4] Notice that all the generated values are close to the maximum size! This phenomenon follows from the exponential growth in the number of possible `Tree`s of `n` constructors as we increase `n`. In other words, the space of `Tree`s up to 400 constructors is composed to a large extent of values with around 400 constructors, and (proportionally) very

[3]We avoid including any source code generated by *Feat*, since it works by synthesizing `Enumerable` type-class instances instead of `Arbitrary` ones. Such instances give no insight into how the derived random generators work.

[4]We choose to use this generation size here since it helps us to compare *MegaDeTH* and *Feat* with the results of our tool in Section 8.

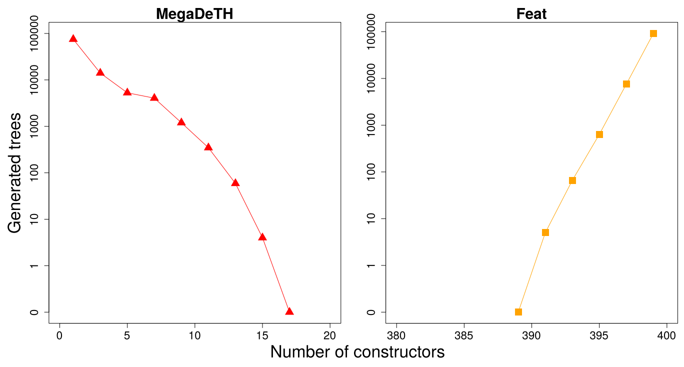


Figure 3: Size distribution of 100000 randomly generated `Tree` values using *MegaDeTH* (▲) with generation size 10, and *Feat* (■) with generation size 400.

few with a smaller number of constructors. Hence, a generation process based on uniform generation of a natural number (which thus ignores the structure of the type being generated) is biased very strongly towards values made up of a large number of constructors. In our tests, no `Tree` with less than 390 constructors was ever generated. In practice, this problem can be partially solved by using a variety of generation sizes in order to get more diversity in the generated values. However, deciding which generation sizes are the best choices is not a trivial task either. As a consequence, in this work we consider only the case of fixed-size random generation.

As we have shown, by using both *MegaDeTH* and *Feat*, the user is tied to the fixed generation distribution that each tool produces, which tends to be highly dependent on the particular data type under consideration in each case. Instead, this work aims to provide a *theoretical framework able to predict and later tune the distributions of automatically derived generators*, giving the user a more flexible testing environment, while keeping it as automated as possible.

3 Simple-Type Branching Processes

Galton-Watson Branching processes (or branching processes for short) are a particular case of Markov processes that model the growth and extinction of populations. Originally conceived to study the extinction of family names in the Victorian era, this formalism has been successfully applied to a wide range of research areas in biology and physics—see the textbook by Haccou et al. [74] for an excellent introduction. In this section, we show how to use this theory to model *QuickCheck*'s distribution of constructors.

We start by analyzing the generation process for the `Node` constructors in the data type `Tree` as described by the generators in Figure 1 and 2. From the code, we can observe that the stochastic process they encode satisfies the following assumptions (which coincide with the assumptions of Galton-Watson branching processes): i) With a certain probability, it starts with some initial

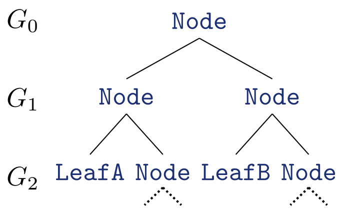


Figure 4: Generation of Node constructors.

Node constructor. ii) At any step, the probability of generating a Node is not affected by the Nodes generated before or after. iii) The probability of generating a Node is independent of where in the tree that constructor is about to be placed.

The original Galton-Watson process is a simple stochastic process that counts the population sizes at different points in time called *generations*. For our purposes, populations consist of Node constructors, and generations are obtained by selecting tree levels.

Figure 4 illustrates a possible generated value. It starts by generating a Node constructor at generation (i.e., depth) zero (G_0), then another two Node constructors as left and right subtrees in generation one (G_1), etc. (Dotted edges denote further constructors which are not drawn, as they are not essential for the point being made.) This process repeats until the population of Node constructors becomes extinct or stable, or alternatively grows forever.

The mathematics behind the Galton-Watson process allows us to predict the expected number of offspring at the nth-generation, i.e., the number of Node constructors at depth n in the generated tree. Formally, we start by introducing the random variable R to denote the number of Node constructors in the next generation generated by a Node constructor in this generation—the R comes from "reproduction" and the reader can think it as a Node constructor reproducing Node constructors. To be a bit more general, let us consider the Tree random generator automatically generated using *derive* (Figure 1), but where the probability of choosing between any constructor is no longer uniform. Instead, we have a p_C probability of choosing the constructor C.

These probabilities are external parameters of the prediction mechanism, and Section 7 explains how they can later be instantiated with actual values found by optimization, enabling the user to tune the generated distribution.

We note p_{Leaf} as the probability of generating a leaf of any kind, i.e., $p_{Leaf} = p_{LeafA} + p_{LeafB} + p_{LeafC}$. In this setting, and assuming a parent constructor Node, the probabilities of generating R numbers of Node offspring in the next generation (i.e., in the recursive calls of `arbitrary`) are as follows:

$$\begin{aligned}
P(R = 0) &= p_{Leaf} \cdot p_{Leaf} \\
P(R = 1) &= p_{Node} \cdot p_{Leaf} + p_{Leaf} \cdot p_{Node} = 2 \cdot p_{Node} \cdot p_{Leaf} \\
P(R = 2) &= p_{Node} \cdot p_{Node}
\end{aligned}$$

One manner to understand the equations above is by considering what *QuickCheck* does when generating the subtrees of a given node. For instance, the cases when generating exactly one `Node` as descendant ($P(R = 1)$) occurs in two situations: when the left subtree is a `Node` and the right one is a `Leaf`; and vice-versa. The probability for those events to occur is $p_{Node} * p_{Leaf}$ and $p_{Leaf} * p_{Node}$, respectively. Then, the probability of having exactly one `Node` as a descendant is given by the sum of the probability of both events—the other cases follow a similar reasoning.

Now that we have determined the distribution of R, we proceed to introduce the random variables G_n to denote the population of `Node` constructors in the nth generation. We write ξ_i^n for the random variable which captures the number of (offspring) `Node` constructors at the nth generation produced by the ith `Node` constructor at the *(n-1)*th generation. It is easy to see that it must be the case that:

$$G_n = \xi_1^n + \xi_2^n + \cdots + \xi_{G_{n-1}}^n$$

To deduce $E[G_n]$, i.e. the expected number of `Node`s in the nth generation, we apply the (standard) Law of Total Expectation $E[X] = E[E[X|Y]]$[5] with $X = G_n$ and $Y = G_{n-1}$ to obtain:

$$E[G_n] = E[E[G_n|G_{n-1}]]. \tag{1}$$

By expanding G_n, we deduce that:

$$\begin{aligned} E[G_n|G_{n-1}] &= E[\xi_1^n + \xi_2^n + \cdots + \xi_{G_{n-1}}^n | G_{n-1}] \\ &= E[\xi_1^n|G_{n-1}] + E[\xi_2^n|G_{n-1}] + \cdots + E[\xi_{G_{n-1}}^n|G_{n-1}] \end{aligned}$$

Since ξ_1^n, ξ_2^n, ..., and $\xi_{G_{n-1}}^n$ are *all governed by the distribution captured by the random variable* R (recall the assumptions at the beginning of the section), we have that:

$$E[G_n|G_{n-1}] = E[R|G_{n-1}] + E[R|G_{n-1}] + \cdots + E[R|G_{n-1}]$$

Since R is *independent of the generation where* `Node` *constructors decide to generate other* `Node` *constructors*, we have that

$$E[G_n|G_{n-1}] = \underbrace{E[R] + E[R] + \cdots + E[R]}_{G_{n-1} \text{ times}} = E[R] \cdot G_{n-1} \tag{2}$$

From now on, we introduce m to denote the mean of R, i.e., *the mean of reproduction.* Then, by rewriting $m = E[R]$, we obtain:

$$E[G_n] \overset{(1)}{=} E[E[G_n|G_{n-1}]] \overset{(2)}{=} E[m \cdot G_{n-1}] \overset{\text{m is constant}}{=} E[G_{n-1}] \cdot m$$

By unfolding this recursive equation many times, we obtain:

[5] $E[X|Y]$ is a function on the random variable Y, i.e., $E[X|Y]y = E[X|Y = y]$ and therefore it is a random variable itself. In this light, the law says that if we observe the expectations of X given the different y_s, and then we do the expectation of all those values, then we have the expectation of X.

$$E[G_n] = E[G_0]{\cdot}m^n \tag{3}$$

As the equation indicates, the expected number of `Node` constructors at the nth generation is affected by the mean of reproduction. Although we obtained this intuitive result using a formalism that may look overly complex, it is useful to understand the methodology used here. In the next section, we will derive the main result of this work following the same reasoning line under a more general scenario.

We can now also predict the total expected number of individuals *up to* the nth generation. For that purpose, we introduce the random variable P_n to denote the population of `Node` constructors up to the nth generation. It is then easy to see that $P_n = \sum_{i=0}^{n} G_i$ and consequently:

$$E[P_n]{=}\sum_{i=0}^{n} E[G_i] \overset{(3)}{=} \sum_{i=0}^{n} E[G_0] \cdot m^i = E[G_0]{\cdot}\left(\frac{1{-}m^{n+1}}{1{-}m}\right) \tag{4}$$

where the last equality holds by the geometric series definition. This is the general formula provided by the Galton-Watson process. In this case, the mean of reproduction for `Node` is given by:

$$m = E[R] = \sum_{k=0}^{2} k \cdot P(R = k) = 2 \cdot p_{Node} \tag{5}$$

By (4) and (5), the expected number of `Node` constructors up to generation n is given by the following formula:

$$E[P_n]{=}E[G_0]{\cdot}\left(\frac{1{-}m^{n+1}}{1{-}m}\right){=}p_{\texttt{Node}}{\cdot}\left(\frac{1{-}(2 \cdot p_{\texttt{Node}})^{n+1}}{1{-}2{\cdot}p_{\texttt{Node}}}\right)$$

If we apply the previous formula to predict the distribution of constructors induced by *MegaDeTH* in Figure 2, where $p_{LeafA} = p_{LeafB} = p_{LeafC} = p_{Node} = 0.25$, we obtain an expected number of `Node` constructors up to level 10 of 0.4997, which denotes a distribution highly biased towards small values since we can only produce further subterms by producing `Node`s. However, if we set $p_{LeafA} = p_{LeafB} = p_{LeafC} = 0.1$ and $p_{Node} = 0.7$, we can predict that, as expected, our general random generator will generate much bigger trees, containing an average number of 69.1173 `Node`s up to level 10! Unfortunately, we cannot apply this reasoning to predict the distribution of constructors for derived generators for ADTs with more than one non-terminal constructor. For instance, let us consider the following data type definition:

data `Tree′` = `Leaf` | `NodeA Tree′ Tree′` | `NodeB Tree′`

In this case, we need to separately consider that a `NodeA` can generate not only `NodeA` but also `NodeB` offspring (similarly with `NodeB`). A stronger mathematical formalism is needed. The next section explains how to predict the generation of this kind of data types by using an extension of Galton-Waston processes known as *multi-type branching processes*.

4 Multi-Type Branching Processes

In this section, we present the basis for our main contribution: *the application of multi-type branching processes to predict the distribution of constructors*. We will illustrate the technique by considering the `Tree′` ADT that we concluded with in the previous section.

Before we dive into technicalities, Figure 5 shows the automatically derived generator for `Tree′` that our tool produces. Our generators depend on the (possibly) different probabilities that constructors have to be generated—variables p_{Leaf}, p_{NodeA}, and p_{NodeB}. These probabilities are used by the function `chooseWith :: [(Double, Gen a)] → Gen a`, which picks a random generator of type `a` with an explicitly given probability from a list. This function can be easily expressed by using *QuickCheck*'s primitive operations and therefore we omit its implementation. Additionally note that, like *MegaDeTH*, our generators use `sized` to limit the number of recursive calls to ensure termination. We note that the theory behind branching processes is able to predict the termination behavior of our generators and we could have used this ability to ensure their termination without the need of a depth limiting mechanism like `sized`. However, using `sized` provides more control over the obtained generator distributions.

To predict the distribution of constructors provided by *DRAGEN* generators, we introduce a generalization of the previous Galton-Watson branching process called multi-type Galton-Watson branching process. This generalization allows us to consider several *kinds of individuals*, i.e., constructors in our setting, to procreate (generate) different *kinds of offspring* (constructors). Additionally, this approach allows us to consider not just one constructor, as we did in the previous section, but rather to consider all of them at the same time.

Before we present the mathematical foundations, which follow a similar line of reasoning as that in Section 3, Figure 6 illustrates a possible generated value of type `Tree′`.

In the generation process, it is assumed that *the kind (i.e., the constructor) of the parent might affect the probabilities of reproducing (generating) offspring of a certain kind.* Observe that this is the case for a wide range of derived ADT generators, e.g., choosing a terminal constructor (e.g., `Leaf`) affects the

```
instance Arbitrary Tree′ where
  arbitrary = sized gen
    where
      gen 0 = pure Leaf
      gen n = chooseWith
        [(p_Leaf,   pure Leaf)
        ,(p_NodeA, NodeA <$> gen (n − 1) <*> gen (n − 1))
        ,(p_NodeB, NodeB <$> gen (n − 1))]
```

Figure 5: *DRAGEN* generator for `Tree′`

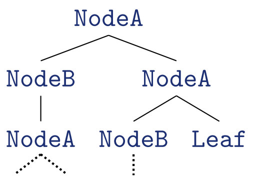


Figure 6: A generated value of type `Tree′`.

probabilities of generating non-terminal ones (by setting them to zero). The population at the nth generation is then characterized as a vector of random variables $G_n = (G_n^1, G_n^2, \cdots, G_n^d)$, where d is the number of different kinds of constructors. Each random variable G_n^i captures the number of occurrences of the ith-constructor of the ADT at the nth generation. Essentially, G_n "groups" the population at level n by the constructors of the ADT. By estimating the expected shape of the vector G_n, it is possible to obtain the expected number of constructors at the nth generation. Specifically, we have that $E[G_n] = (E[G_n^1], E[G_n^2], \cdots, E[G_n^d])$. To deduce $E[G_n]$, we focus on deducing each component of the vector.

As explained above, the reproduction behavior is determined by the kind of the individual. In this light, we introduce random variable R_{ij} to denote a parent ith constructor reproducing a jth constructor. As we did before, we apply the equation $E[X] = E[E[X|Y]]$ with $X = G_n^j$ and $Y = G_{n-1}$ to obtain $E[G_n^j] = E[E[G_n^j|G_{n-1}]]$. To calculate the expected number of jth constructors at the level n produced by the constructors present at level $(n-1)$, i.e., $E[G_n^j|G_{(n-1)}]$, it is enough to count the expected number of children of kind j produced by the different parents of kind i, i.e., $E[R_{ij}]$, times the amount of parents of kind i found in the level $(n-1)$, i.e., $G_{(n-1)}^i$. This result is expressed by the following equation marked as $(\star)$, and is formally verified in the Appendix 2.2.

$$E[G_n^j|G_{n-1}] \overset{(\star)}{=} \sum_{i=1}^{d} G_{(n-1)}^i \cdot E[R_{ij}] = \sum_{i=1}^{d} G_{(n-1)}^i \cdot m_{ij} \qquad (6)$$

Similarly as before, we rewrite $E[R_{ij}]$ as m_{ij}, which now represents a single expectation of reproduction indexed by the kind of both the parent and child constructor.

Mean matrix of constructors In the previous section, m was the expectation of reproduction of a single constructor. Now we have m_{ij} as the expectation of reproduction indexed by the parent and child constructor. In this light, we define M_C, the *mean matrix of constructors* (or mean matrix for simplicity) such that each m_{ij} stores the expected number of jth constructors generated by the ith constructor. M_C is a parameter of the Galton-Watson multi-type process and can be built at compile-time using statically known type information. We are now able to deduce $E[G_n^j]$.

$$\begin{aligned} E[G_n^j] = E[E[G_n^j | G_{n-1}]] &\overset{(6)}{=} E\left[\sum_{i=1}^{d} G^i_{(n-1)} \cdot m_{ij}\right] \\ &= \sum_{i=1}^{d} E[G^i_{(n-1)} \cdot m_{ij}] = \sum_{i=1}^{d} E[G^i_{(n-1)}] \cdot m_{ij} \end{aligned}$$

Using this last equation, we can rewrite $E[G_n]$ as follows.

$$E[G_n] = \left(\sum_{i=1}^{d} E[G^1_{(n-1)}] \cdot m_{i1}, \cdots, \sum_{i=1}^{d} E[G^d_{(n-1)}] \cdot m_{id}\right)$$

By linear algebra, we can rewrite the vector above as the matrix multiplication $E[G_n]^T = E[G_{n-1}]^T \cdot M_C$. By repeatedly unfolding this definition, we obtain that:

$$E[G_n]^T = E[G_0]^T \cdot (M_C)^n \tag{7}$$

This equation is a generalization of (3) when considering many constructors. As we did before, we introduce a random variable $P_n = \sum_{i=0}^{n} G_i$ to denote the population up to the nth generation. It is now possible to obtain the expected population of all the constructors but in a clustered manner:

$$E[P_n]^T = E\left[\sum_{i=0}^{n} G_i\right]^T = \sum_{i=0}^{n} E[G_i]^T \overset{(7)}{=} \sum_{i=0}^{n} E[G_0]^T \cdot (M_C)^n \tag{8}$$

It is possible to write the resulting sum as the closed formula:

$$E[P_n]^T = E[G_0]^T \cdot \left(\frac{I - (M_C)^{n+1}}{I - M_C}\right) \tag{9}$$

where I represents the identity matrix of the appropriate size. Note that equation (9) only holds when $(I - M_C)$ is non-singular, however, this is the usual case. When $(I - M_C)$ is singular, we resort to using equation (8) instead. Without losing generality, and for simplicity, we consider equations (8) and (9) as interchangeable. They are the general formulas for the Galton-Watson multi-type branching processes.

Then, to predict the distribution of our `Tree′` data type example, we proceed to build its mean matrix M_C. For instance, the mean number of `Leaf`s generated by a `NodeA` is:

$$\begin{aligned} m_{NodeA,Leaf} &= \underbrace{1 \cdot p_{Leaf} \cdot p_{NodeA} + 1 \cdot p_{Leaf} \cdot p_{NodeB}}_{\textit{One}\ \text{Leaf as left-subtree}} \\ &+ \underbrace{1 \cdot p_{NodeA} \cdot p_{Leaf} + 1 \cdot p_{NodeB} \cdot p_{Leaf}}_{\textit{One}\ \text{Leaf as right-subtree}} \\ &+ \underbrace{2 \cdot p_{Leaf} \cdot p_{Leaf}}_{\text{Leaf as left- and right-subtree}} \\ &= 2 \cdot p_{Leaf} \end{aligned} \tag{10}$$

The rest of M_C can be similarly computed, obtaining:

$$M_C = \begin{array}{c|ccc|} & Leaf & NodeA & NodeB \\ Leaf & 0 & 0 & 0 \\ NodeA & 2 \cdot p_{Leaf} & 2 \cdot p_{NodeA} & 2 \cdot p_{NodeB} \\ NodeB & p_{Leaf} & p_{NodeA} & p_{NodeB} \end{array} \qquad (11)$$

Note that the first row, corresponding to the `Leaf` constructor, is filled with zeros. This is because `Leaf` is a terminal constructor, i.e., it cannot generate further subterms of any kind.[6]

With the mean matrix in place, we define $E[G_0]$ (the initial vector of mean probabilities) as $(p_{Leaf}, p_{NodeA}, p_{NodeB})$. By applying (9) with $E[G_0]$ and M_C, we can predict the expected number of generated *non-terminal* `NodeA` constructors (and analogously `NodeB`) with a size parameter n as follows:

$$E[\texttt{NodeA}] = \left(E[P_{n-1}]^T\right).\texttt{NodeA} = \left(E[G_0]^T \cdot \left(\frac{I-(M_C)^n}{I-M_C}\right)\right).\texttt{NodeA}$$

Function $(\_).C$ simply projects the value corresponding to constructor C from the population vector. It is very important to note that the sum only includes the population up to level $(n-1)$. This choice comes from the fact that our *QuickCheck* generator can choose between only terminal constructors at the last generation level (recall that `gen 0` generates only `Leaf`s in Figure 5). As an example, if we assign our generation probabilities for `Tree′` as $p_{Leaf} \mapsto 0.2$, $p_{NodeA} \mapsto 0.5$ and $p_{NodeB} \mapsto 0.3$, then the formula predicts that our *QuickCheck* generator with a size parameter of 10 will generate on average 21.322 `NodeA`s and 12.813 `NodeB`s. This result can easily be verified by sampling a large number of values with a generation size of 10, and then averaging the number of generated `NodeA`s and `NodeB`s across the generated values.

In this section, we obtain a prediction of the expected number of non-terminal constructors generated by *DRAGEN* generators. To predict terminal constructors, however, requires special treatment as discussed in the next section.

5 Terminal Constructors

In this section, we introduce the special treatment required to predict the generated distribution of terminal constructors, i.e. constructors with no recursive arguments.

Consider the generator in Figure 5. It generates terminal constructors in two situations, i.e., in the definition of `gen 0` and `gen n`. In other words, the random process introduced by our generators can be considered to be composed of two independent parts when it comes to terminal constructors—refer to

[6]The careful reader may notice that there is a pattern in the mean matrix if inspected together with the definition of `Tree′`. We prove in Section 6 that each m_{ij} can be automatically calculated by simply exploiting type information.

Appendix 2.3 for a graphical interpretation. In principle, the number of terminal constructors generated by the stochastic process described by `gen n` is captured by the multi-type branching process formulas. However, to predict the expected number of terminal constructors generated by exercising `gen 0`, we need to separately consider a random process that *only generates terminal constructors* in order to terminate. For this purpose, and assuming a maximum generation depth n, we need to calculate the number of terminal constructors required to stop the generation process at the recursive arguments of each non-terminal constructor at level $(n-1)$. In our `Tree′` example, this corresponds to two `Leaf`s for every `NodeA` and one `Leaf` for every `NodeB` constructor at level $(n-1)$.

Since both random processes are independent, to predict the overall expected number of terminal constructors, we can simply add the expected number of terminal constructors generated in each one of them. Recalling our previous example, we obtain the following formula for `Tree′` terminals as follows:

$$
\begin{aligned}
E[\texttt{Leaf}] = \underbrace{\left(E[P_{n-1}]^T\right).\texttt{Leaf}}_{\text{branching process}} &+ \underbrace{2\cdot\left(E[G_{n-1}]^T\right).\texttt{NodeA}}_{\text{case } (\texttt{NodeA Leaf Leaf})} \\
&+ \underbrace{1\cdot\left(E[G_{n-1}]^T\right).\texttt{NodeB}}_{\text{case } (\texttt{NodeB Leaf})}
\end{aligned}
$$

The formula counts the `Leaf`s generated by the multi-type branching process up to level $(n-1)$ and adds the expected number of `Leaf`s generated at the last level.

Although we can now predict the expected number of generated `Tree′` constructors regardless of whether they are terminal or not, this approach only works for data types with a single terminal constructor.

If we have a data type with multiple terminal constructors, we have to consider the probabilities of choosing each one of them when filling in the recursive arguments of non-terminal constructors at the previous level. For instance, consider the following ADT:

data `Tree″` = `LeafA` | `LeafB` | `NodeA Tree″ Tree″` | `NodeB Tree″`

Figure 7 shows the corresponding *DRAGEN* generator for `Tree″`. Note there are two sets of probabilities to choose terminal nodes, one for each random process. The p^*_{LeafA} and p^*_{LeafB} probabilities are used to choose between terminal constructors at the last generation level. These probabilities preserve the same proportion as their non-starred versions, i.e., they are normalized to form a probability distribution:

$$
p^*_{LeafA} = \frac{p_{LeafA}}{p_{LeafA} + p_{LeafB}} \qquad p^*_{LeafB} = \frac{p_{LeafB}}{p_{LeafA} + p_{LeafB}}
$$

In this manner, we can use the same generation probabilities for terminal constructors in both random processes—therefore reducing the complexity of our prediction engine implementation (described in Section 7).

To compute the overall expected number of terminals, we need to predict the expected number of terminal constructors at the last generation level which

```
instance Arbitrary Tree'' where
  arbitrary = sized gen
    where
      gen 0 = chooseWith
        [(p*_LeafA, pure LeafA), (p*_LeafB, pure LeafB)]
      gen n = chooseWith
        [(p_LeafA, pure LeafA), (p_LeafB, pure LeafB)
        , (p_NodeA, NodeA <$> gen (n-1) <*> gen (n-1))
        , (p_NodeB, NodeB <$> gen (n-1))]
```

Figure 7: Derived generator for $\mathtt{Tree''}$

could be descendants of non-terminal constructors at level $(n-1)$. More precisely:

$$E[\mathtt{LeafA}] = \underbrace{\left(E[P_{n-1}]^T\right).\mathtt{LeafA}}_{\text{branching process}} + \underbrace{2 \cdot p^*_{LeafA} \cdot \left(E[G_{n-1}]^T\right).\mathtt{NodeA}}_{\text{expected leaves to fill } \mathtt{NodeA}s}$$
$$+ \underbrace{1 \cdot p^*_{LeafA} \cdot \left(E[G_{n-1}]^T\right).\mathtt{NodeB}}_{\text{expected leaves to fill } \mathtt{NodeB}s}$$

where the case of $E[\mathtt{LeafB}]$ follows analogously.

6 Mutually-Recursive and Composite ADTs

In this section, we introduce some extensions to our model that allow us to derive *DRAGEN* generators for data types found in existing off-the-shelf Haskell libraries. We start by showing how multi-type branching processes naturally extend to mutually-recursive ADTs. Consider the mutually recursive ADTs $\mathtt{T_1}$ and $\mathtt{T_2}$ with their automatically derived generators shown in Figure 8.

Note the use of the *QuickCheck*'s function `resize :: Int → Gen a → Gen a`, which resets the generation size of a given generator to a new value. We use it to decrement the generation size at the recursive calls of `arbitrary` that generate subterms of a mutually recursive data type.

The key observation is that we can ignore that `A`, `B`, `C` and `D` are *constructors belonging to different data types* and just consider each of them as a kind of offspring on their own. Figure 9 visualizes the possible offspring generated by the non-terminal constructor `B` (belonging to $\mathtt{T_1}$) with the corresponding probabilities as labeled edges. Following the figure, we obtain the expected number of `D`s generated by `B` constructors as follows:

$$m_{BD} = 1 \cdot p_A \cdot p_D + 1 \cdot p_B \cdot p_D = p_D \cdot (p_A + p_B) = p_D$$

```
data T₁ = A | B T₁ T₂
data T₂ = C | D T₁
instance Arbitrary T₁ where
  arbitrary = sized gen where
    gen 0 = pure A
    gen n = chooseWith
      [(p_A, pure A)
      , (p_B, B ⟨$⟩ gen (n−1) ⟨∗⟩ resize (n−1) arbitrary)]
instance Arbitrary T₂ where
  arbitrary = sized gen where
    gen 0 = pure C
    gen n = chooseWith
      [(p_C, pure C), (p_D, D ⟨$⟩ resize (n−1) arbitrary)]
```

Figure 8: Mutually recursive types T₁ and T₂ and their *DRAGEN* generators.

Doing similar calculations, we obtain the mean matrix M_C for A, B, C, and D as follows:

$$M_C = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \left[\begin{array}{cc:cc} A & B & C & D \\ 0 & 0 & 0 & 0 \\ p_A & p_B & p_C & p_D \\ \hdashline 0 & 0 & 0 & 0 \\ p_A & p_B & 0 & 0 \end{array} \right] \tag{12}$$

We define the mean of the initial generation as $E[G_0] = (p_A, p_B, 0, 0)$—we assign $p_C = p_D = 0$ since we choose to start by generating a value of type T₁. With M_C and $E[G_0]$ in place, we can apply the equations explained through Section 4 to predict the expected number of A, B, C and D constructors.

While this approach works, it completely ignores the types T₁ and T₂ when calculating M_C! For a large set of mutually-recursive data types involving a large number of constructors, handling M_C like this results in a high computational cost. We show next how we cannot only shrink this mean matrix of constructors but also compute it automatically by making use of data type definitions.

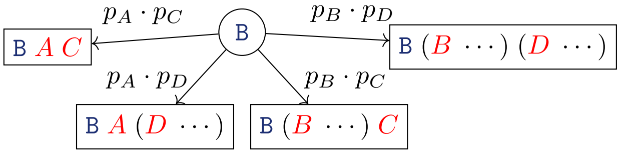


Figure 9: Possible offspring of constructor B.

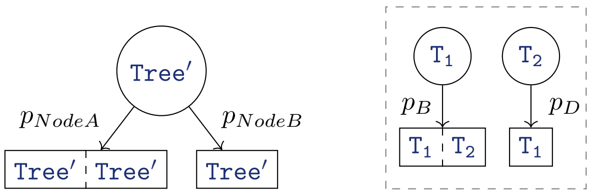


Figure 10: Offspring as types

Mean matrix of types If we analyze the mean matrices of `Tree′` (11) and the mutually-recursive types `T₁` and `T₂` (12), it seems that determining the expected number of offspring generated by a non-terminal constructor requires us to *count the number of occurrences in the ADT to which the offspring belongs to*. For instance, $m_{NodeA,Leaf}$ is $2 \cdot p_{Leaf}$ (10), where 2 is the number of occurrences of `Tree′` in the declaration of `NodeA`. Similarly, m_{BD} is $1 \cdot p_D$, where 1 is the number of occurrences of `T₂` in the declaration of `B`. This observation means that instead of dealing with constructors, we could directly deal with types!

We can think about a branching process as generating "place holders" for constructors, where placeholders can only be populated by constructors of a certain type.

Figure 10 illustrates offspring as types for the definitions `T₁`, `T₂`, and `Tree′`. A place holder of type `T₁` can generate a place holder for type `T₁` and a placeholder for type `T₂`. A placeholder of type `T₂` can generate a placeholder of type `T₁`. A placeholder of type `Tree′` can generate two placeholders of type `Tree′` when generating `NodeA`, one place holder when generating `NodeB`, or zero place holders when generating a `Leaf` (this last case is not shown in the figure since it is void). With these considerations, the mean matrices of types for `Tree′`, written $M_{Tree'}$; and types `T₁` and `T₂`, written $M_{T_1T_2}$ are defined as follows:

$$M_{Tree'} = \begin{array}{c|c} & Tree' \\ \hline Tree' & 2 \cdot p_{NodeA} + p_{NodeB} \end{array} \qquad M_{T_1T_2} = \begin{array}{c|cc} & T1 & T2 \\ \hline T1 & p_B & p_B \\ T2 & p_D & 0 \end{array}$$

Note how $M_{Tree'}$ shows that the mean matrices of types might reduce a multi-type branching process to a simple-type one.

Having the type matrix in place, we can use the following equation (formally stated and proved in the Appendix 1) to soundly predict the expected number of constructors of a given set of (possibly) mutually recursive types:

$$(E[G_n^C]).C_i^t = (E[G_n^T]).T_t \cdot p_{C_i^t} \qquad (\forall n \geq 0)$$

Where G_n^C and G_n^T denotes the nth-generations of constructors and type placeholders respectively. C_i^t represents the ith-constructor of the type T_t. The equation establishes that, the expected number of constructors C_i^t at generation n consists of the expected number of type placeholders of its type (i.e., T_t) at generation n times the probability of generating that constructor. This equation allows us to simplify many of our calculations above by simply using the mean matrix for types instead of the mean matrix for constructors.

6.1 Composite Types

In this subsection, we extend our approach in a *modular* manner to deal with composite ADTs, i.e., ADTs that use already defined types in their constructors' arguments and which are not involved in the branching process. We start by considering the ADT `Tree` modified to carry booleans at the leaves:

$$\textbf{data}\ \texttt{Tree} = \texttt{LeafA Bool} \mid \texttt{LeafB Bool Bool} \mid \cdots$$

Where $\cdots$ denotes the constructors that remain unmodified. To predict the expected number of `True` (and analogously of `False`) constructors, we calculate the multi-type branching process for `Tree` and multiply each expected number of leaves by the number of arguments of type `Bool` present in each one:

$$E[\texttt{True}] = p_{True} \cdot (\underbrace{1 \cdot E[\texttt{LeafA}]}_{\text{case }\texttt{LeafA}} + \underbrace{2 \cdot E[\texttt{LeafB}]}_{\text{case }\texttt{LeafB}})$$

In this case, `Bool` is a ground type like `Int`, `Float`, etc. Predictions become more interesting when considering richer composite types involving, for instance, instantiations of polymorphic types. To illustrate this point, consider a modified version of `Tree` where `LeafA` now carries a value of type `Maybe Bool`:

$$\textbf{data}\ \texttt{Tree} = \texttt{LeafA}\ (\texttt{Maybe Bool}) \mid \texttt{LeafB Bool Bool} \mid \cdots$$

In order to calculate the expected number of `True`s, now we need to consider the cases that a value of type `Maybe Bool` *actually carries a boolean value*, i.e., when a `Just` constructor gets generated:

$$E[\texttt{True}] = p_{True} \cdot (1 \cdot E[\texttt{LeafA}] \cdot p_{Just} + 2 \cdot E[\texttt{LeafB}])$$

In the general case, for constructor arguments utilizing other ADTs, it is necessary to know the chain of constructors required to generate "foreign" values—in our example, a `True` value gets generated if a `LeafA` gets generated with a `Just` constructor "in between." To obtain such information, we create a *constructor dependency graph* (CDG), that is, a directed graph where each node represents a constructor and each edge represents its dependency. Each edge is labeled with its corresponding generation probability. Figure 11 shows the

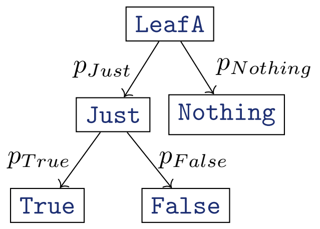


Figure 11: Constructor dependency graph.

CDG for `Tree` starting from the `LeafA` constructor. Having this graph together with the application of the multi-type branching process, we can predict the expected number of constructors belonging to external ADTs. It is enough to multiply the probabilities at each edge of the path between every constructor involved in the branching process and the desired external constructor.

