THESIS FOR THE DEGREE OF LICENTIATE OF PHILOSOPHY

Automated Derivation of Random Generators for Algebraic Data Types

AGUSTÍN MISTA

CHALMERS
UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2020

Automated Derivation of Random Generators for Algebraic Data Types
AGUSTÍN MISTA

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

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

Printed at Chalmers
Göteborg, Sweden 2020

Abstract

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

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

Acknowledgments

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

CONTENTS

Chapter 0

Introduction

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

1 Fuzzing

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

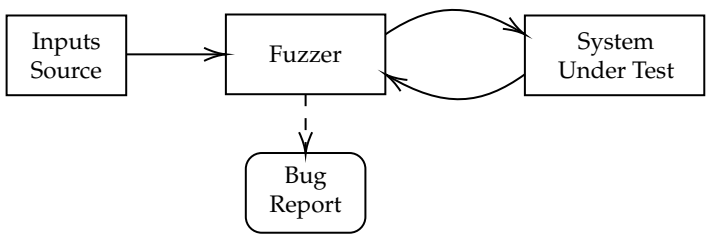


Figure 1: Simplified representation of a fuzzing environment.

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

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

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

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

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

2 QuickCheck

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

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

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

2.1 Testing Properties

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

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

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

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

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

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

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

$$
\begin{aligned}
&\mathit{prop\_reverse\_bad} :: [\mathit{Int}] \to \mathit{Bool} \\
&\mathit{prop\_reverse\_bad}\ \mathit{xs} = \\
&\quad \mathit{reverse}\ \mathit{xs} \equiv \mathit{xs}
\end{aligned}
$$

This property can be easily refuted using QuickCheck as before:

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

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

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

2.2 Random Generators

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

$$
\begin{aligned}
&\textbf{class}\ \mathit{Arbitrary}\ a\ \textbf{where} \\
&\quad \mathit{arbitrary} :: \mathit{Gen}\ a \\
&\quad \mathit{shrink} :: a \to [a]
\end{aligned}
$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3 Automated Derivation of Generators

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

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

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

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

Generation parameters: The generator from Figure 2 simply picks the next random constructor in a uniform basis. This is the simplest approach we can mechanically follow. However, this is hardly the best choice in practice. In particular, generating values of any data type with more ter-

minal than recursive data constructors using uniform choices will be biased towards generating very small values. QuickCheck provides mechanisms for adjusting the generation probability of each random choice it performs. However, doing so carries a second problem: it becomes quite tricky to assign these probabilities without knowing how they will affect the overall distribution of generated values—something that can be seen as a science to its own. Both problems are addressed in detail in Paper 2.

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

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

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

4 Contributions

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

4.1 Paper 1: QuickFuzz Testing For Fun And Profit

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

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

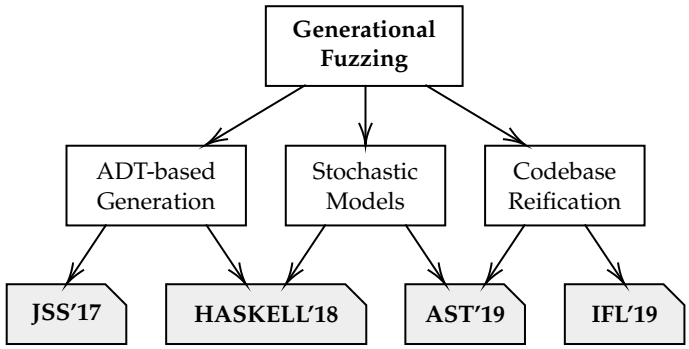


Figure 3: Roadmap of this thesis.

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

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

4.2 Paper 2: Branching Processes for QuickCheck Generators

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

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

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

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

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

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

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

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

On the other hand, it is common that data type definitions simply do not encode enough structural information of the actual data they represent in order for the derivation process to derive useful random generators. This is particularly the case in the presence of shallow embedded

domain-specific languages, where data types are often too generic, and invariants are preserved mostly via their abstract interfaces.

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

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

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

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

4.4 Paper 4: Deriving Compositional Random Generators

The previous paper provides an extension to the automated derivation mechanism of random generators proposed originally. This extension enables us to consider additional sources of structural information when deriving random generators apart from just data type definitions. Each source of structural information introduces a new set of random constructions that can be used by our generators when producing random

values, i.e., one random construction per each data constructor, function input pattern and abstract interface combinator.

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

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

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

4.5 Statement of Contributions

Paper 1: QuickFuzz Testing For Fun And Profit My contributions to this project include: i) a generators derivation extension, which contemplates the common case of existing libraries written using shallow embeddings of the target file format. Before this extension, such libraries simply could

not be used due to the lack of domain-specific structure encoded into data types, which are the main source of information used by QuickFuzz to derive useful random generators for free, and ii) a complete rewrite of the testing harness from scratch, using as much meta-programming as possible in order to ease the task of adding support for new file-format targets.

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

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

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

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

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

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

The majority of the writing was initially done by me.

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

References

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

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

Paper 1

QuickFuzz Testing For Fun And Profit

By Gustavo Grieco, Martin Ceresa, Agustín Mista and Pablo Buiras.

Published in the Journal of Systems and Software, Vol 134, 2017 (JSS'17).

ABSTRACT

Fuzzing is a popular technique to find flaws in programs using invalid or erroneous inputs but not without its drawbacks. At 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 to synthesize 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 on 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 [7].

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 [15, 25, 35] 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 [16, 22, 29], 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 a 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 [10], the well-known property-based random testing library and Hackage [18], the community Haskell software repository in conjunction with off-the-shelf mutational fuzzers to provide automatic fuzzing 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 saves the 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 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[2] 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

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

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 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 [17] 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 the 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 was 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 [26]: *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 producing a random mutation at the bit or byte level.

Nowadays, there are plenty of robust and fast mutational fuzzers. For instance, zzuf [6] 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 [29]. It was developed by the Oulu university secure programming group and works at the byte level randomly adding, removing or changing complete sequence 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 [22] and honggfuzz [16] 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 type 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 [11]. It was originally written in Python in 2007, and later rewritten 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 [31], 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 [27], 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 [19] and GLADE [4] helped to learn and syntetize 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 syntetize 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 model in generational fuzzers [4].

2.2 Haskell

Haskell is a general-purpose purely-functional programming language [23]. 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:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

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

$$\begin{aligned}
&length :: List\ a \to Int\\
&length\ Nil = 0\\
&length\ (Cons\ x\ xs) = 1 + length\ xs\\
&snoc :: a \to List\ a \to List\ a\\
&snoc\ x\ Nil = Cons\ x\ Nil\\
&snoc\ x\ (Cons\ y\ ys) = Cons\ y\ (snoc\ x\ ys)\\
&reverse :: List\ a \to List\ a\\
&reverse\ Nil = Nil\\
&reverse\ (Cons\ x\ xs) = snoc\ x\ (reverse\ xs)
\end{aligned}$$

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 $(\equiv)$ for a given type. Assuming that a is in the Eq type class, we can define an instance of Eq for $List\ a$:

$$
\begin{array}{l}
\textbf{instance}\ Eq\ a \Rightarrow Eq\ (List\ a)\ \textbf{where} \\
\quad \begin{array}{lll}
Nil & \equiv Nil & = True \\
(Cons\ x\ xs) & \equiv (Cons\ y\ ys) & = (x \equiv y) \wedge (xs \equiv ys) \\
\_ & \equiv \_ & = False
\end{array}
\end{array}
$$

Note that the $(\equiv)$ operator is used on two different types: in the expression $x \equiv y$ it uses the definition given in the instance for $Eq\ a$ (equality on a), while in the expression $xs \equiv ys$ it is a recursive call to the $(\equiv)$ 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* [24]. 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 is one of the ways in which we can hide this necessary boiler plate 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 $(\langle\star\rangle)$, with the types given below.

$$
\begin{array}{l}
\textbf{class}\ Applicative\ p\ \textbf{where} \\
\quad pure :: a \to p\ a \\
\quad (\langle\star\rangle) :: p\ (a \to b) \to p\ a \to p\ b
\end{array}
$$

The function $pure$ inserts pure values into the applicative structure (the boiler plate), and $(\langle\star\rangle)$ 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 $(+) :: \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ that adds two numbers, and that we have a type RNG with an instance $\mathit{Applicative\ RNG}$ that represents random-number generation, and moreover that there is a value $\mathit{gen} :: \mathit{RNG\ Int}$ that produces a random Int. We can express a computation that adds two random numbers using the applicative interface as follows: $\mathit{pure}\ (+)\ \langle\star\rangle\ \mathit{gen}\ \langle\star\rangle\ \mathit{gen}$. This expression has type $\mathit{RNG\ Int}$ (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 [36], and it has support for reading and writing different image formats.

Hackage is a fundamental part of QuickFuzz, since it provides all the 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 [10]. 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:

$$\mathit{prop\_reverseSize} :: \mathit{List}\ a \to \mathit{Bool}$$
$$\mathit{prop\_reverseSize}\ xs = \mathit{length}\ xs \equiv \mathit{length}\ (\mathit{reverse}\ xs)$$

QuickCheck will try to falsify the property by generating random values of type $\mathit{List}\ a$ until a counter example is found.

