Synthesizing Random Generators via Higher-Order Representations

Agustín Mista
Chalmers University of Technology
Gothenburg, Sweden
mista@chalmers.se

Alejandro Russo
Chalmers University of Technology
Gothenburg, Sweden
russo@chalmers.se

***Abstract*—Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat [1].

The main contribuitions of this paper are:

- We identify two patological scenarios for which standard type-driven automatic derivation tools fail to synthesize practical random generators, due to a lack of either type structure or domain knowleadge (Section II).
- We present a generation technique able to encode stronger properties of the target data by reifying the static information present on the program codebase (Section III).
- We apply and extend the theory of branching processes to analitically predict the average distribution of generated values. Furthermore, we use the predictions to perform simulation-based optimization of the random generation parameters (Section IV).
- We provide an implementation of our ideas in the form of a Haskell library to perform automatic derivation of random generators capable to extract useful structure information from the user source code.

II. Type-Driven Random Generation

In this section we briefly describe the common technique used for automatically deriving random data generators in Haskell using a type-driven approach. Then, we introduce its main drawbacks by considering two scenarios where this technique gives poor results in practice. We remark that, altough this work makes focus on Haskell data types, this technique is general enough to be applied to most programming languages with some level of support for composite types.

Haskell is a strongly typed programming language with a powerful type system. It lets programmers encode a considerable amount of information about the properties of their systems using data types that can be checked at compilation time. One of its key aspects is the support for Algebraic Data Types (ADTs). Essentially, an ADTs is a composite type defined by combining other types. In the most basic conception, types can be combined by **sums** (also known as *variant types* or *tagged disjoint unions*) and **products** (or tuples) of other data types. To exemplify this, consider the following definition of the data type Exp encoding integer expressions:

$$\begin{aligned}\mathbf{data}\ Exp = {} & Val\ \ Int \\ \mid {} & Add\ Exp\ Exp \\ \mid {} & Mul\ Exp\ Exp\end{aligned}$$

In the previous definition, we declare Exp as the sum of three possible classes of values: Val represents literal values, whereas Add and Mul represent the addition and multiplication of two integer expressions, respectively. In Haskell, Val, Add and Mul are called *data constructors* and are used to

distiguish which variant of the data type we are reffering to. Each data constructor is then defined as a product of zero or more types known as *fields*. In particular, Val contains a field of type Int, while Add and Mul contain two fields of type Exp representing the operands of each operation. Note that Exp is used as a field of at least one of its data constructors (case Add and Mul), making it a recursively defined ADT. 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 reprentation, the expression "2 + (5 * 6)" can be encoded simply by using Exp data constructors as $(Add\ (Val\ 2)\ (Mul\ (Val\ 5)\ (Val\ 6)))$. Furthermore, an evaluation function from Exps to integer values can be defined very idiomatically:

$$
\begin{aligned}
&eval :: Exp \to Int \\
&eval\ (Val\ n) \quad\ = n \\
&eval\ (Add\ x\ y) = eval\ x + eval\ y \\
&eval\ (Mul\ x\ y) = eval\ x * eval\ y
\end{aligned}
$$

In the previous definition, $eval$ is described using *pattern matching* over each possible kind of value. For the input case of a literal values, we simply return the value contained in the Val constructor. On the other hand, if the input value matches either an Add or a Mul data constructor, where x and y are *pattern variables* that match any subexpression, we recursively evaluate these subexpressions, combining them with the apropiate operation on each case.

A. *Type-Driven Generation of Random Values*

In order to perform random testing of a Haskell codebase involving user defined data types, most approaches require the user to provide a random data generator for each one of them. This tends to be a cumbersome and error prone task that closely follows the data type structure. For instance, consider the following definition of a *QuickCheck* random generator for the type Exp:

$$
\begin{aligned}
&genExp :: Gen\ Exp \\
&genExp = sized\ gen \\
&\quad \textbf{where} \\
&\qquad gen\ 0 = Val\ \langle\$\rangle\ arbitrary \\
&\qquad gen\ n = oneof \\
&\qquad\quad [\,Val\ \ \langle\$\rangle\ arbitrary \\
&\qquad\quad , Add\ \langle\$\rangle\ gen\ (n-1)\ \langle*\rangle\ gen\ (n-1) \\
&\qquad\quad , Mul\ \langle\$\rangle\ gen\ (n-1)\ \langle*\rangle\ gen\ (n-1)]
\end{aligned}
$$