The extensions described so far enable our tool (presented in the next section) to make predictions about *QuickCheck* generators for ADTs defined in many existing Haskell libraries.

7 Implementation

DRAGEN is a tool chain written in Haskell that implements the multi-type branching processes (Section 4 and 5) and its extensions (Section 6) together with a distribution optimizer, which calibrates the probabilities involved in generators to fit developers' demands. *DRAGEN* synthesizes generators by calling the Template Haskell function `dragenArbitrary :: Name → Size → CostFunction → Q [Dec]`, where developers indicate the target ADT for which they want to obtain a *QuickCheck* generator; the desired generation size, needed by our prediction mechanism in order to calculate the distribution at the last generation level; and a *cost function* encoding the desired generation distribution.

The design decision to use a probability optimizer rather than search for an analytical solution is driven by two important aspects of the problem we aim to solve. Firstly, the computational cost of exactly solving a non-linear system of equations (such as those arising from branching processes) can be prohibitively high when dealing with a large number of constructors, thus a large number of unknowns to be solved. Secondly, the existence of such exact solutions is not guaranteed due to the implicit invariants the data types under consideration might have. In such cases, we believe it is much more useful to construct a distribution that approximates the user's goal than to abort the entire compilation process. We give an example of this approximate solution-finding behavior later in this section.

7.1 Cost Functions

The optimization process is guided by a user-provided cost function. In our setting, a cost function assigns a real number (a cost) to the combination of a generation size (chosen by the user) and a mapping from constructors to probabilities:

```
type CostFunction = Size → ProbMap → Double
```

Type `ProbMap` encodes the mapping from constructor names to real numbers. Our optimization algorithm works by generating several probability mapping candidates that are evaluated through the provided cost function in order to choose the most suitable one. Cost functions are expected to return a smaller positive number as the predicted distribution obtained from its parameters

Table 1: Predicted and actual distributions for `Tree` generators using different cost functions.

Cost Function	Predicted Expectation				Observed Expectation			
	`LeafA`	`LeafB`	`LeafC`	`Node`	`LeafA`	`LeafB`	`LeafC`	`Node`
`uniform`	5.26	5.26	5.21	14.73	5.27	5.26	5.21	14.74
`weighted` $[('Leaf_A, 3), ('Leaf_B, 1), ('Leaf_C, 1)]$	30.07	9.76	10.15	48.96	30.06	9.75	10.16	48.98
`weighted` $[('Leaf_A, 1), ('Node, 3)]$	10.07	3.15	17.57	29.80	10.08	3.15	17.58	29.82
`only` $['Leaf_A, 'Node]$	10.41	0	0	9.41	10.43	0	0	9.43
`without` $['Leaf_C]$	6.95	6.95	0	12.91	6.93	6.92	0	12.86

gets closer to a certain *target distribution*, which depends on what property that particular cost function is intended to encode. Then, the optimizer simply finds the best `ProbMap` by minimizing the provided cost function.

Currently, our tool provides a basic set of cost functions to easily describe the expected distribution of the derived generator. For instance, `uniform` :: `CostFunction` encodes constructor-wise uniform generation, an interesting property that naturally arises from our generation process formalization. It guides the optimization process to a generation distribution that minimizes the difference between the expected number of each generated constructor and the generation size. Moreover, the user can restrict the generation distribution to a certain subset of constructors using the cost functions `only` :: [`Name`] → `CostFunction` and `without` :: [`Name`] → `CostFunction` to describe these restrictions. In this case, the whitelisted constructors are then generated following the `uniform` behavior. Similarly, if the branching process involves mutually recursive data types, the user could restrict the generation to a certain subset of data types by using the functions `onlyTypes` and `withoutTypes`. Additionally, when the user wants to generate constructors according to certain proportions, `weighted` :: [(`Name`, `Int`)] → `CostFunction` allows to encode this property, e.g. three times more `LeafA`'s than `LeafB`'s.

Table 1 shows the number of expected and observed constructors of different `Tree` generators obtained by using different cost functions. The observed expectations were calculated averaging the number of constructors across 100000 generated values. Firstly, note how the generated distributions are soundly predicted by our tool. In our tests, the small differences between predictions and actual values disappear as we increase the number of generated values. As for the cost functions' behavior, there are some interesting aspects to note. For instance, in the `uniform` case the optimizer cannot do anything to break the implicit invariant of the data type: every binary tree with n nodes has $n + 1$ leaves. Instead, it converges to a solution that "approximates" a uniform distribution around the generation size parameter. We believe this is desirable behavior, to find an approximate solution when certain invariants prevent the optimization process from finding an exact solution. This way the user does not have to be aware of the possible invariants that the target data type may have, obtaining a solution that is good enough for most purposes. On the other hand, notice that in the `weighted` case at the second row of Table 1, the expected number of generated `Node`s is considerably large. This constructor is not listed in the proportions list, hence the optimizer can freely adjust its probability to satisfy the proportions specified for the leaves.

7.2 Derivation Process

DRAGEN's derivation process starts at compile-time with a type reification stage that extracts information about the structure of the types under consideration. It follows an intermediate stage composed of the optimizer for probabilities used in generators, which is guided by our multi-type branching process model, parameterized on the cost function provided. This optimizer is based on a standard local-search optimization algorithm that recursively chooses the best mapping from constructors to probabilities in the current neighborhood. Neighbors are `ProbMap`s, determined by individually varying the probabilities for *each constructor* with a predetermined Δ. Then, to determine the "best" probabilities, the local search applies our prediction mechanism to the immediate neighbors that have not yet been visited by evaluating the cost function to select the most suitable next candidate. This process continues until a local minimum is reached when there are no new neighbors to evaluate, or if each step improvement is lower than a minimum predetermined ε.

The final stage synthesizes a `Arbitrary` type-class instance for the target data types using the optimized generation probabilities. For this stage, we extend some functionality present in *MegaDeTH* in order to derive generators parameterized by our previously optimized probabilities. Refer to Appendix 2.4 for further details on the cost functions and algorithms addressed in this section.

8 Case Studies

We start by comparing the generators for the ADT `Tree` derived by *MegaDeTH* and *Feat*, presented in Section 2, with the corresponding generator derived by *DRAGEN* using a `uniform` cost function. We used a generation size of 10 both for *MegaDeTH* and *DRAGEN*, and a generation size of 400 for *Feat*—that is, *Feat* will generate test cases of maximum 400 constructors, since this is the maximum number of constructors generated by our tool using the generation size cited above. Figure 12 shows the differences between the complexity of the generated values in terms of the number of constructors. As shown in Figure 3, generators derived by *MegaDeTH* and *Feat* produce very narrow distributions, being unable to generate a diverse variety of values of different sizes. In contrast, the *DRAGEN* optimized generator provides a much wider distribution, i.e., from smaller to bigger values.

It is likely that the richer the values generated, the better the chances of covering more code, and thus of finding more bugs. The next case studies provide evidence in that direction.

Although *DRAGEN* can be used to test Haskell code, we follow the same philosophy as *QuickFuzz*, targeting three complex and widely used external programs to evaluate how well our derived generators behave. These applications are *GNU bash 4.4*—a widely used Unix shell, *GNU CLISP 2.49*—the GNU

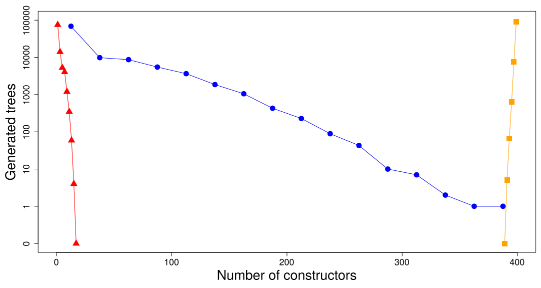


Figure 12: *MegaDeTH* (▲) vs. *Feat* (■) vs. *DRAGEN* (●) generated distributions for type `Tree`.

Common Lisp compiler, and *giffix*—a small test utility from the *GIFLIB 5.1* library focused on reading and writing Gif images. It is worth noticing that these applications are not written in Haskell. Nevertheless, there are Haskell libraries designed to inter-operate with them: *language-bash*, *atto-lisp*, and *JuicyPixels*, respectively. These libraries provide ADT definitions which we used to synthesize *DRAGEN* generators for the inputs of the aforementioned applications. Moreover, they also come with serialization functions that allow us to transform the randomly generated Haskell values into the actual test files that we used to test each external program. The case studies contain mutually recursive and composite ADTs with a wide number of constructors (e.g., GNU bash spans 31 different ADTs and 136 different constructors)—refer to 2.5 for a rough estimation of the scale of such data types and the data types involved with them.

For our experiments, we use the coverage measure known as *execution path* employed by American Fuzzy Lop (AFL) [3]—a well known fuzzer. It was chosen in this work since it is also used in the work by Grieco et al. [71] to compare *MegaDeTH* with other techniques. The process consists of the *instrumentation* of the binaries under test, making them able to return the path in the code taken by each execution. Then, we use AFL to count how many different executions are triggered by a set of randomly generated files—also known as a corpus. In this evaluation, we compare how different *QuickCheck* generators, derived using *MegaDeTH* and using our approach, result in different code coverage when testing external programs, as a function of the size of a set of independently, randomly generated corpora. We have not been able to automatically derive such generators using *Feat*, since it does not work with some Haskell extensions used in the bridging libraries.

We generated each corpus using the same ADTs and generation sizes for each derivation mechanism. We used a generation size of 10 for CLISP and bash files, and a size of 5 for Gif files. For *DRAGEN*, we used `uniform` cost functions to reduce any external bias. In this manner, any observed difference in the code coverage triggered by the corpora generated using each derivation mechanism is entirely caused by the optimization stage that our predictive approach performs, which does not represent an extra effort for the programmer.

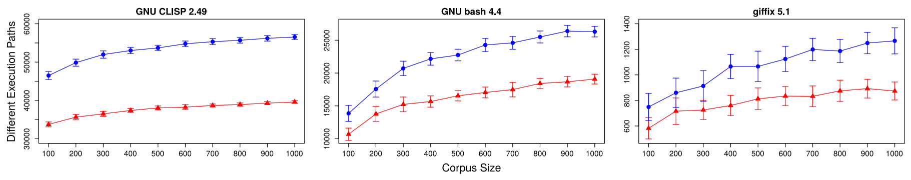


Figure 13: Path coverage comparison between *MegaDeTH* (▲) and *DRAGEN* (●).

Moreover, we repeat each experiment 30 times using independently generated corpora for each combination of derivation mechanism and corpus size.

Figure 13 compares the mean number of different execution paths triggered by each pair of generators and corpus sizes, with error bars indicating 95% confidence intervals of the mean.

It is easy to see how the *DRAGEN* generators synthesize test cases capable of triggering a much larger number of different execution paths in comparison to *MegaDeTH* ones. Our results indicate average increases approximately between 35% and 41% with a standard error close to 0.35% in the number of different execution paths triggered in the programs under test.

An attentive reader might remember that *MegaDeTH* tends to derive generators which produce very small test cases. If we consider that small test cases should take less time (on average) to be tested, is fair to think there is a trade-off between being able to test a bigger number of smaller test cases or a smaller number of bigger ones having the same time available. However, when testing external software like in our experiments, it is important to consider the time overhead introduced by the operating system. In this scenario, it is much more preferable to test interesting values over smaller ones. In our tests, size differences between the generated values of each tool do not result in significant differences in the runtimes required to test each corpora—refer to Appendix 2.5. A user is most likely to get better results by using our tool instead of *MegaDeTH*, with virtually *the same effort.*

We also remark that, if we run sufficiently many tests, then the expected code coverage will tend towards 100% of the reachable code in both cases. However, in practice, our approach is more likely to achieve higher code coverage for the same number of test cases.

9 Related Work

Fuzzers are tools to test programs against randomly generated unexpected inputs. *QuickFuzz* [70], [71] is a tool that synthesizes data with rich structure, that is, well-typed files which can be used as initial "seeds" for state-of-the-art fuzzers—a workflow which discovered many unknown vulnerabilities. Our work could help to improve the variation of the generated initial seeds, by varying the distribution of *QuickFuzz* generators—an interesting direction for future work.

SmallCheck [68] provides a framework to exhaustively test data sets up to a certain (small) size. The authors also propose a variation called *Lazy*

SmallCheck, which avoids the generation of multiple variants which are passed to the test, but not actually used.

QuickCheck has been used to generate well-typed lambda terms in order to test compilers [65]. Recently, Midtgaard et al. extend such a technique to test compilers for impure programming languages [17].

Luck [72] is a domain-specific language for writing testing properties and *QuickCheck* generators at the same time. We see *Luck*'s approach as orthogonal to ours, which is mostly intended to be used when we do not know any specific property of the system under test, although we consider that borrowing some functionality from *Luck* into *DRAGEN* is an interesting path for future work.

Recently, Lampropoulos et al. propose a framework to automatically derive random generators for a large subclass of Coqs' inductively defined relations [75]. This derivation process also provides proof terms certifying that each derived generator is sound and complete with respect to the inductive relation it was derived from.

Boltzmann models [76] are a general approach to randomly generating combinatorial structures such as trees and graphs—also extended to work with closed simply-typed lambda terms [77]. By implementing a *Boltzmann sampler*, it is possible to obtain a random generator built around such models which uniformly generates values of a target size with a certain size tolerance. However, this approach has practical limitations. Firstly, the framework is not expressive enough to represent complex constrained data structures, e.g. red-black trees. Secondly, Boltzmann samplers give the user no control over the distribution of generated values besides ensuring size-uniform generation. They work well in theory but further work is required to apply them to complex structures [78]. Conversely, *DRAGEN* provides a simple mechanism to predict and tune the overall distribution of constructors *analytically at compile-time*, using statically known type information, and requiring no runtime reinforcements to ensure the predicted distributions. Future work will explore the connections between branching processes and Boltzmann models.

Similarly to our work, Feldt et al. propose *GödelTest* [79], a search-based framework for generating biased data. It relies on non-determinism to generate a wide range of data structures, along with metaheuristic search to optimize the parameters governing the desired biases in the generated data. Rather than using metaheuristic search, our approach employs a completely analytical process to predict the generation distribution at each optimization step. A strength of the *GödelTest* approach is that it can optimize the probability parameters even when there is no specific target distribution over the constructors—this allows exploiting software behavior under test to guide the parameter optimization.

The efficiency of random testing is improved if the generated inputs are evenly spread across the input domain [80]. This is the main idea of *Adaptive Random Testing* (ART) [81]. However, this work only covers the particular case of testing programs with numerical inputs and it has also been argued that adaptive random testing has inherent inefficiencies compared to random testing [82]. This strategy is later extended in [83] for object-oriented programs. These approaches present no analysis of the distribution obtained by the heuristics used, therefore we see them as orthogonal work to ours.

10 Final Remarks

We discover an interplay between the stochastic theory of branching processes and algebraic data types structures. This connection enables us to describe a solid mathematical foundation to capture the behavior of our derived *QuickCheck* generators. Based on our formulas, we implement a heuristic to automatically adjust the expected number of constructors being generated as a way to control generation distributions.

One holy grail in testing is the generation of structured data which fulfills certain invariants. We believe that our work could be used to enforce some invariants on data "up to some degree." For instance, by inspecting programs' source code, we could extract the pattern-matching patterns from programs (e.g., (`Cons` (`Cons` `x`))) and derive generators which ensure that such patterns get exercised a certain amount of times (on average)—intriguing thoughts to drive our future work.

Appendix

1 Demonstrations

In this appendix, we provide the formal development to show that the mean matrix of types can be used to soundly predict the distribution of constructors.

We start by defining some terminology. First, let T_t be a data type defined as a sum of type constructors:

$$T_t := C^t_1 + C^t_2 + \cdots + C^t_n$$

Where each constructor is defined as a product of data types:

$$C^t_c := T_1 \times T_2 \times \cdots \times T_m$$

We will define the following observation functions:

$$\begin{aligned} \mathit{cons}(T_t) &= \{C^t_c\}^n_{c=1} \\ \mathit{args}(C^t_c) &= \{T_j\}^m_{j=1} \\ |T_t| &= |\mathit{cons}(T_t)| = n \end{aligned}$$

We will also define the *branching factor from C^u_i to T_v* as the natural number $\beta(T_v, C^u_i)$ denoting the number of occurrences of T_v in the arguments of C^u_i:

$$\beta(T_v, C^u_i) = |\{T_k \in \mathit{args}(C^u_i) \mid T_k = T_v\}|$$

Before showing our main theorem, we need some preliminary propositions. The following one relates the mean of reproduction of constructors with their types and the number of occurrences in the ADT declaration.

Theorem 1. *Let M_C be the mean matrix for constructors for a given, possibly mutually recursive data types $\{T_t\}^n_{t=1}$ and type constructors $\{C^t_i\}^{|T_t|}_{i=1}$. Assuming $p_{C^t_i}$ to be the probability of generating a constructor $C^t_i \in \mathit{cons}(T_t)$ whenever a value of type T^t is needed, then it holds that:*

$$m_{C^u_i C^v_j} = \beta(T_v, C^u_i) \cdot p_{C^v_j} \tag{13}$$

Proof. Let $m_{C^u_i C^v_j}$ be an element of M_C, we know that $m_{C^u_i C^v_j}$ represents the expected number of constructors $C^v_j \in \mathit{cons}(T_v)$ generated whenever a constructor $C_i^u \in cons(T_u)$ is generated. Since every constructor is composed of a product of (possibly) many arguments, we need to sum the expected number of constructors C_j^v generated by each argument of C_i^u of type T_v—the expected number of constructors C_j^v generated by an argument of a type different than T_v is null. For this, we define the random variable $X_k^{C_i^u C_j^v}$ capturing the number of constructors C_j^v generated by the k-th argument of C_i^u as follows:

$$X_k^{C_i^u C_j^v} : cons(T_v) \to \mathbb{N}$$
$$X_k^{C_i^u C_j^v}(C_c^v) = \begin{cases} 1 & \text{if } c = j \\ 0 & \text{otherwise} \end{cases}$$

We can calculate the probabilities of generating zero or one constructors C_j^v by the k-th argument of C_i^u as follows:

$$P(X_k^{C_i^u C_j^v} = 0) = 1 - p_{C_j^v}$$
$$P(X_k^{C_i^u C_j^v} = 1) = p_{C_j^v}$$

Then, we can calculate the expectancy of each $X_k^{C_i^u C_j^v}$:

$$E[X_k^{C_i^u C_j^v}] \;=\; 1 \cdot P(X_k^{C_i^u C_j^v} = 1) + 0 \cdot P(X_k^{C_i^u C_j^v} = 0) \;=\; p_{C_j^v} \tag{14}$$

Finally, we can calculate the expected number of constructors C_j^v generated whenever we generate a constructor C_i^u by adding the expected number of C_j^v generated by each argument of C_i^u of type T_v:

$$\begin{aligned}
m_{C_i^u C_j^v} &= \sum_{\{T_k \in args(C_i^u) \;|\; T_k = T_v\}} E[X_k^{C_i^u C_j^v}] \\
&= \sum_{\{T_k \in args(C_i^u) \;|\; T_k = T_v\}} p_{C_j^v} && \text{(by (14))} \\
&= p_{C_j^v} \cdot \sum_{\{T_k \in args(C_i^u) \;|\; T_k = T_v\}} 1 && (p_{C_j^v} \text{ is constant}) \\
&= p_{C_j^v} \cdot |\{T_k \in args(C_j^v) \mid T_k = T_v\}| && (\sum_S 1 = |S|) \\
&= p_{C_j^v} \cdot \beta(T_v, C_i^u) && \text{(by def. of } \beta)
\end{aligned}$$

□

The next proposition relates the mean of reproduction of types with their constructors.

Theorem 2. *Let M_T be the mean matrix for types for a given, possibly mutually recursive data types $\{T_t\}_{t=1}^n$ and type constructors $\{C_i^t\}_{i=1}^{|T_t|}$. Assuming $p_{C_i^t}$ to be the probability of generating a constructor $C_i^t \in cons(T_t)$ whenever a value of type T_t is needed, then it holds that:*

$$m_{T_u T_v} = \sum_{C_k^u \in cons(T^u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} \tag{15}$$

Proof. Let $m_{T_u T_v}$ be an element of M_T, we know that $m_{T_u T_v}$ represents the expected number of placeholders of type T_v generated whenever a placeholder of type T_u is generated, i.e. by any of its constructors. Therefore, we need to average the number of placeholders of type T_v appearing on each constructor of T_u. For that, we introduce the random variable Y^{uv} capturing this behavior.

$$\begin{gathered} Y^{uv} : cons(T_u) \to \mathbb{N} \\ Y^{uv}(C_k^u) = \beta(T_v, C_k^u) \end{gathered}$$

And we can obtain $m_{T_u T_v}$ by calculating the expected value of Y^{uv} as follows.

$$\begin{aligned} m_{T_u T_v} &= E[Y^{uv}] \\ &= \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot P(Y^{uv} = C_k^u) && \text{(def. of } E[Y^{uv}]) \\ &= \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} && \text{(def. of } p_{C_k^u}) \end{aligned}$$

□

The next proposition relates one entry in M_T with its corresponding in M_C.

Theorem 3. *Let M_C and M_T be the mean matrices for constructors and types respectively for a given, possibly mutually recursive data types $\{T_t\}_{t=1}^n$ and type constructors $\{C_i^t\}_{i=1}^{|T_t|}$. Assuming $p_{C_i^t}$ to be the probability of generating a type constructor $C_i^t \in cons(T_t)$ whenever a value of type T_t is needed, then it holds that:*

$$p_{C_i^v} \cdot m_{T_u T_v} = \sum_{C_j^u \in cons(T_u)} m_{C_j^u C_i^v} \cdot p_{C_j^u} \tag{16}$$

Proof. Let C_i^u and C_j^v be type constructors of T^u and T^v respectively. Then, by (13) and (15) we have:

$$m_{C_i^u C_j^v} = \beta(T_v, C_i^u) \cdot p_{C_j^v} \tag{17}$$

$$m_{T_u T_v} = \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} \tag{18}$$

Now, we can rewrite (17) as follows:

$$\beta(T_v, C_i^u) = \frac{m_{C_i^u C_j^v}}{p_{C_j^v}} \qquad (\text{if } p_{C_j^v} \neq 0) \tag{19}$$

(In the case that $p_{C_j^v} = 0$, the last equation in this proposition holds trivially by (17).) And by replacing (19) in (18) we obtain:

$$\begin{aligned}
m_{T_u T_v} &= \sum_{C_k^u \in cons(T_u)} \frac{m_{C_i^u C_j^v}}{p_{C_j^v}} \cdot p_{C_k^u} \\
m_{T_u T_v} &= \frac{1}{p_{C_j^v}} \cdot \sum_{C_k^u \in cons(T_u)} m_{C_i^u C_j^v} \cdot p_{C_k^u} \qquad (p_{C_j^v} \text{ constant}) \\
p_{C_j^v} \cdot m_{T_u T_v} &= \sum_{C_k^u \in cons(T_u)} m_{C_i^u C_j^v} \cdot p_{C_k^u}
\end{aligned}$$

□

Now, we proceed to prove our main result.

Theorem 4. *Consider a QuickCheck generator for a (possibly) mutually recursive data types $\{T_t\}_{t=1}^{k}$ and type constructors $\{C_i^t\}_{i=1}^{|T^t|}$. We assume $p_{C_i^t}$ as the probability of generating a type constructor $C_i^t \in cons(T_t)$ when a value of type T_t is needed. We will call T_r $(1 \leq r \leq k)$ to the generation root data type, and M_C and M_T to the mean matrices for the multi-type branching process capturing the generation behavior of type constructors and types respectively. The branching process predicting the expected number of type constructors at level n is governed by the formula:*

$$E[G_n^C]^T = E[G_0^C]^T \cdot \left(\frac{I - (M_C)^{n+1}}{I - M_C} \right)$$

In the same way, the branching process predicting the expected number of type placeholders at level n is given by:

$$E[G_n^T]^T = E[G_0^T]^T \cdot \left(\frac{I - (M_T)^{n+1}}{I - M_T} \right)$$

where G_n^C denotes the constructors' population at the level n, and G_n^T denotes the type placeholders population at the level n. The expected number of constructors C_i^t at the n-th level is given by the expected constructors population at the n-level $E[G_n^C]$ indexed by the corresponding constructor. Similarly, the expected number of placeholders of type T_t at the n-th level is given by the expected types' population at the n-level $E[G_n^T]$ indexed by the corresponding type. The initial constructors population $E[G_0^C]$ is defined as the probability of each constructor if it belongs to the root data type, and zero if it belongs to any other data type:

$$E[G_0^C].C_i^t = \begin{cases} p_{C_i^t} & \text{if } t = r \\ 0 & \text{otherwise} \end{cases}$$

The initial type placeholders population is defined as the almost surely probability for the root type, and zero for any other type:

$$E[G_0^T].T_t = \begin{cases} 1 & \text{if } t = r \\ 0 & \text{otherwise} \end{cases}$$

Finally, it holds that:

$$(E[G_n^C]).C_i^t = (E[G_n^T]).T^t \cdot p_{C_i^t}$$

In other words, the expected number of constructors C_i^t at the n-th level consists of the expected number of placeholders of its type (i.e., T_t) at level n times the probability to generate that constructor.

Proof. By induction on the generation size n.

- **Base case**
  We want to prove $(E[G_0^C]).C_i^t = (E[G_0^T]).T_t \cdot p_{C_i^t}$.
  Let T_t be a data type from the Galton-Watson branching process.

  - If $T_t = T_r$ then by the definitions of the initial type constructors and type placeholders populations we have:

    $$(E[C_0^C]).C_i^t = p_{C_i^t} \qquad\qquad (E[G_0^T]).T_t = 1$$

    And the theorem trivially holds by replacing $(E[G_0^C]).C_i^t$ and $(E[G_0^T]).T_t$ with the previous equations in the goal.

  - If $T_t \neq T_r$ then by the definitions of the initial type constructors and type placeholders populations we have:

    $$(E[G_0^C]).C_i^t = 0 \qquad\qquad (E[G_0^T]).T_t = 0$$

    And once again, the theorem trivially holds by replacing $(E[G_0^C]).C_i^t$ and $(E[G_0^T]).T_t$ with the previous equations in the goal.

- **Inductive case**
  We want to prove $(E[G_n^C]).C_i^t = (E[G_n^T]).T_t \cdot p_{C_i^t}$.
  For simplicity, we will call $\Gamma = \{T_t\}_{t=1}^k$.

$$
\begin{aligned}
&(E[G_n^C]).C_i^t \\
&= E\left[\sum_{T_k\in\Gamma}\left(\sum_{C_j^k\in cons(T_k)} (G_{(n-1)}^C).C_j^k \cdot m_{C_j^k C_i^t}\right)\right] && \text{(by G.W. proc.)} \\
&= \sum_{T_k\in\Gamma} E\left[\sum_{C_j^k\in cons(T_k)} (G_{(n-1)}^C).C_j^k \cdot m_{C_j^k C_i^t}\right] && \text{(by prob.)} \\
&= \sum_{T_k\in\Gamma}\left(\sum_{C_j^k\in cons(T_k)} E[(G_{(n-1)}^C).C_j^k \cdot m_{C_j^k C_i^t}]\right) && \text{(by prob.)} \\
&= \sum_{T_k\in\Gamma}\left(\sum_{C_j^k\in cons(T_k)} E[(G_{(n-1)}^C).C_j^k] \cdot m_{C_j^k C_i^t}\right) && \text{(by prob.)} \\
&= \sum_{T_k\in\Gamma}\left(\sum_{C_j^k\in cons(T_k)} (E[G_{(n-1)}^C]).C_j^k \cdot m_{C_j^k C_i^t}\right) && \text{(by linear alg.)} \\
&= \sum_{T_k\in\Gamma}\left(\sum_{C_j^k\in cons(T_k)} (E[G_{(n-1)}^T]).T_t \cdot p_{C_j^k} \cdot m_{C_j^k C_i^t}\right) && \text{(by I.H.)} \\
&= \sum_{T_k\in\Gamma} (E[G_{(n-1)}^T]).T_t \cdot \sum_{C_j^k\in cons(T_k)} p_{C_j^k} \cdot m_{C_j^k C_i^t} && \text{(by linear alg.)} \\
&= \sum_{T_k\in\Gamma} (E[G_{(n-1)}^T]).T_t \cdot p_{C_i^t} \cdot m_{T_k T_t} && \text{(by (16))} \\
&= \sum_{T_k\in\Gamma} (E[G_{(n-1)}^T]).T_t \cdot m_{T_k T_t} \cdot p_{C_i^t} && \text{(rearrange)} \\
&= \sum_{T_k\in\Gamma} E[(G_{(n-1)}^T).T_t] \cdot m_{T_k T_t} \cdot p_{C_i^k} && \text{(by linear alg.)} \\
&= \sum_{T_k\in\Gamma} E[(G_{(n-1)}^T).T_t \cdot m_{T_k T_t}] \cdot p_{C_i^t} && \text{(by prob.)} \\
&= E\left[\sum_{T_k\in\Gamma} (G_{(n-1)}^T).T_t \cdot m_{T_k T_t}\right] \cdot p_{C_i^t} && \text{(by prob.)} \\
&= (E[G_n^T]).T_t \cdot p_{C_i^t} && \text{(by G.W. proc.)}
\end{aligned}
$$

□

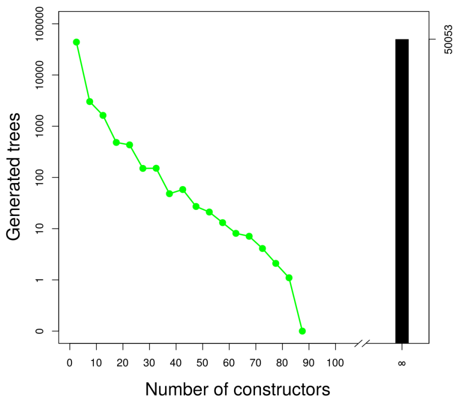


Figure 14: Distribution of (the amount of) `T` constructors induced by *derive*.

2 Additional Information

This appendix is meant to provide further analyses of the aspects presented throughout this work that would not fit into the available space.

2.1 Termination issues with library *derive*

As we have introduced in Section 2, the library *derive* provides an easy alternative to automatically synthesize random generators in compile time. However, in the presence of recursive data types, the generators obtained with this tool lack mechanisms to ensure termination. For instance, consider the following data type definition and its corresponding generator obtained with *derive*:

```
data T = A | B T T | C T T
instance Arbitrary T where
  arbitrary = oneof
    [pure A
    , B ⟨$⟩ arbitrary ⟨∗⟩ arbitrary
    , C ⟨$⟩ arbitrary ⟨∗⟩ arbitrary]
```

When using this generator, *every constructor in the obtained generator has the same probability of being chosen.* Additionally, at each point of the generation process, if we randomly generate a recursive type constructor (either `B` or `C`), then we also need to generate two new `T` values in order to fill the arguments of the chosen type constructor. As a result, it is expected (on average) that each time *QuickCheck* generates a recursive constructor (i.e., `B` or `C`) at one level, *more than one recursive constructor is generated at the next level*—thus, frequently leading to an infinite generation loop.

This behavior can be formalized using the concept known as *probability generating function*, where it is proven that the extinction probability of a generated value d (and thus the termination of the generation) can be calculated by finding the smallest fix point of the generation recurrence. In our example, this is the smallest d such that $d = P_A + (P_B + P_C) \cdot d^2 = (1/3) + (2/3) \cdot d^2$, where P_i denotes the probability of generating a i constructor. In this case $d = 1/2$.

Figure 14 provides an empirical verification of this non-terminating behavior. It shows the distribution (in terms of the number of constructors) of 100000 randomly generated `T` values obtained using the *derive* generator shown above. The black bar on the right represents the number of values that induced an infinite generation loop. Such values were recognized using a sufficiently big timeout. The random generation gets stuck in an infinite generation loop almost exactly half of the time we generate a random `T` value.

In practice, this non-terminating behavior gets worse as we increase either the number of recursive constructors or the number of their recursive arguments in the data type definition, since this increases the probability of choosing a recursive constructor each time we need to generate a subterm.

2.2 Multi-type Branching Processes

We will verify the soundness of the step noted as $(\star)$, used to deduce $E[G_n^j | G_{n-1}]$ in Section 4. In first place, note that $E[G_n^j | G_{n-1}]$ can be rewritten as:

$$E[G_n^j | G_{n-1}] = E\left[\sum_{i=1}^{d} \sum_{p=1}^{G_{n-1}} \xi_{ij}^p\right]$$

Where symbol ξ_{ij}^p denotes the number of offspring of kind j that the parent p of kind i produces. If the parent p has not kind i, then $\xi_{ij}^p = 0$. Essentially, the sums simply iterate on all of the different kinds of parents present in the nth-generation, counting the number of offspring of kind j that they produce. Then, since the expectation of the sum is the sum of expectation, we have that:

$$E[G_n^j | G_{n-1}] = \sum_{i=1}^{d} \sum_{p=1}^{G_{n-1}} E\left[\xi_{ij}^p\right]$$

In the inner sum, some terms are 0 and others are the expected offspring of kind j that a parent of kind i produces. As introduced in Section 4, we capture with random variable R_{ij} the distribution governing that a parent of kind i produces offspring of kind j. Finally, by filtering out all the terms which are 0 in the inner sum, i.e., where $p \neq i$, we obtain the expected result:

$$E[G_n^j | G_{n-1}] = \sum_{i=1}^{d} G_{(n-1)}^i \cdot E[R_{ij}]$$

2.3 Terminal constructors

As we explained in Section 5, our tool synthesizes random generators for which the generation of terminal constructors can be thought of two different random processes. More specifically, the first $(n-1)$ generations of the branching

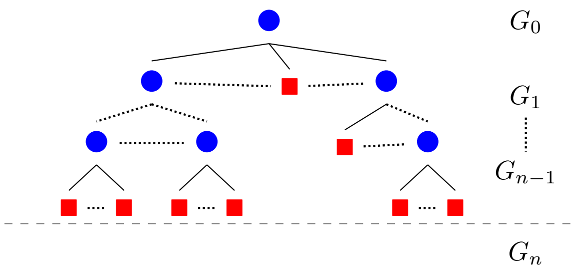


Figure 15: Generation processes of non-terminal (●) and terminal (■) constructors.

process are composed of a mix of non-terminals and terminals constructors. The last level, however, only contains terminal constructors since the size limit has been reached. Figure 15 shows a graphical representation of the overall process.