Generators QuickCheck requires the programmer to implement a generator for $\mathit{List}\ a$ in order to test properties involving such data type, like $\mathit{prop\_reverseSize}$ above. The tool defines an applicative functor Gen and a new type class called $\mathit{Arbitrary}$ 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 Int)
                    ⟨⋆⟩(resize (n−1) arbitrary :: Gen IntList)
```

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* [13] that attempts to fix broken Gif images. Our tool has built-in support for the generation of Gif files using the JuicyPixels library [36].

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 to mention that no instrumentation is required in order 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

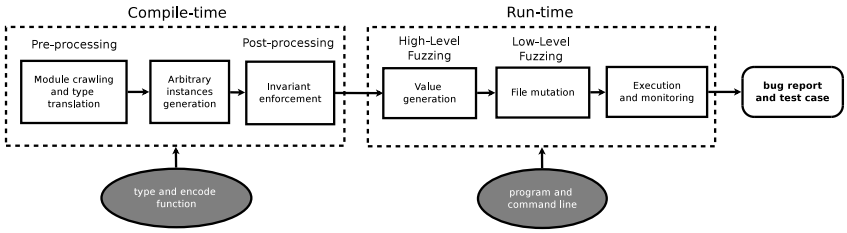


Figure 1: Summary of the random generators deriving using MegaDeTH at compile-time and the test case generation using QuickFuzz at run-time where gray nodes represent inputs provided by a user and bold nodes represent outputs.

data in a compositionally manner. In our example this representation type T (borrowed from JuicyPixels) is called $GifFile$.

$$\begin{aligned}&\textbf{data}\ GifFile = GifFile\ Header\ Images\ Looping\\&\textbf{data}\ Looping\\&\quad = LoopingNever\\&\quad \mid LoopingForever\\&\quad \mid LoopingRepeat\ Int\end{aligned}$$

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 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 [6].
- Radamsa: a general purpose fuzzer developed by the Oulu University Secure Programming Group [29].
- Honggfuzz: a general purpose fuzzer developed by Google [16].

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 of 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 during 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 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 [28] and Address Sanitizer [33] 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 arbitrary generated values to enforce specific semantic properties. In particular, we use such 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 [34], 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 automatic 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 with further detail. The next example introduces a different manner to define a data type which exploits the limitations of MegaDeTH, and serves as introduction to the solution.

Designing a Html manipulating library One of the main decisions involved when designing a domain-specific language [20] (DSL) manipulation library is the level of *embedding* this DSL will have. The most common ap-

proaches are *deep embedding* and *shallow embedding* [1]. Deep 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 conformed by 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 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 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 represents a valid *Html* value. Therefore, this kind of generators are useless for our purpose of discovering bugs on complex software parsing markup languages such as HTML.

This approach to define libraries is common to find in the wild, being *blaze-html* [21] or *language-css* [2] some examples of this. Instead of discarding them, next subsection introduces a different approach we took to derive powerful *Arbitrary* instances for this kind 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* [9]. 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 for 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 to mention 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 [3] in order to manually derive random generators for a particular data type of interest. However, this task also becomes repetitive, specially 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 functions 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:

$$\begin{aligned}&\textbf{instance } Arbitrary\ Html \textbf{ where}\\ &\quad arbitrary = pure\ performHtml\\ &\qquad\qquad \langle\star\rangle (arbitrary :: Gen\ HtmlAction)\end{aligned}$$

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 adding a function for splitting Html values:

$$split :: Html \to (Html, Html)$$

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:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

We can define the following polymorphic functions for all a.

$$append :: a \to List\ a \to List\ a$$
$$concat :: List\ a \to List\ a \to List\ a$$

Our current approach can only handle non-polymorphic functions. We use a naive workaround to solve this consisting on 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:

$$append\_1 :: Int \to List\ Int \to List\ Int$$
$$append\_2 :: String \to List\ String \to List\ String$$
$$concat\_1 :: List\ Int \to List\ Int \to List\ Int$$
$$concat\_2 :: List\ String \to List\ String \to List\ String$$

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 with 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 where QuickFuzz generates a complete program with variables and assignments but without any sense nor 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 [39] 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 variables 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 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 structures provide.

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:

$$\textbf{type}\ \mathit{Cmd} = (\mathit{FilePath}, [\mathit{String}])$$

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 which 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 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 *GifFile*s. 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 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 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.

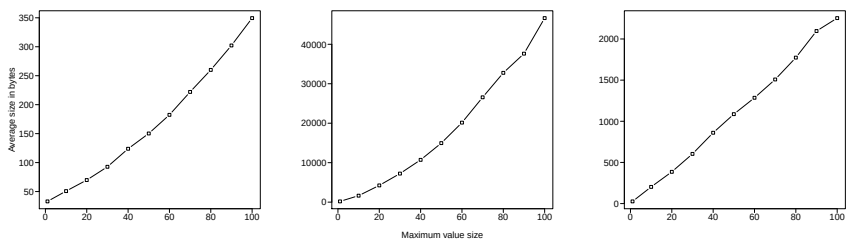
To avoid this pitfall, our instances of *Arbitrary* are carefully crafted to keep the size generated value under control using the *resize* function provided by QuickCheck. Figures 2a, 2b and 2c show how the average size of bytes behaves when the maximum QuickCheck *size* is increased. The size of the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

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.



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

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

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 [22], a well-known fuzzer, because:

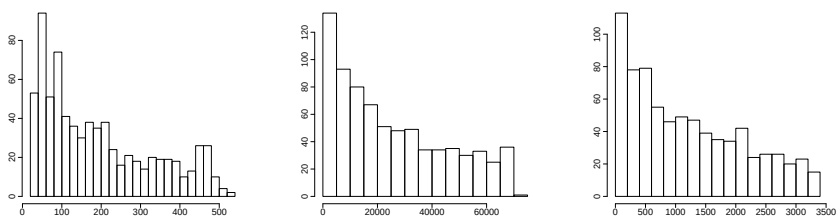
- It was designed to be useful in the fuzzing campaigns: finding more *paths* is highly correlated with the discovery of more bugs [5].
- It was built using a modular approach: we can easily re-use the corresponding command line program to only extract *paths* and count them.
- It has a very fast instrumentation: 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.

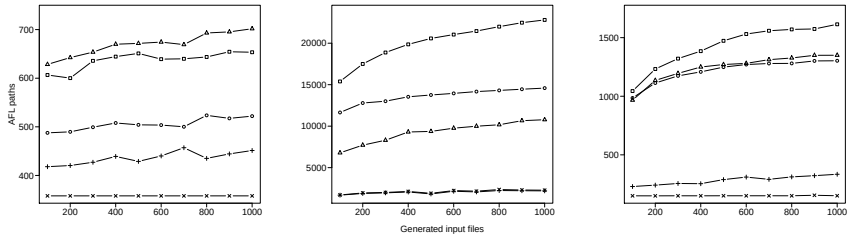
We also included two baseline measures to compare how the file structure created by our tool improves the *path* discovery. The first one generating files of random bytes and the second one using the corresponding magic numbers followed by a random bytes.

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



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

Figure 3: Frequency of generated file sizes in bytes



(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.

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.

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 [22].

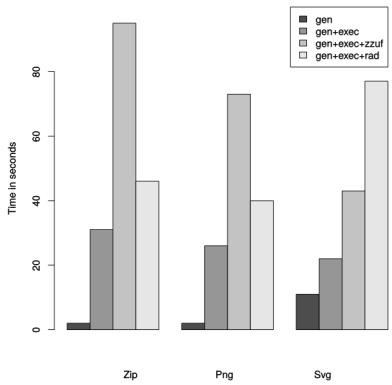


Figure 5: Overhead of QuickFuzz performing the fuzzing process.

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.

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

6.5 Comparison with Other Fuzzers

To make a fair comparison between fuzzers is a challenge. First, it only makes sense to compare between 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 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 [12] 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 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 [37].

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 the 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 on 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 [38] 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 [32]. 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* [36]. For instance, this library lacks of 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 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 with the *encode* function is caused by the use of partial functions. Then the encoding could fail to execute correctly in 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, this kind of issues are avoided manually selecting suitable libraries from Hackage to integrate in 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 [14] 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. [8] 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 to 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 fuzzed generated source code will be very likely rejected in the first steps of the parsing of interpreters or compilers. This is a well-known issue that has been studied extensively by Pałka et al. [30] in the context of testing a compiler.

The approach in that paper always generate 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 [30] 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 serializations 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 [40]. 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 a Generalized Algebraic Data-types that can provides us with more information based in 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.

References

1. Combining deep and shallow embedding of domain-specific languages. *Comput. Lang. Syst. Struct.*, 44(PB):143–165, December 2015.
2. Anton Kholomiov. language-css: a library for building and pretty printing CSS 2.1 code. `https://hackage.haskell.org/package/language-css`, 2010.
3. Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with Quviq Quickcheck. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, pages 1–8, New York, NY, USA, 2008. ACM.
4. O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing program input grammars. In *Programming Language Design and Implementation (PLDI)*, 2017.
5. Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043. ACM, 2016.
6. CACA Labs. zzuf - multi-purpose fuzzer. `http://caca.zoy.org/wiki/zzuf`, 2010.
7. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12. IEEE Computer Society, 2012.
8. Koen Claessen, Jonas Duregård, and Michał H. Pałka. Generating Constrained Random Data with Uniform Distribution. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming: 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, pages 18–34, Cham, 2014. Springer International Publishing.
9. Koen Claessen and John Hughes. Testing monadic code with QuickCheck. *SIGPLAN Not.*, 37(12):47–59, December 2002.
10. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.
11. Deja vu Security. Peach: a smartfuzzer capable of performing both generation and mutation based fuzzing. `http://peachfuzzer.com/`, 2007.
12. Deja vu Security. Peach Pits and Pit Packs. `http://www.peachfuzzer.com/products/peach-pits/`, 2016.
13. Eric S. Raymond. GIFLIB: A library and utilities for processing GIFs. `http://giflib.sourceforge.net/`, 1989.
14. Franco Contanstini. language-python bug report: some stuff missing in Pretty instances. `https://github.com/bjpop/language-python/issues/30`, 2010.
15. Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. *SIGPLAN Not.*, 2008.
16. Google. honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options. `https://github.com/aoh/radamsa`, 2010.
17. Gustavo Grieco, Martín Ceresa, and Pablo Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, pages 13–20, New York, NY, USA, 2016. ACM.
18. Hackage. The Haskell community's central package archive of open source software. `http://hackage.haskell.org/`, 2010.

19. Matthias Höschele and Andreas Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 31–34, 2017.
20. P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society.
21. Jasper Van der Jeugt. blaze-html: a blazingly fast HTML combinator library for Haskell. `https://hackage.haskell.org/package/blaze-html`, 2010.
22. M. Zalewski. American Fuzzy Lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2010.
23. Simon Marlow. Haskell 2010 language report, 2010.
24. Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.
25. Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.
26. Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 2007.
27. Mozilla. Dharma: a generation-based, context-free grammar fuzzer. `https://github.com/MozillaSecurity/dharma`, 2015.
28. Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.
29. Oulu University Secure Programming Group. A Crash Course to Radamsa. `https://github.com/aoh/radamsa`, 2010.
30. M. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing and optimising compiler by generating random lambda terms. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.
31. Pedram Amini and Aaron Portnoy. sulley: a pure-python fully automated and unattended fuzzing framework. `https://github.com/OpenRCE/sulley`, 2012.
32. PNG Development Group. libpng: the official PNG reference library. `http://www.libpng.org/pub/png/libpng.html`, 2000.
33. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. USENIX ATC'12, pages 28–28, 2012.
34. Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.
35. Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
36. V. Berthoux. Juicy.Pixels: Haskell library to load & save pictures. `https://hackage.haskell.org/package/JuicyPixels`, 2012.
37. Vincent Berthoux. svg-tree: SVG loader/serializer for Haskell. `https://github.com/Twinside/svg-tree`, 2007.
38. Willem van Schaik. The öfficialẗest-suite for PNG. `http://www.schaik.com/pngsuite/`, 2011.
39. Xuejun Yang, Yang Chen, Eric Eide, and John 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.
40. Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

Paper 2

Branching Processes for QuickCheck Generators

By Agustín Mista, Alejandro Russo and John Hughes.

Published in the proceedings of the ACM SIGPLAN Haskell Symposium 2018 (HASKELL'18).

ABSTRACT

In *QuickCheck* (or, more generally, random testing), it is challenging to control random data generators' distributions—specially 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 which 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 [3,17,18]. In the Haskell community, *QuickCheck* [10] 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* [27] (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* [24] and *MegaDeTH* [14,15] 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* [12] 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 [9]. 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 [19].

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 which 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* [29], 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 which 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[3] 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 on third-party examples.

2 Background

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

$$\textbf{data}\ \mathit{Tree} = \mathit{LeafA} \mid \mathit{LeafB} \mid \mathit{LeafC} \mid \mathit{Node}\ \mathit{Tree}\ \mathit{Tree}$$

In order to help developers write generators, *QuickCheck* defines the $\mathit{Arbitrary}$ type-class with the overloaded symbol $\mathit{arbitrary} :: \mathit{Gen}\ a$, which denotes a monadic generator for values of type a. Then, to generate random trees, we need to provide an instance of the $\mathit{Arbitrary}$ type-class for

$$\begin{aligned}&\textbf{instance}\ \mathit{Arbitrary}\ \mathit{Tree}\ \textbf{where}\\ &\quad \mathit{arbitrary} = \mathit{oneof}\\ &\qquad [\mathit{pure}\ \mathit{LeafA}, \mathit{pure}\ \mathit{LeafB}, \mathit{pure}\ \mathit{LeafC}\\ &\qquad , \mathit{Node}\ \langle\$\rangle\ \mathit{arbitrary}\ \langle\star\rangle\ \mathit{arbitrary}]\end{aligned}$$

Figure 1: Random generator for Tree.

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

the type *Tree*. Figure 1 shows a possible implementation. At the top level, this generator simply uses *QuickCheck*'s primitive $oneof :: [Gen\ a] \to Gen\ a$ to pick a generator from a list of generators with uniform probability. This list consists of a random generator for each possible choice of data constructor of *Tree*. We use *applicative style* [22] to describe each one of them idiomatically. So, $pure\ LeafA$ is a generator that always generates *LeafA*s, while $Node\ \langle\$\rangle\ arbitrary\ \langle\star\rangle\ arbitrary$ 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* [24]. This library uses Template Haskell [28] 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 which 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 meta-programming tool used intensively by *QuickFuzz* [14, 15]. In 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 $sized :: (Int \to Gen\ a) \to Gen\ a$ to build a random generator based on a function (of type $Int \to Gen\ a$) 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

