Branching Processes for QuickCheck Generators

Agustín Mista
Universidad Nacional de Rosario
amista@dcc.fceia.unr.edu.ar

Alejandro Russo
Chalmers University of Technology
russo@chalmers.se

John Hughes
Chalmers University of Technology
rjmh@chalmers.se

Abstract

QuickCheck is a well-known tool for property-based testing. It often requires users to manually write random data generators for those types involved in the properties under consideration—especially when considering user-defined algebraic data types (ADTs). One challenging aspect of using *QuickCheck* with ADTs is to control generators' distributions. We present a novel idea to automatically synthesize *QuickCheck* generators while providing some control over constructors' distributions. 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 to shape generators' distributions. We provide an implementation of our mechanism in a tool called *DRaGeN* and perform case studies with real-world applications. Our results show improvements in code coverage when compared with state-of-the-art tools to automatically synthesize QuickCheck generators.

Keywords Branching process, QuickCheck, Testing, Haskell

1 Introduction

Random property-based testing is an increasingly popular approach to finding bugs [2, 16, 17]. In the Haskell community, *QuickCheck* [9] 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* [26] (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* [23] and *MegaDeTH* [13, 14] 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* [11] 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 [8]. Lastly, *Luck* is a domain specific language for manually writing *QuickCheck* properties in tandem with generators so that it becomes possible to fine-control the distribution of generated values [18].

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* 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.

One of our main contributions is to provide a mathematical foundation which helps to analytically characterize the distribution of constructors in derived *QuickCheck* generators for ADTs. Our realization is that *branching processes*, a relatively simple stochastic model conceived to study the evolution of populations, can be applied to predict the 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. We design (compile-time) heuristics that automatically search for probability parameters so that distributions of constructors can be adjusted to what developers might want. For instance, it is possible for developers to obtain weighted distribution where one (or many) constructors are required to appear n-times more frequently (on average) than other ones—a property of distributions that is not easy to achieve when considering complex ADTs. We also show how to use type reification to simplify our prediction process and extend our model to mutually recursive and composite types. We provide an implementation of our ideas

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00

in the form of a Haskell library[1] called *DRaGeN* (for Derivation of RAndom GENerators). We evaluate it by generating inputs for real-world programs, where our tool 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.

This paper is organized as follows. Section 2 briefly introduces *QuickCheck* generators. The background on branching processes is presented in Section 3, and extended in Section 4. The special treatment required to predict terminal constructors is addressed in Section 5. Section 6 shows simplifications as well as extensions required to cover richer ADTs. The implementation of our tool is discussed on Section 7 and case studies are given in Section 8. Section 9 positions our work with respect to others and Section 10 concludes.

2 Background

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

data $Tree = Leaf_A \mid Leaf_B \mid Leaf_C \mid Node\ Tree\ Tree$

In order to help developers write generators, *QuickCheck* defines the *Arbitrary* type-class with the overloaded symbol $arbitrary :: Gen\ \ a$, which denotes a monadic generator for values of type a. Then, to generate random trees we need to provide an instance of the *Arbitrary* type-class for the type *Tree*. Figure 1 shows a possible implementation. At the top level, this generator simply uses the *QuickCheck*'s primitive $oneof :: [Gen\ \ a] \rightarrow Gen\ \ a$ to pick a generator from a list of generators with uniform probability. This list consists of a random generator for each possible choice of type constructor of *Tree*. We use *applicative style* [21] to describe each one of them idiomatically. So, $pure\ Leaf_A$ is a generator that always generates $Leaf_A$s, while $Node \langle\$\rangle\ arbitrary \langle*\rangle\ arbitrary$ is a generator that always generates *Node* constructors, "filling" its arguments by calling *arbitrary* recursively on each one 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.

[1] Available at https://bitbucket.org/agustinmista/dragen

instance *Arbitrary Tree* **where**
  $arbitrary = oneof\ [pure\ Leaf_A, pure\ Leaf_B, pure\ Leaf_C$
    $, Node \langle\$\rangle\ arbitrary \langle*\rangle\ arbitrary]$

Figure 1. Random generator for *Tree*.