2.4 Implementation

In this subsection, will give more details on the implementation of our tool. Firstly, Figure 16 shows a schema for the automatic derivation pipeline our tool performs. The user provides a target data type, a cost function and a desired generation size, and our tool returns an optimized random generator. The components marked in red are heavily dependent on Template Haskell and refer to the type introspection and code generation stages of *DRAGEN*, while the intermediate stages (in blue) are composed of our prediction mechanism and the probabilities optimizer.

Cost functions The probabilities optimizer that our tool implements essentially works minimizing a provided cost function that encodes the desired distribution of constructors at the optimized generator. As shown in Section 7, *DRAGEN* comes with a minimal set of useful cost functions. Such functions are built around the *Chi-Square Goodness of Fit Test* [84], a statistical test

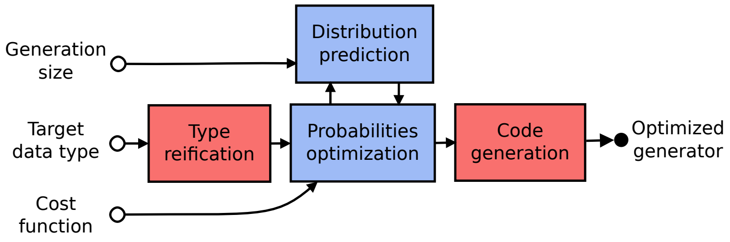


Figure 16: Generation schema.

used to quantify how the observed value of a given phenomenon is significantly different from its expected value:

$$\chi^2 = \sum_{C_i \in \Gamma} \frac{(observed_i - expected_i)^2}{expected_i}$$

Where Γ is a subset of the constructors involved in the generation process; $observed_i$ corresponds to the predicted number of generated C_i constructors; and $expected_i$ corresponds to the amount of constructors C_i desired in the distribution of the optimized generator. This fitness test was chosen for empirical reasons, since it provides better results in practice when finding probabilities that ensure certain distributions.

In this appendix, we will take special attention to the `weighted` cost function, since it is the most general one that our tool provides—the remaining cost functions provided could be expressed in terms of `weighted`. This function uses our previously discussed prediction mechanism to obtain a prediction of the constructors' distribution under the current given probabilities and the generation size (see `obs`), and uses it to calculate the Chi-Square Goodness of Fit Test. A simplified implementation of this cost function is as follows.

```
weighted :: [(Name, Double)] → CostFunction
weighted weights size probs = chiSquare obs exp
  where
    chiSquare = sum ∘ zipWith (λo e → (o − e) squared / e)
    obs = predict size probs
    exp = map weight (Map.keys probs)
    weight con = case lookup con weights of
      Just w → w ∗ size
      Nothing → 0
```

Note how we multiply each weight by the generation size provided by the user (case `Just w`), as a simple way to control the relative size of the generated values. Moreover, the generation probabilities for the constructors not listed in the proportions list do not contribute to the cost (case `Nothing`), and thus they can be freely adjusted by the optimizer to fit the proportions of the listed constructors. In this light, the `uniform` cost function can be seen as a special case of `weighted`, where every constructor is listed with weight 1.

Optimization algorithm As introduced in Section 7, our tool makes use of an optimization mechanism in order to obtain a suitable generation probabilities assignment for its derived generators. Figure 17 illustrates a simplified implementation of our optimization algorithm. This optimizer works by selecting recursively the most suitable neighbor, i.e., a probability assignment that is close to the current one and that minimizes the output of the provided cost function. This process is repeated until a local minimum is found when there are no further neighbors left to visit; or if the step improvement is below a minimum predetermined ε.

```
optimize :: CostFunction → Size → ProbMap → ProbMap
optimize cost size init = localSearch init [ ]
  where
    localSearch focus visited
      | null new   = focus
      | gain ⩽ ε   = focus
      | otherwise = localSearch best frontier
     where
       best = minimumBy (comparing (cost size)) new
       new = neighbors focus \\ (focus : visited)
       frontier = new ++ visited
       gain = cost size focus − cost size best
```

Figure 17: Optimization algorithm.

```
neighbors :: ProbMap → [ProbMap]
neighbors probs = concatMap perturb (Map.keys probs)
  where
    perturb con = [norm (adj             (+Δ)  con)
                  , norm (adj (max 0 ∘ (−Δ)) con)]
    norm m = fmap (/sum (Map.elems m)) m
    adj f con = Map.adjust f con probs
```

Figure 18: Immediate neighbors of a probability distribution.

In our setting, neighbors are obtained by taking the current probability distribution, and constructing a list of paired probability distributions, where each one is constructed from the current distribution, adjusting each constructor probability by $\pm\Delta$. This behavior is shown in Figure 18. Note the need of bound checking and normalization of the new neighbors in order to enforce a probability distribution (`max 0` and `norm`). Each pair of neighbors is then joined together and returned as the current probability distribution immediate neighborhood.

2.5 Case studies

As explained in Section 8, our test cases targeted three complex programs to evaluate the power of our derivation tool, i.e. *GNU CLISP 2.49*, *GNU bash 4.4* and *GIFLIB 5.1*. We derived random generators for each test case input format using some existent Haskell libraries. Each one of these libraries contains data types definition encoding the structure of the input format of its corresponding test case, as well as serialization functions that we use to convert randomly generated Haskell values into actual test input files. Table 2 illustrates the complexity of the bridging libraries used in our case studies.

Table 2: Type information for ADTs used in the case studies.

Case Study	#Types	#Constructors	Composite types	Mut. Rec. types
Lisp	7	14	Yes	Yes
Bash	31	136	Yes	Yes
Gif	16	30	Yes	No

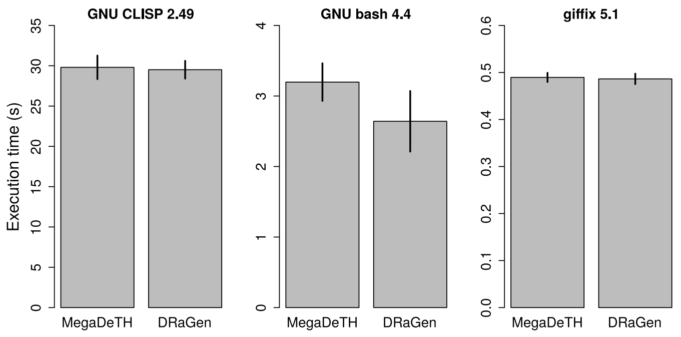


Figure 19: Execution time required to test the biggest randomly generated corpora consisting of 1000 files.

Testing runtimes As we have shown, *MegaDeTH* tends to derive generators that produce very small test cases. However, in our tests, the size differences in the test cases generated by each tool do not produce remarkable differences in the runtimes required to test each corpora. Figure 19 shows the execution time required to test each case of the biggest corpora previously generated by each tool consisting of 1000 test cases.

Paper III

Generating Random Structurally Rich Algebraic Data Type Values

Agustín Mista and Alejandro Russo

14th IEEE/ACM International Workshop on Automation of Software Test (2019)

Abstract

Automatic generation of random values described by algebraic data types (ADTs) is often a hard task. State-of-the-art random testing tools can automatically synthesize random data generators based on ADTs definitions. In that manner, generated values comply with the structure described by ADTs, something that proves useful when testing software that expects complex inputs. However, it sometimes becomes necessary to generate structurally richer ADTs values in order to test deeper software layers. In this work, we propose to leverage static information found in the codebase as a manner to improve the generation process. Namely, our generators are capable of considering how programs branch on input data as well as how ADTs values are built via interfaces. We implement a tool, responsible for synthesizing generators for ADTs values while providing compile-time guarantees about their distributions. Using compile-time predictions, we provide a heuristic that tries to adjust the distribution of generators to what developers might want. We report on preliminary experiments where our approach shows encouraging results.

1 Introduction

Random testing is a promising approach for finding bugs [19], [20], [67]. *QuickCheck* [16] is the dominant tool of this sort used by the Haskell community. It requires developers to specify (i) *testing properties* describing programs' expected behavior and (ii) *random data generators* based on the *types* of the expected inputs (e.g., integers, strings, etc.). *QuickCheck* then generates random test cases and reports violating testing properties.

QuickCheck comes equipped with random generators for built-in types, while it requires to manually write generators for user-defined ADTs. Recently, there has been a proliferation of tools to automatically derive *QuickCheck* generators for ADTs [29], [68], [69], [71], [85]. The main difference between these tools lies in the guarantees provided to ensure *the termination of the generation process* and the *distribution of random values.* Despite their differences, these tools guarantee that generated values are *well-typed.* In other words, generated values follow the structure described by ADT definitions.

Well-typed ADT values are especially useful when testing programs which expect highly structured inputs like compilers [17], [65], [86]. Generating ADT values also proves fruitful when looking for vulnerabilities in combination with fuzzers [70], [71]. Despite these success stories, ADT type-definitions do not often capture all the invariants expected from the data that they are intended to model. As a result, even if random values are well-typed, they might not be built with enough structure to penetrate deep software layers.

In this work, we identify two different sources of structural information that can be statically exploited to improve the generation process of ADT values (Section 3). Then, we show how to capture this information into our (automatically) derived random generators. More specifically, we propose a generation process that is capable of considering how programs branch on input ADTs values as well as how they get manipulated by abstract interfaces (Section 4). Furthermore, we show how to predict the *expected* distribution of the ADT constructors, values fitting certain branching patterns, and calls to interfaces that our random generators produce. For that, we extend some recent results on applying *branching processes*[73]—a simple stochastic model conceived to study population growth (Section 5). We implement our ideas as an extension of the already existing derivation tool called *DRAGEN* [85]. We call our extension as *DRAGEN2*[7] to make it easy the distinction for the reader. *DRAGEN2* is capable of automatically synthesizing *QuickCheck* generators which produce *rich ADT values*, where the distributions of random values can be adjusted at compile-time to what developers might want. Finally, we provide empirical evaluations showing that including static information from the user codebase improves the code coverage of two external applications when tested using random values generated following our ideas (Section 6).

We remark that, although this work focuses on Haskell algebraic data types, this technique is general enough to be applied to most programming languages.

[7] *DRAGEN2* is available at `http://github.com/OctopiChalmers/dragen2`

2 Background

In this section, we briefly introduce the common approach for automatically deriving random data generators for ADTs in QuickCheck. To exemplify this, and for illustrative purposes, let us consider the following ADT definition to encode simple `Html` pages:

```
data Html =
    Text   String
  | Single String
  | Tag    String Html
  | Join   Html   Html
```

The type `Html` allows to build pages via four possible constructions: `Text`—which represents plain text values—, `Single` and `Tag`—which represent singular and paired HTML tags, respectively—, and `Join`—which concatenates two HTML pages one after another. In Haskell, `Text`, `Single`, `Tag`, and `Join` 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 `Join` 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* if it has at least one field of such nature. With this representation, the example page `<html>hello<hr>bye</html>` can be encoded as:

```
Tag "html" (Join (Text "hello")
                 (Join (Single "hr")
                       (Text "bye")))
```

2.1 Type-driven generation of random values

In order to generate random ADTs values, most approaches require users to provide a random data generator for each ADT definition. This is a cumbersome and error-prone task that usually *follows closely the structure of the ADTs*. For instance, consider the following definition of a *QuickCheck* random generator for the type `Html`:

```
genHtml = sized (λsize →
  if size == 0
  then frequency
  [(2, Text   ⟨$⟩ genString)
  ,(1, Single ⟨$⟩ genString)]
  else frequency
  [(2, Text   ⟨$⟩ genString)
  ,(1, Single ⟨$⟩ genString)
  ,(4, Tag    ⟨$⟩ genString ⟨∗⟩ smaller genHtml)
  ,(3, Join   ⟨$⟩ smaller genHtml ⟨∗⟩ smaller genHtml)])
```

We use the Haskell syntax [ ] and (,) for denoting lists and pairs of elements, respectively (e.g., [(1, 2), (3, 4)] is a list of pairs of numbers.) The random generator `genHtml` is defined using *QuickCheck*'s function `sized` to parameterize the generation process up to an external natural number known as the *generation size*—captured in the code with variable `size`. This parameter is chosen by the user, and it is used to limit the maximum amount of recursive calls that this random generator can perform and thus ensuring the termination of the generation process. When called with a positive generation size, this generator can pick to generate among any `Html` data constructor with an explicitly *generation frequency* that can be chosen by the user—in this example, 2, 1, 4 and 3 for `Text`, `Single`, `Tag`, and `Join`, respectively. When it picks to generate a `Text` or a `Single` data constructor, it also generates a random `String` value using the standard *QuickCheck* generator `genString`.[8] On the other hand, when it picks to generate a `Join` constructor, it also generates two independent random sub-expressions recursively, decreasing the generation size by a unit on each recursive invocation (`smaller genHtml`). The case of random generation of `Tag` constructors follows analogously. This random process keeps calling itself recursively until the generation size reaches zero, where the generator is constrained to pick among terminal data constructors, being `Text` and `Single` the only possible choices in our particular case.

The previous definition is rather mechanical, except perhaps for the chosen generation frequencies. *DRAGEN* [85] is a tool conceived to mitigate the problem of finding the appropriate generation frequencies. It uses the theory of branching processes [73] to model and predict analytically the expected number of generated data constructors. This prediction mechanism is used to feedback a simulation-based optimization process that adjusts the generation frequency of each data constructor in order to obtain a particular distribution of values that can be specified by the user—thus providing a flexible testing environment while still being mostly automated.

As many other tools for automatic derivation of generators (e.g.,[29], [68]–[70]), *DRAGEN* synthesizes random generators similar to the one shown before, where the generation process is limited to pick *a single data constructor at the time and then recursively generate each required sub-expression independently*. In practice, this procedure is often too generic to generate random data with enough structural complexity required for testing certain applications.

3 Sources of Structural Information

In this section, we describe the motivation for considering two additional sources of structural information which lead us to obtain better random data generators. We proceed to exemplify the need to consider such sources with examples.

[8]The operators ⟨\$⟩ and ⟨∗⟩ are used in Haskell to combine values obtained from calling random generators and they are not particularly relevant to the point being made in this work.

3.1 Branching on input data

To exemplify the first source of structural information, consider that we want to use randomly generated Html values to test a function simplify :: Html → Html. In Haskell, the notation f :: T means that program f has type T. In our example, function simplify takes an Html as input and produces an Html value as an output—thus its type Html → Html. Intuitively, the purpose of this function is to assemble sequences of Text constructors into a single big one. More specifically, the code of simplify is as follows:

```
simplify :: Html → Html
simplify (Join (Text t₁) (Text t₂)) =
  Text (concat t₁ t₂)
simplify (Join (Join (Text t₁) x) y) =
  simplify (Join (Text t₁) (simplify (Join x y)))
simplify (Join x y) =
  Join (simplify x) (simplify y)
simplify (Tag t x) =
  Tag t (simplify x)
simplify x = x
```

Function concat just concatenates two strings. The body of simplify is described using *pattern matching* over possible kinds of Html values. Pattern matching allows defining functions idiomatically by defining different function clauses for each input pattern we are interested in. In other words, pattern matching is a mechanism that functions have to branch on input arguments. In the code above, we can see that simplify patterns match against sequences of Text constructors combined by a Join constructor—see first and second clauses. Generally speaking, patterns can be defined to match specific constructors, literal values or variable sub-expressions (like x in the last clause of simplify). Patterns can also be nested in order to match very specific values.

Ideally, we would like to put approximately the same amount of effort into testing each clause of the function simplify. However, each data constructor is generated independently by those generators automatically derived by just considering ADT definitions. Observe that the probability of generating a value satisfying a nested pattern (like Join (Text t_1) (Text t_2)) decreases multiplicatively with the number of constructors we simultaneously pattern against. As evidence of that, in our tests, we found at the first two clauses of simplify get exercised only approximately between 1.5% and 6% of the time when using the state-of-the-art tools for automatically deriving *QuickCheck* generators *MegaDeTH* [70] and *DRAGEN* [85]. Most of the generated values were exercising the simplest clauses of our function, i.e, simplify (Join x y), simplify (Tag t x), and simplify x.

Although the previous example might seem rather simple, branching against specific patterns of the input data is not an uncommon task. In that light, and in order to obtain interesting test cases, it is desirable to conceive generators able to produce random values capable of exercising patterns with certain frequency—Section 4 shows how to do so.

```
hr :: Html
hr = Single "hr"

div :: Html → Html
div x = Tag "div" x

bold :: Html → Html
bold x = Tag "b" x
```

Figure 1: Abstract interface of the type `Html`.

3.2 Abstract interfaces

A common choice when implementing ADTs is to transfer the responsibility of preserving structural invariants to the interfaces that manipulate values of such types. To illustrate this point, let us consider three new primitives responsible to handle `Html` data as shown in Figure 1. These functions encode additional information about the structure of `Html` values in the form of specific HTML tags. Primitive `hr` represents the tag `<hr>` used to separate content in an HTML page. Function `div` and `bold` place an `Html` value within the tags `div` and `b` in order to introduce divisions and activate bold fonts, respectively. For instance, the page `<html><b>hello</b><hr>bye</html>` can be encoded as:

```
Tag "html" (Join (bold (Text "hello"))
                 (Join hr (Text "bye")))
```

Observe that, instead of including a new data constructor for each possible HTML tag in the `Html` definition (recall Section 2), we defined a minimal general representation with a set of high-level primitives to build valid `Html` tags. This programming pattern is often found in a variety of Haskell libraries. As a consequence of this practice, generators derived by only looking into ADT definitions often fail to synthesize useful random values, e.g., random HTML pages with valid tags. After all, most of the *valid structure* of values has been encoded into the primitives of the ADT abstract interface. When considering the generator described in Section 2, the chances of generating a `Tag` value representing a commonly used HTML tag such as `div` or `b` are extremely low.

So far, we have introduced two scenarios where derivation approaches based only on ADT definitions are unable to capture all the available structural information from the user codebase. Fortunately, this information can be automatically exploited and used to generate interesting and more structured random values. The next section introduces a model capable of encoding structural information presented in this section into our automatically derived random generators in a modular and flexible way.

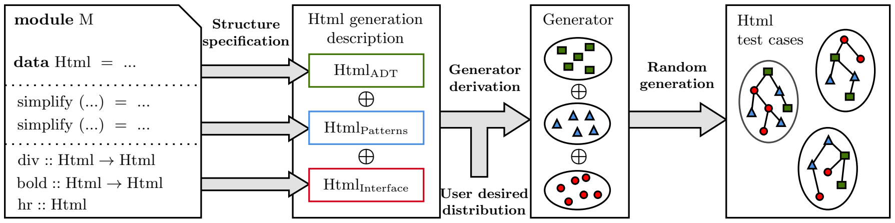


Figure 2: Deriving a generator for the ADT `Html` with the structural information found in module `M`.

4 Capturing ADTs Structure

In this section, we show how to augment the automatic process of deriving random data generators with the structural information expressed by pattern matchings and abstract interfaces. The key idea of this work is to represent the different sources in an homogeneous way.

Figure 2 shows the workflow of our approach for the `Html` ADT. Based on the codebase, the user of *DRAGEN2* specifies: (i) the ADT definition to consider (noted as Html_{ADT}), (ii) its patterns of interest (noted $\text{Html}_{\text{Patterns}}$), and (iii) the primitives from abstract interfaces to involve in the generation process (noted as $\text{Html}_{\text{Interface}}$). Our tool then *automatically derives generators* for each source of structural information. These generators produce random *partial ADT values* in a way that it is easier to combine them in order to create structurally richer ones. For instance, the generator obtained from Html_{ADT} only generates constructors of the ADT but leaves the generation at the recursive fields incomplete, e.g., it generates values of the form (`Text "xA2sx"`), (`Single "xj32da"`), (`Tag "divx234jx"` •) and (`Join` • •), where • is a placeholder denoting a "yet-to-complete" value. Similarly, the generator obtained from $\text{Html}_{\text{Patterns}}$ generates values satisfying the expected patterns where recursive fields are also left uncompleted, e.g., it generates values of the form (`Join (Text "xxa34") (Text "yxa123")`) and (`Join (Join (Text "xd32sa")` •) •). Finally, the generator derived from $\text{Html}_{\text{Interface}}$ generates calls to the interface's primitives, where each argument of type `Html` is left uncompleted, e.g., (`div` •) and (`bold` •).

Observe that *partial ADT values* can be combined easily and the result is still a well-formed value of type `Html`. For instance, if we want to combine the following random generated ADT value (`Text "xx34s"`), pattern (`Join (Join (Text "xd32sa")` •), and interface call (`div` •), we can obtain the following well-typed `Html` value:

```
Join (Join (Text "xd32sa") (div (Text "xx34s"))
```

Finally, our tool puts all these three generators together into one that combines partial ADT values into fully formed ones. Importantly, the user can specify the desired distribution of the expected number of constructors, pattern matching values, and interface calls that the generator will produce.

All in all, our approach offers the following advantages over usual derivation of random generators based only on ADT definitions:

- **Composability:** our tool can combine different partial ADT values arising from different structural information sources depending on what property or sub-system becomes necessary to test using randomly generated values.

- **Extensibility:** the developer can specify new sources of structural information and combine them with the existing ones simply by adding them to the existing specification of the target ADT.

- **Predictability:** the tool is capable of synthesizing generators with adjustable distributions based on developers' demands. For instance, a uniform distribution of pattern matching values, or a distribution where some constructors are generated twice as often as others. We explain the prediction of distributions in the next section.

We remark that, for space reasons, we were only able to introduce the specification of a rather simple target ADT like `Html`. In practice, this reasoning can be extended to mutually recursive and parametric ADT definitions as well.

5 Predicting Distributions

Characterizing the distribution of values of an arbitrary random generator is a hard task. It requires modeling every random choice that a generator could possibly make to generate a value. In a recent work [85], we have shown that it is possible to *analytically* predict the average distribution of data constructors produced by random generators automatically derived considering only ADT definitions—like the one presented in Section 2. For this purpose, we found that random generation of ADT values can be characterized using the theory of *branching processes* [73]. This probabilistic theory was originally conceived to predict the growth and extinction of royal family trees in the Victorian Era, later being applied to a wide variety of research areas. In this work, we adapt this model to predict the average distribution of values of random generators derived considering structural information coming from functions' pattern matchings and abstract interfaces.

Essentially, a branching process is a special kind of Markov process that models the evolution of a population of *individuals of different kinds* across discrete time steps known as *generations*. Each kind of individual is expected to produce an average number of offspring of (possibly) different kinds from one generation to the next one. Mista el at. [85] show that branching processes can be adapted to predict the generation of ADT values by simply considering each data constructor as a kind of its own. In fact, any ADT value can be seen as a tree where each node represents a root data constructor and has its sub-expressions as sub-trees—hence note the similarity with family trees. In this light, each tree level of a random value can be seen as a generation of individuals in this model.

We characterize the numbers of constructors that a random generator produces in the n-th generation as a vector G_n, a vector that groups the number of constructors of each kind produced in that generation—in our `Html` example, this vector has four components, i.e., one for each constructor. From the branching processes theory, the following equation captures the expected distribution of constructors at the generation n, noted $E[G_n]$, as follows:

$$E[G_n]^T = E[G_0]^T \cdot M^n \tag{20}$$

Vector $E[G_0]$ represents the initial distribution of constructors that our generator produces, which simply consists of the generation probability of each one. The interesting aspect of the prediction mechanism is encoded in the matrix M, known as the *mean matrix* of this stochastic process. M is a squared matrix with as rows and columns as different data constructors involved in the generation process. Each element $M_{i,j}$ of this matrix encodes the average number of data constructors of kind j that get generated in a given generation, provided that we generated a constructor of kind i at the previous one. In this sense, this matrix encodes the "branching" behavior of our random generation from one generation to the next one. Each element of the matrix can be automatically calculated by exploiting ADT definitions, as well as the individual probability of generating each constructor. For instance, the average number of `Text` data constructors that we will generate provided that we generated a `Join` constructor on the previous level results:

$$M_{\text{Join},\text{Text}} = 2 \cdot p_{\text{Text}}$$

where 2 is the number of holes present when generating a partial ADT value `Join` (i.e., `Join` $\bullet$ $\bullet$) and p_{Text} is the probability of individually generating the constructor `Text`. This reasoning can be used to build the rest of the mean matrix analogously.

5.1 Extending predictions for structural information

In this work, we show how to naturally fit structural information beyond ADT definitions into the prediction mechanism of branching processes. Our realization is that it suffices to consider *each different pattern matching and function call as a kind of individual on its own.* In that manner, we can extend our mean matrix M by adding a row and a column for each different pattern matching and function call as shown in Figure 3. Symbol $C_1 \cdots C_i$ denotes constructors, $P_1 \cdots P_j$ pattern matchings, and $F_1 \cdots F_k$ function calls. The light-red colored matrix is what we had before, whereas the light-blue colored cells are new—we encourage readers to obtain a colored copy of this work.

The new cells are filled as before: we need to consider the number of holes when generating partial pattern matching values and function calls as well as their individual probabilities. For instance, if we consider P_j as the second pattern of function `simplify` and F_1 as function `div`, then the marked cell above has the value $2 \cdot p_{div}$, i.e., the number of holes in the partially generated pattern (`Join` (`Join` (`Text` `s`) $\bullet$) $\bullet$), where `s` is some random string, times the

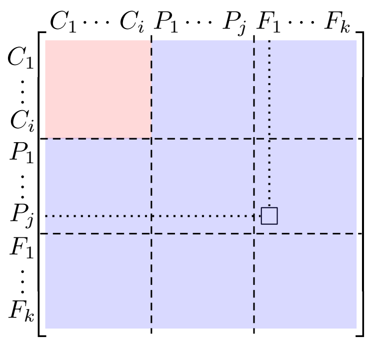


Figure 3: Mean matrix M including pattern matching and function calls information.

probability to generate a call to function `div`. The rest of this matrix can be computed analogously.

As another contribution, we found that the whole prediction process can be factored in terms of two vectors β and $\mathcal{P}$, such that β represents the number of holes in each partial ADT value that we generate, whereas $\mathcal{P}$ simply represents the probability of generating that partial ADT value. Then, the equation (20) can be rewritten as:

$$E[G_n]^T = \beta^T \cdot (\beta \cdot \mathcal{P}^T)^n$$

For instance, β and $\mathcal{P}$ for our generation specification of HTML values are as shown in Figure 4. We note `simplify#1` and `simplify#2` to the patterns occurring in the first and second clauses of `simplify`, respectively.

Note that by varying the shape of the vector $\mathcal{P}$ we can tune the distribution of our random generator in a way that can be always characterized and predicted. *DRAGEN2* follows a similar approach as *DRAGEN* and uses a heuristic to tune the generation probabilities of each source of structural information. This is done by running a simulation-based optimization process at compile-time. This process is parameterized by the desired distribution of values set

$$\beta = \begin{array}{r} \text{Text} \\ \text{Single} \\ \text{Tag} \\ \text{Join} \\ \text{simplify\#1} \\ \text{simplify\#2} \\ \text{hr} \\ \text{div} \\ \text{bold} \end{array} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 2 \\ \hdashline 0 \\ 2 \\ \hdashline 0 \\ 1 \\ 1 \end{bmatrix} \qquad \mathcal{P} = \begin{bmatrix} p_{\text{Text}} \\ p_{\text{Single}} \\ p_{\text{Tag}} \\ p_{\text{Join}} \\ \hdashline p_{\text{simplify\#1}} \\ p_{\text{simplify\#2}} \\ \hdashline p_{\text{hr}} \\ p_{\text{div}} \\ p_{\text{bold}} \end{bmatrix}$$

Figure 4: Prediction vectors of our `Html` generation specification.

by the user. In this manner, developers can specify, for instance, a uniform distribution of data constructors, pattern matching values and function calls or, alternatively, a distribution of values with some constructions appearing in a different proportion as others, e.g., two times more function calls to `div` than `Join` constructors.

5.2 Overall prediction

It is possible to provide an overall prediction of the expected number of constructors when restricting the generation process to only bare data constructors and pattern matching values. To achieve that, we should stop considering pattern matching values as atomic constructions and start seeing them as compositions of several data constructors. In that manner, it is possible to obtain the expected *total* number of generated data constructors that our generators will produce—regardless if they are generated on their own, or as part of a pattern matching value. We note this number as $E^{\downarrow}[\_]$ and, to calculate it, we only need to add the expected number of bare constructors that are included within each pattern matching. For instance, we can calculate the total expected number of constructors `Text` and `Join` that we will generate by simply expanding the expected number of generated pattern matching values `simplify#1` and `simplify#2` into their corresponding data constructors:

$$E^{\downarrow}[\texttt{Text}] = E\,[\texttt{Text}] + 2 \cdot E[\texttt{simplify\#1}] + 1 \cdot E[\texttt{simplify\#2}]$$
$$E^{\downarrow}[\texttt{Join}] = E\,[\texttt{Join}] + 1 \cdot E[\texttt{simplify\#1}] + 2 \cdot E[\texttt{simplify\#2}]$$

Observe that each time we generate a value satisfying the first pattern matching of the function `simplify`, we add two `Text` and one `Join` data constructors to our random value. The case of the second pattern matching of `simplify` follows analogously. Note that the overall prediction cannot be applied if we also generate random values containing function calls, as we cannot predict the output of an arbitrary function.

6 Case Studies

This section describes two case studies showing that considering additional structural information when deriving generators can consistently produce better testing results in terms of code coverage. Instead of restricting our scope to Haskell, in this work we follow a broader evaluation approach taken previously to compare state-of-the-art techniques to derive random data generators based on ADT definitions [71], [85].

We evaluate how including additional structural information when generating a set of random test cases (often referred to as a *corpus*) affects the code coverage obtained when testing a given target program. For that, we considered two external programs which expect highly structured inputs, namely *GNU CLISP* [9]—the GNU Common Lisp compiler, and *HTML Tidy* [10]—a well

[9]https://www.gnu.org/software/gcl/
[10]http://www.html-tidy.org

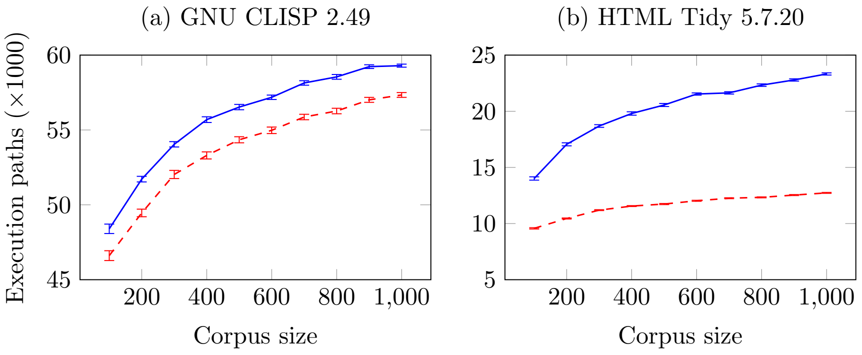


Figure 5: Path coverage comparison between *DRAGEN* (- - -) and *DRAGEN2* (——).

known HTML refactoring and correction utility. We remark that these applications are not written in Haskell. However, there exist Haskell libraries defining ADTs encoding their input structure, i.e., Lisp and HTML values respectively. These libraries are: *hs-zuramaru*[11], implementing an embedded Lisp interpreter for a small subset of this programming language, and *html*[12], defining a combinator library for constructing HTML values. These libraries also come with serialization functions to map Haskell values into corresponding test case files.

We first compiled instrumented versions of the target programs in a way that they also return the execution path followed in the source code every time we run them with a given input test case. This let us distinguish the number of different execution paths that a randomly generated corpus can trigger. We then used the ADTs defined on the chosen libraries to derive random generators using *DRAGEN* and *DRAGEN2*, including structural information extracted from the library's codebase in the case of the latter. Then, we proceeded to evaluate the code coverage triggered by independent, randomly generated corpora of different sizes varying from 100 to 1000 test cases each. In order to remove any external bias, we derived generators optimized to follow *a uniform distribution of constructors (and pattern matchings or function calls in the case* DRAGEN2*), and carefully adjusted their generation sizes to match the average test case size in bytes*. This way, any noticeable difference in the code coverage can be attributed to the presence (or lack thereof) of structural information when generating the test cases. Additionally, to achieve statistical significance we repeated each experiment 30 times with independently generated sets of random test cases.

Figure 5 illustrates the mean number of different execution paths triggered for different combinations of corpus size and derivation tool, including error bars indicating the standard error of the mean on each case. We proceed to describe each case study and our findings in detail as follows.

[11]http://hackage.haskell.org/package/zuramaru
[12]http://hackage.haskell.org/package/html

6.1 Branching on input data

In this first case study we wanted to evaluate the observed code coverage differences when considering structural information present in functions pattern matchings.

Our chosen library encodes Lisp S-expressions essentially as lists of symbols, represented as plain strings; and literal values like booleans or integers. In order to interpret Lisp programs, this unified representation of data and code requires this library to pattern match against common patterns like let-bindings, if-then-else expressions and arithmetic operators among others. In particular, each one of these patterns matches against a special symbol of the Lisp syntax like `"let"`, `"if"` or `"+"`; and their corresponding sub-expressions. We extracted this structural information and included it into the generation specification of our random Lisp values—which were generated by randomly picking from a total of 6 data constructors and 8 different pattern matchings. By doing this, we obtained a code coverage improvement of approximately 4% using *DRAGEN2* with respect to the one obtained with *DRAGEN* (see Figure 5 (a)). While it seems an small improvement, we argue that an improvement of 4% is not negligible considering (a) the little effort that took us to specify the pattern matchings and (b) that we are testing a full-fledged compiler.

6.2 Abstract interfaces

For our second case study, we wanted to evaluate how including structural information coming from abstract interfaces when generating random HTML values might improve the testing performance.

The library we used for this purpose represents HTML values very much in the same way as we exemplify in Section 2, i.e., defining a small set of general constructions representing plain text and tags—although this library also supports HTML tag attributes as well. Then, this representation is extended with a large abstract interface consisting of combinators representing common HTML tags and tag attributes—equivalent to the combinators `div`, `bold` and `hr` illustrated in Section 3.

In this case study we included the structural information present in the abstract interface of this library into the generation specification of random HTML values, resulting in a generation process that randomly picked among 4 data constructors and 163 abstract functions. With this large amount of additional structural information, we observed an increase of up to 83% in the code coverage obtained with *DRAGEN2* with respect to the one observed with *DRAGEN* (see Figure 5 (b)). A manual inspection of the corpora generated with each tool revealed that, in general terms, the test cases generated with *DRAGEN* rarely represent syntactically correct HTML values, consisting to a large extent of random strings within and between HTML tag delimiters (`"<"`, `">"` and `"/>"`). On the other hand, test cases generated with *DRAGEN2* encode much more interesting structural information, being mostly syntactically correct. We found that, in many cases, the test cases generated with *DRAGEN2* were parsed, analyzed and reported as valid HTML values by the target application.