This random generator is defined using *QuickCheck*'s combinator $sized :: (Int \to Gen\ a) \to Gen\ a$ to paremetrize the generation process up to an external integer number known as the *generation size*. This parameter is used to limit the maximum amount of recursive call that this random generator can perform. When called with a positive generation size (case $gen\ n$), the generator is free to randomly pick with uniform probability among any data constructor of Exp. In the case it picks to generate a Val data constructor, it also generates a random Int value using the standard overloaded generator $arbitrary$ (*QuickCheck* provides standard random generators for most base types like Int, $Bool$, etc.). On the other hand, when it randomly picks to generate either an Add or a Mul data constructor, it also generates independently two random subexpressions corresponding to the data constructor fields by calling itself recursively ($gen\ (n-1)$), ensuring to decrease the generation size by a unit on each recursive invocation.

This procedure keeps calling itself recursively until the generation size reaches zero (case $gen\ 0$), where the generator is constrained to pick only among terminal data constructors, being Val the only possible choice in our particular case. Strictly decreasing the the generation size by one on each recursive call results in a useful property over the generated data: every generated value has at most n levels, where n is the generation size set by the user. This property enables us to model the generation process using the theory of branching processes introduced by **CITATION NEEDED**, and extended in this work as described in Section IV.

Having discussed the previous random generator definition, it easy to understand how to extend this generation procedure to any data type defined in an algebraic fashion. Fortunately, there exists different meta-programming tools to avoid the user from having to mechanically write random generators over and over again for each user-defined ADT. The simplest tool for such purpose is *MegaDeTH*. Given the name of the target data type, it synthesizes a random generator for it that behaves pretty much like the previously defined one. However, picking among different data constructors with uniform probability can lead to a generation process biased to generate (in average) very small values, regardless of the generation sized set by the user **CITATION NEEDED**.

DRAGEN is meta-programming tool conceived to mitigate this problem. Instead of setting a uniform generation probability of data constructors, this tool considers the scenario where each one can be generated following a different (although fixed over time) generation probability. Then, this tool uses the theory of branching processes to analitically predict the average distribution of data constructors generated on each random value. This prediction mechanism is used to feedback a simulation-based optimization process that adjusts the generation probability of each data constructor in order to obtain a particular distribution of values that can be specified by the user, offering a more flexible testing environment while still being mostly automated.

Both *MegaDeTH* and **DRAGEN** synthesize random generators that are theoretically capable to generate the whole space of values of the target data type, provided that we set a generation size suficiently big. However, the limitations arise quickly when we consider that the underlying generation model is essentially the same: they pick a single data constructor and recusively generate each required subexpression independently. In practice, this procedure is too generic to generate complex data that is useful enough to be used for random testing.

In this work we identify two sources of complications that appear when using the aforementioned automatic derivation tools to obtain random data generators:

1) The target code behaves differently over inputs matching very specific patterns.
2) The target code behaves differently according to some dependency over subexpressions of its inputs.

To exemplify the first point, suppose we want to use randomly generated *Exp*s to test a property comprising the following function:

```
foo :: Exp → Exp
foo (Add (Add x (Val 50)) (Add (Val 25) y))
  = error "pattern #1"
foo (Mul (Mul x (Val 50)) (Mul (Val 25) y))
  = error "pattern #2"
foo x = x
```

This function behaves mostly like an identity function, except for two very specific patterns of input values defined using three nested levels of pattern matching. This is, *foo* does not only matches against the root data constructor, but also against the data constructors of its subexpressions and sub-subexpressions.

If we consider that *QuickCheck*'s default implementation for random *Int*s pick a number in the interval $[-n, n]$ with uniform distribution, we need to be able to identify which one is a sensible generation size to set based on wether our code branches on numbers or not, otherwise our random generator will not be able to generate random values to match against such numbers, leaving fragments of code completely untested. Suppose that we pick the a generation size 50, i.e. the minimum size that is big enough to produce an *Int* number equal to 50. Under this considerations, the probability of generating a value matching the first clause of *foo* (and hence triggering the error) are as follows:

$$\begin{aligned} P(match(foo_1)) &= P(Add) \\ &\quad * P(Add) * 1 * (P(Val) * (1/100)) \\ &\quad * P(Add) * (P(Val) * (1/100)) * 1 \end{aligned}$$

AM: COMPLETE

Despite the fact that the previous example might look artificial, branching against specific patterns of the input data is a common task. For instance, the balancing function of a Red-Black tree need to consider a quite specific combination of color, left- and right subtrees and sub-subtrees in order to preserve the height invariant. Moreover, Common Subexpression Elimination (CSE) is a compiler optimization that needs to consider very specific sequences of instructions that may be regrouped in a computationally more efficient way.