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* [23]. This library uses Template Haskell [27] 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 is generated, which can lead to infinite generation loops. A detailed example of this phenomenon is given in Appendix B.1. Because of this non-terminating behavior, we do not compare *derive* with the other derivation tools addressed by this work.

2.2 *MegaDeTH*

The second approach we will discuss is the one taken by *MegaDeTH*, a meta-programming tool used intensively by *QuickFuzz* [13, 14]. Firstly, *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 $sized :: (Int \rightarrow Gen\ a) \rightarrow Gen\ a$ to build a random generator based on a function (of type $Int \rightarrow Gen\ a$) that limits the possible recursive calls performed when creating random values. The integer passed to *sized*'s argument is called the *generation size*. When the generation size is zero (see definition $gen\ \ 0$), the generator only chooses between the *Tree*'s terminal constructors—thus ending the generation process. If the generation size is strictly positive, it is free to randomly generate any *Tree* constructor (see definition $gen\ n$). When it chooses to generate a recursive constructor, it reduces the generation size for its subsequent recursive calls by a factor that depends on the number of recursive arguments this constructor has ($div\ \ n\ 2$). In this way, *MegaDeTH* ensures that all generated values are finite.

Although *MegaDeTH* generators always terminate, they have a major practical drawback: in our example, the use of *oneof* to uniformly decide the next constructor to be generated produces a generator that generates leaves approximately

instance *Arbitrary Tree* **where**
  $arbitrary = sized\ gen$ **where**
    $gen\ 0 = oneof$
      $[pure\ Leaf_A, pure\ Leaf_B, pure\ Leaf_C]$
    $gen\ n = oneof$
      $[pure\ Leaf_A, pure\ Leaf_B, pure\ Leaf_C$
      $, Node \langle\$\rangle\ gen\ (div\ \ n\ 2) \langle*\rangle\ gen\ (div\ \ n\ 2)]$

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

three quarters of the time. 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* [11]. 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.[2] However, the distribution of size, given by the number of constructors in the generated values, is highly dependent on the structure of the data type being considered. Figure 3 (right) shows the overall distribution shape of a *QuickCheck* generator derived using *Feat* for *Tree* using a generation size of 400, i.e., generating values of up to 400 constructors.[3] Notice that all the generated values are close to the maximum size! This phenomenon follows from the exponential grow in the number of possible *Tree*s of n constructors as we increase n. In other words, the space of *Tree*s of 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 bound to be 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 only consider 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.

3 Simple-Type Branching Processes

Galton-Watson Branching processes (or branching processes for short) is an area of mathematics dedicated to study 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 [15] for an excellent introduction. In this section, we show how to use this theory to model *QuickCheck*'s distribution of constructors.

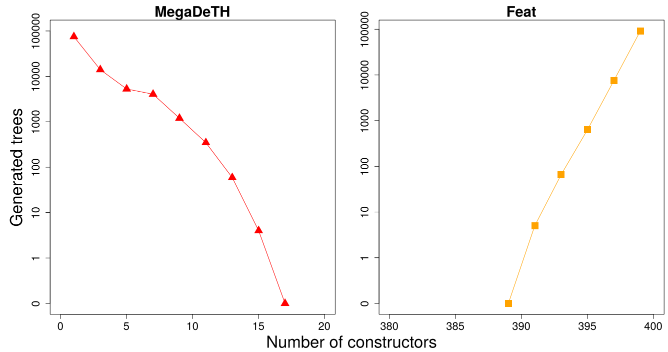

Figure 3. Size distribution of 100000 randomly generated *Tree* values using *MegaDeTH* (▲) with generation size 10, and *Feat* (■) with generation size 400.

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 illustrate 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 denotes 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 it may grow forever.