With these results, we are confident that including the structural information present on the user codebase improves the overall testing performance.

7 Related Work

Boltzmann models [76] are a general approach to randomly generating combinatorial structures such as trees, graphs, closed simply-typed lambda terms, etc. A random generator built around such models uniformly generates values of a target size with a certain size tolerance. However, it has been argued that this approach has theoretical and practical limitations in the context of software testing [79]. In a recent work, Bendkowski et al. provide a framework called *boltzmann-brain* to specify and synthesize standalone Haskell random generators based on Boltzmann models [87]. This framework mixes parameter tuning and rejection of samples of unwanted sizes to approximate the desired distribution of values according to user demands. The overall discard ratio then depends on how constrained the desired sizes of values are. On the other hand, our work is focused on approximating the desired distribution as much as possible via parameter optimization, without discarding any generated value at runtime. Although promising, we found difficulties to compare both approaches in practice due that *boltzmann-brain* is considered a conceptual standalone utility that produces self-contained samplers. In this light, data specifications have to be manually written using a special syntax, and cannot include Haskell ground types like `String` or `Int`, difficulting the integration of this tool to existing Haskell codebases like the ones we consider in this work.

From the practical point of view, Feldt and Poulding propose *GödelTest* [79], a search-based framework for generating biased data. Similar to our approach, *GödelTest* works by optimizing the parameters governing the desired biases on the generated data. However, the optimization mechanism uses meta-heuristic search to find the best parameters at runtime. *DRAGEN2* on the other hand implements an analytic and composable prediction mechanism that is only used at compile time to optimize the generation parameters, thus avoiding performing any kind of runtime reinforcement.

Directed Automated Random Testing (DART) is a technique that combines random testing with symbolic execution for C programs [88]. It requires instrumenting the target programs in order to introduce testing assertions and obtain feedback from previous testing executions, which is used to explore new paths in the source code. This technique has been shown to be remarkably useful, although it forces a strong coupling between the testing suite and the target code. Our tool intends to provide better random generation of values following an undirected fashion, without having to instrument the target code, but still extracting useful structural information from it.

8 Final Remarks

We extended the standard approach for automatically deriving random generators in Haskell. Our generators are capable of producing complex and interesting random values by exploiting static structural information found in the user code-base. Based on the theory of branching processes, we adapt our previous prediction mechanism to characterize the distribution of random values representing the different sources of structural information that our generators might produce. These predictions let us optimize the generation parameters in compile time, resulting in an improved testing performance according to our experiments.

Paper IV

Deriving Compositional Random Generators

Agustín Mista and Alejandro Russo

31st Symposium on Implementation and Application of Functional Languages
(2019)

Abstract

Generating good random values described by algebraic data types is often quite intricate. State-of-the-art tools for synthesizing random generators serve the valuable purpose of helping with this task, while providing different levels of invariants imposed over the generated values. However, they are often not built for composability nor extensibility, a useful feature when the shape of our random data needs to be adapted while testing different properties or subsystems.

In this work, we develop an extensible framework for deriving compositional generators, which can be easily combined in different ways in order to fit developers' demands using a simple type-level description language. Our framework relies on familiar ideas from the à la Carte technique for writing composable interpreters in Haskell. In particular, we adapt this technique with the machinery required in the scope of random generation, showing how concepts like generation frequency or terminal constructions can also be expressed in the same type-level fashion. We provide an implementation of our ideas, and evaluate its performance using real-world examples.

1 Introduction

Random property-based testing is a powerful technique for finding bugs [17], [19], [20], [67]. In Haskell, *QuickCheck* is the predominant tool for this task [16]. The developers specify i) the testing properties their systems must fulfill, and ii) random data generators (or generators for short) for the data types involved at their properties. Then, *QuickCheck* generates random values and uses them to evaluate the testing properties in search of possible counterexamples, which always indicate the presence of bugs, either in the program or in the specification of our properties.

Although *QuickCheck* provides default generators for the common base types, like `Int` or `String`, it requires implementing generators for any user-defined data type we want to generate. This process is cumbersome and error-prone, and commonly follows closely the shape of our data types. Fortunately, there exists a variety of tools helping with this task, providing different levels of invariants on the generated values as well as automation [29], [70], [72], [85]. We divide the different approaches into two kinds: those which are *manual*, where generators are often able to enforce a wide range of invariants on the generated data, and those which are *automatic* where the generators can only guarantee lightweight invariants like generating well-typed values.

On the manual side, *Luck* [72] is a domain-specific language for manually writing testing properties and random generators in tandem. It allows obtaining generators specialized to produce random data which is proven to satisfy the preconditions of their corresponding properties. In contrast, on the automatic side, tools like *MegaDeTH* [70], [71], *DRAGEN* [85] and *Feat* [29] allow obtaining random generators automatically at compile time. *MegaDeTH* and *DRAGEN* derive random generators following a simple recipe: to generate a value, they simply pick a random data constructor from our data type with a given probability, and proceed to generate the required sub-terms recursively. *MegaDeTH* pays no attention to the generation frequencies, nor the distribution induced by the derived generator—it just picks among data constructors with uniform probability. Differently, *DRAGEN* analyzes type definitions and tunes the generation frequencies to match the desired distribution of random values specified by developers. Finally, *Feat* relies on functional enumerations, deriving random generators that sample random values uniformly across the whole search space of values of up to a given size of the data type under consideration. In this work, we focus on automatic approaches to derive generators.

While *MegaDeTH*, *DRAGEN*, and *Feat* provide a useful mechanism for automating the task of writing random generators by hand, they implement a derivation procedure which is often too generic to synthesize useful generators in common scenarios, mostly because *they only consider the structural information encoded in type definitions.* To illustrate this point, consider the following type definition encoding basic HTML pages—inspired by the widely used *html* package:[13]

[13]http://hackage.haskell.org/package/html

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

This type allows building HTML pages via four possible data constructors: Text is used for plain text values; Sing and Tag represent singular and paired HTML tags, respectively; whereas the infix (:+:) constructor simply concatenates two HTML pages one after another. Note that the constructors Tag and (:+:) are recursive, as they have at least one field of type Html. Then, the example page:

```
<html>hi<br><b>bye</b></html>
```

can be encoded with the following Html value:

```
Tag "html" (Text "hi" :+: Sing "br" :+: Tag "b" (Text "bye"))
```

In this work, we focus on two scenarios where deriving generators following only the information extracted from type definitions does not work well. The first case is when type definitions are too general (like the case of Html) where, as a consequence, the generation process leaves a large room for ill-formed values, e.g., invalid HTML pages. For instance, when generating an Html value using the Sing constructor, it is very likely that an automatically derived generator will choose a random string not corresponding to any valid HTML singular tag. In such situations, a common practice is to rely on existing abstract interfaces to generate random values—such interfaces are often designed to preserve our desired invariants. As an example, consider that our Html data type comes equipped with the following abstract interface:

```
br :: Html
bold :: Html → Html
list :: [Html] → Html
(⟨+⟩) :: Html → Html → Html
```

These high-level combinators let us represent structured HTML constructions like line breaks (br), bold blocks (bold), unordered lists (list) and concatenation of values one below another (⟨+⟩). This methodology of generating random data employing high-level combinators has shown to be particularly useful in the presence of monadic code [57], [71].

The second scenario that we consider is where derived generators fail to produce very specific patterns of values which might be needed to trigger bugs. For instance, a function for simplifying Html values might be defined to branch differently over complex sequences of Text and (:+:) constructors:

```
simplify :: Html → Html
simplify (Text t₁ :+: Text t₂) = ···
simplify (Text t  :+: x :+: y) = ···
simplify ···                   = ···
```

(Symbol $\cdots$ denotes code that is not relevant for the point being made.) Generating values that match, for instance, the pattern `Text t :+: x :+: y` using *DRAGEN* under a uniform distribution will only occur 6% of the time! Clearly, these input pattern matchings should also be included into our generators, allowing them to produce random values satisfying such inputs. This structural information can help increase the chances of reaching portions of our code which otherwise would be very difficult to test. Functions' pattern matchings often expose interesting relationships between multiple data constructors, a valuable asset for testing complex systems expecting highly structured inputs [86].

Our previous work [89] focuses on extending *DRAGEN*'s generators as well as its predictive approach to include all these extra sources of structural information, namely high-level combinators and functions' input patterns, while allowing tuning the generation parameters based on the developers' demands. In turn, this work focuses on an orthogonal problem: that of *modularity*. In essence, all the automatic tools cited above work by synthesizing *rigid* monolithic generator definitions. Once derived, these generators have almost no parameters available for adjusting the shape of our random data. Sadly, this is something we might want to do if we need to test different properties or subsystems using random values generated in slightly different ways. As the reader might appreciate, it can become handy to cherry-pick, for each situation, which data constructors, abstract interface functions, or functions' input patterns to consider when generating random values.

The contribution of this work is an automated framework for synthesizing compositional random generators, which can be naturally extended to include the extra sources of structural information mentioned above. Using our approach, a user can obtain random generators following different *generation specifications* whenever necessary, all of them built upon the *same* underlying machinery which only needs to be derived *once*.

Figure 1 illustrates a possible usage scenario of our approach. We first invoke a derivation procedure (1a) to extract the structural information of the type `Html` encoded on i) its data constructors, ii) its abstract interface, and iii) the patterns from the function `simplify`. Then, two different generation specifications, namely $\texttt{Html}_{\texttt{valid}}$ and $\texttt{Html}_{\texttt{simplify}}$ can be defined using a simple type-level idiom (1b). Each specification mentions the different sources of structural information to consider, along with (perhaps) their respective generation frequency. Intuitively, $\texttt{Html}_{\texttt{valid}}$ chooses among the constructors `Text` and `:+:` , as well as functions from `Html`'s abstract interface; while $\texttt{Html}_{\texttt{simplify}}$ chooses among all `Html`'s constructors and the patterns of the first and second clauses in the function `simplify`. The syntax used there will be addressed in detail in Sections 3 to 5. Finally, we obtain two concrete random generators following such specifications by writing `genRep` $\texttt{@Html}_{\texttt{valid}}$ and `genRep` $\texttt{@Html}_{\texttt{simplify}}$, respectively.

The main contributions of this paper are:

- We present an extensible mechanism for representing random values built upon different sources of structural information, adopting ideas from *Data Types à la Carte* [90] (Section 3).

```
derive [constructors ''Html, interface ''Html, patterns 'simplify]
```

(a) Machinery derivation

```
type Html_valid =                       type Html_simplify =
      Con "Text" ⊗ 2                          Con "Text"          ⊗ 2
    ⊕ Con ":+:"  ⊗ 4                        ⊕ Con "Sing"          ⊗ 1
    ⊕ Fun "hr"   ⊗ 3                        ⊕ Con "Tag"           ⊗ 3
    ⊕ Fun "bold" ⊗ 2                        ⊕ Con ":+:"           ⊗ 4
    ⊕ Fun "list" ⊗ 3                        ⊕ Pat "simplify" 1 ⊗ 3
    ⊕ Fun "<+>"  ⊗ 5                        ⊕ Pat "simplify" 2 ⊗ 5

genHtml_valid    = genRep @Html_valid
genHtml_simplify = genRep @Html_simplify
```

(b) Generators specification

Figure 1: Usage example of our framework. Two random generators obtained from the same underlying machinery.

- We develop a modular generation scheme, extending our representation to encode information relevant to the generation process at the type level (Section 4).
- We propose a simple type-level idiom for describing extensible generators, based on the types used to represent the desired shape of our random data (Section 5).
- We provide a Template Haskell tool[14] for automatically deriving all the required machinery presented throughout this paper, and evaluate its generation performance with three real-world case studies and a type-level runtime optimization (Section 6).

Overall, we present a novel technique for reusing automatically derived generators in a composable fashion, in contrast to the usual paradigm of synthesizing rigid, monolithic generators.

2 Random Generators in Haskell

In this section, we introduce the common approach for writing random generators in Haskell using *QuickCheck*, along with the motivation for including extra information into our generators, discussing how this could be naively implemented in practice.

[14]Available at `https://github.com/OctopiChalmers/dragen2`

In order to provide a common interface for writing generators, *QuickCheck* uses Haskell's overloading mechanism known as *type classes* [27], defining the `Arbitrary` class for random generators as:

```
class Arbitrary a where
  arbitrary :: Gen a
```

where the overloaded symbol `arbitrary :: Gen a` denotes a monadic generator for values of type `a`. Using this mechanism, a user can define a sensible random generator for our `Html` data type as follows:

```
instance Arbitrary Html where
  arbitrary = sized gen
    where
      gen 0 = frequency
        [(2, Text ⟨$⟩ arbitrary)
        ,(1, Sing ⟨$⟩ arbitrary)]
      gen d = frequency
        [(2, Text ⟨$⟩ arbitrary)
        ,(1, Sing ⟨$⟩ arbitrary)
        ,(4, Tag  ⟨$⟩ arbitrary ⟨*⟩ gen (d − 1))
        ,(3, (:+:) ⟨$⟩ gen (d − 1) ⟨*⟩ gen (d − 1))]
```

At the top level, this definition parameterizes the generation process using *QuickCheck*'s `sized` combinator, which lets us build our generator via an auxiliary, locally defined function `gen::Int → Gen Html`. The `Int` passed to `gen` is known as the *generation size*, and is threaded seamlessly by *QuickCheck* on each call to `arbitrary`. We use this parameter to limit the maximum amount of recursive calls that our generator can perform, and thus the maximum depth of the generated values. If the generation size is positive (case `gen d`), our generator picks a random `Html` constructor with a given generation frequency (denoted here by the arbitrarily chosen numbers 2, 1, 4 and 3) using *QuickCheck*'s `frequency` combinator. Then, our generator proceeds to fill its fields using randomly generated sub-terms—here using Haskell's applicative notation [48] and the default `Arbitrary` instance for `String`s. For the case of the recursive sub-terms, this generator simply calls the function `gen` recursively with a smaller depth limit (`gen (d − 1)`). This process repeats until we reach the base case (`gen 0`) on each recursive sub-term. At this point, our generator is limited to picking only among terminal `Html` constructors, hence ending the generation process.

As one can observe, the previous definition is quite mechanical, and depends only on the generation frequencies we choose for each constructor. This simple generation procedure is the one used by tools like *MegaDeTH* or *DRAGEN* when synthesizing generators.

2.1 Abstract Interfaces

A common choice when implementing abstract data types is to transfer the responsibility of preserving their invariants to the functions on their abstract interface. Take for example our `Html` data type. Instead of defining a different constructor for each possible HTML construction, we opted for a small generic representation that can be extended with a set of high-level combinators:

```
br :: Html
br = Sing "br"

bold :: Html → Html
bold = Tag "b"

list :: [Html] → Html
list []  = Text "empty list"
list xs = Tag "ul" (foldl1 (:+:) (Tag "li" ⟨$⟩ xs))

(⟨+⟩) :: Html → Html → Html
(⟨+⟩) x y = x :+: br :+: y
```

Note how difficult it would be to generate random values containing, for example, structurally valid HTML lists, if we only consider the structural information encoded in our `Html` type definition. After all, much of the valid structure of HTML has been encoded in its abstract interface.

A synthesized generator could easily contemplate this structural information by creating random values arising from applying such functions to randomly generated inputs:

```
instance Arbitrary Html where
  arbitrary = ···
    frequency
    [...
    , (1, pure br)
    , (5, bold ⟨$⟩ gen (d − 1))
    , (2, list ⟨$⟩ listOf (gen (d − 1)))
    , (3, (⟨+⟩) ⟨$⟩ gen (d − 1) ⟨*⟩ gen (d − 1))]
```

where (...) represents the rest of the code of the random generator introduced before. From now on, we will refer to each choice given to the `frequency` combinator as a different *random construction*, since we are not considering generating only single data constructors anymore, but more general value fragments.

2.2 Functions' Pattern Matchings

A different challenge appears when we try to test functions involving complex pattern matchings. Consider, for instance, the full definition of the function `simplify` introduced in Section 1:

```
simplify :: Html → Html
simplify (Text t₁ :+: Text t₂) = Text (t₁ ++ t₂)
simplify (Text t  :+: x :+: y) =
  simplify (Text t :+: simplify (x :+: y))
simplify (x :+: y) = simplify x :+: simplify y
simplify (Tag t x) = Tag t (simplify x)
simplify x = x
```

This function traverses `Html` values, joining together every contiguous pair of `Text` constructors. Ideally, we would like to put approximately the same testing effort into each clause of `simplify`, or perhaps even more to the first two ones, since those are the ones performing actual simplifications. However, these two clauses are the most difficult ones to test in practice! The probability of generating a random value satisfying nested patterns decreases multiplicatively with the number of constructors we simultaneously pattern match against. In our tests, we were not able to exercise any of these two patterns more than 6% of the overall testing time, using random generators derived using both *MegaDeTH* and *DRAGEN*. As expected, most of the random test cases were exercising the simplest (and rather uninteresting) patterns of this function.

To solve this issue, we could opt to consider each complex pattern as a new kind of random construction. In this light, we can simply generate values satisfying patterns directly by returning their corresponding expressions, where each variable or wildcard pattern is filled using a random sub-expression:

```
instance Arbitrary Html where
  arbitrary = ···
    frequency
    [...
    , (2, do t₁ ← arbitrary
             t₂ ← arbitrary
             return (Text t₁ :+: Text t₂))
    , (4, do t ← arbitrary
             x ← gen (d − 1)
             y ← gen (d − 1);
             return (Text t :+: x :+: y))]
```

While the ideas presented in this section are plausible, accumulating cruft from different sources of structural information into a single, global `Arbitrary` instance is unwieldy, especially if we consider that some random constructions might not be relevant or desired in many cases, e.g., generating the patterns of the function `simplify` might only be useful when testing properties involving such function, and nowhere else.

In contrast, the following sections of this paper present our extensible approach for deriving generators, where the required machinery is derived once, and each variant of our random generators is expressed on a per-case basis.

3 Modular Random Constructions

This section introduces a unified representation for the different constructions we might want to consider when generating random values. The key idea of this work is to lift each different source of structural information to the type level. In this light, the shape of our random data is determined entirely by the types we use to represent it during the generation process.

For this purpose, we will use a set of simple “open” *representation types*, each one encoding a single random construction from our *target* data type, i.e., the actual data type we want to randomly generate. These types can be i) combined in several ways depending on the desired shape of our test data (applying the familiar à la Carte technique); ii) randomly generated (see Section 4); and finally, iii) transformed to the corresponding values of our target data type automatically. This representation can be automatically derived from our source code at compile time, relieving programmers of the burden of manually implementing the required machinery.

3.1 Representing Data Constructors

When generating values of algebraic data types, the simplest piece of meaningful information we ought to consider is the one given by each one of its constructors. In this light, each constructor of our target type can be represented using a single-constructor data type. Recalling our Html example, its constructors can be represented as:

```
data Con_Text  r = Mk_Text  String
data Con_Sing  r = Mk_Sing  String
data Con_Tag   r = Mk_Tag   String r
data Con_(:+:) r = Mk_(:+:) r r
```

Each representation type has the same fields as its corresponding constructor, except for the recursive ones which are abstracted away using a type parameter r. This parametricity lets us leave the type of recursive sub-terms unspecified until we have decided on the final shape of our random data. Then, for instance, the value $\mathtt{Mk}_{\mathtt{Tag}}$ "div" x :: $\mathtt{Con}_{\mathtt{Tag}}$ r represents the Html value Tag "div" x, for some sub-term x :: r that can be transformed to Html as well. Note how these representations types encode the minimum amount of information they need, leaving everything else unspecified.

An important property of these parametric representations is that, in most cases, they form a functor over its type parameter, thus we can use Haskell’s **deriving** mechanism to obtain suitable Functor instances for free—this will be useful for the next steps.

The next building block of our approach consists of providing a mapping from each constructor representation to its corresponding target value, provided that each recursive sub-term has already been translated to its corresponding target value. This notion is often referred to as an *F-Algebra* over the functor used to represent each different construction. Accordingly, to represent this mapping, we will define a type class Algebra with a single method alg as follows:

```
class Functor f ⇒ Algebra f a | f → a where
  alg :: f a → a
```

where f is the functor type used to represent a construction of the target type a. The functional dependency f → a helps the type system to solve the type of the type variable a, which appears free on the right hand side of the ⇒. This means that every representation type f will uniquely determine its target type a. Then, we need to instantiate this type class for each data constructor representation we are considering, providing an appropriate implementation for the overloaded alg function:

```
instance Algebra Con_Text Html where
  alg (Mk_Text x) = Text x
instance Algebra Con_Sing Html where
  alg (Mk_Sing x) = Sing x
instance Algebra Con_Tag Html where
  alg (Mk_Tag t x) = Tag t x
instance Algebra Con_(:+:) Html where
  alg (Mk_(:+:) x y) = x :+: y
```

There, we simply transform each constructor representation into its corresponding data constructor, piping its fields unchanged.

3.2 Composing Representations

So far we have seen how to represent each data constructor of our Html data type independently. In order to represent interesting values, we need to be able to combine single representations into (possibly complex) composite ones. For this purpose, we will define a functor type ⊕ to encode the choice between two given representations:

```
data ((f :: * → *) ⊕ (g :: * → *)) r = In_L (f r) | In_R (g r)
```

This infix type-level operator lets us combine two representations f and g into a composite one f ⊕ g, encoding either a value drawn from f (via the In$_L$ constructor) or a value drawn from g (via the In$_R$ constructor). This operator works pretty much in the same way as Haskell's Either data type, except that, instead of combining two base types, it works by combining two *parametric type constructors*, hence the kind signature $* \to *$ in both f and g. For instance, the type Con$_{Text}$ ⊕ Con$_{Tag}$ encodes values representing either plain text HTML or paired tags. Such values can be constructed using the injections In$_L$ and In$_R$ on each case, respectively.

The next step consists of providing a mapping from composite representations to target types, provided that each component can be translated into the same target type:

```
instance (Algebra f a, Algebra g a) ⇒ Algebra (f ⊕ g) a where
  alg (In_L fa) = alg fa
  alg (In_R ga) = alg ga
```

There, we use the appropriate Algebra instance of the inner representation, based on the injection used to create the composite value.

Worth remarking, the order in which we associate each operand of $\oplus$ results semantically irrelevant. However, in practice, associativity takes a dramatic role when it comes to generation speed. This phenomenon is addressed in detail in Section 6.

3.3 Tying the Knot

Even though we have already seen how to encode single and composite representations for our target data types, there is a piece of machinery still missing: our representations are not recursive, but parametric on their recursive fields. We can think of them as encoding a *single layer* of our target data. In order to represent recursive values, we need to close them *tying the knot* recursively, i.e., once we have fixed a suitable representation type for our target data, each one of its recursive fields has to be instantiated with itself. This can be easily achieved by using a type-level fixed point operator:

```
data Fix (f :: * → *) = Fix {unFix :: f (Fix f)}
```

Given a representation type `f` of kind $* \to *$, the type `Fix f` instantiates each recursive field of `f` with `Fix f`, closing the definition of `f` into itself—thus the kind of `Fix f` results $*$.

In general, if a type `f` is used to represent a given target type, we will refer to `Fix f` as a *final representation*, since it cannot be further combined or extended—the $\oplus$ operator has to be applied *within* the `Fix` type constructor.

The effect of a fixed point combinator is easier to interpret with an example. Let us imagine we want to represent our `Html` data type using all of its data constructors, employing the following type:

$$\textbf{type}\ \texttt{Html}' = \texttt{Con}_{\texttt{Text}} \oplus \texttt{Con}_{\texttt{Sing}} \oplus \texttt{Con}_{\texttt{Tag}} \oplus \texttt{Con}_{(:+:)}$$

Then, for instance, the value `x = Text "hi" :+: Sing "hr" :: Html` can be represented with a value `x' :: Fix Html'` as:

$$\begin{aligned}\texttt{x}' = \texttt{Fix}\ &(\texttt{In}_{\texttt{R}}\ (\texttt{In}_{\texttt{R}}\ (\texttt{In}_{\texttt{R}}\ (\texttt{Mk}_{(:+:)} \\ &\quad (\texttt{Fix}\ (\texttt{In}_{\texttt{L}}\ (\texttt{Mk}_{\texttt{Text}}\ \texttt{"hi"}))) \\ &\quad (\texttt{Fix}\ (\texttt{In}_{\texttt{R}}\ (\texttt{In}_{\texttt{L}}\ (\texttt{Mk}_{\texttt{Sing}}\ \texttt{"hr"}))))))))\end{aligned}$$

where the sequences of $\texttt{In}_{\texttt{L}}$ and $\texttt{In}_{\texttt{R}}$ data constructors *inject* each value from an individual representation into the appropriate position of our composite representation `Html'`.

Finally, we can define a generic function `eval` to evaluate any value of a final representation type `Fix f` into its corresponding value of the target type `a` as follows:

```
eval :: Algebra f a ⇒ Fix f → a
eval = alg ∘ fmap eval ∘ unFix
```

This function exploits the `Functor` structure of our representations, unwrapping the fixed points and mapping their algebras to the result of evaluating recursively each recursive sub-term.

In our particular example, this function satisfies `eval x′ == x`. More specifically, the types `Html` and `Fix Html′` are in fact isomorphic, with `eval` as the witness of one side of this isomorphism—though this is not the case for any arbitrary representation.

3.4 Representing Additional Constructions

The representation mechanism we have developed so far lets us determine the shape of our target data based on the type we use to represent its constructors. However, it is hardly useful for random testing, as the values we can represent are still quite unstructured. It is not until we start considering more complex constructions that this approach becomes particularly appealing.

3.4.1 Abstract Interfaces

Let us consider the case of generating values obtained by abstract interface functions. If we recall our `Html` example, the functions on its abstract interface can be used to obtain `Html` values based on different input arguments. Fortunately, it is easy to extend our approach to incorporate the interesting structure arising from these functions into our framework. As before, we start by defining a set of open data types to encode each function as a random construction:

```
data Fun_br   r = Mk_br
data Fun_bold r = Mk_bold r
data Fun_list r = Mk_list [r]
data Fun_⟨+⟩  r = Mk_⟨+⟩ r r
```

Each data type represents a value resulting from evaluating its corresponding function, using the values encoded on its fields as input arguments. Once again, we replace each recursive field (representing a recursive input argument) with a type parameter `r` in order to leave the type of the recursive sub-terms unspecified until we have decided on the final shape of our data.

By representing values obtained from function application this way, we are not performing any actual computation—we simply store the functions' input arguments. Instead, these functions are evaluated when transforming each representation into its target type, by the means of an `Algebra`:

```
instance Algebra Fun_br Html where
  alg Mk_br = br
instance Algebra Fun_bold Html where
  alg (Mk_bold x) = bold x
instance Algebra Fun_list Html where
  alg (Mk_list xs) = list xs
instance Algebra Fun_⟨+⟩ Html where
  alg (Mk_⟨+⟩ x y) = x ⟨+⟩ y
```

Where we simply return the result of evaluating each corresponding function, using its representation fields as input arguments.

It is important to remark that this approach inherits any possible downside from the functions we use to represent our target data. In particular, representing non-terminating functions might produce a non-terminating behavior when calling the `eval` function.

3.4.2 Functions' Pattern Matchings

The second source of structural information that we consider in this work is the one present in functions' pattern matchings. If we recall our `simplify` function, we can observe it has two complex, non-trivial patterns that we might want to satisfy when generating random values. We can extend our approach in order to represent these patterns as well. We start by defining data types for each one of them, this time using the fields of each single data constructor to encode the free pattern variables (or wildcards) appearing on its corresponding pattern:

```
data Pat_simplify#1 r = Mk_simplify#1 String String
data Pat_simplify#2 r = Mk_simplify#2 String r r
```

where the number after the # distinguishes the different patterns from the function `simplify` by the index of the clause they belong to. As before, we abstract away every recursive field (corresponding to a recursive pattern variable or wildcard) with a type variable `r`.

Then, the `Algebra` instance of each pattern will expand each representation into the corresponding target value resembling such a pattern, where each pattern variable gets instantiated using the values stored in its representation field:

```
instance Algebra Pat_simplify#1 Html where
  alg (Mk_simplify#1 t1 t2) = Text t1 :+: Text t2
instance Algebra Pat_simplify#2 Html where
  alg (Mk_simplify#1 t x y) = Text t :+: x :+: y
```

3.5 Lightweight Invariants for Free!

Using the machinery presented so far, we can represent the values of our target data coming from different sources of structural information in a compositional way.

Using this simple mechanism we can obtain values exposing light-weight invariants very easily. For instance, a value of type `Html` might encode invalid HTML pages if we construct them using invalid tags in the process (via the `Sing` or `Tag` constructors). To avoid this, we can explicitly disallow the direct use of the `Sing` and `Tag` constructors, replacing them with safe constructions from its abstract interface. In this light, a value of type:

$$\mathtt{Con}_{\mathtt{Text}} \oplus \mathtt{Con}_{(:+:)} \oplus \mathtt{Fun}_{\mathtt{br}} \oplus \mathtt{Fun}_{\mathtt{bold}} \oplus \mathtt{Fun}_{\mathtt{list}} \oplus \mathtt{Fun}_{\langle + \rangle}$$

always represents a valid HTML page.

Similarly, we can enforce that every `Text` constructor within a value will always appear in pairs of two, by using the following type:

$$\texttt{Con}_{\texttt{Sing}} \oplus \texttt{Con}_{\texttt{Tag}} \oplus \texttt{Con}_{(:+:)} \oplus \texttt{Pat}_{\texttt{simplify\#1}}$$

Since the only way to place a `Text` constructor within a value of this type is via the construction $\texttt{Pat}_{\texttt{simplify\#1}}$, which always contains two consecutive `Text`s.

As a consequence, generating random data exposing such invariants will simply become using an appropriate representation type while generating random values, without having to rely on runtime reinforcements of any sort. The next section introduces a generic way to generate random values from our different representations, extending them with a set of combinators to encode information relevant to the generation process directly at the type level.

4 Generating Random Constructions

So far we have seen how to encode different random constructions representing interesting values from our target data types. Such representations follow a modular approach, where each construction is independent from the rest. This modularity allows us to derive each different construction representation individually, as well as to specify the shape of our target data in simple and extensible manner.

In this section, we introduce the machinery required to randomly generate the values encoded using our representations. This step also follows the modular fashion, resulting in a random generation process that is entirely compositional. In this light, our generators are built from simpler ones (each one representing a single random construction), and are solely based on the types we use to represent the shape of our random data.

Ideally, our aim is to be able to obtain random generators with a behavior similar to the one presented for `Html` in Section 2. If we take a closer look at its definition, there we can observe three factors happening simultaneously:

- We use *QuickCheck*'s generation size to limit the depth of the generated values, reducing it by one on each recursive call of the local auxiliary function `gen`.
- We differentiate between *terminal* and *non-terminal (i.e. recursive) constructors*, picking only among terminal ones when we have reached the maximum depth (case `gen 0`).
- We generate different constructions with different frequencies.

For the rest of this section, we will focus on modeling these aspects in our modular framework, in such a way that does not compromise the compositionality obtained so far.

4.1 Depth-Bounded Modular Generators

The first obstacle that arises when trying to generate random values with a limited depth using our approach is related to modularity. If we recall the random generator for `Html` from Section 2 we can observe that the depth parameter `d` is threaded to the different recursive calls of our generator, always within the scope of the local function `gen`. Since each construction will have a specialized random generator, we cannot group them as we did before using an internal `gen` function. Instead, we will define a new type for depth-bounded generators, wrapping *QuickCheck*'s `Gen` type with an external parameter representing the maximum recursive depth:

```
type BGen a = Int → Gen a
```

A `BGen` is, essentially, a normal *QuickCheck* `Gen` with the maximum recursive depth as an input parameter. Using this definition, we can generalize *QuickCheck*'s `Arbitrary` class to work with depth-bounded generators simply as follows:

```
class BArbitrary (a :: ∗) where
  barbitrary :: BGen a
```

From now on, we will use this type class as a more flexible substitute of `Arbitrary`, given that now we have two parameters to tune: the maximum recursive depth, and the *QuickCheck* generation size. The former is useful for tuning the overall size of our random data, whereas the latter can be used for tuning the values of the *leaf types*, such as the maximum length of the random strings or the biggest/smallest random integers.

Here we want to remark that, even though we could have used *QuickCheck*'s generation size to simultaneously model the maximum recursive depth and the maximum size of the leaf types, doing so would imply generating random values with a decreasing size as we move deeper within a random value, obtaining for instance, random trees with all zeroes on their leaves, or random lists skewed to be ordered in decreasing order. In addition, one can always obtain a trivial `Arbitrary` instance from a `BArbitrary` one, by setting the maximum depth to be equal to *QuickCheck*'s generation size:

```
instance BArbitrary a ⇒ Arbitrary a where
  arbitrary = sized barbitrary
```

Even though this extension allows *QuickCheck* generators to be depth-aware, here we also need to consider the parametric nature of our representations. In the previous section, we defined each construction representation as being parametric on the type of its recursive sub-terms, as a way to defer this choice until we have specified the final shape of our target data. Hence, each construction representation is of kind $* \to *$. If we want to define our generators in a modular way, we also need to parameterize somehow the generation of the recursive sub-terms! If we look at *QuickCheck*, this library already defines a type class `Arbitrary1` for parametric types of kind $* \to *$, which solves this issue by receiving the generator for the parametric sub-terms as an argument:

```
class Arbitrary1 (f :: ∗ → ∗) where
  liftArbitrary :: Gen a → Gen (f a)
```

Then, we can use this same mechanism for our modular generators, extending `Arbitrary1` to be depth-aware as follows:

```
class BArbitrary1 (f :: ∗ → ∗) where
  liftBGen :: BGen a → BGen (f a)
```

Note the similarities between `Arbitrary1` and `BArbitrary1`. We will use this type class to implement random generators for each construction we are automatically deriving. Recalling our `Html` example, we can define modular random generators for the constructions representing its data constructors as follows:

```
instance BArbitrary1 Con_Text where
  liftBGen bgen d = Mk_Text ⟨$⟩ arbitrary
instance BArbitrary1 Con_Sing where
  liftBGen bgen d = Mk_Sing ⟨$⟩ arbitrary
instance BArbitrary1 Con_Tag where
  liftBGen bgen d = Mk_Tag ⟨$⟩ arbitrary ⟨∗⟩ bgen (d − 1)
instance BArbitrary1 Con_(:+:) where
  liftBGen bgen d = Mk_(:+:) ⟨$⟩ bgen (d − 1) ⟨∗⟩ bgen (d − 1)
```

Note how each instance is defined to be parametric of the maximum depth (using the input integer `d`) and of the random generator used for the recursive sub-terms (using the input generator `bgen`). Every other non-recursive sub-term can be generated using a normal `Arbitrary` in-stance—we use this to generate random `String`s in the previous definitions.