```
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*.

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 how large the chosen generation size is—see Figure 3 (left).

2.3 *Feat*

The last approach we discuss is *Feat* [12]. 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.[4] 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.,

[4]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.

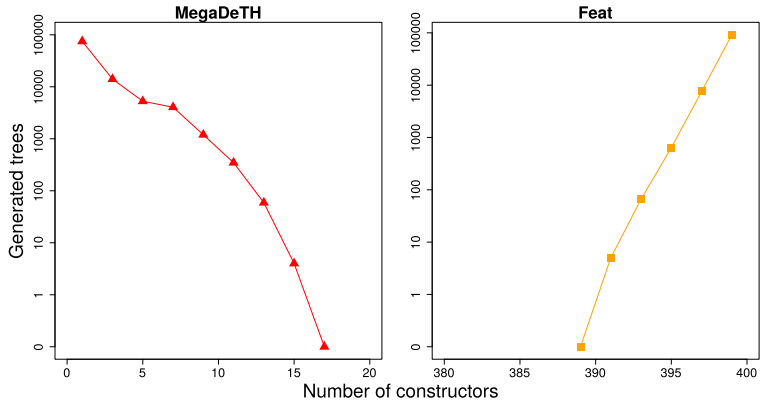


Figure 3: Size distribution of 100000 randomly generated *Tree* values using *MegaDeTH* (▲) with generation size 10, and *Feat* (■) with generation size 400.

generating values of up to 400 constructors.[5] 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 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, to decide which generation sizes are the best choices is not a trivial task either. As 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 on 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.

[5] 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.

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. [16] 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 *Node* constructor. ii) At any step, the probability of generating a *Node* is not affected by the *Node*s 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.

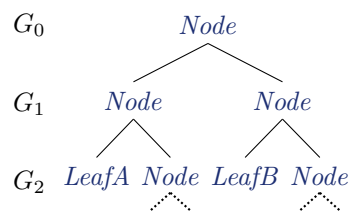


Figure 4: Generation of *Node* constructors.

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:

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

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 $E[X] = E[E[X|Y]]$[6] with $X = G_n$ and $Y = G_{n-1}$ to obtain:

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

[6] $E[X|Y]$ 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.

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:

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

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:

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

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.

```
instance Arbitrary Tree' where
  arbitrary = sized gen
    where
      gen 0 = pure Leaf
      gen n = chooseWith
        [(pLeaf,  pure Leaf)
        ,(pNodeA, NodeA <$> gen (n-1) <*> gen (n-1))
        ,(pNodeB, NodeB <$> gen (n-1))]
```

Figure 5: *DRAGEN* generator for *Tree'*

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 $chooseWith :: [(Double, Gen\ a)] \rightarrow Gen\ a$, 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′*.

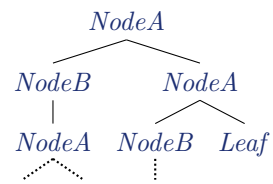


Figure 6: A generated value of type *Tree′*.

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 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 gener-

ation. 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.

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

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 $E[G_n^j]$.

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

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

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

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:

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

The rest of M_C can be similarly computed, obtaining:

$$M_C = \begin{array}{c} \\ Leaf \\ NodeA \\ NodeB \end{array} \begin{array}{c} \begin{array}{ccc} Leaf & NodeA & NodeB \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ 2 \cdot p_{Leaf} & 2 \cdot p_{NodeA} & 2 \cdot p_{NodeB} \\ p_{Leaf} & p_{NodeA} & p_{NodeB} \end{bmatrix} \end{array} \tag{11}$$

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.[7]

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:

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

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 a 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 Appendix 2.3 for a graphical interpretation. In principle, the number of terminal constructors generated by the stochastic process described in *gen n* is captured by the multi-type branching process formulas. However, to predict the expected number of terminal

[7] 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.