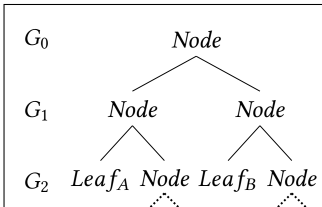

Figure 4. Generation of *Node* constructors.

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

[2]We avoid to include any source code generated by *Feat*, since it works synthetizing *Enumerable* type-class intances instead of *Arbitrary* ones. Such instances give no insight on how the derived random generators work.
[3] 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.

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 able 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 *Node* offspring in the next generation (i.e., in the recursive calls of *arbitrary*) are as follows:

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

Now we have determined the distribution of R, we proceed to introduce 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 $E[G_n]$, we apply the (standard) Law of Total Expectation $E[X] = E[E[X|Y]]$ with $X = G_n$ and $Y = G_{n-1}$ to obtain:

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

By expanding G_n, we deduce that:

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

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

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

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

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

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

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

By unfolding this recursive equation many times, we obtain:

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

As the equation indicates, the expected number of *Node* constructors at the nth generation is affected by the mean of reproduction. 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:

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

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:

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

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 $Tree' = Leaf \mid Node_A\ Tree'\ Tree' \mid Node_B\ Tree'$

In this case, we need to separately consider that a $Node_A$ can generate not only $Node_A$ but also $Node_B$ offspring (similarly with $Node_B$). 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-Walton processes known as *multi-type branching process*.

4 Multi-Type Branching Processes

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

Before we dive into technicalities, Figure 5 shows the automatically derived generator for $Tree'$ that our tool produces. Our generators depend on the (possibly) different probabilities that constructors have to be generated—variables p_{Leaf},

p_{NodeA}, and p_{NodeB}. These probabilities are used by the function *chooseWith* :: [(*Double*, *Gen a*)] → *Gen a*, which picks a random generator of type a with an explicitly given probability from a list. This function can be easily expressed by using *QuickCheck*'s primitive operations and therefore we omit its implementation. Additionally note that, like *MegaDeTH*, our generators use *sized* to limit the number of recursive calls to ensure termination. We note that the theory behind branching processes is able to predict the termination behavior of our generators and we could have used this ability to ensure their termination without the need of a depth limiting mechanism like *sized*. However, using *sized* provides more control over the obtained generator distributions.

To predict the distribution of constructors provided by *DRaGeN* generators, we introduce a generalization of the previous Galton-Watson branching process called multi-type Galton-Watson branching process. This generalization allows us to consider several *kinds of individuals*, i.e., constructors in our setting, to procreate (generate) different *kind 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′*.

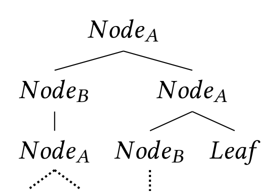

Figure 6. A generated value of type *Tree′*.

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

```
instance Arbitrary Tree′ where
  arbitrary = sized gen where
    gen 0 = pure Leaf
    gen n = chooseWith
      [(p_Leaf   , pure Leaf)
      , (p_NodeA, Node_A ⟨$⟩ gen (n−1) ⟨∗⟩ gen (n−1))
      , (p_NodeB, Node_B ⟨$⟩ gen (n−1))]
```

Figure 5. *DRaGeN* generator for *Tree′*

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 deduce $E[G_n^j|G_{n-1}]$, we realize that the expected number of jth constructors at the nth generation is the expected number of jth constructors produced by the different parent constructors at the *(n-1)*th generation. Refer to Appendix B.2 for a formal verification of this step (★). Similarly as before, we rewrite $E[R_{ij}]$ as m_{ij}, obtaining:

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

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 call the *mean matrix of constructors* (or mean matrix for simplicity) to the matrix M_C such that each $(m_{ij})_{i,j=1}^d$ 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 in compile-time using statically known type information. We are now able to deduce $E[G_n^j]$.