The rest of our representations can be generated analogously. For example, the `BArbitrary1` instances for `Fun`$_{\texttt{bold}}$ and `Pat`$_{\texttt{simplify\#2}}$ are as follows:

```
instance BArbitrary1 Fun_bold where
  liftBGen bgen d = Mk_bold ⟨$⟩ bgen (d − 1)
instance BArbitrary1 Pat_simplify#2 where
  liftBGen bgen d =
    Mk_simplify#2 ⟨$⟩ arbitrary ⟨∗⟩ bgen (d − 1) ⟨∗⟩ bgen (d − 1)
```

Then, having the modular generators for each random construction in place, we can obtain a concrete depth-aware generator (of kind $*$) for any final representation `Fix f` as follows:

```
instance BArbitrary1 f ⇒ BArbitrary (Fix f) where
  barbitrary d = Fix ⟨$⟩ liftBGen barbitrary d
```

There, we use the `BArbitrary1` instance of our representation `f` to generate sub-terms recursively by lifting itself as the parameterized input generator (`liftBGen barbitrary`), wrapping each recursive sub-term with a `Fix` data constructor.

The machinery developed so far lets us generate single random constructions in a modular fashion. However, we still need to develop our generation mechanism a bit further in order to generate composite representations built using the $\oplus$ operator. This is the objective of the next sub-section.

4.2 Encoding Generation Behavior Using Types

As we have seen so far, generating each representation is rather straightforward: there is only one data constructor to pick, and every field is generated using a mechanical recipe. In our approach, most of the generation complexity is encoded in the random generator for composite representations, built upon the $\oplus$ operator. Before introducing it, we need to define some additional machinery to encode the notions of terminal construction and generation frequency.

Recalling the random generator for `Html` presented in Section 2, we can observe that the last generation level (see `gen 0`) is constrained to generate values only from the subset of terminal constructions. In order to model this behavior, we will first define a data type `Term` to tag every terminal construction explicitly:

```
data Term (f :: * → *) r = Term (f r)
```

Then, if `f` is a terminal construction, the type `Term f ⊕ g` can be interpreted as representing data generated using values drawn both from `f` and `g`, but closed using only values from `f`. Since this data type will not add any semantic information to the represented values, we can define suitable `Algebra` and `BArbitrary1` instances for it simply by delegating the work to the inner type:

```
instance Algebra f a ⇒ Algebra (Term f) a where
  alg (Term f) = alg f
instance BArbitrary1 f ⇒ BArbitrary1 (Term f) where
  liftBGen bgen d = Term ⟨$⟩ liftBGen bgen d
```

Worth mentioning, our approach does not require the final user to manually specify terminal constructions—a repetitive task that might lead to obscure non-termination errors if a recursive construction is wrongly tagged as terminal. In turn, this information can be easily extracted at derivation time and included implicitly in our refined type-level idiom, described in detail in Section 5.

The next building block of our framework consists in a way of specifying the generation frequency of each construction. For this purpose, we can follow the same reasoning as before, defining a type-level operator $\otimes$ to explicitly tag the generation frequency of a given representation:

```
data ((f :: * → *) ⊗ (n :: Nat)) r = Freq (f r)
```

This operator is parameterized by a type-level natural number `n` (of kind `Nat`) representing the desired generation frequency. In this light, the type `(f⊗3)⊕(g⊗1)` represents data generated using values from both `f` and `g`, where `f` is randomly chosen three times more frequently than `g`. In practice, we defined $\otimes$ such that it associates more strongly than ⊕, thus avoiding the need for parenthesis in types like the previous one. Analogously as `Term`, the operator ⊗ does not add any semantic information to the values it represents, so we can define its `Algebra` and `BAbitrary1` instance by delegating the work to the inner type as before:

```
instance Algebra f a ⇒ Algebra (f ⊗ n) a where
  alg (Freq f) = alg f
instance BArbitrary1 f ⇒ BArbitrary1 (f ⊗ n) where
  liftBGen bgen d = Freq ⟨$⟩ liftBGen bgen d
```

With these two new type-level combinators, `Term` and ⊗, we are now able to express the behavior of our entire generation process based solely on the type we are generating.

In addition to these combinators, we will need to perform some type-level computations based on them in order to define our random generator for composite representations. Consider for instance the following type—expressed using parenthesis for clarity:

```
(f ⊗ 2) ⊕ ((g ⊗ 3) ⊕ (Term h ⊗ 5))
```

Our generation process will traverse this type one combinator at a time, processing each occurrence of ⊕ independently. This means that, in order to select the appropriate generation frequency for each operand we need to calculate the overall sum of frequencies on each side of the ⊕. For this purpose, we rely on Haskell's type-level programming feature known as *type families* [91]. In this light, we can implement a type-level function `FreqOf` to compute the overall sum of frequencies of a given representation type:

```
type family FreqOf (f :: ∗ → ∗) :: Nat where
  FreqOf (f ⊕ g)    = FreqOf f + FreqOf g
  FreqOf (f ⊗ n)    = n ∗ FreqOf f
  FreqOf (Term f) = FreqOf f
  FreqOf _          = 1
```

This type-level function takes a representation type as an input and traverses it recursively, adding up each frequency tag found in the process, and returning a type-level natural number. Note how in the second equation we multiply the frequency encoded in the ⊗ tag with the frequency of the type it is wrapping. This way, the type ((f⊗2)⊕g)⊗3 is equivalent to (f⊗6)⊕(g⊗3), following the natural intuition for the addition and multiplication operations over natural numbers. Moreover, if a type does not have an explicit frequency, then its generation frequency defaults to one.

Furthermore, the last step of our generation process, which only generates terminal constructions, could be seen as considering the non-terminal ones as having generation frequency zero. This way, we can introduce another type-level computation to calculate the *terminal generation frequency* `FreqOf′` of a given representation:

```
type family FreqOf′ (f :: ∗ → ∗) :: Nat where
  FreqOf′ (f ⊕ g)    = FreqOf′ f + FreqOf′ g
```

```
FreqOf′ (f ⊗ n)    = n ∗ FreqOf′ f
FreqOf′ (Term f) = FreqOf f
FreqOf′ _          = 0
```

Similar to FreqOf, the type family above traverses its input type adding the terminal frequency of each sub-type. However, FreqOf′ only considers the frequency of those representation sub-types that are explicitly tagged as terminal, returning zero in any other case.

Then, using the Term and ⊗ combinators introduced at the beginning of this sub-section, along with the previous type-level computations over frequencies, we can finally define our random generator for composite representations:

```
instance (BArbitrary1 f, BArbitrary1 g)
  ⇒ BArbitrary1 (f ⊕ g) where
  liftBGen bgen d =
    if d > 0
    then frequency
      [(freqVal @(FreqOf f), In_L ⟨$⟩ liftBGen bgen d)
      ,(freqVal @(FreqOf g), In_R ⟨$⟩ liftBGen bgen d)]
    else frequency
      [(freqVal @(FreqOf′ f), In_L ⟨$⟩ liftBGen bgen d)
      ,(freqVal @(FreqOf′ g), In_R ⟨$⟩ liftBGen bgen d)]
```

Like the generator for Html introduced in Section 2, this generator branches over the current depth d. In the case we can still generate values from any construction ($d > 0$), we will use *QuickCheck*'s frequency operation to randomly choose between generating a value of each side of the ⊕, i.e., either a value of f or a value of g, following the generation frequencies specified for both of them, and wrapping the values with the appropriate injection In_L or In_R on each case. Such frequencies are obtained by *reflecting* the type-level natural values obtained from applying FreqOf to both f and g, using a type-dependent function freqVal that returns the number corresponding to the type-level natural value we apply to it:

```
freqVal :: ∀n . KnownNat n ⇒ Int
```

Note that the type of freqVal is ambiguous, since it quantifies over every possible known type-level natural value n. We use a *visible type application* [92] (employing the @(...) syntax) to disambiguate to which natural value we are actually referring to. Then, for instance, the value

```
freqVal @(FreqOf (f ⊗ 5 ⊕ g ⊗ 4))
```

evaluates to the concrete value 9 :: Int.

The **else** clause of our random generator works analogously, except that, this time we only want to generate terminal constructions, hence we use the FreqOf′ type family to compute the terminal generation frequency of each operand. If any of FreqOf′ f or FreqOf′ g evaluates to zero, it means that such operand does not contain any terminal constructions, and frequency will not consider it when generating terminal values.

Moreover, if it happens that both `FreqOf′ f` and `FreqOf′ g` compute to zero simultaneously, then this will produce a runtime error triggered by the function `frequency`, as it does not have anything with a positive frequency to generate. These kinds of exceptions will arise, for example, if we forget to include at least one terminal construction in our final representation—thus leaving the door open for potential infinite generation loops. Fortunately, such runtime exceptions can be caught at compile time. We can define a type constraint `Safe` that ensures we are trying to generate values using a representation with a strictly positive terminal generation frequency—thus containing at least a single terminal construction:

```
type family Safe (f :: * → *) :: Constraint where
  Safe f = IsPositive (FreqOf′ f)
type family IsPositive (n :: Nat) :: Constraint where
  IsPositive 0 = TypeError "No terminals"
  IsPositive _ = ()
```

These type families compute the terminal generation frequency of a representation type `f`, returning either a type error, if its result is zero; or, alternatively, an empty constraint () that is always trivially satisfied. Finally, we can use this constraint to define a safe generation primitive `genRep` to obtain a concrete depth-bounded generator for every target type `a`, specified using a “safe” representation `f`:

```
genRep :: ∀f a . (BArbitrary1 f, Safe f, Algebra f a) ⇒ BGen a
genRep d = eval ⟨$⟩ barbitrary @(Fix f) d
```

Note how this primitive is also ambiguous in the type used for the representation. This allows us to use a visible type application to obtain values from the same target type but generated using different underlying representations. For instance, we can obtain two different concrete generators of our `Html` type simply by changing its generation representation type as follows:

```
genHtml_valid :: BGen Html
genHtml_valid = genRep @Html_valid
genHtml_simplify :: BGen Html
genHtml_simplify = genRep @Html_simplify
```

where $\texttt{Html}_{\texttt{valid}}$ and $\texttt{Html}_{\texttt{simplify}}$ are the representations types introduced in Figure 1b—the syntax used to define them is completed in the next section.

So far we have seen how to represent and generate values for our target data type by combining different random constructions, as well as a series of type-level combinators to encode the desired generation behavior. The next section refines our type-level machinery in order to provide a simple idiom for defining composable random generators.

5 Type-Level Generation Specifications

This section introduces refinements to our basic language for describing random generators, making it more flexible and robust in order to fit real-world usage scenarios.

The first problem we face is that of naming conventions. In practice, the actual name used when deriving the representation for each random construction needs to be generated such that it complies with Haskell's syntax, and also that it is *unique within our namespace*. This means that, type names like $\texttt{Fun}_{\langle+\rangle}$ or $\texttt{Pat}_{\texttt{simplify\#1}}$ are, technically, not valid Haskell data type names, thus they will have to be synthesized as something like `Fun_lt_plus_gt_543` and `Pat_simplify_1_325`, where the last sequence of numbers is inserted by Template Haskell to ensure uniqueness.

This naming convention results hard to use, especially if we consider that we do not know the actual type names until they are synthesized during compilation, due to their unique suffixes. Fortunately, it is easy to solve this problem using some type-level machinery. Instead of imposing a naming convention in our derivation tool, we define a set of open type families to hide each kind of construction behind meaningful names:

```
type family Con (c :: Symbol)
type family Fun (f :: Symbol)
type family Pat (p :: Symbol) (n :: Nat)
```

where `Symbol` is the kind of type-level strings in Haskell. Then, our derivation process will synthesize each representation using unique names, along with a type instance of the corresponding type family, i.e., `Con` for data constructors, `Fun` for interface functions, and `Pat` for functions' patterns. For instance, along with the constructions representations $\texttt{Con}_{\texttt{Text}}$, $\texttt{Fun}_{\langle+\rangle}$ and $\texttt{Pat}_{\texttt{simplify\#1}}$, we will automatically derive the following type instances:

```
type instance Con "Text"         = Term Con_Text_123
type instance Fun "<+>"          = Fun_lt_plus_gt_543
type instance Pat "simplify" 1 = Term Pat_simplify_1_325
```

As a result, the end user can simply refer to each particular construction by using these synonyms, e.g., with representation types like `Con "Text"` $\oplus$ `Fun "<+>"`. The additional `Nat` type parameter on `Pat` simply identifies each pattern number uniquely.

Moreover, notice how we include the appropriate `Term` tags for each terminal construction automatically—namely `Con "Text"` and `Pat "simplify" 1` in the example above. Since this information is statically available, we can easily extract it during derivation time. This relieves us of the burden of manually identifying and declaring the terminal constructions for every generation specification. Additionally, it helps ensure the static termination guarantees provided by our `Safe` constraint mechanism.

Using the type-level extension presented so far, we are now able to write the generation specifications presented in Figure 1b in a clear and concise way.

5.1 Parametric Target Data Types

So far we have seen how to specify random generators for our simple self-contained `Html` data type. In practice, however, we are often required to write random generators for parametric target data types as well. Consider, for example, the following `Tree` data type definition encoding binary trees with generic information of type `a` in the leaves:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

In order to represent its data constructors, we can follow the same recipe presented in Section 3, but also parameterizing our representations over the type variable `a` as well:

```
data Con_Leaf a r = Mk_Leaf a
data Con_Node a r = Mk_Node r r
```

The rest of the machinery can be derived in the same way as before, carrying this type parameter and including the appropriate `Arbitrary` constraints all along the way:

```
instance Algebra (Con_Leaf a) (Tree a) where ···
instance Algebra (Con_Node a) (Tree a) where ···
instance Arbitrary a ⇒ BArbitrary1 (Con_Leaf a) where ···
instance Arbitrary a ⇒ BArbitrary1 (Con_Node a) where ···
```

Then, instead of carrying this type parameter in our generation specifications, we can avoid it by hiding it behind an existential type:

```
data Some (f :: ∗ → ∗ → ∗) (r :: ∗) = ∀(a :: ∗) . Some (f a r)
```

The type constructor `Some` is a wrapper for a 2-parametric type that hides the first type variable using an explicit existential quantifier. Note thus that the type parameter `a` does not appear at the left-hand side of `Some` on its definition. In this light, when deriving any `Con`, `Fun` or `Pat` type instance, we can use this type wrapper to hide the additional type parameters of each construction representation:

```
type instance Con "Leaf" = Term (Some Con_Leaf)
type instance Con "Node" = Some Con_Node
```

As a consequence, we can write generation specifications for our `Tree` data type without having to refer to its type parameter anywhere. For instance:

```
type Tree_Spec = Con "Leaf" ⊗ 2
               ⊕ Con "Node" ⊗ 3
```

Instead, we defer handling this type parameter until we actually use it to define a concrete generator. For instance, we can write a concrete generator of `Tree Int` as follows:

```
genIntTree :: BGen (Tree Int)
genIntTree = genRep @(Tree_Spec ◁ Int)
```

Where ◁ is a type family that simply traverses our generation specification, applying the `Int` type to each occurrence of `Some`, thus eliminating the existential type:

```
type family (f :: ∗ → ∗) ◁ (a :: ∗) :: ∗ → ∗  where
  (Some f) ◁ a = f a
  (f ⊕ g)  ◁ a = (f ◁ a) ⊕ (g ◁ a)
  (f ⊗ n)  ◁ a = (f ◁ a) ⊗ n
  (Term f) ◁ a = Term (t ◁ a)
  f        ◁ a = f
```

As a result, in `genIntTree`, the ◁ operator will reduce the type (`Tree`$_{\texttt{Spec}}$ ◁ `Int`) to the following concrete type:

```
(Term (Con_Leaf Int) ⊗ 2)  ⊕  ((Con_Node Int) ⊗ 3)
```

Worth mentioning, this approach for handling parametric types can be extended to multi-parametric data types with minor effort.

Along with our automated constructions derivation mechanism, the machinery introduced in this section allows us to specify random generators using a simple type-level specification language.

The next section evaluates our approach in terms of performance using a set of case studies extracted from real-world Haskell implementations, along with an interesting runtime optimization.

6 Benchmarks and Optimizations

The random generation framework presented throughout this paper allows us to write extensible generators in a very concise way. However, this expressiveness comes attached to a perceptible runtime overhead, primarily inherited from the use of Data Types à la Carte—a technique which is not often scrutinized for performance. In this section, we evaluate the implicit cost of composing generators using three real-world case studies, along with a type-level optimization that helps avoiding much of the runtime bureaucracy.

Balanced Representations As we have shown in Section 4, the random generation process we propose in this paper can be seen as having two phases. First, we generate random values from the representation types used to specify the shape of our data; and then we use their algebras to translate them to the corresponding values of our target data types. In particular, this last step is expected to pattern match repeatedly against the `In`$_{\texttt{L}}$ and `In`$_{\texttt{R}}$ constructors of the ⊕ operators when traversing each construction injection. Because of this, in general, we expect a performance impact with respect to manually-written concrete generators.

As recently analyzed by Kiriyama et al., this slowdown is expected to be linear in the depth of our representation type [93]. In this light, one can drastically reduce the runtime overhead by associating each $\oplus$ operator in a balanced fashion. So, for instance, instead of writing $(\texttt{f} \oplus \texttt{g} \oplus \texttt{h} \oplus \texttt{i})$, which is implicitly parsed as $(\texttt{f} \oplus (\texttt{g} \oplus (\texttt{h} \oplus \texttt{i})))$; we can associate constructions as $((\texttt{f} \oplus \texttt{g}) \oplus (\texttt{h} \oplus \texttt{i}))$, thus reducing the depth of our representation from four to three levels and, in general, from a $\mathcal{O}(n)$ to a $\mathcal{O}(log(n))$ complexity in the runtime overhead, where n is the number of constructions under consideration.

Worth mentioning, this balancing optimization cannot be applied to the original fashion of Data Types à la Carte by Swierstra. This limitation comes from that the linearity of the representation types is required in order to define *smart injections*, allowing users to construct values of such types in an easy way, injecting the appropriate sequences of $\texttt{In}_\texttt{L}$ and $\texttt{In}_\texttt{R}$ constructors automatically. There, a naïve attempt to use smart injections in a balanced representation may fail due to the nature of Haskell's type checker, and in particular on the lack of backtracking when solving type-class constraints. Fortunately, smart injections are not required for our purposes, as users are not expected to construct values by hand at any point—they are randomly constructed by our generators.

Benchmarks We analyzed the performance of generating random values using three case studies: i) Red-Black Trees (RBT), inspired by Okasaki's formulation [94], ii) Lisp S-expressions (SExp), inspired by the package *hs-zuramaru*[15], and iii) HTML expressions (HTML), inspired by the *html* package, which follows the same structure as our motivating `Html` example. The magnitude of each case study can be outlined as shown in Table 2.

These case studies provide a good combination of data constructors, interface functions and patterns, and cover from smaller to larger numbers of constructions.

Then, we benchmarked the execution time required to generate and fully evaluate 10000 random values corresponding to each case study, comparing both manually-written concrete generators, and those obtained using our modular approach. For this purpose, we used the *Criterion* [95] benchmarking tool for Haskell, and limited the maximum depth of the generated values to five levels. Additionally, our modular generators were tested using both linear and balanced generation specifications. Figure 3 illustrates the relative execution time of each case study, normalized to their corresponding manually-written counterpart—we encourage the reader to obtain a colored version of this work.

[15] `http://hackage.haskell.org/package/zuramaru`

Case Study	#`Con`	#`Fun`	#`Pat`	Total Constructions
RBT	2	5	6	13
SExp	6	-	9	15
HTML	4	132	-	136

Figure 2: Overview of the size of our case studies.

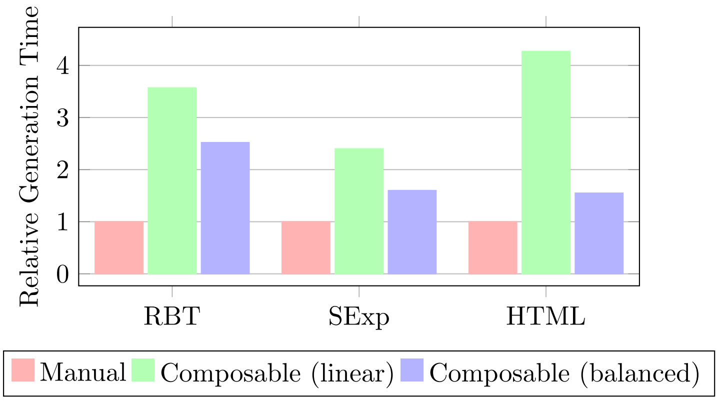


Figure 3: Generation time comparison between manually written and automatically derived composable generators.

As it can be observed, our approach suffers from a noticeable runtime overhead when using linearly defined representations, especially when considering the HTML case study, involving a large number of constructions in the generation process. However, we found that, by balancing our representation types, the generation performance improves dramatically. At the light of these improvements, *our tool includes an additional type-level computation that automatically balances our representations* in order to reduce the generation overhead as much as possible.

On the other hand, it has been argued that the generation time is often not substantial with respect to the rest of the testing process, especially when testing complex properties over monadic code, as well as using random values for penetration testing [71], [85].

All in all, we consider that these results are fairly encouraging, given that the flexibility obtained from using our compositional approach does not produce severe slowdowns when generating random values in practice.

7 Related Work

Extensible Data Types Swierstra proposed Data Types à la Carte [90], a technique for building extensible data types, as a solution for the *expression problem* coined by Wadler [96]. This technique has been successfully applied in a variety of scenarios, from extensible compilers, to composable machine-mechanized proofs [97]–[100]. In this work, we take ideas from this approach and extend them to work in the scope of random data generation, where other parameters come into play apart from just combining constructions, e.g., generation frequency and terminal constructions.

From the practical point of view, Kiriyama et al. propose an optimization mechanism for Data Types à la Carte, where a concrete data type has to be derived for each different composition of constructions defined by the user [93].

This solution avoids much of the runtime overhead introduced when internally pattern matching against sequences of `In`$_L$ and `In`$_R$ data constructors. However, this approach is not entirely compositional, as we still need to rely on Template Haskell to derive the machinery for *each* specialized instance of our data type. In our particular setting, we found that our solution has a fairly acceptable overhead, achieved by automatically balancing our representation types.

Domain Specific Languages Testing properties using small values first is a good practice, both for performance and for obtaining small counterexamples. In this light, *SmallCheck* [68] is a library for defining *exhaustive* generators inspired by *QuickCheck*. Such generators can be used to test properties against *all* possible values of a data type up to a given depth. The authors also present *Lazy SmallCheck*, a variation of *SmallCheck* prepared to use partially defined inputs to explore large parts of the search space at once.

Luck [72] is a domain-specific language for describing testing properties and random generators in parallel. It allows obtaining random generators producing highly-constrained random data by using a mixture of backtracking and constraint solving while generating values. While this approach can lead to quite good testing results, it still requires users to manually think about how to generate their random data. Moreover, the generators obtained are not compiled, but interpreted. In consequence, *Luck*'s generators are rather slow, typically around 20 times slower than compiled ones.

In contrast to these tools, this work lies on the automated side, where we are able to provide lightweight invariants over our random data by following the structural information extracted from the users' codebase.

Automatic Derivation Tools In the past few years, there has been a bloom of automated tools for helping the process of writing random generators.

MegaDeTH [70], [71] is a simple derivation tool that synthesizes generators solely based on their types, paying no attention whatsoever to the generation frequency of each data constructor. As a result, it has been shown that its synthesized generators are biased towards generating very small values [85].

Feat [29] provides a mechanism to uniformly generating values from a given data type of up to a given size. It works by enumerating all the possible values of such type, so that sampling uniformly from it simply becomes sampling uniformly from a finite prefix of natural numbers—something easy to do. This tool has been shown to be useful for generating unbiased random values, as they are drawn uniformly from their value space. However, sampling uniformly may not be ideal in some scenarios, especially when our data types are too general, e.g., using *Feat* to generate valid HTML values as in our previous examples would be quite ineffective, as values drawn uniformly from the value space of our `Html` data type represent, in most cases, invalid HTML values.

On the other hand, *DRAGEN* is a tool that synthesizes optimized generators, tuning their generation frequencies using a simulation-based optimization process, which is parameterized by the distribution of values desired by the user [85]. This simulation is based on the theory of *branching processes*, which models the growth and extinction of populations across successive generations.

In this setting, populations consist of randomly generated data constructors, where generations correspond to each level of the generated values. This tool has been shown to improve code coverage over complex systems, when compared to other automated generator derivation tools.

In a recent work, we extended this approach to generate random values considering also the other sources of structural information covered here, namely abstract interfaces and function pattern matchings [89]. There, we focus on the generation model problem, extending the theory of branching processes to obtain sound predictions about distributions of random values considering these new kinds of constructions. Using this extension, we show that using extra information when generating random values can be extremely valuable, in particular under situations like the ones described in Section 2, where the usual derivation approaches fail to synthesize useful generators due to a lack of structural information. In turn, this paper tackles the representation problem, exploring how a compositional generation process can be effectively implemented and automated in Haskell using advanced type-level features.

In light of that none of the aforementioned automated derivation tools are designed for composability, we consider that the ideas presented in this paper could perhaps be applied to improve the state-of-the-art in automatic derivation of random generators in the future.

8 Conclusions

We presented a novel approach for automatically deriving flexible, composable random generators inspired by the seminal work on Data Types à la Carte. In addition, we incorporate valuable structural information into our generation process by considering not only data constructors, but also the structural information statically available in abstract interfaces and functions' pattern matchings.

In the future, we aim to extend our mechanism for obtaining random generators with the ability to perform stateful generation. In this light, a user could indicate which random constructions interact with their environment, obtaining random generators ensuring strong invariants like well-scopedness or type-correctness, all this while keeping the derivation process as automatic as possible.

Paper V

BinderAnn: Automated Reification of Source Annotations for Monadic EDSLs

Agustín Mista and Alejandro Russo

21st International Symposium on Trends in Functional Programming (2020)

Abstract

Embedded Domain-Specific Languages (EDSLs) are an alternative to quickly implement specialized languages without the need to write compilers or interpreters from scratch. In this territory, Haskell is a prime choice as the host language. EDSLs in Haskell, however, are often incapable of reifying useful static information from the source code, namely variable binding names and source locations. Not having access to variable names directly affects EDSLs designed to generate low-level code, where the variables names in the generated code do not match those found in the source code—thus broadening the semantic gap between source and target code. Similarly, many existing EDSLs produce poor error messages due to the lack of knowledge of source locations where errors are generated.

In this work, we propose a simple technique for enhancing monadic EDSLs expressed using **do** notation. This technique employs *source-to-source plugins*, a relatively new feature of GHC, to annotate every **do** statement of our EDSLs with relevant information extracted from the source code at compile time. We show how these annotations can be incorporated into EDSL designs either directly inside values or as monadic effects. We provide *BinderAnn*, a GHC source plugin implementing our ideas, and evaluate it by enhancing existing real-world EDSLs with relatively minor modification efforts to contemplate the source-level static information related to variables names and source locations.

1 Introduction

Embedded Domain-Specific Languages (EDSLs) are ubiquitous in Haskell. Its powerful type system and extensible syntax are among the reasons making it a very suitable programming language for implementing EDSLs [101]. Especially, monads [102] and monadic **do** notation [103] are part of the programmers' toolbox to implement all sorts of EDSLs. Monadic **do** notation enables users to write domain-specific code in a sequential-like manner that is easy to adopt by programmers not familiar to Haskell's syntax or even functional programming languages.

As a result of being embedded, Haskell EDSLs often lack the ability of reflecting some of the static source information that is intrinsic and available to the host language (Haskell) but not to the guest (the embedded DSL), namely bound names and source locations. These limitations are especially known by designers of EDSLs which generate low-level code, e.g., FeldSpar [104], Ivory [105], or Copilot [106]. In these EDSLs, developers adopted, as the best-case scenario, ad-hoc measures to enforce that variables names in the generated code match those in the host language. In this paper, we instead propose a systematic solution to such problems as a *source-to-source* plugin [107] called *BinderAnn*. We will illustrate the aforementioned limitations of Haskell EDSLs using a series of real-world examples of code generation, while we will show in tandem how our approach can be used to overcome them.

1.1 Motivating examples

We consider as a motivating example the monadic EDSL from the `dotgen` package for generating DOT code[16] from inside Haskell [108]. This EDSL creates new graph nodes and connects them using **do** notation. A simple example of this is shown in Fig. 1a, where we create a graph of the alternating colors of a street semaphore.

Internally, this EDSL sequentially creates a fresh node name for each invocation of the `node` combinator, i.e, `n0`, `n1`, and so on. Then, the corresponding DOT code is generated referring to these generated names, as it is shown in Fig. 1b. Sadly, the generated code does not quite reflect the nature of our particular graph: *sequential names are of little help for interpreting the semantics of the generated code.* To make things worse, this is not a just limitation of this particular EDSL. The variable names to the left of binds ($\leftarrow$) do not belong to an EDSL itself, but to the host language in which it is embedded—thus, such EDSL cannot make use of this useful source information directly.

1.1.1 Common practices

To address this recurrent limitation, some EDSLs resolve in using redundant strings to indicate variable names when synthesizing code [106], [109], [110]. For instance, consider the EDSL for synthesizing C programs via SMT solvers in the `sbv` package [109]. This EDSL enables to express relationships between

[16]DOT is a graph description language used by many open source applications.

```
semaphore = do
  green  ← node
  yellow ← node
  red    ← node
  green  --> yellow
  yellow --> red
  red    --> green
```

(a) EDSL code describing a semaphore color cycle.

```
digraph G
{
 n0; n1; n2;
 n0 -> n1;
 n1 -> n2;
 n2 -> n0;
}
```

(b) Generated code without source information.

```
digraph G
{
 green; yellow; red;
 green -> yellow;
 yellow -> red;
 red -> green;
}
```

(c) Generated code using the BinderAnn plugin.

Figure 1: Enhancing the `dotgen` code generating EDSL with source information.

the inputs and outputs of a function, and based on that, it generates its C body accordingly. Fig. 2a presents a very simple example of this, where we use the `cgInput` combinator to bind the function inputs `"x"` and `"y"` to the Haskell variables `x` and `y`, respectively, and then we specify how the outputs are calculated based on them. In this example, the function will simply return the sum of both inputs (line 5), while storing their difference in the output pointer `"diff"` (line 4). Then, the EDSL will generate the following C code:

```
SInt32 AddSub(SInt32 x, SInt32 y, SInt32 *diff){
  ...
  *diff = x - y;
  return (x + y);
}
```

where `...` simply indicates the rest of the generated code which is not relevant to the point being made here. Notice how the EDSL expects the users to give strings denoting variable names to the expressions they already bind with the *same* variable name but using **do** notation. While this common technique works in practice, this added redundancy requires maintenance and might be hard to keep in sync with the concrete Haskell bind variable names they replicate.

1.2 BinderAnn

In this paper, we present a novel technique to enhance existing (and future) EDSLs with the static information that is missing to generate faithful code, and without relying on redundant string names. In essence, our approach consists of automatically transforming the syntactic representation of our Haskell code to make the static information related to bound names explicitly available to EDSLs. This is now possible due to the recent addition of *source-to-source* plugins [107] to the GHC Haskell compiler.

Recalling our `dotgen` example, our approach can be used to generate DOT code that accurately reflects the one written by the user of the EDSL—see Fig. 1c Furthermore, Fig 2b shows how our approach can simplify the `sbv`

```
genAddSub = do
  x ← cgInput "x"
  y ← cgInput "y"
  cgOutput "diff" (x − y)
  cgReturn (x + y)
```

(a) EDSL code with redundant string names for generating terms.

```
genAddSub = do
  x ← cgInput
  y ← cgInput
  diff ← cgOutput (x − y)
  cgReturn (x + y)
```

(b) Simplified EDSL where names are extracted automatically by BinderAnn.

Figure 2: Avoiding redundant string names in the sbv EDSL via source annotations.

EDSL by not requiring string names to be passed around while generating the same C code.

1.3 Beyond bindings

In practice, bound names are not the only kind of useful static information that can be extracted from EDSL code. Many EDSLs lack descriptive error messages which could be improved by having access to the source locations. To illustrate this point, we consider the EDSL provided by the `shellmate` package for executing shell scripts from Haskell [111]. With this EDSL, we can create computations capturing the output of existing shell commands:

```
cpuinfo = capture (run "cat" ["/proc/cpuinfo"])
meminfo = capture (run "cat" ["/proc/meminfo"])
```

And use them to build complex shell-like scripts:

```
saveInfo = do
  cpu ← cpuinfo
  mem ← meminfo
  output "info.txt" (cpu ++ mem)
```

Let us suppose that we mistype the `"/proc/meminfo"` path. If we run our `saveInfo` script, the mangled path given to the command `cat` will produce a runtime exception that will be captured by the EDSL and printed to the user simply as:

```
Command "cat" failed with error code 1
```

This error message is hardly helpful for debugging the problem of our shell script, especially considering that many functions may be defined in terms of capturing the output of the `cat` command.

By using BinderAnn, it is also possible to extract the exact position in the user code where the error is triggered. In this light, we can enhance this EDSL to support more precise and useful error messages. For instance, the error message above can be improved to:

```
Exception raised at src/MyScript.hs:(3,3):
The value "mem" produced the following error:
Command "cat" failed with error code 1
```

Note how this error message now includes not only the name bound to the problematic command (`mem`), but also its position in the code.