We believe that if a certain pattern of values is relevant enough to appear in the codebase being tested, a practical random testing methodology should be able to produce values satisfying such pattern in a substantial proportion.

```
fortyTwo :: Exp
fortyTwo = Val 42
twice :: Exp → Exp
twice x = Add x x
square :: Exp → Exp
square x = Mul x x
neg :: Exp → Exp
neg x = Mul x (Val (−1))
minus :: Exp → Exp → Exp
minus x y = Add x (neg y)
```

III. Extracting Structure

What to say here:

- Haskell ADTs
- Type driven generation
- Limitations
- Key idea: Higher order representations
- Canonical representation of the data type
- Pattern matchings representation
- Abstract interface representation
- Composing representations

Pellentesque dapibus suscipit ligula. Donec posuere augue in quam. Etiam vel tortor sodales tellus ultricies commodo. Suspendisse potenti. Aenean in sem ac leo mollis blandit. Donec neque quam, dignissim in, mollis nec, sagittis eu, wisi. Phasellus lacus. Etiam laoreet quam sed arcu. Phasellus at dui in ligula mollis ultricies. Integer placerat tristique nisl. Praesent augue. Fusce commodo. Vestibulum convallis, lorem a tempus semper, dui dui euismod elit, vitae placerat urna tortor vitae lacus. Nullam libero mauris, consequat quis, varius et, dictum id, arcu. Mauris mollis tincidunt felis. Aliquam feugiat tellus ut neque. Nulla facilisis, risus a rhoncus fermentum, tellus tellus lacinia purus, et dictum nunc justo sit amet elit.

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis,

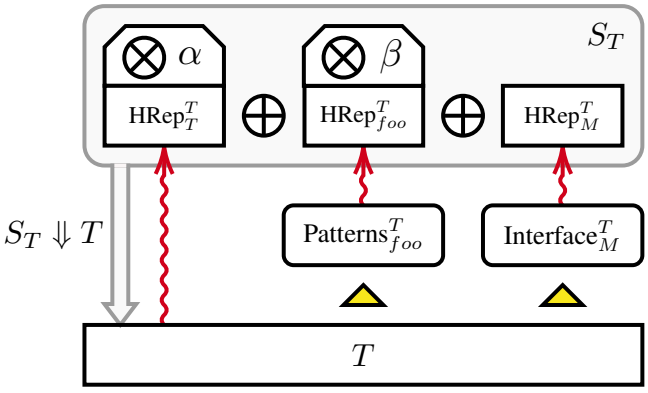

Fig. 1: The idea

magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

IV. Random Generators Synthesis

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

V. Case Studies

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

VI. Related Work

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Donec hendrerit tempor tellus. Donec pretium posuere tellus. Proin quam nisl, tincidunt et, mattis eget, convallis nec, purus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Nulla posuere. Donec vitae dolor. Nullam tristique diam non turpis. Cras placerat accumsan nulla. Nullam rutrum. Nam vestibulum accumsan nisl.

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

VII. Final Remarks

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare

nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

Nullam eu ante vel est convallis dignissim. Fusce suscipit, wisi nec facilisis facilisis, est dui fermentum leo, quis tempor ligula erat quis odio. Nunc porta vulputate tellus. Nunc rutrum turpis sed pede. Sed bibendum. Aliquam posuere. Nunc aliquet, augue nec adipiscing interdum, lacus tellus malesuada massa, quis varius mi purus non odio. Pellentesque condimentum, magna ut suscipit hendrerit, ipsum augue ornare nulla, non luctus diam neque sit amet urna. Curabitur vulputate vestibulum lorem. Fusce sagittis, libero non molestie mollis, magna orci ultrices dolor, at vulputate neque nulla lacinia eros. Sed id ligula quis est convallis tempor. Curabitur lacinia pulvinar nibh. Nam a sapien.

Aliquam erat volutpat. Nunc eleifend leo vitae magna. In id erat non orci commodo lobortis. Proin neque massa, cursus ut, gravida ut, lobortis eget, lacus. Sed diam. Praesent fermentum tempor tellus. Nullam tempus. Mauris ac felis vel velit tristique imperdiet. Donec at pede. Etiam vel neque nec dui dignissim bibendum. Vivamus id enim. Phasellus neque orci, porta a, aliquet quis, semper a, massa. Phasellus purus. Pellentesque tristique imperdiet tortor. Nam euismod tellus id erat.

REFERENCES

[1] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "QuickFuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, no. Supp. C, 2017.