$$E[G_n^j] \overset{\text{by prob.}}{=} E[E[G_n^j|G_{n-1}]] \overset{(6)}{=} E\left[\sum_{i=1}^{d} G_{(n-1)}^i \cdot m_{ij}\right]$$

$$\overset{\text{by prob.}}{=} \sum_{i=1}^{d} E[G_{(n-1)}^i \cdot m_{ij}] \overset{\text{by prob.}}{=} \sum_{i=1}^{d} E[G_{(n-1)}^i] \cdot m_{ij}$$

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:

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

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. Is now possible to obtain the expected population of all the constructors but in a clustered manner:

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

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

$$E[P_n]^T \overset{\text{by geo. series}}{=} E[G_0]^T \cdot \left(\frac{I - (M_C)^{n+1}}{I - M_C}\right) \quad (9)$$

Where symbol I is 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 $Node_A$ is:

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

The rest of M_C can be similarly computed, obtaining:

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

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.[4]

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

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

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 only choose between terminal constructors at the last generation level (recall that *gen* 0 only generates *Leaf*s in Figure 5). As an example, if we assign our generation probabilities for *Tree′* as $p_{Leaf} \mapsto 0.2, p_{Node_A} \mapsto 0.5$ and $p_{Node_B} \mapsto 0.3$, then the formula predicts that our *QuickCheck* generator with a size parameter of 10 will generate on average 21.322 $Node_A$s and 12.813 $Node_B$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 $Node_A$s and $Node_B$s across the generated values.

[4] The careful reader might have noticed 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.

In this section, we precisely obtain the 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*, respectively. 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 B.3 for a graphical interpretation. On one hand, 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 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 $Node_A$ and one *Leaf* for every $Node_B$ constructor at level $(n-1)$.

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

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

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

Although we can now predict the expected number of generated *Tree′* constructors regardless of whether they are terminal or not, this approach only works for data types with a single terminal constructor. If we have a data type with multiple terminal constructors, we have to consider the probabilities

```
instance Arbitrary Tree″ where
  arbitrary = sized gen where
    gen 0 = chooseWith
      [(p*_LeafA, pure Leaf_A), (p*_LeafB, pure Leaf_B)]
    gen n = chooseWith
      [(p_LeafA, pure Leaf_A), (p_LeafB, pure Leaf_B)
      , (p_NodeA, Node_A ⟨$⟩ gen (n−1) ⟨∗⟩ gen (n−1))
      , (p_NodeB, Node_B ⟨$⟩ gen (n−1))]
```

Figure 7. Derived generator for $Tree''$

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:

data $Tree'' = Leaf_A \,|\, Leaf_B \,|\, Node_A\ Tree''\ Tree'' \,|\, Node_B\ Tree''$

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:

$$p^*_{Leaf_A} = \frac{p_{Leaf_A}}{p_{Leaf_A} + p_{Leaf_B}} \qquad p^*_{Leaf_B} = \frac{p_{Leaf_B}}{p_{Leaf_A} + p_{Leaf_B}}$$

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

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

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

Where the case of $E[Leaf_B]$ 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* → *Gen a* → *Gen a*, it resets the generation size of a given generator to a new value. We use it to decrement the

```
data T1 = A | B T1 T2
data T2 = C | D T1

instance Arbitrary T1 where
  arbitrary = sized gen where
    gen 0 = pure A
    gen n = chooseWith
      [(p_A, pure A)
      , (p_B, B ⟨$⟩ gen (n−1) ⟨∗⟩ resize (n−1) arbitrary)]

instance Arbitrary T2 where
  arbitrary = sized gen where
    gen 0 = pure C
    gen n = chooseWith
      [(p_C, pure C), (p_D, D ⟨$⟩ resize (n−1) arbitrary)]
```

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

generation size at the recursive calls of *arbitrary* that generates 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:

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

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

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

We define the mean of the initial generation as $E[G_0] = (p_A, p_B, 0, 0)$—we assing $p_C = p_D = 0$ 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.