The examples presented so far have motivated the development of BinderAnn to improve the capabilities of monadic EDSLs considerably. To summarize, the contributions of this paper are:

- We propose a simple yet powerful syntactic transformation technique for annotating monadic computations expressed using **do** notation with useful source information (Section 2).
- We propose two different *annotation styles* depending on how EDSLs can consume the static information provided to them, i.e., binding names and source locations (Section 3).
- We extend our simple transformation technique with support for annotating monadic computations returning and pattern matching against tuples, as well as a mechanism for controlling the transformation scope (Section 4).
- We provide an implementation of our ideas, in the shape of a GHC source-to-source plugin called BinderAnn.[17] With our plugin in mind from the beginning, we develop a complete case study from scratch, demonstrating how the ability to reify source information automatically might unlock attractive new features in future EDSLs (Section 5).
- We discuss other possible approaches to fill the static information gap between host and guest embedded languages and their implications. Additionally, we reflect on the limitations of BinderAnn, as well as possible extensions to make it applicable to a larger space of EDSL (Section 6).

2 Generating Source Annotations Using Source Plugins

This section briefly describes *source-to-source plugins* (or source plugins for short), a new mechanism included in the GHC compiler for inspecting and transforming the parsed representation of the compiled code before any other transformation is performed. Moreover, we show how it is possible to take advantage of this mechanism to transparently enhance monadic code written using **do** notation with useful source information.

Essentially, a GHC plugin is a Haskell function that can be inserted into the compilation pipeline to transform the output of the compiled code in different ways [107], [112]. These transformations can alter the compiled code

[17]Available at `https://github.com/OctopiChalmers/BinderAnn`

at different stages, where each stage defines a different interface for its corresponding kind of plugin, dependent on the representation of the code used by the compiler at that point. Historically, this mechanism only allowed plugins to be inserted during type-checking, and after the code was transformed to GHC's Core intermediate representation [113]. Recently, GHC 8.6.1 also added support for plugins to be inserted after parsing and after renaming, and this work focuses on the former kind.

In GHC, the plugin interface is condensed in a record data type `Plugin`, containing a field for each of the transformation stages available. In particular, source-to-source plugins are given by the record field `parsedResultAction` of this data type:

```
data Plugin = Plugin {
  parsedResultAction :: [CommandLineOption] → ModSummary
                           → HsParsedModule → Hsc HsParsedModule
  ...
}
```

This field exposes the interface of a transformation function over the abstract syntax tree of the module under compilation (of type `HsParsedModule`). This abstract syntax tree includes relevant static information not available to the programmer, such as the variable name of every binding, as well as the source location of every syntactic object in the module—two valuable resources that one might want to have access to when implementing EDSLs in Haskell.

Using this interface, we can implement our source plugin by providing a module exporting a value `plugin :: Plugin`, which executes our code transformation:

```
module BinderAnn (plugin) where
import GhcPlugins
plugin :: Plugin
plugin = defaultPlugin {parsedResultAction = <our code here>}
```

Later, our plugin can be enabled by passing the name of its module as a flag to the GHC compiler (`-fplugin=BinderAnn`), or using a compiler-options pragma in the module we want our plugin to transform:

```
{-# OPTIONS_GHC -fplugin BinderAnn #-}
```

The next subsection introduces a simple syntactic transformation procedure based on source plugins for transposing useful static information from the source code representation into the internal state of our EDSLs automatically.

2.1 Enhancing EDSLs with Source Information

We have seen that it is possible to expose static source information from our code using source plugins. However, for our EDSLs to take advantage of this information, we need to transform the user code so that it explicitly communicates this information to the EDSL after our plugin runs at compile time.

In this work, we propose a simple transformation over **do** statements: we will *annotate* each statement with the static information that can be extracted from the parsed representation of the code, which we will simply refer to as a *source annotation*. To achieve this, the first step consists of defining a concrete representation for source annotations, which will be used both by our plugin and by the target EDSLs it annotates. For this purpose, we will rely on a new data type `SrcInfo` to hold the static information relative to a **do** statement:

```
data SrcInfo = Info (Maybe String) (Maybe Loc)
```

This data type stores the name bound to the statement (if any), and the location in the source code where it is defined, being the latter a conjunction of a file path, and a row and column within such file:

```
type Loc = (FilePath, Int, Int)
```

The option type used for the location information in the definition of `SrcInfo` is required to represent the fact that the GHC compiler might not know the specific source location of a statement. A situation that might occur, for instance, if such a statement was automatically generated by another source plugin.

Later, our source plugin can easily populate a source annotation (of type `SrcInfo`) for each **do** statement it finds. However, we still need to insert each annotation into our EDSL in a predictable way. For this purpose, we will define a function `annotateM`, taking a monadic computation and a source annotation, and returning a new monadic computation which internalizes such annotation:

```
annotateM :: Monad m ⇒ m a → SrcInfo → m a
```

Note how this function refers neither to a specific monadic type (`m`) nor to a specific return type of the monadic computation (`a`). This generality lets our plugin blindly transform every **do** statement it finds in the user code in a type-safe manner. To do so, it simply wraps every statement with its static information using our generic annotation function. For instance, our plugin will transform the semaphore example from Section 1 to the following concrete code:

```
semaphore = do
green  ← node    `annotateM` Info (Just "green")  (Just ("Main.hs", 2, 3))
yellow ← node    `annotateM` Info (Just "yellow") (Just ("Main.hs", 3, 3))
red    ← node    `annotateM` Info (Just "red")    (Just ("Main.hs", 4, 3))
green  --> yellow `annotateM` Info Nothing        (Just ("Main.hs", 5, 3))
yellow --> red    `annotateM` Info Nothing        (Just ("Main.hs", 6, 3))
red    --> green  `annotateM` Info Nothing        (Just ("Main.hs", 7, 3))
```

Notice, for instance, how the bound name `red` is reflected in the source annotation for the `red ← node` statement with the value `Just "red"`, whereas the `green --> yellow` statement in the next line is not given any name, which gets represented by the `Nothing` constructor on its corresponding source annotation.

Additionally, each annotation carries the source location within the user code of its corresponding statement—assuming here that the first **do** statement is defined in line number 2 of the file *Main.hs*.

After this transformation is automatically applied, the user will be able to make use of this useful source information, which is now explicit in the source code—and without the burden of maintaining manually written annotations.

Even though this transformation is rather mechanical, the behavior of the annotating function `annotateM` is not trivial, and is subject to *which* types of our EDSLs are expected to be annotated, and *how* the source annotations should be consumed by them. The next section addresses the challenges of implementing this function in depth.

3 Consuming Source Annotations

In the previous section, we demonstrated how it is possible to annotate expressions written using **do** notation with source information via source plugins. Such annotations rely on a generic function `annotateM` to produce the annotation effect. This section explores the details of this function in two possible variants.

Haskell gives the programmer the freedom to implement EDSLs in many ways, depending on the nature of the embedded language. As a consequence, a concrete solution for annotating EDSLs would likely not fit many use cases. In this light, our approach supports two different *annotation styles* that the programmer can use depending on the particular implementations of their EDSLs:

- *Effect-free annotations:* the annotations are stored directly on the values they refer to, e.g, using a specialized data constructor, or an option type.
- *Effect-full annotations:* the annotations are kept in a monadic context as a side effect, e.g., using a mapping from values to annotations inside a state monad.

On one hand, the effect-free style lets us annotate values in place, regardless of the monadic context producing them, which might come in handy if our EDSL defines several monadic types to be used by the end user. On the other hand, the effect-full style lets us insert the source annotations in the monadic context without having to modify the return value of each computation. This style might be useful if our EDSL already carries an internal monadic state, or if the source annotations should not be available to the end user.

Both annotation styles are *independent* of each other and provide different interfaces to interact with BinderAnn. Programmers will then have to choose the most suitable one depending on the nature of their EDSLs, and adapt their code to be able to consume the annotations generated by our plugin.

The rest of this section addresses each annotation style in detail.

3.1 Effect-Free Annotations

The simplest way to annotate a value with source information is given when its type already supports annotations. For instance, suppose that the graph-building EDSL from Section 1 defines graph nodes as having an identifier, and an associative list of attributes as payload:

```
data Node = Node Id [(Attr, Value)]
```

With this in place, the rest of the EDSL combinators can be implemented in terms of nodes as inputs and outputs:

```
node  :: Dot Node
(-->) :: Node → Node → Dot ()
```

where Dot is the main monad defined by this EDSL, whose details are not very relevant for this annotation style. To support generating faithful code, we can extend the definition of the Node data type to also carry an optional field representing the name of each node:

```
data Node = Node Id (Maybe String) [(Attr, Value)]
```

Then, we need to somehow specify that every monadic computation returning a Node should (potentially) be annotated with its bound name. To encode this, we can define a new *type class* [114] Annotated, representing types (of type a) that can be annotated directly:

```
class Annotated a where
  annotate :: a → SrcInfo → a
```

The function annotate simply takes a value and an annotation and returns an annotated value of the same type. Then, we can specify how the source-bound names can be inserted into nodes by giving an appropriate Annotated instance:

```
instance Annotated Node where
  annotate (Node id _ attrs) (SrcInfo name loc) = Node id name attrs
```

where we simply extract the bind name from the source annotation and use it as the node name—for simplicity, we discard the location information here.

Using this type class, we can finally implement our desired annotateM function which transforms **do** statements by unwrapping the return value from the monadic computation and returning the corresponding annotated one:

```
annotateM :: (Monad m, Annotated a) ⇒ m a → SrcInfo → m a
annotateM ma ann = do
  a ← ma
  return (annotate a ann)
```

This is an extensible mechanism that lets us support automatic annotations over the return types of our interest. We simply need to provide an instance of the Annotated type class for every return type of a **do** statement we want to annotate using our plugin.

While simple, this transformation is not safe (yet). Recalling from Section 2, our plugin knows nothing about the return type of a **do** statement. Hence, it transforms every statement it finds under the assumption that this transformation will not produce a type error—as annotateM universally quantifies over any possible return type of the monadic computation it transforms. However, our annotateM function now carries an additional Annotated constraint! In practice, this means that our plugin will break the well-typedness of our EDSL if it happens to find a monadic computation returning a value of a type without an Annotated instance. And even though we could potentially provide an Annotated instance for every type used by our EDSL, a user could always write a statement returning a value of a type not known by our EDSL:

```
x ← return False
```

Here, the lack of an instance for Annotated Bool will break the module during type checking.

To attenuate this problem, we can make every type trivially annotatable by simply discarding the annotation altogether:

```
instance {-# INCOHERENT #-} Annotated a where
  annotate a _ = a
```

This generic instance works as a default trivial annotation method, where any concrete Annotated instance written by the programmer will take precedence against this one [115]. Furthermore, note how this default instance requires to be declared as *incoherent*. This ensures that the type checker will pick a concrete instance written by the user whenever possible, but it will conservatively use the default one in case of an overlapping arising from annotating fully-polymorphic functions—we discuss this in detail in Section 6.4.

3.2 Effect-Full Annotations

EDSLs might also be implemented in a fully stateful manner, where the important data is kept in the monadic context, and the user only gets a reference to handle it. For instance, suppose that our graph-building EDSL from Section 1 does not return nodes directly, but references to them instead:

```
data NodeRef = NodeRef Id
node  :: Dot NodeRef
(-->) :: NodeRef → NodeRef → Dot ()
```

Here, the node payload will be kept in an internal state of the Dot monad defined by the EDSL, which could be defined in terms of a state monad:

```
newtype Dot a = Dot (State DotState a)
data DotState = DotState {
  node_attrs :: Map NodeRef [(Attr, Value)]
}
```

In this case, we might as well want our annotation mechanism to follow the same pattern, inserting the annotations in the monadic context instead of directly in the value they refer to. For this purpose, we can extend our DotState type to also carry the source names given to the bound nodes (if any) using a partial mapping:

```
data DotState = DotState {
  node_attrs :: Map NodeRef [(Attr, Value)],
  node_names :: Map NodeRef String
}
```

Similarly as before, we can define a new type class to specify how to annotate values of different types, except that this time we also need to quantify over the specific monadic context in which the annotation takes place:

```
class Monad m ⇒ AnnotatedM m a where
  annotateM :: m a → SrcInfo → m a
```

Notice that this new type class defines our desired `annotateM` function directly. In contrast to the previously seen `Annotated` type class from the previous subsection, this type class lets us specify how **do** statements can be annotated depending not only on their result type but also on their specific monadic type. In this light, computations returning new node references can be annotated within the `Dot` monad by inserting the bound names in the extended internal state:

```
instance AnnotatedM Dot NodeRef where
  annotateM mref (Info name loc) = do
    ref ← mref
    when (isJust name) $ modify $ λs →
      s {node_names = Map.insert ref (fromJust name) (node_names s)}
    return ref
```

As before, we also need to provide a default instance for our new type class, to ensure that our plugin will not break the well-typedness of the user code:

```
instance {-# INCOHERENT #-} Monad m ⇒ AnnotatedM m a where
  annotateM ma _ = ma
```

All in all, the two annotation styles presented in this section cover a wide variety of EDSL implementation patterns.

4 Extensions

This section describes two useful extensions to our annotation approach that are currently supported by our plugin.

4.1 Annotating Computations Returning Tuples

The syntactic transformation described so far contemplates monadic computations with and without bound names. However, in principle we could only use it to extract the names bound to complete resulting values, i.e, when the pattern at the left-hand side of (←) is a plain variable pattern. In practice, a computation could produce multiple values and return them in a tuple. For instance, suppose that our graph-building EDSL example from Section 1 provides a combinator `nodes` returning multiple new nodes at once:

```
(green, yellow, red) ← nodes
```

For this common programming practice, we would want to insert an annotation for each element of this tuple, following the same pattern as we did before. However, our source annotations can only associate a single name bound to a complete result value of a monadic computation.

Fortunately, we can extend our plugin to support tuple results by inserting a function that lifts our annotation mechanism to each element of the resulting tuple:

```
(green, yellow, red) ← nodes
  `annotateM3`
    (Info (Just "green")  (Just ("Main.hs", 2, 4)),
     Info (Just "yellow") (Just ("Main.hs", 2, 10)),
     Info (Just "red")    (Just ("Main.hs", 2, 18))
```

where `annotateM3` simply extracts each tuple element from the monadic computation, annotates it using the ordinary annotation function, and returns a new tuple containing each annotated value:

```
annotateM3 :: Monad m ⇒ m (a, b, c) → (SrcInfo, SrcInfo, SrcInfo) → m (a, b, c)
annotateM3 mabc (ia, ib, ic) = do
  (a, b, c) ← mabc
  a′ ← return a `annotateM` ia
  b′ ← return b `annotateM` ib
  c′ ← return c `annotateM` ic
  return (a′, b′, c′)
```

It is easy to see how this lifting primitive can be trivially generalized to tuples of any fixed size.

4.2 Specifying the Annotation Scope

By default, our annotation plugin will transform *every* **do** expression present on the module it runs over. Even though a module can contain **do** expressions of different monadic types, we have shown in Section 3 how this transformation can effectively affect only those expressions of the types the user is interested in.

Nonetheless, for a given type to be annotated with source information, a user might still want to limit the scope of the annotations to a certain subset of **do** expressions. To support this, our plugin can also be set to work in a selective mode, where the user specifies which **do** expressions should be transformed.

On one hand, if the target expression is bound to a top-level name, we can use a GHC *annotation pragma* to specify that we are interested in annotating it:

```
{-# ANN semaphore SrcInfo #-}
semaphore = do
  <annotated do statements>
```

This way, BinderAnn will begin by reifying all the annotation pragmas defined in the module, and will proceed to transform only those **do** expression for which a corresponding annotation pragma exists.

However, annotation pragmas can only refer to top-level bindings, limiting the applicability of this technique. In practice, writing **do** expressions at the right hand side of the ($) infix function application operator is quite common. For instance, a user might define a graph and render its DOT code right away:

```
semaphoreCode = showDot $ do
  <do statements>
```

There is no top-level name we can use to specify our plugin to annotate this nested **do** expression. To solve this, we can introduce an *infix annotation operator*. This is, we can replace the infix function application operator ($) with a new syntactic operator, e.g., (|$|), that can be sought within the user's code in order to transform nested **do** expressions:

```
semaphoreCode = showDot |$| do
  <annotated do statements>
```

Then, our plugin will transform every **do** expression at the right hand side of a (|$|) operator to include the appropriate source annotations, replacing it with a normal function application in the process. In practice, the programmer can specify the annotation operator to be any valid infix operator name using a plugin option in BinderAnn (`-fplugin-opt BinderAnn:infix=|$|`).

This gives us the freedom to choose the most appropriate operator according to the nature of the embedded language. Additionally, the infix annotation operator can be defined as a synonym to the actual function application operator:

```
(|$|) :: (a → b) → a → b
(|$|) = ($)
```

This way, the behavior of our code does not change when the plugin is disabled.

The next section develops a complete case study, exploring some interesting features that our plugin enables and can aid in implementing future EDSLs.

5 Case Study: Theorem Proving EDSL

So far we have seen how source annotations can be automatically extracted from the source code using a GHC source plugin (Section 2), as well as consumed by our EDSLs in different ways depending on how they are implemented (Section 3).

Using this approach, we enhanced several existing EDSLs [108], [109], [111], [116] (including the ones presented in Section 1) to support source annotations, obtaining attractive results[18]with relatively small effort.

To demonstrate the full potential of our automated transformation technique, this section introduces a novel case study we designed from scratch having source annotations in mind. In this light, we implemented a simple proof assistant EDSL for propositional logic formulas,[19] based on Coq's [117] tactic style, i.e., our proofs will consist of a series of monadic commands (the tactics) which will manipulate our goals and hypotheses to construct a proof for a given target formula.

Despite not being academically enlightening, this EDSL uses the effect-full annotation style to take advantage of the source information present in the user code, in order to provide useful interactive (modulo recompilation time) proof-state reports—an attractive feature that was not possible to achieve before using monadic EDSLs. To give an example of this, Fig. 3a shows a proof of *Modus ponens* discharged using our EDSL. Firstly, we use the combinator `variables` to create two new propositional variables `p` and `q` (line 3). These variables are used immediately in line 4, where the `proof` combinator establishes the current proof goal (`p` ∧ (`p` ⇒ `q`) ⇒ `q`) and we can proceed to prove it using the **do** expression starting after the ($) operator.

The proof itself uses a series of tactic combinators to progressively manipulate our goal and hypotheses in order to prove our goal. In the first place, we introduce the left-hand side of the top-level implication goal as a new hypothesis named `hand` using the `intro` combinator (line 5), leaving us with the responsibility of proving its consequence, i.e., `q`. From here, we use the `destruct` combinator to split our conjunction hypothesis `hand` into two new hypotheses named `hp` and `hpq`, representing each side of the conjunction (line 6). Having the hypotheses `p` and `p ⇒ q` now in scope, we use the `apply` tactic to eliminate the latter applying it the former, obtaining a new hypothesis `hq` which represents our goal (line 7). Our proof concludes in line 8 by telling the EDSL to use the specific hypothesis `hq` as a proof of our goal, using the `exact` com-

[18]Available at `http://github.com/OctopiChalmers/BinderAnn-examples`
[19]Available at `http://github.com/OctopiChalmers/PropProver`

```
1 modus_ponens :: Proof Prop
2 modus_ponens = do
3   (p, q) ← variables
4   proof (p ∧ (p ⇒ q) ⇒ q) $ do
5     hand ← intro
6     (hp, hpq) ← destruct hand
7     hq ← apply hp hpq
8     exact hq
9     qed
```

(a) A proof of Modus Ponens using **do** notation in our EDSL.

```
At Proofs.hs:(7,5):
1 subgoal left
p, q:  Prop
hand:  p ∧ (p ⇒ q)
hp:  p
hpq:  p ⇒ q
hq:  q
===================
q
```

(b) Proof state using source annotations.

```
Incomplete proof:
1 subgoal left
V0, V1:  Prop
H0:  V0 ∧ (V0 ⇒ V1)
H1:  V0
H2:  V0 ⇒ V1
H3:  V1
===================
V1
```

(c) Proof state using internal names.

Figure 3: User interface of our Coq-like, tactics-based proof assistant EDSL.

binator. The final `qed` command at line 9 simply asserts that the proof given matches the current goal, and returns the proven proposition.

While writing this proof, our EDSL assists the user with a report of the current proof state on each step. For instance, by removing the last tactic we apply (line 8), the corresponding proof state given to the user is the one shown in Fig. 3b. Notice how this report reflects the same variable and hypothesis names introduced by the user in the proof code, i.e., `p`, `q`, `hand`, and so on. Additionally, it indicates the current proof position within our file, which is also used to emit a precise error message whenever some tactic is applied incorrectly—all these features are now possible thanks to our plugin.

To illustrate how helpful this information is for our EDSL, Fig. 3c illustrates the same proof state report we would obtain without reified source annotations (by disabling our plugin for instance). There, both variable and hypothesis names are just printed out using their internal names. Moreover, the current proof-state source position is not available either. Together, these two compromises limit the attractiveness of implementing elegant embedded proof assistants in Haskell.

5.0.1 Implementation

To implement our EDSL, we will start by defining our main monadic data type `Proof` by stacking two monads: a `StateT` transformer to keep an implicit proof state, on top of an `Except` monad to raise and catch proof-related errors:

```
newtype Proof a = Proof (StateT ProofState (Except ProofError a))
```

The most interesting bit here is how we define our proof state. In essence, we will keep a set of propositional variables in scope, along with a stack of subgoals (propositional formulas to construct) and their corresponding context:

```
data ProofState = ProofState {
  ps_vars     :: Set Var,
  ps_subgoals :: [(Prop, Context)]
}
```

Here, variables are represented simply as numbers, whereas contexts are mappings from hypotheses (also represented as numbers) to propositions:

```
newtype Var = Var Int
newtype Hyp = Hyp Int
type Context = Map Hyp Prop
```

Finally, propositions are represented using a simple recursive data type encoding each logical connective:

```
data Prop = Var Var | Prop ∧ Prop | Prop ⇒ Prop | ···
```

The machinery introduced so far is enough to implement the core logic of our EDSL and its proof tactics. However, to take advantage of the source information extracted by our plugin using the effect-full annotation style, we will further extend our proof state with three additional fields to keep track of the source information relevant to our proofs:

```
data ProofState = ProofState {
  ...
  ps_var_names :: Map Var String,
  ps_hyp_names :: Map Hyp String,
  ps_curr_pos  :: Maybe Loc
}
```

These new fields will help us keep track of: the source name given to each propositional variable (introduced by the `variables` combinator); the source name given to each new hypothesis (introduced by our different tactics); and the location in the source code of the last command evaluated by the EDSL (if any).

Then, to connect this internal state to the source annotations generated by our plugin, we need to consider the different result types that each combinator of our EDSL produces. In the first place, our `variables` combinator is used to instantiate new propositional variables (of type `Var`). In this light, we can create an annotation rule (using an `AnnotatedM` instance) to store the source name each variable is given by the user (if any) into the internal names mapping of our proof state:

```
1 instance AnnotatedM Proof Var where
2   annotateM mvar (Info name loc) = do
3     updateCurrentPosition loc
4     var ← mvar
5     when (isJust name) (recordVarName var (fromJust name))
6     return var
```

where `recordVarName` (line 5) inserts the bind name (if any) coming from the source annotation into the internal variable names mapping:

```
recordVarName :: Var → String → Proof ()
recordVarName var name = modify $ λs →
  s {ps_var_names = Map.insert var name (ps_var_names s)}
```

Additionally, the function `updateCurrentPosition` (line 3) simply updates the location in the code of the last command executed by the EDSL (if any):

```
updateCurrentPosition :: Maybe Loc → Proof ()
updateCurrentPosition loc = modify $ λs → s {ps_curr_pos = loc}
```

The next thing we need to consider is how the result of each tactic affects the source information collected in the internal proof state. In principle, proof tactics can return either a new hypothesis (or a tuple of them), when they cause new hypotheses to appear in the proof state, e.g., `intro` or `apply` tactics; or a unit value, when they transform the proof state without introducing any new hypothesis, e.g., the `exact` tactic. With this in mind, we will provide two additional annotation rules to be executed whenever a proof tactic returns either a new hypothesis (of type `Hyp`) or nothing (of type ()):

```
 1 instance AnnotatedM Proof Hyp where
 2   annotateM mhyp (Info name loc) = do
 3     updateCurrentPosition loc
 4     hyp ← mhyp
 5     when (isJust name) (recordHypName hyp (fromJust name))
 6     return hyp
 7 instance AnnotatedM Proof () where
 8   annotateM munit (Info name loc) = do
 9     updateCurrentPosition loc
10     munit
```

The first `AnnotatedM` instance (line 1) will store the source name each hypothesis is given by the user (if any) into the internal proof state—the function `recordHypName` from line 5 works analogously as `recordVarName`. As before, we keep track of the last command evaluated by the EDSL in case of a proof error.

For the case of the second `AnnotatedM` instance (line 7), tactics not producing new hypotheses will not bring new source names to store in the internal proof state. However, this instance makes sure that if such a tactic fails, we have its position logged into our internal proof state in order to report a precise error message (line 9).

With these `AnnotatedM` instances in place, our plugin will seamlessly interact with them, keeping track automatically of source names introduced by the users in their code, as well as the location of each tactic invocation in case of having to report a proof-related error.

6 Discussion

We have presented a simple mechanism based on source plugins for enhancing Haskell EDSLs with source information. This section reflects on other approaches for supporting the extraction of source information without relying on source plugins. Moreover, we discuss limitations and possible extensions to our approach.

6.1 Preprocessing Haskell Code

Our approach is based on transforming the user code adding explicitly some of the useful information that gets lost during compilation. The main advantage of source plugins is that they provide a simple way of doing so without relying on external machinery. Before their existence, achieving the same kind of functionality would have required a substantial amount of effort.

For an overview of other possible (and arguably less pleasant) solutions to this problem, we refer the reader to the work of Dévai et al. [118]. There, the authors propose different indirect techniques for enhancing Haskell EDSLs with static information, e.g., using *cpphs*, the Haskell implementation of the C preprocessor; as well as transforming the Haskell source AST using existing parsers and pretty printers before feeding it to the actual compiler.

6.2 Implementing EDSLs Using QuasiQuotation

In contrast to preprocessing our Haskell code to include static information, it is also somewhat possible to achieve the same goal using meta-programming.

Template Haskell [30] is the Haskell meta-programming framework bundled in the GHC compiler. This tool can be used to inspect the typing information present in the user's codebase and synthesize new code depending on it but, for technical reasons, inspecting term definitions or modifying existing Haskell code is not possible, making this framework unsuitable for implementing a transformation-based approach. Nonetheless, a useful feature of Template Haskell used by many existing EDSLs [105], [119]–[122] is the support for *quasiquotation* [123]. Essentially, quasiquotation allows to embed code written using arbitrary, domain-specific syntax into our Haskell code. To do so, this approach relies on implementing *quasi quoters*, i.e., interpretations from arbitrary strings to their corresponding Haskell expressions:

```
data QuasiQuoter = QuasiQuoter {
  quoteExp :: String → Q Exp,
  ...
}
```

where `Q` is the quasiquotation monad defined by Template Haskell.

Using this approach, it would be possible to implement our Coq-like EDSL from Section 5 as a quasi quoter `coq` :: `QuasiQuoter` accepting concrete Coq syntax. Then, we could use it to embed Coq proofs into our Haskell EDSL using quasiquotation brackets syntax (`[|` ··· `|]`):

```
modus_ponens :: Proof Prop
modus_ponens = [coq|
  Variables P Q.
  Theorem (P ∧ (P → Q) → Q).
    Proof.
    intro hand.
    destruct hand as [hp hpq].
    apply hp hpq as hq.
    exact hq.
  Qed.
|]
```

An advantage of this approach is that the arbitrary code written inside of the quasiquotation brackets has (almost) no syntactic restrictions. Hence, it can be used to embed domain-specific code written using the syntax that fits best the nature of a given EDSL, as opposed to the syntactic restrictions imposed by the use of Haskell syntax and **do** notation—which are exploited by BinderAnn.

However, all this flexibility does not come for free. Implementing a quasiquoter for a language with a novel syntax implies writing a lexer and a parser from a plain string to a Haskell expression—a task that might overcome the benefits of having a new specialized syntax. Moreover, the interaction between quasiquoters and native Haskell code tends to be intricate. In particular, enabling quasiquoters to support embedding native Haskell code inside quasiquotation brackets (something known as *antiquotation*) requires a considerable amount of work and knowledge [123]—without this feature, our quasiquoters can only accept constant EDSL expressions inside the quasiquotation brackets.

Extracting bound names becomes possible using quasi quoters, since, as we mention above, we have access to the literal string written by the user. Source locations, on the other hand, are more tricky to infer. By default, quasiquoters will only be able to recognize source locations relative to where the quasiquotation brackets are interpolated in our Haskell code (line 2 in our example above), difficulting the task of giving the end-user error messages referring to absolute locations within their code.

6.3 Source Annotations for Non-Monadic EDSLs

In this work, we decided to focus only on automatically annotating monadic EDSLs expressed using **do** notation. Although it may seem arbitrary, the reason behind this decision is simple: **do** notation gives us a good level of granularity. Our plugin performs statement-wise transformations, matching the natural notion of having one domain-specific command or instruction per **do** statement. This symmetry lets us annotate EDSL very transparently for the end-user.

On the other hand, there exist many remarkable non-monadic EDSLs written in Haskell and not supporting them by default constitutes a noticeable limitation of our current approach. In principle, we could use the pure annotation style introduced in Section 3 to insert annotations into pure values. However, it is the lack of a well-defined statement structure what complicates deciding *where* to insert source annotations. On one hand, annotating only top-level bindings might be too sporadic for practical purposes, while doing so for every subexpression within a value might blow up the size of our transformed code exponentially, so an acceptable annotation granularity would seem to lay somewhere in between of these two extremes—an intriguing problem to drive our future work.

6.4 Use of Incoherent Instances

As mentioned in Section 3, our approach let us inject source annotations into the values of certain types of interest, and relies on default instances to provide trivial implementations of the annotation functions for any other possible value.

Instead of having to provide concrete annotation instances for each possible type present in the user code, these default instances are a convenient feature that allows doing so on a per-case basis while preserving the type-correctness of the user code after it is transformed by our plugin. Sadly, this convenience has as a limitation that *annotations inserted into fully-polymorphic functions* *will be systematically discarded.* To illustrate this, consider for example the following function that duplicates the output of a monadic computation:

```
twice :: Monad m ⇒ m a → m (a, a)
twice ma = do
  x ← ma
  return (x, x)
```

If written by the user of the EDSL, and then annotated by our plugin, this function will trigger a type error when there exists at least a single more concrete `Annotated` or `AnnotatedM` instance. The reason behind this is simple: while type-checking the annotated statement `x ← ma`, only the default annotation instance is polymorphic enough to match the type of `ma`, however, it cannot be chosen directly, as the existence of other more concrete ones would make this choice inconsistent, e.g, using the default instance even when `twice` is instantiated in the user code with a type that has a more concrete one. Then, declaring our default instances as incoherent loosens this constraint, allowing the compiler to choose the default instance whenever it has to solve an overlap while compiling fully-polymorphic functions like `twice`, but leaving us with the aforementioned limitation as a result of this conservative behavior.

The complexity around the use of overlapping instances is well-known by the Haskell community. In this light, this problem has been solved using more sophisticated approaches relying on type-level programming, e.g., using *closed type families* [124]. Adopting them in our plugin without sacrificing its transparency and ease of use is an ambitious problem that we keep as future work.

7 Conclusions

We developed a simple mechanism to facilitate the automatic extraction of useful source code information that is otherwise lost during compilation. Having access to such information when implementing embedded domain-specific languages is extremely valuable, making it possible to implement attractive features such as faithful code generation and precise error messages. In the past, such features were more complicated if not impossible to achieve without involving undesirable trade-offs like repeated code or quasiquotations.

In the future, we aim to investigate how to extend our approach to a wider set of EDSL programming patterns, especially to those implemented using non-monadic combinators, and for which the use of **do** notation is not available. Additionally, we intend to evaluate how our annotation framework could be extended using generic programming techniques, so programmers should not need to adapt their existing EDSL data type definitions to work with it.

Paper VI

Short Paper: Weak Runtime-Irrelevant Typing for Security

Matthías Páll Gissurarson and **Agustín Mista**

ACM SIGSAC 15th Workshop on Programming Languages and Analysis for Security (2020)

Abstract

Types indexed with extra type-level information are a powerful tool for statically enforcing domain-specific security properties. In many cases, this extra information is runtime-irrelevant, and so it can be completely erased at compile-time without degrading the performance of the compiled code. In practice, however, the added bureaucracy often disrupts the development process, as programmers must completely adhere to new complex constraints in order to even compile their code.

In this work we present WRIT, a plugin for the GHC Haskell compiler that relaxes the type-checking process in the presence of runtime-irrelevant constraints. In particular, WRIT can automatically coerce between runtime equivalent types, allowing users to run programs even in the presence of some classes of type errors. This allows us to gradually secure our code while still being able to compile at each step, separating security concerns from functional correctness.

Moreover, we present a novel way to specify which types should be considered equivalent for the purpose of allowing the program to run, how ambiguity at the type level should be resolved and which constraints can be safely ignored and turned into warnings.

1 Programming with Type Constraints

Enforcing domain-specific properties is a complicated task that developers are forced to carefully address when designing complex systems. In the functional programming realm, strongly-typed languages like Haskell are an advantage since *one can use the type system to enforce domain-specific constraints*! However, this technique is not without flaws. To illustrate some of the issues with this technique, suppose we are writing a library for information-flow control over labeled pure values – loosely inspired by the `MAC` library by Russo [125]. For simplicity, we assume that the only labels are `L` for public and `H` for secret data. Then, we can use *phantom* types [126], [127] to label arbitrary data with security labels:

```
data Label = L | H


newtype Labeled (l :: Label) a = Labeled a
```

As an example, the value `Labeled 42 :: Labeled L Int` represents a public integer, whereas `Labeled "1234" :: Labeled H String` represents a secret string. It is important to note that in Haskell, **newtype**s are representationally equal to the type they wrap, meaning that the runtime representation of `Labeled 42` is the same as the one for `42`. Later, labeled values can be combined according to different security policies using type constraints [114], [115], as an example, we can enforce that no information flows from `H` to `L` by defining the empty type class:

```
class ((l :: Label) ⩽ (l′ :: Label))
```

and defining instances of (⩽) only for the flows we allow:

```
instance (L ⩽ L)
instance (L ⩽ H)
instance (H ⩽ H)
```

Since there is no instance for the forbidden flow `H ⩽ L`, any code that triggers the constraint `H ⩽ L` during compilation will produce a type error. Note that the class (⩽) has no methods, so it is represented by a computationally-irrelevant empty dictionary at runtime.