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:

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

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 the recursive arguments of non-terminal constructors at the previous level. For instance, consider the following ADT:

$$\textbf{data}\ Tree'' = LeafA \mid LeafB \mid NodeA\ Tree''\ Tree'' \mid NodeB\ Tree''$$

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).

```
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 $Tree''$

To compute the overall expected number of terminals, we need to predict the expected number of terminal constructors at the last generation level which could be descendants of non-terminal constructors at level $(n-1)$. More precisely:

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

where the case of $E[LeafB]$ 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 T_1 and T_2 with their automatically derived generators shown in Figure 8.

Note the use of the *QuickCheck*'s function $resize :: Int \to Gen\ a \to Gen\ a$, 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 its own. Figure 9 visualizes the possible offspring generated by the non-terminal constructor B (belonging to T_1) with the corresponding probabilities as labeled edges. Following the figure, we obtain the expected number of Ds generated by B constructors as follows:

```
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
      [(pA, pure A)
      ,(pB, B <$> gen (n-1) <*> resize (n-1) arbitrary)]
instance Arbitrary T2 where
  arbitrary = sized gen where
    gen 0 = pure C
    gen n = chooseWith
      [(pC, pure C),(pD, D <$> resize (n-1) arbitrary)]
```

Figure 8: Mutually recursive types T_1 and T_2 and their *DRAGEN* generators.

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

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

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

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 T_1. 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.

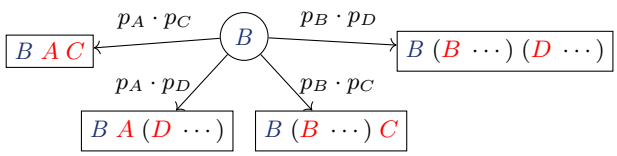


Figure 9: Possible offspring of constructor B.

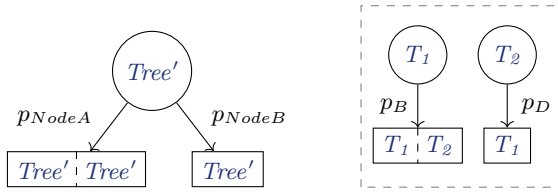


Figure 10: Offspring as types

While this approach works, it completely ignores the types T_1 and T_2 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.

Mean matrix of types If we analyze the mean matrices of $Tree'$ (11) and the mutually-recursive types T_1 and T_2 (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 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 T_2 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 place holders can only be populated by constructors of a certain type.

Figure 10 illustrates offspring as types for the definitions T_1, T_2, and $Tree'$. A place holder of type T_1 can generate a place holder for type T_1 and a place holder for type T_2. A place holder of type T_2 can generate a place holder of type T_1. A place holder of type $Tree'$ can generate two place holders 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 T_1 and T_2, written $M_{T_1T_2}$ are defined as follows:

$$M_{Tree'} = \begin{matrix} \\ Tree' \end{matrix}\begin{bmatrix} Tree' \\ 2 \cdot p_{NodeA} + p_{NodeB} \end{bmatrix} \qquad M_{T_1T_2} = \begin{matrix} \\ T1 \\ T2 \end{matrix}\begin{bmatrix} T1 & T2 \\ p_B & p_B \\ p_D & 0 \end{bmatrix}$$

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 place holders 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 place holders 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 which 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:

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

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:

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

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*:

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

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:

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

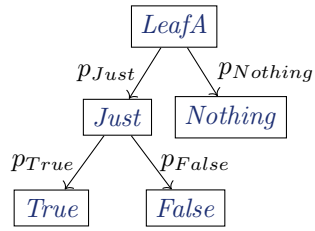


Figure 11: Constructor dependency graph.

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 of 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 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 $dragenArbitrary :: Name \to Size \to CostFunction \to Q\ [Dec]$, 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 for. 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:

$$\textbf{type } \mathit{CostFunction} = \mathit{Size} \to \mathit{ProbMap} \to \mathit{Double}$$

Type $\mathit{ProbMap}$ 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 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 $\mathit{ProbMap}$ 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, $\mathit{uniform} :: \mathit{CostFunction}$ 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 $\mathit{only} :: [\mathit{Name}] \to \mathit{CostFunction}$ and $\mathit{without} :: [\mathit{Name}] \to \mathit{CostFunction}$ to describe these restrictions. In this case, the whitelisted constructors are then generated following the $\mathit{uniform}$ 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 $\mathit{onlyTypes}$ and $\mathit{withoutTypes}$. Additionally, when the user wants to generate constructors according to certain proportions, $\mathit{weighted} :: [(\mathit{Name}, \mathit{Int})] \to \mathit{CostFunction}$ 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 $\mathit{uniform}$ case the optimizer cannot do anything to break the implicit invariant of the data

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

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 by 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.

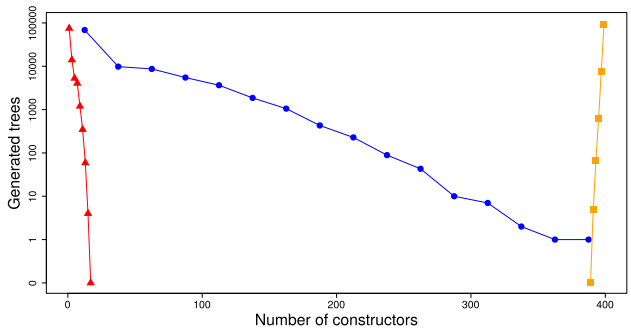


Figure 12: *MegaDeTH* (▲) vs. *Feat* (■) vs. *DRAGEN* (●) generated distributions for type *Tree*.

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 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 interoperate 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) [21]—a well known fuzzer. It was chosen in this work since it is also used in the work by Grieco et al. [15] 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.

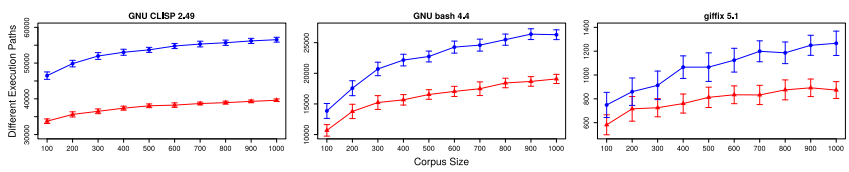


Figure 13: Path coverage comparison between *MegaDeTH* (▲) and *DRAGEN* (●).

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. 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 an 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 does 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 tests programs against randomly generated unexpected inputs. *QuickFuzz* [14, 15] 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 work flow 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 [27] 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 [25]. Recently, Midtgaard et al. extend such a technique to test compilers for impure programming languages [23].

Luck [19] 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 [20]. 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 [11] are a general approach to randomly generating combinatorial structures such as trees and graphs—also extended to work with closed simply-typed lambda terms [5]. 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 [26]. 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* [13], 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 [6]. This is the main idea of *Adaptive Random Testing* (ART) [7]. 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 [1]. This strategy is later extended in [8] 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.

References

1. A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proc. of the International Symposium on Software Testing and Analysis*, ISSTA '11. ACM, 2011.
2. B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security Privacy*, 2005.
3. T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*, 2015.
4. N. Balakrishnan, V. Voinov, and M.S Nikulin. *Chi-Squared Goodness of Fit Tests with Applications. 1st Edition*. Academic Press, 2013.
5. M. Bendkowski, K. Grygiel, and P. Tarau. Boltzmann samplers for closed simply-typed lambda terms. In *In Proc. of International Symposium on Practical Aspects of Declarative Languages*. ACM, 2017.
6. F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775 – 782, 1996.
7. T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In Michael J. Maher, editor, *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*. Springer Berlin Heidelberg, 2005.
8. I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In *Proc. of International Conference on Software Engineering*. ACM/IEEE, 2008.
9. K. Claessen, J. Duregård, and M. H. Palka. Generating constrained random data with uniform distribution. In *Proc. of the Functional and Logic Programming FLOPS*, 2014.
10. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.
11. P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing.*, 13, 2004.
12. J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2012.
13. R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *Proc. of International Symp. on Software Reliability Engineering (ISSRE)*. IEEE, 2013.
14. G. Grieco, M. Ceresa, and P. Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *Proc. of the ACM SIGPLAN International Symposium on Haskell*, 2016.
15. G. Grieco, M. Ceresa, A. Mista, and P. Buiras. QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134, 2017.
16. P. Haccou, P. Jagers, and V. Vatutin. *Branching processes. Variation, growth, and extinction of populations*. Cambridge University Press, 2005.
17. J. Hughes, C. Pierce B, T. Arts, and U. Norell. Mysteries of DropBox: Property-based testing of a distributed synchronization service. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation*, 2016.
18. 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.

19. 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 *Proc. of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, 2017.
20. L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *In Proc. ACM on Programming Languages*, 2(POPL), 2017.
21. M. Zalewski. American Fuzzy Lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/`, 2010.
22. C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), January 2008.
23. J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson. Effect-driven QuickChecking of compilers. *In Proceedings of the ACM on Programming Languages, Volume 1*, (ICFP), 2017.
24. N. Mitchell. Deriving generic functions by example. In *Proc. of the 1st York Doctoral Syposium*, pages 55–62. Tech. Report YCS-2007-421, Department of Computer Science, University of York, UK, 2007.
25. M. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing and optimising compiler by generating random lambda terms. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.
26. S. M. Poulding and R. Feldt. Automated random testing in multiple dispatch languages. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.
27. C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Proc. of the ACM SIGPLAN Symposium on Haskell*, 2008.
28. T. Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.
29. 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.

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:

$$T_t := C_1^t + C_2^t + \cdots + C_n^t$$

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

$$C_c^t := T_1 \times T_2 \times \cdots \times T_m$$

We will define the following observation functions:

$$\begin{aligned} cons(T_t) &= \{C_c^t\}_{c=1}^n \\ args(C_c^t) &= \{T_j\}_{j=1}^m \\ |T_t| &= |cons(T_t)| = n \end{aligned}$$

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

$$\beta(T_v, C_i^u) = |\{T_k \in args(C_i^u) \mid T_k = T_v\}|$$

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 $\{T_t\}_{t=1}^n$ 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:*

$$m_{C_i^u C_j^v} = \beta(T_v, C_i^u) \cdot p_{C_j^v} \tag{13}$$

Proof 1 *Let $m_{C_i^u C_j^v}$ be an element of M_C, we know that $m_{C_i^u C_j^v}$ represents the expected number of constructors $C_j^v \in cons(T_v)$ 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 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 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 $X_k^{C_i^u C_j^v}$:

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

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:

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

The next propositions 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:*

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

Proof 2 *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 place holders of type T_v appearing on each constructor of T_u. For that, we introduce the random variable Y^{uv} capturing this behavior.

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

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

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

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:*

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

Proof 3 *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:*

$$m_{C_i^u C_j^v} = \beta(T_v, C_i^u) \cdot p_{C_j^v} \tag{17}$$
$$m_{T_uT_v} = \sum_{C_k^u \in cons(T_u)} \beta(T_v, C_k^u) \cdot p_{C_k^u} \tag{18}$$

Now, we can rewrite (17) as follows:

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

(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:

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

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

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

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 belong 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 4 *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:*

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

  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:*

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

  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 $\Gamma = \{T_t\}_{t=1}^k$.

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

2 Additional Information

This appendix is meant to provide further analyses for 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 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 $d = 1/2$.

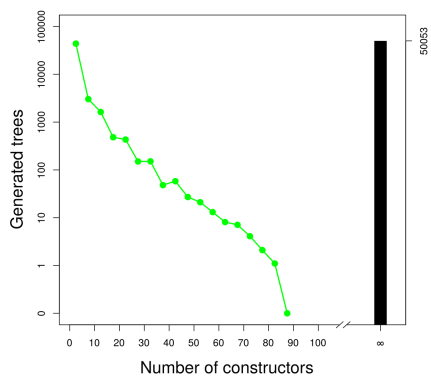


Figure 14: Distribution of (the amount of) T constructors induced by *derive*.

Figure 14 provides an empirical verification of this non-

terminating behavior. It shows the distribution (in terms of amount of constructors) of 100000 randomly generated T values obtained using the *derive* generator shown above. The black bar on the right represents the amount 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 times 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 kind 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, there are some terms which are 0 and others which 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 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.

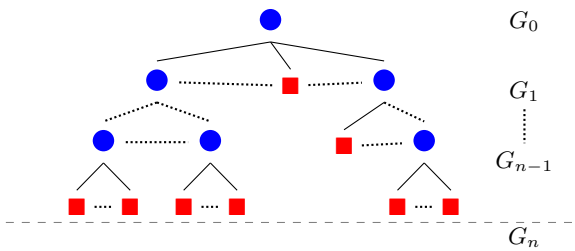


Figure 15: Generation processes of non-terminal (●) and terminal (■) constructors.

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 by 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.

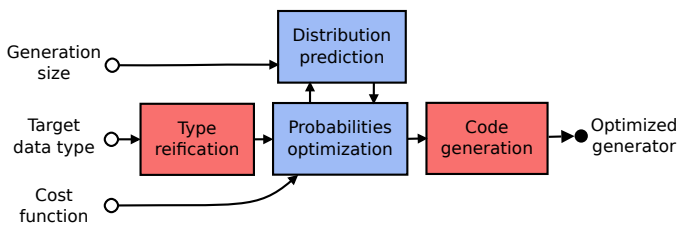


Figure 16: Generation schema.

Such functions are built around the *Chi-Square Goodness of Fit Test* [4], a statistical test used quantify how the observed value of a given phenomena 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 selecting recursively the most suitable neighbor, i.e., a probability