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,

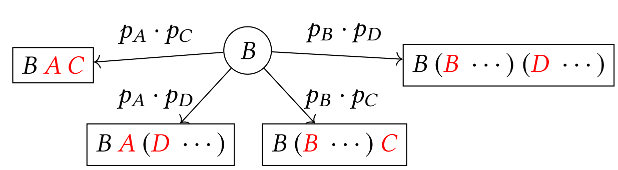

Figure 9. Possible offspring of constructor B.

handling M_C like this results in a high computational cost. We show next how we can not 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 $2 \cdot p_L$ (10), where 2 is the number of occurrences of *Tree′* in the declaration of $Node_A$. 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 $Node_A$, one place holder when generating $Node_B$, or zero place holders when generating a *Leaf* (this 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:

$$M_{Tree'} = \begin{array}{c} \\ Tree' \end{array} \begin{array}{c} Tree' \\ \left[2 \cdot p_{NodeA} + p_{NodeB} \right] \end{array} \qquad M_{T_1T_2} = \begin{array}{c} \\ T1 \\ T2 \end{array} \begin{array}{cc} T1 & T2 \\ \end{array} \begin{bmatrix} p_B & p_B \\ p_D & 0 \end{bmatrix}$$

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 A) 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 *n*th-generations of constructors and type place holders respectively. C_i^t represents the *i*th-constructor of the type T_t. In other words, 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.

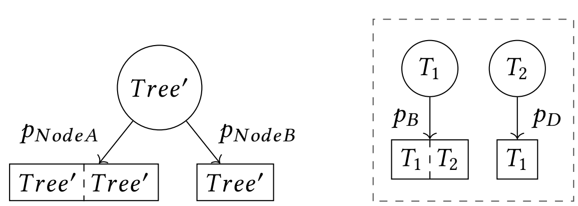

Figure 10. Offspring as types

6.1 Composite types

In this section, 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:

data *Tree* = $Leaf_A$ *Bool* | $Leaf_B$ *Bool Bool* | $\cdots$

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:

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

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 $Leaf_A$ now carries a value of type *Maybe Bool*:

data *Tree* = $Leaf_A$ (*Maybe Bool*) | $Leaf_B$ *Bool Bool* | $\cdots$

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:

$$E[True] = p_{True} \cdot (1 \cdot E[Leaf_A] \cdot p_{Just} + 2 \cdot E[Leaf_B])$$

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 $Leaf_B$ 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 $Leaf_A$ constructor. Having this graph together with the application of the multi-type branching process, we can

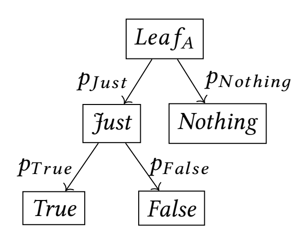

Figure 11. Constructor dependency graph.

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 describe 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 $dragenArbitrary :: Name \rightarrow Size \rightarrow CostFunction \rightarrow Q\ [Dec]$, where developers indicates 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 of 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. We proceed to explain the concept of *cost functions*, an essential piece in charge to provide the flexibility to our framework.

7.1 Cost functions

The optimization process is guided by a user provided cost function. In our setting, a cost function is a mapping between the generation size chosen by the user and a constructors' probabilities assignment to a real number—the cost.

$$\textbf{type}\ CostFunction = Size \rightarrow ProbMap \rightarrow Double$$

ProbMap encodes a constructors' probabilities assignment candidate as a map from constructor names to real numbers. Our optimization algorithm works generating several *ProbMap* 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 is that particular cost function intended to encode. Then, the optimization process simply finds the best probabilities assignment by minimizing the provided cost function.