We can now use (⩽) to implement combinators over labeled values that ensure that secrets do not leak into public data, e.g. the familiar `zip` combinator can be given the type:

```
zip :: (x ⩽ z, y ⩽ z) ⇒ Labeled x [a]
                      → Labeled y [b]
                      → Labeled z [(a, b)]
```

where `(x ⩽ z, y ⩽ z)` ensures that the label `z` of the output is greater or equal to both its inputs. Then, the definition:

```
bad :: Labeled L [(Usr, Pwd)]
bad = zip (Labeled [11111, 222222] :: Labeled L [Usr])
          (Labeled ["hun", "ter2"] :: Labeled H [Pwd])
```

will be rejected by GHC with a generic error indicating that we are missing a type class instance for the forbidden flow:

```
error:  No instance for H <= L (...)
```

and indeed, we can see that there is a leak from the secret passwords in the list `["hun", "ter2"]` to the public list `[(11111, "hun"), (222222, "ter2")]`. Ouch!

As shown so far, we can use Haskell's type system to accommodate domain-specific constraints about security labels using phantom types and type classes. Although this is a powerful strategy when it comes to writing domain-specific libraries [121], [128]–[130], it can be hard to use in practice:

- The code cannot be run unless it is provably secure, preventing users from testing the functional correctness of the program separately from its security properties.
- Users must tag all their data with an explicit `Label`, and cannot use features such as pattern matching without explicitly unwrapping and rewrapping the labels.
- Moreover, they need to tag both the secret and the public data, even though there might exist a sane default tag.
- The type errors are too general and hard to understand for users unfamiliar with Haskell's type system, and;
- Synthesizing type based suggestions [131] becomes harder, due to domain-specific constraints and ambiguous types.

2 Weakening Runtime-Irrelevant Typing

In GHC, type checking is based on constraint-based type inference. Albeit intricate in practice, the algorithm works by traversing the code to accumulate a set of type constraints (defined as part of the type system specification) and then invokes the constraint solver to solve those constraints [132]. In the latest GHC, constraints come in three main flavours [133]:

- *Givens* from type signatures, for which we have evidence,
- *Wanteds* from expressions, for which we want evidence,
- *Deriveds*, which are constraints that any solution must satisfy but we do not require evidence of (e.g. equalities arising from functional dependencies and superclasses).

The constraint solver solves the wanteds with respect to the givens and the typing rules of GHC (which include creating and unifying type variables), making sure that the solution satisfies the deriveds [132], [133]. This process is capable of type-checking complex programs, but is not perfect when it comes to domain-specific constraints like ($\leqslant$).

Luckily, the type checker can be extended with plugins to handle additional type-checking rules, for example, to simplify naturals or invoking an SMT solver [134], [135]. Type checker plugins are invoked by the compiler in order to

1. simplify givens, where a plugin might find a contradiction, and,
2. whenever there are unsolved constraints that the type checker could not solve.

For the purpose of weakening the type checking of runtime-irrelevant types, we developed WRIT,[20] a plugin that extends GHC's type system by adding the rules seen in Figure 1 for when type checking would not be able to proceed otherwise. Users of the plugin can selectively apply these rules to runtime-irrelevant constraints and equalities by writing instances of the `Ignore`, `Discharge`, `Promote`, and `Default` type families [136], [137] as described in the rest of this section.

2.1 Ignoring Runtime-Irrelevant Constraints

In Haskell, users can define empty typeclasses that have no methods (like ($\leqslant$)), which represent runtime-irrelevant constraints. However, we would like to be able to turn these constraints into compile-time warnings, so that functional correctness of the program can be verified separately from its security. The IGNORE rule applies whenever there is an unsolved empty typeclass constraint with an instance of the `Ignore` family:

```
type family Ignore (c :: Constraint) :: Message
```

By defining an instance of the `Ignore` family for ($\leqslant$):

```
type instance Ignore (H ⩽ L) =
  Msg (Text "Found forbidden flow from H to L!")
```

Users can specify that the constraint `H` $\leqslant$ `L` can be ignored with the message shown above. With this instance in scope and WRIT enabled, the error for the `bad` function defined earlier will be turned into the following warning:

```
warning:  Found forbidden flow from H to L!
```

[20]The WRIT plugin is available at `https://github.com/tritlo/writ-plugin`.

2.2 Discharging Runtime-Irrelevant Equalities

With runtime-irrelevant types, we often want to ignore nominal equalities of the form `a ~ b`, which are specially handled GHC primitives. As an example, we might want to turn `L ~ H` into a warning when compiling insecure programs. The Discharge rule applies to unsolved equalities of the form `a ~ b`, for which there is an instance of the `Discharge` family for `a` and `b`:

```
type family Discharge (a :: k) (b :: k) :: Message
```

By defining an instance of `Discharge` for `L` and `H`:

```
type instance Discharge L H =
  Msg (Text "Using a public L as a secret H!")
```

Users can allow `L ~ H` with the message shown above. This in conjunction with ignoring `H ⩽ L` effectively negates any guarantees that our library provides.

2.3 Promoting Representationally-Equivalent Types

A special case of discharging is when `a` and `b` have the kind $(*)$, the kind of base types in Haskell. Discharging the equality `a ~ b` effectively *promotes* `a` to `b`, meaning that `a` is treated as a `b`. This is only runtime-irrelevant when `a` and `b` have the same runtime representation, making `a ~ b` runtime-irrelevant. This coincides with the `Coercible` constraint in GHC [138], so to handle this common case we define `Promote`:

```
type family Promote (a :: *) (b :: *) :: Message
```

And define an instance of `Discharge` for types of kind $(*)$:

```
type instance Discharge (a :: *) (b :: *) =
  OnlyIf (Coercible a b) (Promote a b)
```

Then, by defining an instance of `Promote` for labeled values:

```
type instance Promote a (Labeled l a) =
  Msg (Text "Promoting unlabeled " :<>: ShowType a
    :<>: Text " to " :<>: ShowType (Labeled l a))
```

Users can use any base type `a` (like `Int`) as a `Labeled l a`, where `l` is either `L` or `H`, e.g., it becomes possible to write: `[1, 2] :: Labeled L [Int]`, where `[1, 2]` is promoted and treated as a public `[Int]`.

2.4 Defaulting Runtime-Irrelevant Type Variables

When programming using runtime-irrelevant types, it frequently occurs that the type of a phantom type variable cannot be inferred. However, it is often the case that there is a “sane” value to choose when there are no restrictions, such as the label `L` for labeled data. The Default rule applies whenever there is an unsolved constraint with a free type variable of kind k for which there is an instance of the `Default` family:

```
type family Default k :: k
```

By defining an instance of the `Default` family for `Label`:

```
type instance Default Label = L
```

Users can specify that any free type variables of kind `Label` in an unsolved constraint should be set to `L`.

$$\frac{\Gamma, \mathrm{Default}\, k, a \sim \mathrm{Default}\, k \vdash c : \mathrm{Constraint},\ M}{\Gamma, a : k \in \mathrm{FV}(c), \mathrm{Default}\, k \vdash c : \mathrm{Constraint},\ M \cup \{m_{\mathrm{def}}\}}\ \textsc{Default}$$

$$\frac{\Gamma, \mathrm{Ignore}\, c \vdash \mathrm{Ignore}\, c \sim \mathrm{Msg}\, m,\ M}{\Gamma, \mathrm{Ignore}\, c \vdash c : \mathrm{Constraint},\ M \cup \{m\}}\ \textsc{Ignore}$$

$$\frac{\Gamma, \mathrm{Discharge}\, a\, b \vdash \mathrm{Discharge}\, a\, b \sim \mathrm{Msg}\, m,\ M}{\Gamma, \mathrm{Discharge}\, a\, b \vdash a \sim b,\ M \cup \{m\}}\ \textsc{Discharge}$$

$$\frac{\Gamma \vdash c : \mathrm{Constraint},\ M_c \qquad \Gamma \vdash m_a \sim m_b,\ M}{\Gamma \vdash \mathrm{OnlyIf}\, c\, m_a \sim m_b,\ M_c \cup M}\ \textsc{OnlyIf}$$

Figure 1: The typing rules that WRIT extends GHC's type system with. The judgement Γ, $\mathrm{F}\, a_1 \ldots a_n \vdash c$, M here judges that with an instance $\mathrm{F}\, a_1 \ldots a_n$ in the context the constraint (or equality) c holds with the set of output messages M. Here, we write $c : \mathrm{Constraint}$ to denote a well-formed constraint c, and m_{def} is a compiler-generated message based on the source expression.

Now With Less Cruft! After defining the instances as shown above, WRIT can use them to weaken our library's domain-specific constraints. Users can then easily express and run (possibly insecure) programs operating on labeled values as if they had the underlying type without overhead:

```
labeledOr :: Labeled L [Bool] → Labeled L Bool
labeledOr (x : xs) = if x then True else labeledOr xs
labeledOr _        = True
```

2.5 Ensuring Runtime-Irrelevance

Since it is not always safe to ignore or discharge, we allow users to recover some safety by using the `OnlyIf` constructor, as used above in the `Discharge` instance for $(*)$ to assert `Coercible`. The OnlyIf rule is used to unravel

$\mathrm{F}\, a_1 \ldots a_n \sim \mathrm{Msg}\, m_b$' when $\mathrm{F}\, a_1 \ldots a_n$ reduces to an $\mathrm{OnlyIf}\, c\, m_a$, and adds the additional constraints c and $m_a \sim m_b$ as obligations. This eventually results in an equality of the form $\mathrm{Msg}\, m_a \sim \mathrm{Msg}\, m_b$, causing GHC to unify m_b with m_a, inferring the message to be emitted. Note that $\mathrm{OnlyIf}\, a\, b$ only holds if both a and b hold, and b is only emitted if a holds.

2.6 Turning Type-Errors into Warnings

To model the fact that we often want to turn type errors into warnings, all our rules produce a set of messages, M, which is a union of the messages produced by any obligations. The DISCHARGE and IGNORE rule add a user-defined message to the set, whereas the DEFAULT rule adds a standardized message. The user-defined messages are built using GHC's user type-error combinators, which allows them to use type families to compute the message [139]. The resulting set of messages is reported as warnings at the end of type-checking, or alternatively, as type errors if the user passes the plugin the `keep-errors` flag.

3 Implementation

WRIT operates by examining the wanted and derived constraints passed to the plugin by GHC. Messages are handled as a set of logs with type variables for the messages and their origin. The logs are finalized before they are output, with the type variables representing messages are replaced with the messages themselves.

The plugin applies the DEFAULT rule by generating constraints of the form `a` $\sim$ `Default k` for any free type variable `a` of kind `k` in unsolved constraints, Then, e.g. `Default Label` will reduce to `L`, and the variable `a` is set to `L` in the context. In Haskell, there are two types of type variables, rigid and flexible. Rigid type variables are variables mentioned in the givens, i.e. the constraints. Flexible type variables are type variables instantiated from a $\forall$. For example, in `return` :: `Monad m` $\Rightarrow$ `a` $\rightarrow$ `m a`, `m` is a rigid type variable, while `a` is flexible type variable. When we default a type variable, we must distinguish between rigid and flexible type variables: for rigid type variables, the generated constraints take the form of a given, with assertion from WRIT that `a` is equivalent to `Default Label` as the evidence. For flexible type variables, we do not require evidence, so it suffices to emit a derived to unify `a` with `Default Label`.

For the IGNORE rule, the plugin asserts that the constraint holds, which corresponds to the empty typeclass having an instance. It also emits a constraint that applying the `Ignore` family to the constraint results in a message wrapped in the `Msg` constructor, and adds it to the set of messages as a new type variable that will unify with the message itself.

Similarly for the DISCHARGE rule, WRIT generates a proof by assertion that `a` $\sim$ `b` holds (e.g. `L` $\sim$ `H`), and adds the obligation that $\mathrm{Discharge}\, a\, b$ reduces to a $\mathrm{Msg}\, m$, with the fresh flexible type variable m added to the set of messages M. The evidence is an assertion in the form of a zero-cost coercion [138], which is safe for runtime-irrelevant types which have the same runtime representation.

WRIT applies the OnlyIf rule by generating an assertion that $\text{OnlyIf}\, c\, m_a \sim m_b$ and checking that both c and $m_a \sim m_b$ hold. As an optimization, we solve equalities of the form:

$$\text{OnlyIf}\, c_1\, (\text{OnlyIf}\, c_2\, (\ldots (\text{OnlyIf}\, c_n\;\, \text{Msg}\, m_a))) \sim \text{Msg}\, m_b$$

by checking all the constraints $c_1, \ldots, c_n$ and $\text{Msg}\, m_a \sim \text{Msg}\, m_b$, causing GHC to unify m_a with m_b.

4 Conclusions and Future Work

We presented WRIT, a type-checker plugin for GHC to weaken the type-checking process for runtime-irrelevant constraints and representationally-equivalent types. We believe our work will facilitate developers to adopt more secure programming practices in Haskell with less overhead, since it is now possible to start doing so in a more gradual manner. As this is a work in progress, there are a few avenues for future work:

Safety The WRIT plugin gives users a lot of freedom and allows them to override the typing rules used in Haskell. We have yet to investigate which rules can be safely defined by the user, what can go wrong if they define an invalid rule, and whether we can prevent users from defining such rules.

Overlaps Neither the compiler plugin nor the formalization deal with what happens when the user-defined instances overlap, which can cause the typing rules of WRIT to overlap and it is unclear which one to choose. In the plugin itself, this is handled by preferring Discharge to Ignore and Ignore to Default. It is clear however that the choice should not affect the semantics of the compiled program (something yet to be proven), but which typing rule is preferred can affect the errors or warnings emitted in the process. One possibility is to design a heuristic that selects the most specific typing rule applicable, to emit more concrete (and useful) messages, as opposed to more generic ones.

Dynamic and Gradual Typing We want to investigate how relaxing the type-checking process could interact with Haskell's dynamic typing capabilities [140]. Whenever the type checker finds two expressions producing a type mismatch error, it might be possible to promote them both to Haskell's dynamic representation, `Dynamic`. In this light, the invalid list expression [`42`, `"hello"`] could be promoted to a list of dynamic values by promoting both `42` and `"hello"` to a unified dynamic representation, i.e. [`42`, `"hello"`] :: [`Dynamic`]. Then, dynamically typed values could be demoted to concrete types via runtime checks inserted automatically. This mechanism could shorten the gap between Haskell, a strongly typed language, and dynamically typed languages like Python or Erlang by simply toggling a compiler plugin, enabling us to do module-based *gradual* typing [141].

Paper VII

MUTAGEN: Reliable Coverage-Guided, Property-Based Testing using Exhaustive Mutations

Agustín Mista and Alejandro Russo

16th IEEE International Conference on Software Testing, Verification and Validation (2023)

Abstract

Automatically-synthesized random data generators are an appealing option when using property-based testing. There exists a variety of techniques that extract static information from the codebase to produce random test cases. Unfortunately, such techniques cannot enforce the complex invariants often needed to test properties with sparse preconditions.

Coverage-guided, property-based testing (CGPT) tackles this limitation by enhancing synthesized generators with structure-preserving mutations guided by execution traces. Albeit effective, CGPT relies largely on randomness and exhibits poor scheduling, which can prevent bugs from being found.

We present MUTAGEN, a CGPT framework that tackles such limitations by generating mutants *exhaustively*. Our tool incorporates heuristics that help to minimize scalability issues as well as cover the search space in a principled manner. Our evaluation shows that MUTAGEN not only outperforms existing CGPT tools but also finds previously unknown bugs in real-world software.

1 Introduction

Random Property-Based Testing (RPBT) is a popular technique for finding bugs using executable testing properties [16], [22], [142]–[144]. A practical limitation of RPBT is the need for random data generators used to instantiate the testing properties, and writing highly-tuned generators can take several thousand person-hours of trial and error [34]. Luckily, there exist several approaches that automatically synthesize random data generators by extracting static information from the codebase, e.g., data type definitions and application public interfaces (APIs). [29], [70], [75], [77], [85], [89]. These approaches, however, are unable to synthesize generators capable of producing data satisfying complex invariants not easily derivable from the codebase. Generating random valid programs to test compilers is a clear example of this limitation [145], where developers are forced to write specialized generators by hand [65], [146], [147].

Coverage-Guided, Property-Based Testing (CGPT) [34] is a technique that borrows ideas from the fuzzing community to generate highly-structured values while still using automatically derived generators. CGPT keeps queues of *interesting* previously executed test cases that can be transformed using structure-preserving mutations to produce new ones. Intuitively, mutating an existing interesting test case is more likely to produce a new interesting test case than generating a new one from scratch. Moreover, unlike the generic bit-level mutators often used by the fuzzing community [5], [7], structure-preserving mutations specified at the data type level can effectively produce only syntactically valid mutants. Such an approach has shown to be effective when fuzzing systems accepting structurally complex inputs [9], [148], [149]. Notably, CGPT uses the data type information of the inputs to the testing properties to derive specialized structure-preserving mutators directly and without the need for external grammars — making strongly-typed programming languages an ideal match for this technique. In addition, CGPT relies on execution traces to distinguish interesting test cases — a technique popularized by *coverage-guided fuzzers* like *AFL* [3]. Here, test cases are interesting (and therefore worth mutating) only when they exercise new parts of the code in the system under test.

In this work, we establish several aspects of the seminal CGPT approach by Lampropoulos *et al.* that leave room for improvement (see Section 2). In particular: i) if not done carefully, automatically derived structure-preserving mutators can become "shallow", unlikely to transform deep test cases more than superficially; ii) the queuing mechanism can cause delays if interesting test cases are enqueued frequently and there is no way to prioritize them; and iii) the heuristic used to assign a "mutation budget" to each interesting test case (often referred to as a *power schedule*) requires fine tuning and can be hard to generalize. Overcoming these obstacles is important to make CGPT more suitable for testing real-world software.

To tackle these limitations, we introduce Mutagen, a CGPT framework that applies mutations *exhaustively* (see Section 3). That is, given an interesting test case, our tool forces every structure-preserving mutation that can be applied to it to be evaluated exactly once. This has two main advantages. Firstly, every subexpression of the input test case is mutated on the same basis, ensuring that deep transformations are not omitted due to randomness. Moreover, computing mutations exhaustively eliminates the need for a heuristic power schedule.

Internally, MUTAGEN distinguishes two kinds of mutations. On one hand, *deterministic (pure) mutations* encode transformations that yield a single mutated test case obtained by swapping data constructors around, as well as rearranging or returning subexpressions. On the other hand, *non-deterministic (random) mutations* are used to represent transformations over large enumeration types. This mechanism let us selectively escape the scalability issues of exhaustiveness by yielding a random generator that replaces a specific subexpression of an input test case with a randomly generated one. This generator is later sampled a *relatively small* number of times. This way MUTAGEN avoids, for instance, mutating every number inside a test case into every other number of its range.

MUTAGEN's testing loop incorporates two novel heuristics that help finding bugs more reliably (Section 4). In the first place, our tool uses last-in-first-out (LIFO) scheduling with priority when enqueueing interesting test cases for mutation. This way, interesting test cases that discover larger parts of untested code are given a higher priority. Moreover, LIFO scheduling allows the testing loop to jump back and forth between enqueued test cases as soon as new more interesting ones become available, eliminating potential delays when the mutation queues grow more often than they shrink.

The second heuristic controls the number of test cases sampled from random mutations by monitoring how often we generate interesting test cases. Whenever this frequency stalls, MUTAGEN resets the testing loop and increases the effort put into sampling random mutations. This way, our tool automatically adjusts this parameter on the fly.

We validated our ideas in two different ways. We first compared MUTAGEN against *FuzzChick*, the reference CGPT implementation by Lampropoulos *et al.*, on all the existing cases studies described in their original work. These case studies focus on finding counterexamples for buggy variations of two Information-Flow Control (IFC) machines of different complexity. Our results (Section 6) indicate that: when bugs are relatively easy to find, MUTAGEN can reliably find them faster than *FuzzChick*. On the other hand, when bugs are harder to find, our tool outperforms *FuzzChick* in terms of failure rate at the cost of (possibly) needing more time to find them. Notably, MUTAGEN is capable of finding bugs that *FuzzChick* was not able to find in our evaluation nor in its original one.

Additionally, we compared MUTAGEN against *QuickCheck* [16], the most widely used RPBT tool in Haskell, on an existing WebAssembly engine implementation of industrial strength. There, MUTAGEN is capable of reliably finding 15 planted bugs in the validator and interpreter, as well as 3 previously unknown ones. Moreover, this case study lets us evaluate the performance versus the overhead of our tool (and its custom code instrumentation mechanism). All in all, our evaluation indicates that *testing mutants exhaustively together with our heuristics to escape scalability issues* can be an appealing technique for finding bugs reliably without sacrificing speed.

We additionally present threats to validity in Section 7 and discuss related work in Section 8, to finally conclude in Section 9.

2 Background

This section briefly introduces the motivation, ideas and limitations behind CGPT [34]. To illustrate it, we focus on a simple property defined over binary trees. Such a data structure can be defined in Haskell with a custom data type with two data constructors for leaves and branches respectively:

```
data Tree a = Leaf a | Branch (Tree a) a (Tree a)
```

The type parameter `a` indicates that trees can be instantiated using any type as payload, so the value `Leaf True` has type `Tree Bool`, whereas the value `Branch (Leaf 1) 2 (Leaf 3)` has type `Tree Int`. If we assume the existence of a function `balanced` of type `Tree a → Bool` that asserts that a tree satisfies some notion of balancedness, we can write properties to validate that the operations defined over binary trees preserve this invariant. For instance, to validate the implementation of an `insert` function, we assert that, given an element `x` and a balanced tree `t` as input, inserting `x` into `t` will produce a balanced tree as output:

```
prop_insert :: a → Tree a → Property
prop_insert x t = balanced t ⇒ balanced (insert x t)
```

(The definitions of `balanced` and `insert` are not important here.) The arrow operator (⇒) indicates that `balanced t` is a precondition of this property, so test cases where the input tree is unbalanced will get *discarded* prematurely.

The only missing piece is a random generator of trees. For this, we can define a naïve generator for trees of integers as:

```
genTree (size) = if size == 0
  then do {x ← genInt; return (Leaf x)}
  else oneof [do {x ← genInt; return (Leaf x)},
              do {l ← genTree (size − 1);
                  x ← genInt;
                  r ← genTree (size − 1);
                  return (Branch l x r)}]
```

This definition (simplified to make it more accessible) follows a common type-directed approach used by some existing generator synthesizer tools. At each step, `genTree` picks a `Tree` data constructor with uniform probability, and calls itself to generate recursive subexpressions, carefully reducing the input size limit `size` by a unit at a time. This ensures termination by generating only leaves when the size reaches zero (case `size == 0`). Integers payloads are generated by calling an external random generator (`genInt`) defined elsewhere.

Readers familiar with RPBT will notice that `genTree` is not suitable for testing `prop_insert` with *QuickCheck*, as this generator produces mostly unbalanced trees which do not satisfy the property's precondition, thus leaving its postcondition (`balanced (insert x t)`) largely untested.

2.1 Coverage-Guided Property-Based Testing

CGPT alleviates the problem of testing properties with non-trivial preconditions while using automatically derived generators by enhancing the testing process with: i) *target code instrumentation*, to capture execution information from each test case; and ii) *high-level, structure-preserving mutations*, to produce syntactically valid test cases from existing ones.

Using code instrumentation in tandem with mutations is a well-known technique in the fuzzing community [3], [4], [31]–[33]. Notably, CGPT can additionally use the result of the testing properties' preconditions to distinguish semantically valid test cases from invalid ones. This is useful to favor mutating valid test cases over discarded ones.

The CGPT testing loop uses two queues to store valid and discarded previously executed test cases along with a mutation budget that controls how many times they can be mutated before being finally thrown away. This budget is calculated using a heuristic derived from AFL's power schedule, i.e., more budget to test cases that lead to shorter executions, or that discover more parts of the code. On each iteration, the testing loop selects the next test case by mutating the first value on the queue of valid test cases. If such queue is empty, it mutates the first test case from the queue of discarded test cases. If both queues are empty, CGPT generates a new random value from scratch. The loop then runs this test case and evaluates whether it was interesting. If the test case was interesting, it gets enqueued into its corresponding queue (either valid or discarded), This process alternates between random generation and mutation until a bug is found or the test limit is reached.

Limitations of CGPT Lampropoulos *et al.* compared the mean-time-to-failure (MTTF) of CGPT against random testing using both automatically derived generators and manually-written ones, where their results show that CGPT lies in between these two approaches. While MTTF is a useful global metric, we argue that a meticulous evaluation ought to consider failure rate, i.e., the ability to find a bug in a given run as an important metric when comparing PBT tools. After repeating each original experiment 10 times, we observed that *FuzzChick* was only able to find 7 (out of 20) and 18 (out of 33) bugs with 100% failure rate in the two IFC machine case studies. Notably, *FuzzChick* was unable to find any counterexample for 3 of the planted bugs. With this observation in mind, we consider three aspects to tackle CGPT's reliability issues:

- *Mutators distribution:* for simplicity, the mutators proposed by Lampropoulos *et al.* are derived to follow a top-down approach: mutations can happen at the top level or be recursively applied to an immediate subexpression of the input test case with approximately the same probability. This makes deep recursive mutations very unlikely, as their probability decreases multiplicatively with each recursive call. Hence, these mutators can only effectively transform shallow test cases, excluding scenarios involving deeply nested data structures. *Ideally, mutations should happen on every subexpression of the input test case on a reasonable basis.*

- *Scheduling:* CGPT uses single-ended queues to store valid and discarded interesting test cases, where new test cases are placed *at the end* of their corresponding queue. If a test case discovered a whole new portion of the target code, it will not be mutated until the rest of the queue ahead of it gets processed, limiting the effectiveness of the testing loop whenever queues grow more often than they shrink. In an extreme case, interesting test cases might not get processed at all within the testing budget. *Thus, we should prioritize mutating novel test cases right away.*

- *Power schedule:* it is unclear how well this heuristic assigns a budget to each interesting test case. On one hand, assigning too much budget to not-so-interesting wastes precious testing time. On the other hand, assigning too little budget to interesting test cases might prevent bugs from being discovered at all! Finding a balanced heuristic can be quite challenging in the general case. *Ideally, the scheduling mechanism should be as unbiased as possible.*

3 Mutagen

This section describes the main ideas behind MUTAGEN, our revised CGPT tool written in Haskell.[21]

MUTAGEN works by mutating test cases in an exhaustive and precise manner, where i) each subexpression of a test case is associated with a set of structure-preserving mutations, and ii) each one of these mutations is scheduled *exactly once.* We realized that, by using an exhaustive mutation approach, we avoid needing a heuristic power schedule to assign a budget to each interesting test case. Moreover, computing mutants exhaustively ensures that interesting mutations are not omitted or overly exercised due to randomness. This approach is inspired by exhaustive bounded testing tools like *SmallCheck* [68] or *Korat* [150] — refer to Section 8 for a detailed discussion.

3.1 Exhaustive Mutations

In MUTAGEN, mutators are defined as the set of mutants that can be obtained by transforming the input test case *at the top-level* (the root data constructor). For a given type `a`, we represent a mutator of `a`'s with a function from `a`'s to a list of mutants. In Haskell, we introduce the type synonym:

```
type Mutator a = a → [Mutant a]
```

As mentioned earlier, concrete mutants can be obtained either from a pure or a random mutation, which we define as follows:

```
data Mutant a = PURE a | RAND (Gen a)
```

We first focus on pure mutants, which encode deterministic transformations over the *outermost* data constructor of the input — recursive mutations will be introduced soon.

[21]Although we make use of Haskell's powerful type system, our ideas should apply to other statically typed languages with minor effort.

```
mutate t = case t of
  Leaf x → [
    PURE (Branch def x def)
  ]
  Branch l x r → [
    PURE l, PURE r,
    PURE (Leaf x),
    PURE (Branch l x l),
    PURE (Branch r x r),
    PURE (Branch r x l)
  ]
```

Figure 1: MUTAGEN `Tree` mutator.

These transformations can either i) return an immediate subexpression of the same type as the input, or ii) swap the outermost data constructor with a (possibly) different one of the same type, reusing the immediate subexpressions of the input in any combination that produces a well-typed value.

Fig. 1 illustrates a mutator for the `Tree` data type. This definition simply enumerates mutants that transform the outermost data constructor. Moreover, notice how a default value `def` used to fill the subtrees when "growing" a leaf into a branch. In practice, `def` corresponds to the simplest expression we can construct for the mutant to be type-correct. In our example, the default `Tree` value is a leaf containing the smallest value of the payload type, e.g., `Leaf 0` is the default value of `Tree Int`. Using a small default value, as opposed to a randomly generated one (as done by the original CGPT) is also inspired by exhaustive bounded testing tools, and avoids introducing unnecessary randomness when growing data constructors.

Formally, for a type T defined in terms of the data constructors C_i, each one with fields of (possibly different) types t^i_j:

$$T \ := \ C_1\ t^1_1\ t^1_2\ ...\ |\ C_2\ t^2_1\ t^2_2\ ...\ |\ ...$$

MUTAGEN synthesizes the `mutate` function so it pattern matches on the root data constructor of the input as follows:

$$\texttt{mutate}(C_i\, x_1\, x_2\, ...) = \texttt{mut}_\texttt{r}(C_i\, x_1\, x_2\, ...) \cup \texttt{mut}_\texttt{s}(C_i\, x_1\, x_2\, ...)$$

Firstly, $\texttt{mut}_\texttt{r}$ computes the set of possible mutations that return an immediate subexpression of the same type as the input:

$$\texttt{mut}_\texttt{r}(C_i\, x_1\, x_2\, ...) = \{\texttt{PURE}\, x_k \mid x_k \in \texttt{filter}(T, \{x_1,\, x_2,\, ...\})\}$$

Then, $\texttt{mut}_\texttt{s}$ builds every mutation that swaps the root data constructor with a (possibly) different one, reusing (or defaulting to) compatible subexpressions whenever possible:

$$\texttt{mut}_{\texttt{s}}(C_i\, x_1\, x_2 \ldots) = \left\{ \texttt{PURE}(C_j\, x'_1\, x'_2 \ldots) \;\middle|\; \begin{array}{l} C_j \in \{C_1, C_2, \ldots\} \\ x'_k \in \overline{\texttt{filter}}(t^j_k, \{x_1,\, x_2,\, \ldots\}) \\ C_j\, x'_1\, x'_2 \ldots \;\neq\; C_i\, x_1\, x_2 \ldots \end{array} \right\}$$

The helper `filter` simply returns the subset of the input values X that match the type t, whereas $\overline{\texttt{filter}}$ returns the default value of the type t ($\texttt{def}_t$) if the result of `filter` is empty:

$$\texttt{filter}(t, X) = \{x \mid x \in X, \texttt{typeof}(x) = t\}$$

$$\overline{\texttt{filter}}(t, X) = \begin{cases} \texttt{filter}(t, X) & \texttt{if}\ \texttt{filter}(t, X) \neq \emptyset \\ \{\texttt{def}_t\} & \texttt{otherwise} \end{cases}$$

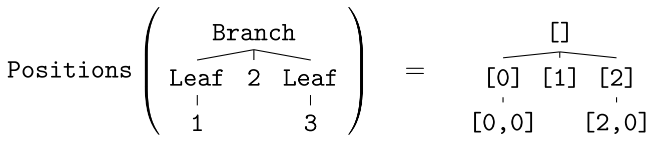
This ensures that a small constructor can always be grown into a larger one by inserting default subexpressions whenever needed. (Recalling the `Tree` mutator from Fig. 1, we show this for the case of mutating a `Leaf` into a `Branch`.)

We can finally focus on random mutants, which let us *selectively avoid exhaustiveness* when mutating values of large enumeration types (e.g. numbers). Instead of creating a `PURE` mutant for every numerical subexpression exhaustively, we condense them into a generator that can be sampled to produce new random values. This way, a mutator for integers becomes:

$$\texttt{mutate}\,\texttt{n} = [\texttt{RAND}\ \texttt{genInt}]$$

This approach allows MUTAGEN to control the amount of effort put into mutating any subexpression of an input test case associated with a random mutation. This can avoid dedicating unnecessary effort to mutating data payloads when the execution of the testing property or the system under test is independent of their values (see Section 4).

Mapping top-level mutations everywhere So far we have defined mutations that transform only the root node of the input. To apply these mutations to every subexpression we use two utility functions. Firstly, a function `Positions` traverses the input and builds a Rose tree [151] of *mutable positions*, i.e., lists of indices encoding the path from the root to every mutable subexpression. For instance, the mutable positions of the value `Branch (Leaf 1) 2 (Leaf 3)` are:



Then, we define a function `MutateInside` that takes a desired position within an input test case and mutates its corresponding subexpression, returning

Algorithm 1: MUTAGEN Testing Loop

```
Function Loop(P, N, R, gen):
    i ← 0
    TLog, QValid, QDiscarded ← ∅
    while i < N do
        x ← Pick(QValid, QDiscarded, gen)
        (result, trace) ← WithTrace(P(x))
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            if not Discarded(result) then
                batch ← CreateMutationBatch(x, R)
                Enqueue(QValid, batch)
            else if not Discarded(Parent(x)) then
                batch ← CreateMutationBatch(x, R)
                Enqueue(QDiscarded, batch)
        i ← i+1
    return Ok
```

a list of mutants. This function traverses the desired position, calling itself recursively until it reaches the desired subexpression, where a mutation encoded by `mutate` can be applied directly. The definition of these functions consists of boilerplate code that our tool synthesizes automatically, thus we omit them to preserve space.

3.2 Testing loop

We now introduce the base testing loop of MUTAGEN, outlined in Algorithm 1. Like in CGPT, we use two queues, QValid and QDiscarded to store valid and discarded interesting test cases, respectively. Our tool precomputes all the mutations of a given test case before enqueueing them. These mutations are put together into lists we call *mutation batches* — one for each mutated test case. To initialize a mutation batch (outlined in Algorithm 2), we first flatten all the mutable positions of the input test case in level order. Then, we iterate over all these positions, retrieving all the mutants associated to each corresponding subexpression. For each one of these: i) if it is a pure mutant carrying a concrete mutated value, we enqueue it into the mutation batch directly; otherwise ii) it is a random mutant that carries a random generator with it, in which case we sample and enqueue R random values using this generator, where R is a parameter set by the user. At the end, we simply return the accumulated batch.

Then, the seed selection algorithm (Algorithm 3) picks the next test case using the same criteria as CGPT, prioritizing valid test cases over discarded ones, falling back to random generation when necessary. For this, we simply pick the next mutated test case from the current precomputed batch, jumping to the next batch in line when the current one becomes empty.