```
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.

assignment that it 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 the are no further neighbors that remains unvisited; or if the step improvement is below a minimum predetermined ε.

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

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

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.

Testing runtimes As we have shown, *MegaDeTH* tends to derive generators which produce very small test cases. However, in our tests, the size differences in the test cases generated by each tool does 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.

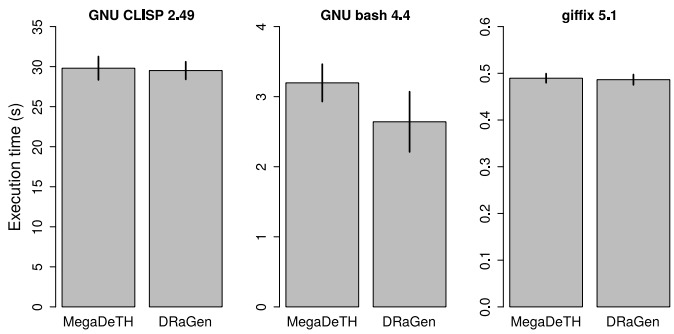


Figure 19: Execution time required to test the biggest randomly generated corpora consisting of 1000 files.

Paper 3

Generating Random Structurally Rich Algebraic Data Type Values

By Agustín Mista and Alejandro Russo.

Published in the proceedings of the IEEE/ACM International Workshop on Automation of Software Test 2019 (AST'19).

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 which expects complex inputs. However, it sometimes becomes necessary to generate structural 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 [1, 10, 11]. *QuickCheck* [3] 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 [5,9,14,15,17]. The main difference about these tools lies on 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 specially useful when testing programs which expect highly structured inputs like compilers [12, 13, 16]. Generating ADT values also proves fruitful when looking for vulnerabilities in combination with fuzzers [8, 9]. 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 into 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* [18]—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* [14]. We call our extension as *DRAGEN2*[8] 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

[8] *DRAGEN2* is available at `http://github.com/OctopiChalmers/dragen2`

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.

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*. [9] 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* [14] is a tool conceived to mitigate the problem of finding the appropriated generation frequencies. It uses the theory of branching processes [18] 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., [5, 8, 15, 17]), *DRAGEN* synthesizes random generators similar to the one

[9] The operators ⟨$⟩ and ⟨⋆⟩ are used in Haskell to combine values obtained from calling random generators and they are not particularly relevant for the point being made in this work.

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.

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 $simplify :: Html \rightarrow Html$. In Haskell, the notation $f :: T$ means that program f has type T. In our example, function $simplify$ takes an $Html$ as an input and produces an $Html$ value as an output—thus its type $Html \rightarrow Html$. 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:

$$
\begin{aligned}
&simplify :: Html \rightarrow Html \\
&simplify\ (Join\ (Text\ t_1)\ (Text\ t_2)) = \\
&\quad Text\ (concat\ t_1\ t_2) \\
&simplify\ (Join\ (Join\ (Text\ t_1)\ x)\ y) = \\
&\quad simplify\ (Join\ (Text\ t_1)\ (simplify\ (Join\ x\ y))) \\
&simplify\ (Join\ x\ y) = \\
&\quad Join\ (simplify\ x)\ (simplify\ y) \\
&simplify\ (Tag\ t\ x) = \\
&\quad Tag\ t\ (simplify\ x) \\
&simplify\ x = x
\end{aligned}
$$

Function $concat$ just concatenates two strings. The body of $simplify$ is described using *pattern matching* over possible kinds of $Html$ values. Pattern matching allows to define 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 an 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* [8] and *DRAGEN* [14]. 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.

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:

```
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*.

```
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

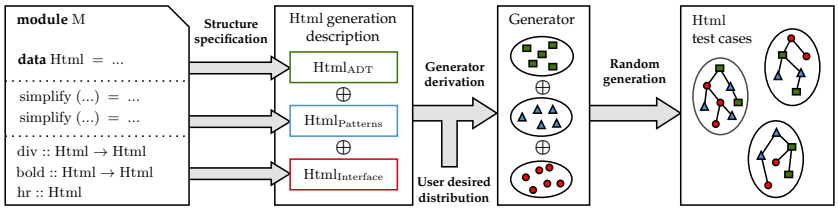


Figure 2: Deriving a generator for the ADT *Html* with the structural information found in module *M*.

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.

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 $\mathrm{Html}_{\mathrm{ADT}}$), (ii) its patterns of interest (noted $\mathrm{Html}_{\mathrm{Patterns}}$), and (iii) the primitives from abstract interfaces to involve in the generation process (noted as $\mathrm{Html}_{\mathrm{Interface}}$). 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 $\mathrm{Html}_{\mathrm{ADT}}$ 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 $\mathrm{Html}_{\mathrm{Patterns}}$ 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 $\mathrm{Html}_{\mathrm{Interface}}$ 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 desire 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 [14], 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

on Section 2. For this purpose, we found that random generation of ADT values can be characterized using the theory of *branching processes* [18]. This probabilistic theory was originally conceived to predict the growth and extinction of royal family trees 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. [14] 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 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 a 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 gets 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:

$$M_{\text{Join,Text}} = 2 \cdot p_{\text{Text}}$$

where 2 is the number of holes present when generating a partial ADT value *Join* (i.e., *Join* • •) 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 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.

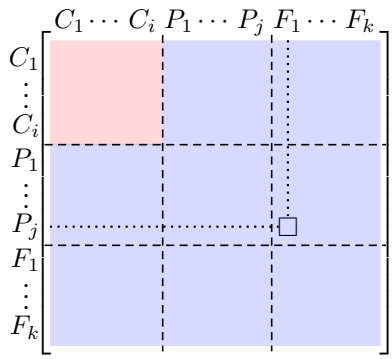


Figure 3: Mean matrix M including pattern matching and function calls information.