Currently, our tool provides a basic set of cost functions to easily describe the expected distribution of the derived generator. For instance, $uniform :: CostFunction$ encodes constuctor-wise uniform generation, an interesting property that naturally arises from our generation process formalization. It guides the optimization process to a generation distribution that minimizes the difference between the expected number of each generated constructor and the generation size. Moreover, the user can restrict the generation distribution to a certain subset of constructors using the cost functions $only :: [Name] \rightarrow CostFunction$ and $without :: [Name] \rightarrow CostFunction$ to describe these restrictions. In this case, the whitelisted constructors are then generated following the *uniform* behavior. Similarly, if the branching process involves mutually recursive data types, the user could restrict the generation to a certain subset of data types by using the functions *onlyTypes* and *withoutTypes*. Additionally, when the user wants to generate constructors according to certain proportions, $weighted :: [(Name, Int)] \rightarrow CostFunction$ allows to encode this property, e.g. three times more $Leaf_A$'s than $Leaf_B$'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 dissapear as we increase the number of generated values. As for the cost functions behavior, there are some interesting aspects to note. For instance, in the *uniform* the optimizer cannot do anything to break the implicit invariant of the data type: every binary tree with n nodes has $n+1$ leaves. Instead, it converges to a solution that "approximates" a uniform distribution around the generation size parameter. We believe this is desirable behavior, to find an approximate solution when certain invariants prevent the optimization process from finding an exact solution. This way the user does not have to be aware of the possible invariants that the target data type may have, obtaining a solution that is good enough for most purposes. On the other hand, notice that in the *weighted* case at the second row of Table 1, the expected number of generated *Node*s is considerably large. This constructor is not listed at 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, parametrized on the cost function provided.

Cost Function	Predicted Expectation				Observed Expectation			
	$Leaf_A$	$Leaf_B$	$Leaf_C$	*Node*	$Leaf_A$	$Leaf_B$	$Leaf_C$	*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

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

This optimizer is based on a standard local-search optimization algorithm that recursively chooses the best constructor probability assignment 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 mechanishm 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 parametrized by our previously optimized probabilities. Refer to Appendix B.4 for further details on the cost functions and algorithms addressed by this section.

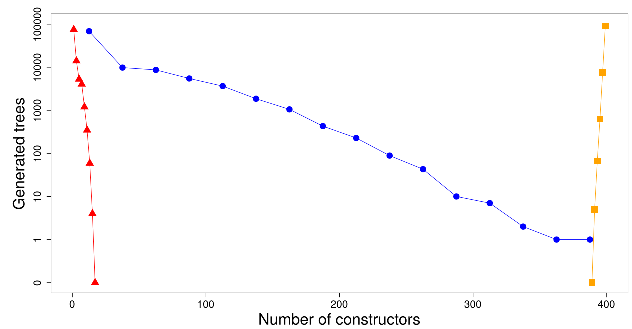

Figure 12. *MegaDeTH* (▲) vs. *Feat* (■) vs. *DRaGeN* (●) generated distributions for type *Tree*.

8 Case Studies

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

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

We target three complex and widely used 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 inter-operate with them: *language-bash*, *atto-lisp*, and *JuicyPixels*, respectively. These libraries provide ADT definitions which we used to synthesize *DRaGeN* generators for the inputs to 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 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 constuctors)—refer to B.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) [20]—a well known fuzzer. It was chosen in this work since it is also used in [14] to compare *MegaDeTH* with other techniques. The process consists of the *instrumentation* of the binaries under test, making them able to return the path in the code taken by each execution. Then, we use AFL to count how many different executions are triggered by a set of randomly generated files—also known as a corpus. In this evaluation, we compare how different *QuickCheck* generators, derived using *MegaDeTH* and using our approach, result in different code coverage when testing external programs, as a function of the size of a set of independently, randomly generated corpora. We have not been able to automatically derive such generators using *Feat*, since it does not work with some Haskell extensions used in the bridging libraries.

We generated each corpora 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—see Appendix B.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* [13, 14] 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 provides a framework to exhaustively test data sets up to a certain (small) size [26]. 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 [24]. Recently, Midtgaard et al. extends such a technique to test compilers for impure programming languages [22].

Luck [18] 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 including in *DRaGeN* some functionalities borrowed from *Luck* is an interesting path for future work. Recently, Lampropoulos et al. propose a framework to automatically derive random generators for a large subclass of Coq's inductively defined relations [19]. 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 [10] is a general approach to randomly generating combinatorial structures such as trees and graphs—also extended to work with closed simply-typed lambda terms [4]. 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 gives 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 [25]. 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.