Algorithm 2: Mutation Batch Initialization

```
Function CreateMutationBatch($x$, $R$):
    batch ← ∅
    for pos in Flatten(Positions(x)) do
        for mutant in MutateInside(pos, x) do
            switch mutant do
                case PURE $\hat{x}$ do
                    Enqueue($\hat{x}$, batch)
                case RAND gen do
                    repeat R times
                        $\hat{x}$ ← Sample(gen)
                        Enqueue($\hat{x}$, batch)
    return batch
```

Algorithm 3: MUTAGEN Seed Selection

```
Function Pick($QValid$, $QDiscarded$, $gen$):
    if not Empty(QValid) then
        batch ← Deque(QValid)
        if Empty(batch) then  Pick(QValid, QDiscarded, gen)
        else
            PushFront(QValid, Rest(batch))
            return First(batch)
    if not Empty(QDiscarded) then
        batch ← Deque(QDiscarded)
        if Empty(batch) then  Pick(QValid, QDiscarded, gen)
        else
            PushFront(QDiscarded, Rest(batch))
            return First(batch)
    else return Sample(gen)
```

Having selected the next test case, the testing loop proceeds to execute it, capturing both the result (valid, discarded, or failed) and its execution trace. If the test case fails, it is reported as a bug. If not, the algorithm evaluates if it was interesting based on its trace information and the one from previously executed test cases (represented by TLog). If the test case was interesting, its mutants are precomputed and enqueued on its corresponding queue. This process is repeated until finding a bug or reaching the test limit N.

A notable difference with CGPT's testing loop is the criterion for enqueuing discarded tests. We found that, especially for large data types, the queue of discarded candidates tends to grow disproportionately large, making it hardly usable while consuming large amounts of memory. To improve this, we resort to mutating discarded tests cases only when we have some evidence that they are "almost valid." For this, each mutated test case remembers whether its parent (the original test case they derive from) was valid. Then, we enqueue discarded test cases only if they descend from a valid parent (see Algorithm 1). This way we fill the discarded queue with lesser but more interesting test cases.

4 Mutagen Heuristics

In this section we introduce two heuristics implemented on top of the base testing loop of Mutagen described in Section 3.

4.1 Priority LIFO Scheduling

This heuristic tackles the issue of enqueuing new interesting test cases at the end of possibly long queues of not-so-interesting ones. For this, Mutagen captures the execution trace of each test case and computes its novelty relative to previously executed ones with respect to their *edge coverage*, i.e, test cases that discover new edges in the system under test are considered interesting, and their priorities are proportional to the number of edges they discovered.

Using this mechanism, we can modify Mutagen's base testing loop replacing each mutation queue with a priority queue indexed by the novelty of their test cases. These changes are illustrated in Algorithm 4. Statements in red indicate important changes to the base algorithm, whereas ellipses denote parts of the code that remain unchanged.

To pick the next test case, we retrieve the first one with the highest priority. Then, when we find a new interesting test case, it gets enqueued at the *beginning* of the queue of its corresponding priority. This allows the testing loop to jump immediately onto mutating new interesting test cases as soon as they are found (even at the same priority), and to jump back to previous test cases as soon as mutants become less novel.

4.2 Tuning Random Mutations Parameter

As introduced in Section 3, our tool is parameterized by the number of times it samples the random generators associated with random mutations (R). But, how many test cases should we sample? Answering this question precisely can be challenging, so this second heuristic aims to alleviate the problem.

We found that the smaller the number of times we sample from random mutations, the easier it is for the trace log that records executions to get saturated, i.e., when interesting test cases stop getting discovered or are discovered very seldom. We realized that we can use this information to dynamically adapt the number of times we sample from random mutations. This idea is described in Algorithm 5. The process is as follows: i) we start the testing loop with the R parameter set to one, and ii) each time we find that a test is not interesting (i.e. boring), we increment a counter. Then, iii) if we have not produced any interesting test case after a certain number of tests (1000 tests seems to be a reasonable threshold in practice), we duplicate the number of random mutations and the threshold. Additionally, we reset the trace log so interesting test cases found on a previous iteration can be found

Algorithm 4: Priority LIFO Heuristic

```
Function Loop(P, N, R, gen):
    ...
    x ← Pick(QValid, QDiscarded, gen)
    (result, trace) ← WithTrace(P(x))
    ...
    if Interesting(TLog, trace) then
        if not Discarded(result) then
            batch ← CreateMutationBatch(x, R)
            prio ← TracePriority(TLog, trace)
            PushFront(QValid, prio, batch)
        ...
Function Pick(QValid, QDiscarded, gen):
    if not Empty(QValid) then
        (batch, prio) ← DequeMax(QValid)
        if Empty(batch) then  Pick(QValid, QDiscarded, gen)
        else
            PushFront(QValid, prio, Rest(batch))
            return First(batch)
    if not Empty(QDiscarded) then
        /* Analogous to the case above */
    ...
```

and enqueued for mutation again with a higher effort dedicated to sampling from random mutations.

Notably, this heuristic can be useful when the execution of the system under test depends on invariants over numeric data (e.g, the number of pixels declared by the header of an image matching the size of its actual payload). There, starting with a single random mutation will quickly saturate the trace log with discarded (invalid) tests, and this heuristic will continuously increase the effort put into sampling from random mutations until some randomly generated value satisfies the required invariant, making the overall test case valid.

5 Case studies

We evaluated the performance of MUTAGEN using three case studies. The first two are IFC abstract machines that enforce *noninterference* [152], [153] using runtime checks. While similar in spirit, these abstract machines have a completely different complexity. The first one follows a relatively simple stack-based execution model, with a limited number of instructions. The second one is substantially more featureful, including registers, dynamic memory allocation and a larger instruction set, among others. Notably, both machines were originally proven correct by Amorim *et al.* [154] in Coq, and later degraded by systematically introducing bugs in their IFC policy enforcing mechanism.

Algorithm 5: Adaptive Random Mutations Heuristic

```
Function Loop(P, N, gen):
    boring ← 0; reset ← 1000; R ← 1
    ...
    while i < N do
        if boring > reset then
            TLog ← ∅
            reset ← reset * 2
            R ← R * 2
        ...
        if not result then return Bug(x)
        if Interesting(TLog, trace) then
            boring ← 0
            ...
        else boring ← boring + 1
        ...
```

Lampropoulos *et al.* borrowed these case studies from existing literature [38], [39] to compare *FuzzChick* against RPBT using automatically derived and hand-tuned random generators. Here, we reproduce all their experiments and compare them against our tool. Worth mentioning, we mechanically translated these case studies to Haskell in order to run MUTAGEN on their test suites.

The third case study evaluates MUTAGEN in a realistic scenario, and targets *haskell-wasm* [155], an existing WebAssembly engine of industrial strength. Unfortunately, the current state of *FuzzChick*'s development does not allow to easily port new case studies into its framework, so comparing MUTAGEN with *FuzzChick* on this case study has been out of the scope of this work. Instead, we compare MUTAGEN against *QuickCheck*, evaluating its effectiveness versus the relative overhead of our custom code instrumentation.

5.1 IFC Stack and Register Machines

These abstract machines enforce noninterference, a hyper-property based on the notion of *indistinguishability*. Intuitively, two machine states are indistinguishable if they only differ on secret data. Using this notion, the variant of noninterference we are interested in is called *single-step noninterference* [38] (SSNI). Given two indistinguishable machine states, SSNI asserts that running a single instruction on both machines brings them to resulting states that are also indistinguishable. To achieve this, every runtime value handled by these abstract machines is labeled with a security level, i.e., L (for "low" or public) or H (for "high" or secret). Security labels are then propagated throughout the execution of the program whenever the machines execute an instruction. For this, both machines use a different rule table to specify their IFC policy. These tables store the dynamic check that each machine needs to perform before running each instruction, along with the resulting security labels corresponding to the program counter and the instruction result. For instance, to execute the `Store` instruction (which stores a value in a memory pointer), the IFC Stack machine checks that both the labels of the program counter and the pointer together can flow to the label of the destination memory cell. If this condition is not met, this machine immediately halts its execution. After this check, the machine overwrites the value at the destination cell and updates its label with the maximum sensibility of the involved labels. In the rule table, this looks as follows:

Instruction	**Precondition Check**	**Final PC Label**	**Final Result Label**
`Store`	$l_{pc} \vee l_p \sqsubseteq l_v$	l_{pc}	$l_{v'} \vee l_{pc} \vee l_p$

Where l_{pc}, l_p, l_v, and $l_{v'}$ represent the labels of: the program counter, the memory pointer, and the old and new values stored in that memory cell. The symbol $\vee$ simply denotes the join of two labels, i.e., the *maximum* of their sensibilities. To preserve space, we encourage the reader to refer to the work of Hritcu *et al.* [38], [39] and Lampropoulos *et al.* [34] for further details about the implementation and semantics of these case studies.

Bugs are systematically injected in the IFC enforcing mechanism of both machines by weakening the checks stored in their corresponding IFC rule table. For instance, the following are the buggy rule variations (in red) for the `Store` instruction of the IFC Stack machine:

Instruction	**Precondition Check**	**Final PC Label**	**Final Result Label**
`Store`	$l_{pc} \sqsubseteq l_v$	l_{pc}	$l_{v'} \vee l_{pc} \vee l_p$
`Store`	$l_p \sqsubseteq l_v$	l_{pc}	$l_{v'} \vee l_{pc} \vee l_p$
`Store`	$l_{pc} \vee l_p \sqsubseteq l_v$	$\bot$	$l_{v'} \vee l_{pc} \vee l_p$
`Store`	$l_{pc} \vee l_p \sqsubseteq l_v$	l_{pc}	$l_{pc} \vee l_p$
`Store`	$l_{pc} \vee l_p \sqsubseteq l_v$	l_{pc}	$l_{v'} \vee l_p$
`Store`	$l_{pc} \vee l_p \sqsubseteq l_v$	l_{pc}	$l_{v'} \vee l_{pc}$

This way, there are 20 different buggy ways the IFC Stack machine can be tampered with to violate its IFC policy and invalidate SSNI. Likewise, 33 different IFC-violating bugs can be inserted in the IFC Register machine.

The challenge with testing SSNI for these two case studies is to satisfy its sparse precondition: we need to generate two valid indistinguishable machine states to even proceed to execute the next instruction. Lampropoulos *et al.* demonstrated that generating two independent machine states using *QuickCheck* has virtually no chance of producing valid indistinguishable ones. However, using the mutation mechanism, we can obtain a pair of valid indistinguishable machine states by generating a single valid machine state (something still hard but much easier than before), and then producing a similar mutated copy. This way, we have a higher chance of producing two almost identical states that pass the sparse precondition.

5.2 WebAssembly Engine

WebAssembly [156] is a language designed for executing low-level code on the web. WebAssembly programs are first validated and later executed in a sandboxed environment. The language is relatively simple: in essence i) it

Id	Subsystem	Category	Description
1	Validator	Bug	Invalid memory alignment validation
2	Validator	Discrepancy	Validator accepts returning multi-value blocks
3	Interpreter	Bug	Function invoker ignores arity mismatch
4	Interpreter	Bug	Allowed out-of-bounds memory access
5	Interpreter	Discrepancy	Non-standard NaN reinterpretation

Table 3: Issues found by Mutagen in *haskell-wasm*.

Id	Subsystem	Description
1	Validator	Wrong if-then-else type validation on else branch
2	Validator	Wrong stack type validation
3	Validator	Removed function type mismatch assertion
4	Validator	Removed max memory instances assertion
5	Validator	Removed function index out-of-range assertion
6	Validator	Wrong type validation on `i64.eqz` instruction
7	Validator	Wrong type validation on `i32` binary operations
8	Validator	Removed memory index out-of-range assertion
9	Validator	Wrong type validation on `i64` constants
10	Validator	Removed alignment validation on `i32.load` instruction
11	Interpreter	Wrong interpretation of `i32.sub` instruction
12	Interpreter	Wrong interpretation of `i32.lt_u` instruction
13	Interpreter	Wrong interpretation of `i32.shr_u` instruction
14	Interpreter	Wrong local variable initialization
15	Interpreter	Wrong memory address casting on `i32.load8_s` instruction

Table 4: Bugs injected into *haskell-wasm*.

contains only four base types, representing both integers and IEEE754 floating-point numbers of either 32 or 64 bits; ii) values of these types are manipulated by functions written using sequences of stack instructions; iii) functions are organized in modules and must be explicitly imported/exported; iv) memory blocks can be imported, exported, and grown dynamically; among others. WebAssembly semantics are fully specified, and programs must be consistently interpreted across engines — despite some subtle details we will address soon. For this, the WebAssembly standard provides a reference implementation with all the functionality expected from a compliant engine.

Our tool is an attractive match for testing WebAssembly engines: most of the programs that can be represented using WebAssembly's AST are invalid, and automatically derived random generators cannot satisfy the invariants required to produce interesting test cases.

In this work, we apply Mutagen to test the two most complex subsystems of *haskell-wasm*: the *validator* and the *interpreter* — both being previously tested using a unit test suite. For this, we took advantage of the reference implementation to find discrepancies (that could potentially lead to bugs) via differential testing [157]. Our testing properties assert that any result produced by *haskell-wasm* matches that of the reference implementation. Notably, Mutagen discovered three latent bugs that the existing test suite was unable to reveal. Moreover, MUTAGEN exposed two other discrepancies between *haskell-wasm* and the reference implementation. These discrepancies trigger parts of the specification that were either not yet supported by the reference implementation (multi-value blocks), or that produce a well-known non-deterministic undefined behavior (NaN reinterpretation) [146]. All these findings (see Table 3) were confirmed by the authors of *haskell-wasm*.

Having sorted these issues out, we mechanically injected 10 new bugs in the validator as well as 5 new bugs in the interpreter of this engine (see Table 4). These bugs either i) remove an existing integrity check (to weaken the WebAssembly type-system/validator); or ii) simulate a copy-paste bug [158], replacing the implementation of an instruction with a compatible one (e.g., `i32.add` by `i32.sub`).[22]

Testing the WebAssembly Validator Our approach is to assert that, whenever a randomly generated (or mutated) WebAssembly module is valid according to *haskell-wasm*, then the reference implementation agrees upon it. In Haskell, we write the property:

```
prop_validator m = isValidHaskellWasm m ⇒ isValidRef m
```

The precondition (`isValidHaskellWasm m`) runs the input WebAssembly module `m` against *haskell-wasm*'s validator, whereas the postcondition (`isValidRef m`) serializes `m`, runs it against the reference implementation and checks that no errors are produced.

We note that, although we only focus on finding false negatives, a comprehensive test suite should also test for false positives, i.e., when a module is valid and *haskell-wasm* rejects it.

Testing the WebAssembly Interpreter Testing the WebAssembly interpreter is substantially more complex than testing the validator since it requires running and comparing the output of the test case programs. To achieve this, the generated test cases need to comply with a common interface that can be invoked both by *haskell-wasm* and the reference implementation. For simplicity, we write a helper function `mkModule` to build a stub WebAssembly module that initializes one memory block and exports a single function. This helper is parameterized by the definition of the module's single function, along with its type signature and name. Then, we can use `mkModule` to define a testing property parameterized by a function type, along with its definition and invocation arguments:

```
prop_interpreter ty fun args =
  do let m = mkModule fun ty "f"
     resHaskellWasm ← invokeHaskellWasm m "f" args
     resRef         ← invokeRef         m "f" args
     return (resHaskellWasm === resRef)
```

[22]These bugs were inspired by the real bug #1 we found prior to this step.

This testing property instantiates a module stub `m` with the input WebAssembly function (`fun`) and its type signature (`ty`). Then, it invokes the function `f` of the module `m` both on *haskell-wasm* and the reference interpreter with the provided arguments (`args`). Finally, the property asserts whether their results are equivalent.[23] Interestingly, equivalence does not imply equality: Non-deterministic operations in WebAssembly like NaN reinterpretations can produce different equivalent results (as exposed by the discrepancy #5 in Table 3), and our equivalence relation needs to take that into account.

Using this testing property directly might not sound wise, as randomly generated lists of input arguments will be very unlikely to match the type signature of randomly generated functions. However, it lets us test what happens when programs are not properly invoked, and it quickly discovered the previously unknown bug #3 in *haskell-wasm* mentioned above. Having fixed this issue, we define a specialized version of `prop_interpreter` that fixes the type of the generated function to take two arguments `x` and `y` (of type `I32` and `F32`, respectively) and return an `I32` as a result:

```
prop_interpreter_i32 f x y =
  prop_interpreter (Type {args = [I32, F32], res = [I32]})
                   f [VI32 x, VF32 y]
```

This property lets us generate functions of a fixed type and invoke them with randomly generated inputs of the expected types. We use this property to find all the bugs injected into *haskell-wasm*'s interpreter in the next section.

6 Evaluation

We repeated each experiment 10 times in a workstation with 64GB of RAM and an Intel Core i7-8700 CPU. In all cases, we used a one-hour timeout to stop the execution if an experiment had not yet found a counterexample. Moreover, we followed the approach taken by Lampropoulos *et al.* and collected the mean-time-to-failure (MTTF) of each bug, i.e., how quickly a bug can be found in wall clock time. In addition, we collected the failure rate (FR) observed for each bug, i.e. the proportion of times each tool finds each bug within the one-hour testing budget. Unlike Lampropoulos *et al.*, *we only aggregate the MTTF of successful runs*, i.e, when a bug was found, since doing so for all runs heavily inflates results when the failure rate is below 100%. In all case studies, we additionally show the effect of the heuristics described in Section 4 by individually disabling them. We call these variants *no LIFO* and *no reset*. For *no reset*, the number of times we sample random mutants (R) is no longer controlled by Mutagen, so we arbitrarily fixed it to R=25 throughout the experiment. [24]

[23]We also set a short timeout to discard potentially diverging programs.

[24]A replication package [159] is available for reviewing purposes.

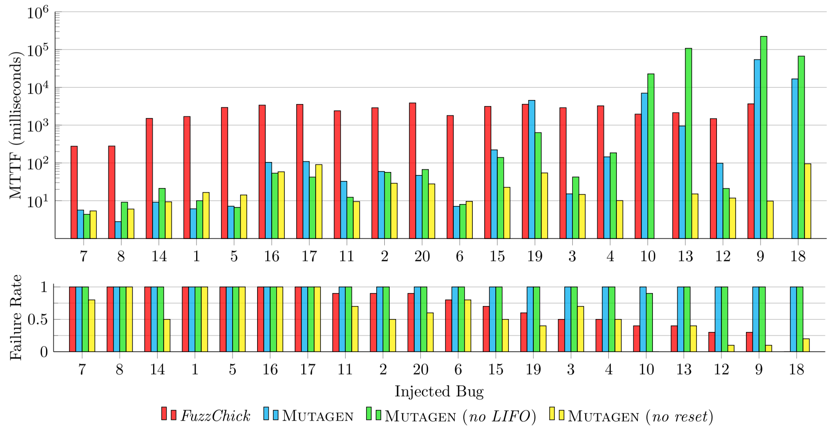


Figure 2: *FuzzChick* versus MUTAGEN in the IFC stack machine.

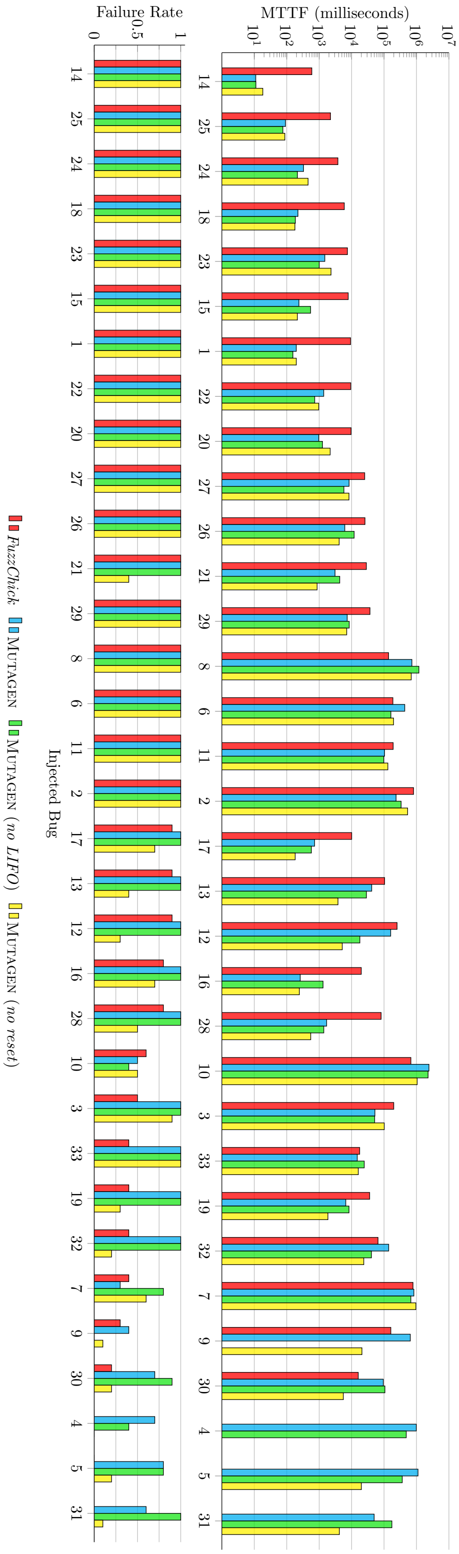


Figure 3: *FuzzChick* versus MUTAGEN in the IFC register machine.

6.1 IFC Stack and Register Machines

The results of these case studies are shown in Fig. 2 and Fig. 3, respectively. In both cases, the injected bugs are ordered by the failure rate achieved by *FuzzChick* in decreasing order. Moreover, notice the logarithmic scale used on the MTTF.

Firstly, we observed a statistically significant improvement in terms of the overall failure rate: Mutagen achieved 100% and 90.9% failure rate (versus 71% and 74.2% for *FuzzChick*) in the IFC Stack and Register cases studies, respectively.

Moreover, if we observe the MTTF achieved by each tool, we recognize in both cases that Mutagen is significantly faster than *FuzzChick* when finding "easy" bugs, i.e., those which both tools can reliably find on each run. However, the results are not as intuitive for the "harder" bugs, i.e, where the failure rate of some tool drops below 100%. To better understand the tradeoffs between these two metrics, we grouped bugs into four categories based on the statistical evidence[25] we observed over the corresponding MTTF achieved by each tool: i) when *FuzzChick* is faster than Mutagen, ii) when Mutagen is faster than *FuzzChick*, iii) when results are inconclusive, i.e., no statistical evidence in favor of either tool, and iv) when *FuzzChick* always fails to find a bug. We avoid considering the case where Mutagen always fails to find a bug as this scenario did not occur in our experiments. In each case, we additionally computed the mean failure rate across bugs for each tool. These curated results are shown in Tables 5 and 6. In both cases, we can observe that our tool is faster than *FuzzChick* for a significant number of bugs, while there are only two bugs in the IFC Register Machine case study where *FuzzChick* consistently outperforms Mutagen. The inconclusive cases reveal that Mutagen achieves a considerably larger failure rate without being significantly slower than *FuzzChick*.

In terms of the Mutagen heuristics, we observed that disabling our LIFO scheduling (case *no LIFO*) does not show a large impact on the results. We noticed that, for these case studies, when a new interesting test case gets enqueued, all its mutants (and their descendants) are quickly processed before new ones start piling up, keeping the mutation queues empty most of the time (generation mode). On the other hand, dynamically tuning the random mutation parameter seems critical to find the harder bugs, as disabling it (case *no reset*) heavily affects Mutagen's failure rate in such cases.

6.2 WebAssembly Engine

The results of this case study are shown in Fig. 4, ordered by the MTTF achieved by Mutagen. We first focus on the bugs injected in the validator (Fig. 4 left). There, we quickly conclude that *QuickCheck* is not well suited to find most of the bugs — it merely finds the easier bugs #5 and #3 in just 1 out of 10 runs. The reason behind this is simple: an automatically derived generator is virtually unable to produce valid WebAssembly modules other than the trivial empty one. Using the same random generator, however, Mutagen

[25]Based on each tail of a Mann-Whitney U-Test with threshold $p < 0.05$.

Bugs where	Count	Mean Failure Rate	
		FuzzChick	MUTAGEN
FuzzChick is faster	0	-	-
Neither tool is faster	2	0.35	1
MUTAGEN is faster	17	0.79	1
FuzzChick always times out	1	0	1

Table 5: Curated results for the IFC Stack Machine case study.

Bugs where	Count	Mean Failure Rate	
		FuzzChick	MUTAGEN
FuzzChick is faster	2	0.8	0.75
Neither tool is faster	8	0.52	0.8
MUTAGEN is faster	20	0.93	1
FuzzChick always times out	3	0	0.7

Table 6: Curated results for the IFC Register Machine case study.

consistently finds every bug in less than 4 seconds. Moreover, disabling the heuristics does not affect the failure rate but tends to add some time overhead to the MTTF, where the *no LIFO* and *no reset* variants are 2.1x and 1.4x slower than the baseline on average.

If we now focus on the bugs injected into the interpreter (Fig. 4 right), we notice that finding bugs now requires minutes instead of seconds, as both interpreters need to validate and run the inputs before producing a result to compare. We also observe a significant improvement in the performance of *QuickCheck* in terms of failure rate. This is of no surprise: we deliberately reduced the search problem to generating functions bodies instead of complete WebAssembly modules. Notably, *QuickCheck* finds counterexamples for the bug #14 almost instantly. This is because this bug can be found using a very small counterexample, and *QuickCheck* prefers sampling small test cases at the beginning of the testing loop. While MUTAGEN uses this approach when in generation mode, our scheduler does not take the size of an interesting test case into account when computing its priority — future work will investigate this possibility. Nonetheless, MUTAGEN still outperforms *QuickCheck* on the remaining bugs in terms of MTTF. Moreover, the *no reset* variant resulted in a 2.9x average slowdown with respect to the baseline, whereas the *no LIFO* variant shows a subtle speedup at the cost of no longer finding the bug #15 with 100% failure rate.

Finally, this case study allows us to analyze the overhead introduced by the custom code instrumentation and internal processing used in MUTAGEN versus the black-box approach used by *QuickCheck*. Table 7 compares the total number of executed and passed tests per second achieved by each tool. Although MUTAGEN is considerably slower than *QuickCheck* at executing tests (roughly 9x and 49x slower when testing `prop_validator` and `prop_interpreter_i32` respect-

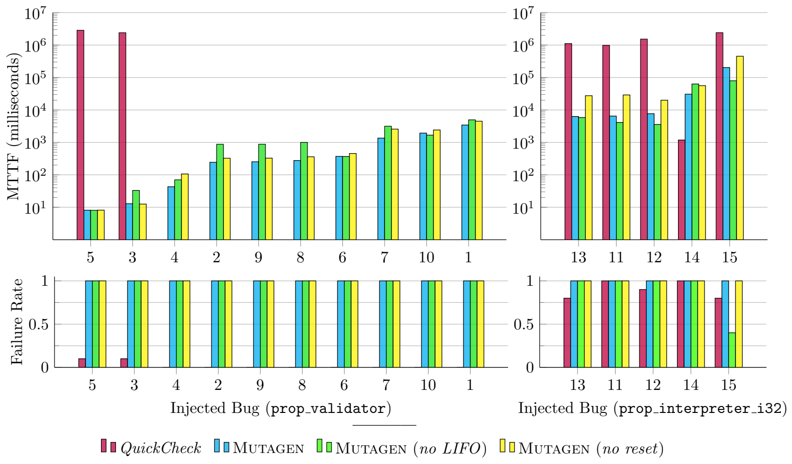


Figure 4: *QuickCheck* versus Mutagen in the WebAssembly case study.

ively), it runs substantially more valid tests that pass the sparse preconditions in the same amount of time, *which can ultimately lead us to find bugs faster.*

7 Threats to Validity

We evaluated Mutagen in three different scenarios that require generating highly-structured inputs, where it was able to find several planted and real previously-unknown bugs. In particular, we compared our tool against all the existing case studies previously considered by Lampropoulos *et al.* However, we cannot generalize that our tool will be effective at finding bugs in other scenarios. To compensate, Mutagen is a fully-automated tool that synthesizes all the needed boilerplate, making it an appealing alternative whenever *QuickCheck* is unable to penetrate properties with sparse preconditions.

As mentioned in Section 5, our evaluation required us to translate both cases studies used by Lampropoulos *et al.* from Coq to Haskell. This was partly aided by Coq's code extraction mechanism. However, this approach still requires some manual intervention and produces Haskell code that is often hard to read. To our advantage, both case studies are implemented using basic Coq features, so translating them to equivalent Haskell code could be done syntactically. As such, we do not have *formal* guarantees that our Haskell version of the code behaves exactly as the original one.

Property	*QuickCheck*		Mutagen	
	Total	Passed	Total	Passed
`prop_validator`	31882.78	0.0003	3505.68	756.52
`prop_interpreter_i32`	106619.31	18.02	2142.14	500.67

Table 7: Tests per second on the WebAssembly case study across tools.

8 Related work

Automated Random Data Generation DRAGEN [85] is a meta-programming tool that synthesizes random generators from data types definitions, using stochastic models to predict and optimize their distribution toward user-defined target ones. DRAGEN2 [89] extends this idea with support for extracting library APIs and function input patterns from the codebase. *QuickFuzz* [70], [71] is a fuzzer that exploits these ideas to synthesize random generators from existing Haskell libraries, which are combined with off-the-shelf low-level fuzzers to find bugs in heavily used programs.

Automatically deriving random generators is substantially more complicated when the generated data must satisfy sparse preconditions. Claessen *et al.* [64] developed an algorithm for generating inputs constrained by boolean preconditions with almost-uniform distribution. Lampropoulos *et al.* [72] extended this approach by adding a limited form of constraint solving controllable by the user. Recently, Lampropoulos *et al.* [75] proposed a mechanism to obtain constrained generators automatically from inductively defined relations in Coq.

All these generational approaches are somewhat orthogonal to the ideas behind MUTAGEN, and while our tool is tailored to improve the performance of poor automatically derived generators, it can benefit from using better generators to find initial (valid) interesting seeds faster.

Coverage-Guided Fuzzing *AFL* [3] is the reference tool when it comes to coverage-guided fuzzing. *AFLFast* [160] extends AFL using Markov chain models to tune the power scheduler toward testing low-frequency paths. MUTAGEN's scheduler is deliberately simple and does not account for path frequency — future work should investigate this possibility. *CollAFL* [161] is a variant of AFL that uses path- instead of edge-based coverage to distinguish executions more precisely by reducing path collisions. We tested this approach in MUTAGEN and found that some bugs can be found faster and more reliably using a prefix-tree-based prioritization of interesting test cases. However, storing the trace of every executed test case a in prefix tree consumes large amounts of memory and the lookup performance heavily degrades over time. Future work should investigate the tradeoffs of this approach in depth.

Havrikov and Zeller [162] have proposed an algorithm that uses input grammars to systematically cover the input space in a bounded fashion, which closely relates to our approach given the similarities between input grammars and values described by algebraic data types. However, MUTAGEN uses the execution trace feedback to decide when to grow recursive grammar nodes one step at a time, whereas the approach by Havrikov and Zeller unfolds these steps into exhaustively testing k-grams pairs of grammar constructions.

BeDivFuzz [163] is a fuzzing approach that separates mutations into "structure-changing" and "structure-preserving", which closely relate to MUTAGEN's pure and random mutant kinds, respectively. BeDivFuzz uses this distinction to search for diverse input structures (via structure-changing mutations) and then apply structure-preserving mutations to them to produce structure-equivalent variants. In turn, MUTAGEN uses this distinction to avoid testing *every* structure-equivalent variant exhaustively.

Zest [164] and Crowbar [32] are two fuzzing tools that mutate the bits representing the pseudo-random choices taken by the input generators instead of relying on specialized structure-preserving mutators. While our approach carries the burden of synthesizing such mutators, it allows us to implement future high-level optimizations, e.g, leveraging lazy evaluation to avoid mutating unevaluated parts of an interesting test case.

Exhaustive Bounded Testing A different category of property-based testing tools does not rely on randomness. Instead, test cases are exhaustively enumerated and tested from smaller to larger up to a certain size bound. *Feat* [29] formalizes the notion of *functional enumerations.* For any algebraic type, it synthesizes a bijection between a finite prefix of the natural numbers and a set of increasingly larger values of the input type. This bijection can be traversed exhaustively or, more interestingly, randomly accessed. This allows the user to easily generate random test cases uniformly simply by sampling natural numbers. However, test cases are enumerated based only on their type definition, so this technique is not suitable for testing properties with sparse preconditions expressed elsewhere. *SmallCheck* [68] is a Haskell tool that also follows this approach. It progressively executes the testing properties against all possible input test cases of up to a certain depth. Similarly, *Korat* [150] is a Java tool that uses method specification predicates to automatically generate all non-isomorphic test cases up to a given small size.

These approaches rely on pruning mechanisms to avoid generating too many equivalent test cases before their exhaustiveness becomes impractical. *LazySmallCheck* is a variant of *SmallCheck* that uses lazy evaluation to automatically prune the search space by detecting unevaluated subexpressions. In *Korat*, pruning is done by instrumenting method precondition predicates and analyzing which parts of the execution trace correspond to each evaluated subexpression.

Our tool uses exhaustiveness as a way to reliably enforce that all possible mutants of an interesting test case are scheduled. In contrast to fully-exhaustive tools, Mutagen relies on randomly generated test cases as a shortcut to find initial interesting test cases without enumerating them exhaustively. Mutagen additionally supports lazy pruning, i.e., it can detect unevaluated subexpressions and avoid producing mutations over their corresponding positions. This can improve the overall performance when testing non-strict properties. In our case studies, however, the precondition of the testing properties fully evaluate their inputs before executing the postconditions, thus we avoided including this optimization in our evaluation. Our future work will investigate the effect of Mutagen's lazy pruning against non-strict testing properties.

9 Conclusions

We presented Mutagen, a coverage-guided, property-based testing tool that extends the original CGPT approach with an exhaustive mutation mechanism that generates every possible mutant for each interesting test case, scheduling them to be tested exactly once.

Our experimental results indicate that MUTAGEN outperforms the simpler CGPT approach implemented in *FuzzChick* in terms of both failure rate and tests until first failure. Moreover, we show how our tool can be applied in a real-world testing scenario, where it quickly discovers 15 planted and 3 previously unknown bugs.