The new cells are filled as before: we need to consider the amount 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 amount of holes in the partially generated pattern (*Join* (*Join* (*Text* s) •) •), where s is some random string, times the 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 character-

$$\beta = \begin{array}{r} \\ \text{Text} \\ \text{Single} \\ \text{Tag} \\ \text{Join} \\ \text{simplify\#1} \\ \text{simplify\#2} \\ \text{hr} \\ \text{div} \\ \text{bold} \end{array} \begin{bmatrix} \\ 0 \\ 0 \\ 1 \\ 2 \\ \hline 0 \\ 2 \\ \hline 0 \\ 1 \\ 1 \end{bmatrix} \qquad \mathcal{P} = \begin{bmatrix} \\ p_{\text{Text}} \\ p_{\text{Single}} \\ p_{\text{Tag}} \\ p_{\text{Join}} \\ \hline p_{\text{simplify\#1}} \\ p_{\text{simplify\#2}} \\ \hline p_{\text{hr}} \\ p_{\text{div}} \\ p_{\text{bold}} \end{bmatrix}$$

Figure 4: Prediction vectors of our *Html* generation specification.

ized and predicted. *DRAGEN2* follows a similar approach as *DRAGEN* and uses an 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 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 functions 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:

$$E^{\downarrow}[\mathit{Text}] = E[\mathit{Text}] + 2 \cdot E[\mathit{simplify\#1}] + 1 \cdot E[\mathit{simplify\#2}]$$
$$E^{\downarrow}[\mathit{Join}] = E[\mathit{Join}] + 1 \cdot E[\mathit{simplify\#1}] + 2 \cdot E[\mathit{simplify\#2}]$$

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 [9, 14].

We evaluate how including additional structural information when generating a set of random test cases (often referred 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* [10]—the GNU Common Lisp compiler, and *HTML Tidy* [11]—a well 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*[12], implementing an embedded Lisp interpreter for a small subset of this programming language, and *html*[13], 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 firstly 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 amount 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) 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.

[10] https://www.gnu.org/software/gcl/
[11] http://www.html-tidy.org
[12] http://hackage.haskell.org/package/zuramaru
[13] http://hackage.haskell.org/package/html

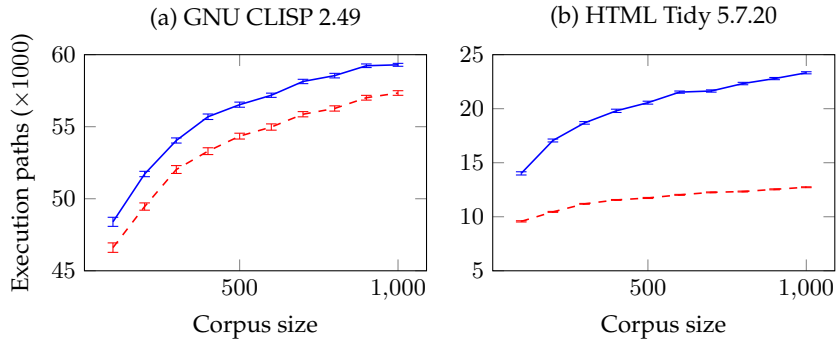


Figure 5: Path coverage comparison between *DRAGEN* (- - -) and *DRAGEN2* (——).

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.

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 on 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 match a against 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 on 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 us 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 [4] are a general approach to randomly generating combinatorial structures such as trees and 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 [6]. In a recent work, Bendkowski et al. provides a framework called *boltzmann-brain* to specify and synthesize standalone Haskell random generators based on Boltzmann models [2]. 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* [6], 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 [7]. 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 codebase. 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. This predictions let us optimize the generation parameters in compile time, resulting in an improved testing performance according to our experiments.

References

1. T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*, 2015.
2. Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. Polynomial tuning of multiparametric combinatorial samplers. In *Proc. of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, 2018.
3. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.
4. P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing.*, 13, 2004.
5. J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2012.
6. R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *Proc. of International Symp. on Software Reliability Engineering (ISSRE)*. IEEE, 2013.
7. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, 2005.
8. G. Grieco, M. Ceresa, and P. Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *Proc. of the ACM SIGPLAN International Symposium on Haskell*, 2016.
9. G. Grieco, M. Ceresa, A. Mista, and P. Buiras. QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134, 2017.
10. J. Hughes, C. Pierce B, T. Arts, and U. Norell. Mysteries of DropBox: Property-based testing of a distributed synchronization service. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation*, 2016.
11. 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.
12. Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2009.
13. J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson. Effect-driven QuickChecking of compilers. *In Proceedings of the ACM on Programming Languages, Volume 1*, (ICFP), 2017.
14. Agustín Mista, Alejandro Russo, and John Hughes. Branching processes for QuickCheck generators. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2018.
15. N. Mitchell. Deriving generic functions by example. In *Proc. of the 1st York Doctoral Syposium*, pages 55–62. Tech. Report YCS-2007-421, Department of Computer Science, University of York, UK, 2007.
16. M. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing and optimising compiler by generating random lambda terms. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.
17. C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Proc. of the ACM SIGPLAN Symposium on Haskell*, 2008.

18. 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.

Paper 4

Deriving Compositional Random Generators

By Agustín Mista and Alejandro Russo

Accepted for publication in the proceedings of the 31st Symposium on Implementation and Application of Functional Languages 2019 (IFL'19)

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 which expects complex inputs. However, it sometimes becomes necessary to generate structural 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 property-based testing is a powerful technique for finding bugs [1, 10, 11, 16]. In Haskell, *QuickCheck* is the predominant tool for this task [2]. 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 [6, 8, 14, 18]. We divide the different approaches in 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* [14] 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* [8,9], *DRAGEN* [18] and *Feat* [6] 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 which 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:[14]

```
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 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 [3, 9].

[14] http://hackage.haskell.org/package/html

The second scenario that we consider is that where derived generators fails at producing 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:

$$
\begin{aligned}
&\mathit{simplify} :: \mathit{Html} \to \mathit{Html} \\
&\mathit{simplify}\ (\mathit{Text}\ t_1 :+: \mathit{Text}\ t_2) = \cdots \\
&\mathit{simplify}\ (\mathit{Text}\ t\ \ :+: x :+: y) = \cdots \\
&\mathit{simplify}\ \cdots\ = \cdots
\end{aligned}
$$

(Symbol $\cdots$ denotes code that is not relevant for the point being made.) Generating values that match, for instance, the pattern $\mathit{Text}\ t :+: x :+: y$ 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 [13].

Our previous work [17] 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 sub-systems 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 interfaces 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

```
derive [
  constructors ''Html,
  interface    ''Html,
  patterns     'simplify
]
```

(a) Machinery derivation

```
type Html_valid =
      Con "Text" ⊗ 2
  ⊕ Con ":+:"  ⊗ 4
  ⊕ Fun "hr"   ⊗ 3
  ⊕ Fun "bold" ⊗ 2
  ⊕ Fun "list" ⊗ 3
  ⊕ Fun "<+>"  ⊗ 5
```

```
type Html_simplify =
      Con "Text"          ⊗ 2
  ⊕ Con "Sing"          ⊗ 1
  ⊕ Con "Tag"           ⊗ 3
  ⊕ Con ":+:"           ⊗ 4
  ⊕ Pat "simplify" 1 ⊗ 3
  ⊕ 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.

generation specifications, namely $Html_{valid}$ and $Html_{simplify}$ 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, $Html_{valid}$ chooses among the constructors *Text* and $:+:$, as well as functions from *Html*'s abstract interface; while $Html_{simplify}$ 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 $genRep\ @Html_{valid}$ and $genRep\ @Html_{simplify}$, respectively.

The main contribution 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* [24] (Section 3).
- 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[15] 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.

In order to provide a common interface for writing generators, *QuickCheck* uses Haskell's overloading mechanism known as *type classes* [25], 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 $gen :: Int \to Gen\ Html$. The *Int* passed to *gen* is known as the *generation size*, and is threaded seamlessly

[15] Available at `https://github.com/OctopiChalmers/dragen2`

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 [15] 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 pick 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 on 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 the final shape of our random data. Then, for instance, the value Mk_{Tag} `"div"` $x :: Con_{Tag}\ r$ 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 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 $f \to a$ helps the type system to solve type of the type variable a, which appears free on the right hand side of the $\Rightarrow$. 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 $\oplus$ to encode the choice between two given representations:

```
data ((f :: * -> *) ⊕ (g :: * -> *)) r = InL (f r) | InR (g r)
```

This infix type-level operator lets us combine two representations f and g into a composite one $f \oplus g$, 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 combining two *parametric type constructors*, hence the kind signature $* \to *$ in both f and g. For instance, the type $Con_{Text} \oplus Con_{Tag}$ encodes values representing either plain text HTMLs 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 of can be translated to the same target type:

```
instance (Algebra f a, Algebra g a) ⇒ Algebra (f ⊕ g) a where
  alg (InL fa) = alg fa
  alg (InR 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 its recursive fields. We can think of them as a 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:

$$\textbf{data}\ \mathit{Fix}\ (f :: * \to *) = \mathit{Fix}\ \{\mathit{unFix} :: f\ (\mathit{Fix}\ f)\}$$

Given a representation type f of kind $* \to *$, the type $\mathit{Fix}\ f$ instantiates each recursive field of f with $\mathit{Fix}\ f$, closing the definition of f into itself—thus the kind of $\mathit{Fix}\ f$ results $*$.

In general, if a type f is used to represent a given target type, we will refer to $\mathit{Fix}\ f$ 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:

$$\textbf{type}\ \mathit{Html}' = \mathit{Con}_{\mathit{Text}} \oplus \mathit{Con}_{\mathit{Sing}} \oplus \mathit{Con}_{\mathit{Tag}} \oplus \mathit{Con}_{(:+:)}$$

Then, for instance, the value $x = \mathit{Text}\ \texttt{"hi"} :+: \mathit{Sing}\ \texttt{"hr"} :: \mathit{Html}$ can be represented with a value $x' :: \mathit{Fix}\ \mathit{Html}'$ as:

$$\begin{aligned} x' = \mathit{Fix}\ (\mathit{In}_R\ (\mathit{In}_R\ (\mathit{In}_R\ (\mathit{Mk}_{(:+:)} \\ \quad (\mathit{Fix}\ (\mathit{In}_L\ (\mathit{Mk}_{\mathit{Text}}\ \texttt{"hi"}))) \\ \quad (\mathit{Fix}\ (\mathit{In}_R\ (\mathit{In}_L\ (\mathit{Mk}_{\mathit{Sing}}\ \texttt{"hr"}))))))))) \end{aligned}$$

where the sequences of In_L and In_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 $\mathit{Fix}\ f$ into its corresponding value of the target type a as follows:

$$\begin{aligned} &\mathit{eval} :: \mathit{Algebra}\ f\ a \Rightarrow \mathit{Fix}\ f \to a \\ &\mathit{eval} = \mathit{alg} \circ \mathit{fmap}\ \mathit{eval} \circ \mathit{unFix} \end{aligned}$$

This function exploits the $\mathit{Functor}$ 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 $\mathit{eval}\ x' \equiv x$. More specifically, the types Html and $\mathit{Fix}\ \mathit{Html}'$ 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.

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 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 an 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 to the *eval* function.

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 to 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:

$$\begin{aligned}&\textbf{data } Pat_{simplify\#1}\ r = Mk_{simplify\#1}\ String\ String\\&\textbf{data } Pat_{simplify\#2}\ r = Mk_{simplify\#2}\ String\ r\ r\end{aligned}$$

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 pattern, where each pattern variable gets instantiated using the values stored in its representation field:

$$\begin{aligned}&\textbf{instance } Algebra\ Pat_{simplify\#1}\ Html\ \textbf{where}\\&\quad alg\ (Mk_{simplify\#1}\ t_1\ t_2) = Text\ t_1\ :+:\ Text\ t_2\\&\textbf{instance } Algebra\ Pat_{simplify\#2}\ Html\ \textbf{where}\\&\quad alg\ (Mk_{simplify\#1}\ t\ x\ y) = Text\ t\ :+:\ x\ :+:\ y\end{aligned}$$

3.5 Lightweight Invariants for Free!

Using the machinery presented so far, we can represent 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 lightweight 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:

$$Con_{Text}\ \oplus\ Con_{(:+:)}\ \oplus\ Fun_{br}\ \oplus\ Fun_{bold}\ \oplus\ Fun_{list}\ \oplus\ Fun_{\langle+\rangle}$$

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:

$$Con_{Sing}\ \oplus\ Con_{Tag}\ \oplus\ Con_{(:+:)}\ \oplus\ Pat_{simplify\#1}$$

Since the only way to place a *Text* constructor within a value of this type is via the construction $Pat_{simplify\#1}$, 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 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 which 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 in a different frequency.

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:

$$\textbf{type}\ BGen\ a = Int \to Gen\ a$$

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:

$$\begin{aligned}&\textbf{class}\ BArbitrary\ (a :: *)\ \textbf{where}\\ &\quad barbitrary :: BGen\ a\end{aligned}$$

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 its 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:

$$\begin{aligned}&\textbf{instance}\ BArbitrary\ a \Rightarrow Arbitrary\ a\ \textbf{where}\\ &\quad arbitrary = sized\ barbitrary\end{aligned}$$

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:

$$\begin{aligned}&\textbf{class } Arbitrary1\ (f :: * \to *)\ \textbf{where}\\&\quad liftArbitrary :: Gen\ a \to Gen\ (f\ a)\end{aligned}$$

Then, we can use this same mechanism for our modular generators, extending $Arbitrary1$ to be depth-aware as follows:

$$\begin{aligned}&\textbf{class } BArbitrary1\ (f :: * \to *)\ \textbf{where}\\&\quad liftBGen :: BGen\ a \to BGen\ (f\ a)\end{aligned}$$

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:

$$\begin{aligned}&\textbf{instance } BArbitrary1\ Con_{Text}\ \textbf{where}\\&\quad liftBGen\ bgen\ d = Mk_{Text}\ \langle\$\rangle\ arbitrary\\&\textbf{instance } BArbitrary1\ Con_{Sing}\ \textbf{where}\\&\quad liftBGen\ bgen\ d = Mk_{Sing}\ \langle\$\rangle\ arbitrary\\&\textbf{instance } BArbitrary1\ Con_{Tag}\ \textbf{where}\\&\quad liftBGen\ bgen\ d = Mk_{Tag}\ \langle\$\rangle\ arbitrary\ \langle\star\rangle\ bgen\ (d-1)\\&\textbf{instance } BArbitrary1\ Con_{(:+:)}\ \textbf{where}\\&\quad liftBGen\ bgen\ d = Mk_{(:+:)}\ \langle\$\rangle\ bgen\ (d-1)\ \langle\star\rangle\ bgen\ (d-1)\end{aligned}$$

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 $Arbitrary$ instance—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_{bold} and $Pat_{simplify\#2}$ are as follows:

$$\begin{aligned}&\textbf{instance } BArbitrary1\ Fun_{bold}\ \textbf{where}\\&\quad liftBGen\ bgen\ d = Mk_{bold}\ \langle\$\rangle\ bgen\ (d-1)\\&\textbf{instance } BArbitrary1\ Pat_{simplify\#2}\ \textbf{where}\\&\quad liftBGen\ bgen\ d =\\&\qquad Mk_{simplify\#2}\ \langle\$\rangle\ arbitrary\ \langle\star\rangle\ bgen\ (d-1)\ \langle\star\rangle\ bgen\ (d-1)\end{aligned}$$

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* $f \oplus g$ 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 requires the final user to manually specify terminal constructions—a repetitive task which 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:

$$\textbf{data}\ ((f :: * \to *) \otimes (n :: \mathit{Nat}))\ r = \mathit{Freq}\ (f\ r)$$

This operator is parameterized by a type-level natural number n (of kind Nat) representing the desired generation frequency. In this light, the type $(f \otimes 3) \oplus (g \otimes 1)$ 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 $\oplus$, thus avoiding the need of parenthesis in types like the previous one. Analogously as Term, the operator $\otimes$ does not add any semantic information to the values it represents, so we can define its $\mathit{Algebra}$ and $\mathit{BArbitrary1}$ instance by delegating the work to the inner type as before:

$$\begin{aligned}
&\textbf{instance}\ \mathit{Algebra}\ f\ a \Rightarrow \mathit{Algebra}\ (f \otimes n)\ a\ \textbf{where}\\
&\quad \mathit{alg}\ (\mathit{Freq}\ f) = \mathit{alg}\ f\\
&\textbf{instance}\ \mathit{BArbitrary1}\ f \Rightarrow \mathit{BArbitrary1}\ (f \otimes n)\ \textbf{where}\\
&\quad \mathit{liftBGen}\ \mathit{bgen}\ d = \mathit{Freq}\ \langle\$\rangle\ \mathit{liftBGen}\ \mathit{bgen}\ d
\end{aligned}$$

With these two new type level combinators, Term and $\otimes$, 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:

$$(f \otimes 2) \oplus ((g \otimes 3) \oplus (\mathit{Term}\ h \otimes 5))$$

Our generation process will traverse this type one combinator at a time, processing each occurrence of $\oplus$ independently. This means that, in order to select the appropriate generation frequency of each operand we need to calculate the overall sum of frequencies on each side of the $\oplus$. For this purpose, we rely on Haskell's type-level programming feature known as *type families* [23]. In this light, we can implement a type-level function FreqOf to compute the overall sum of frequencies of a given representation type:

$$\begin{aligned}
&\textbf{type family}\ \mathit{FreqOf}\ (f :: * \to *) :: \mathit{Nat}\ \textbf{where}\\
&\quad \mathit{FreqOf}\ (f \oplus g) = \mathit{FreqOf}\ f + \mathit{FreqOf}\ g\\
&\quad \mathit{FreqOf}\ (f \otimes n) = n * \mathit{FreqOf}\ f\\
&\quad \mathit{FreqOf}\ (\mathit{Term}\ f) = \mathit{FreqOf}\ f\\
&\quad \mathit{FreqOf}\ _ = 1
\end{aligned}$$

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 $\otimes$ tag with the frequency of the type it is wrapping. This way, the type $((f \otimes 2) \oplus g) \otimes 3$ is equivalent to $(f \otimes 6) \oplus (g \otimes 3)$, 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 is defaulted 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 $\otimes$ combinators introduced at the beginning of this sub-section, along with the previous type-level computations over frequencies, we are finally in position of defining 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), InL <$> liftBGen bgen d)
      ,(freqVal @(FreqOf g), InR <$> liftBGen bgen d)]
    else frequency
      [(freqVal @(FreqOf' f), InL <$> liftBGen bgen d)
      ,(freqVal @(FreqOf' g), InR <$> 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 $\oplus$, 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:

$$freqVal :: \forall\, n\ .\ KnownNat\ n \Rightarrow Int$$

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* [7] (employing the `@(...)` syntax) to disambiguate to which natural value we are actually referring to. Then, for instance, the value

$$freqVal\ \ @(FreqOf\ \ (f \otimes 5 \oplus g \otimes 4))$$

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. This kind 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:

$$genRep :: \forall f\ a\ .\ (BArbitrary1\ f, Safe\ f, Algebra\ f\ a) \Rightarrow BGen\ a$$
$$genRep\ d = eval\ \langle\$\rangle\ barbitrary\ @(Fix\ f)\ d$$

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 $Html_{valid}$ and $Html_{simplify}$ 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 $Fun_{\langle+\rangle}$ or $Pat_{simplify\#1}$ 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, specially 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:

$$\textbf{type family}\ Con\ (c :: Symbol)$$
$$\textbf{type family}\ Fun\ (f :: Symbol)$$
$$\textbf{type family}\ Pat\ (p :: Symbol)\ (n :: Nat)$$

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 Con_{Text}, $Fun_{\langle +\rangle}$ and $Pat_{simplify\#1}$, 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 ensuring 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:

$$\textbf{instance}\ \mathit{Algebra}\ (\mathit{Con}_{\mathit{Leaf}}\ a)\ (\mathit{Tree}\ a)\ \textbf{where}\ \cdots$$
$$\textbf{instance}\ \mathit{Algebra}\ (\mathit{Con}_{\mathit{Node}}\ a)\ (\mathit{Tree}\ a)\ \textbf{where}\ \cdots$$
$$\textbf{instance}\ \mathit{Arbitrary}\ a \Rightarrow \mathit{BArbitrary1}\ (\mathit{Con}_{\mathit{Leaf}}\ a)\ \textbf{where}\ \cdots$$
$$\textbf{instance}\ \mathit{Arbitrary}\ a \Rightarrow \mathit{BArbitrary1}\ (\mathit{Con}_{\mathit{Node}}\ a)\ \textbf{where}\ \cdots$$

Then, instead of carrying this type parameter in our generation specifications, we can avoid it by hiding it behind an existential type:

$$\textbf{data}\ \mathit{Some}\ (f :: * \to * \to *)\ (r :: *) = \forall (a :: *)\ .\ \mathit{Some}\ (f\ a\ r)$$

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 appears 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 it to hide the additional type parameters of each construction representation:

$$\textbf{type instance}\ \mathit{Con}\ \texttt{"Leaf"} = \mathit{Term}\ (\mathit{Some}\ \mathit{Con}_{\mathit{Leaf}})$$
$$\textbf{type instance}\ \mathit{Con}\ \texttt{"Node"} = \mathit{Some}\ \mathit{Con}_{\mathit{Node}}$$

As a consequence, we can write generation specifications for our *Tree* data type without having to refer to its type parameter anywhere. For instance:

$$\begin{aligned}\textbf{type}\ \mathit{Tree}_{\mathit{Spec}} = {} & \mathit{Con}\ \texttt{"Leaf"} \otimes \mathit{2} \\ \oplus\ & \mathit{Con}\ \texttt{"Node"} \otimes \mathit{3}\end{aligned}$$

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:

$$\begin{aligned}&\mathit{genIntTree} :: \mathit{BGen}\ (\mathit{Tree}\ \mathit{Int}) \\ &\mathit{genIntTree} = \mathit{genRep}\ @(\mathit{Tree}_{\mathit{Spec}} \lhd \mathit{Int})\end{aligned}$$

Where $\lhd$ is a type family that simply traverses our generation specification, applying the *Int* type to each occurrence of *Some*, thus eliminating the existential type:

$$\begin{aligned}&\textbf{type family}\ (f :: * \to *) \lhd (a :: *) :: * \to *\ \ \textbf{where} \\ &\quad (\mathit{Some}\ f) \lhd a = f\ a \\ &\quad (f \oplus g) \lhd a = (f \lhd a) \oplus (g \lhd a) \\ &\quad (f \otimes n) \lhd a = (f \lhd a) \otimes n \\ &\quad (\mathit{Term}\ f) \lhd a = \mathit{Term}\ (t \lhd a) \\ &\quad f \lhd a = f\end{aligned}$$

As a result, in *genIntTree*, the $\lhd$ operator will reduce the type $(\mathit{Tree}_{\mathit{Spec}} \lhd \mathit{Int})$ to the following concrete type:

$$(Term\ (Con_{Leaf}\ Int) \otimes 2)\ \oplus\ ((Con_{Node}\ Int) \otimes 3)$$

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_L and In_R constructors of the $\oplus$ 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 [12]. 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 $(f \oplus g \oplus h \oplus i)$, which is implicitly parsed as $(f\ \oplus\ (g\ \oplus\ (h\ \oplus\ i)))$; we can associate constructions as $((f\ \oplus\ g)\ \oplus\ (h\ \oplus\ i))$, 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 amount 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 In_L and In_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 typeclass 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 [19], (ii) Lisp S-expressions (SExp), inspired by the package *hs-zuramaru*[16], 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* [20] 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.

As it can be observed, our approach suffers from a noticeable runtime overhead when using linearly defined representations, specially 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

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.

[16] `http://hackage.haskell.org/package/zuramaru`

when testing complex properties over monadic code, as well as using random values for penetration testing [9, 18].

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.

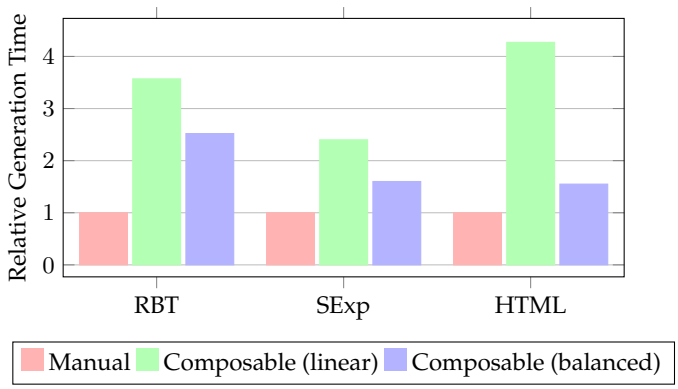


Figure 3: Generation time comparison between manually written and automatically derived composable generators.

7 Related Work

Extensible Data Types Swierstra proposed Data Types à la Carte [24], a technique for building extensible data types, as a solution for the *expression problem* coined by Wadler [26]. This technique has been successfully applied in a variety of scenarios, from extensible compilers, to composable machine-mechanized proofs [4, 5, 21, 27]. 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 [12]. 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* [22] 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 of 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 [14] 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 [8,9] 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 [18].

Feat [6] 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, specially 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 [18]. 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 shown to improve the code coverage over complex systems, when compared to other automated generators 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 [17]. 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 shown 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 the 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 of performing 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.

References

1. T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*, 2015.
2. K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.
3. Koen Claessen and John Hughes. Testing monadic code with QuickCheck. *SIGPLAN Not.*, 37(12):47–59, 2002.
4. L. E. Day and G. Hutton. Compilation À la carte. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, 2014.
5. Benjamin Delaware, Bruno C d S Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *ACM SIGPLAN Notices*, volume 48, pages 207–218. ACM, 2013.
6. J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2012.
7. Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. Visible type application. In *European Symposium on Programming*, pages 229–254. Springer, 2016.
8. G. Grieco, M. Ceresa, and P. Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *Proc. of the ACM SIGPLAN International Symposium on Haskell*, 2016.
9. G. Grieco, M. Ceresa, A. Mista, and P. Buiras. QuickFuzz testing for fun and profit. *Journal of Systems and Software*, 134, 2017.
10. J. Hughes, C. Pierce B, T. Arts, and U. Norell. Mysteries of DropBox: Property-based testing of a distributed synchronization service. In *Proc. of the Int. Conf. on Software Testing, Verification and Validation*, 2016.
11. 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.
12. H. Kiriyama, H. Aotani, and H. Masuhara. A lightweight optimization technique for data types a la carte. In *Companion Proceedings of the 15th Int. Conference on Modularity*, MODULARITY 2016, New York, NY, USA, 2016. ACM.
13. Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2009.
14. 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 *Proc. of the ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, 2017.
15. C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), January 2008.
16. J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson. Effect-driven QuickChecking of compilers. *In Proceedings of the ACM on Programming Languages, Volume 1*, (ICFP), 2017.
17. Agustín Mista and Alejandro Russo. Generating random structurally rich algebraic data type values. In *Proceedings of the 14th International Workshop on Automation of Software Test*, 2019.
18. Agustín Mista, Alejandro Russo, and John Hughes. Branching processes for QuickCheck generators. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2018.

19. Chris Okasaki. Red-black trees in a functional setting. *Journal of functional programming*, 9(4):471–477, 1999.
20. Bryan O'Sullivan. Criterion: a haskell microbenchmarking library, 2014.
21. Anders Persson, Emil Axelsson, and Josef Svenningsson. Generic monadic constructs for embedded languages. In *International Symposium on Implementation and Application of Functional Languages*, pages 85–99. Springer, 2011.
22. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and lazy SmallCheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, volume 44, pages 37–48. ACM, 2008.
23. 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.
24. Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.
25. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, 1989.
26. Philip Wadler. The expression problem, 1998.
27. Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. 2014.