Similarly to our work, Feldt and Poulding propose *GödelTest* [12], 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 to exploit 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 [5]. This is the main idea of *Adaptive Random Testing* (ART) [6]. However, this work only covers the particular case of testing programs with numerical inputs and it has also been argued

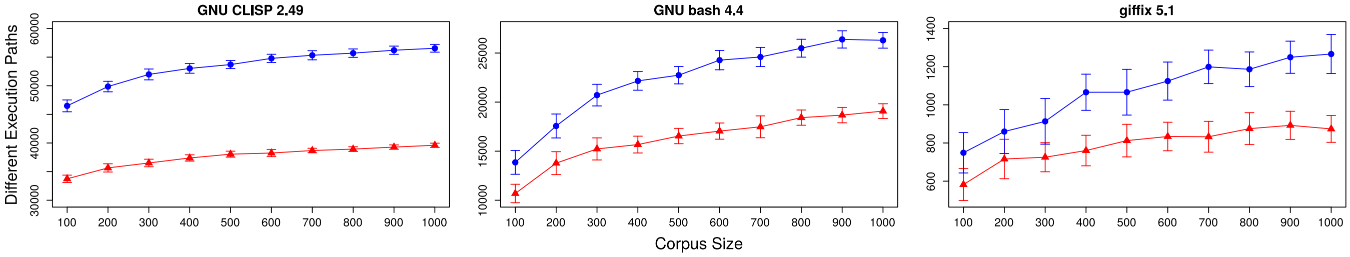

Figure 13. Path coverage comparison between *MegaDeTH* (▲) and *DRaGeN* (●).

that adaptive random testing has inherent inefficiencies compared to random testing [1]. This strategy is later extended in [7] 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 ADTs 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 manner 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. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM.

[2] T. Arts, J. Hughes, U. Norell, and H. Svensson. 2015. Testing AUTOSAR software with QuickCheck. In *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*.

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

[4] M. Bendkowski, K. Grygiel, and P. Tarau. 2017. Boltzmann Samplers for Closed Simply-Typed Lambda Terms. In *In Proc. of International Symposium on Practical Aspects of Declarative Languages*. ACM.

[5] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. 1996. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology* 38, 12 (1996), 775 – 782.

[6] T. Y. Chen, H. Leung, and I. K. Mak. 2005. Adaptive Random Testing. In *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, Michael J. Maher (Ed.). Springer Berlin Heidelberg.

[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *In Proc. of International Conference on Software Engineering*. ACM/IEEE.

[8] K. Claessen, J. Duregård, and M. H. Palka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Proc. of the Functional and Logic Programming FLOPS*.

[9] K. Claessen and J. Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[10] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. 2004. Boltzmann Samplers for the Random Generation of Combinatorial Structures. *Comb. Probab. Comput.* 13 (2004).

[11] J. Duregård, P. Jansson, and M. Wang. 2012. Feat: Functional enumeration of algebraic types. In *Proc. of the ACM SIGPLAN Symposium on Haskell*.

[12] R. Feldt and S. Poulding. 2013. Finding test data with specific properties via metaheuristic search. In *In Proc. of International Symp. on Software Reliability Engineering (ISSRE)*. IEEE.

[13] G. Grieco, M. Ceresa, and P. Buiras. 2016. QuickFuzz: An automatic random fuzzer for common file formats. In *Proc. of the International Symposium on Haskell*. ACM.

[14] G. Grieco, M., A. Mista, and P. Buiras. 2017. QuickFuzz testing for fun and profit. *Journal of Systems and Software* 134, Supplement C (2017).

[15] P. Haccou, P. Jagers, and V. Vatutin. 2005. *Branching processes. Variation, growth, and extinction of populations.* Cambridge University Press.

[16] J. Hughes, C. Pierce B, T. Arts, and U. Norell. 2016. Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service. In *Proc. of the Int. Conference on Software Testing, Verification and Validation, ICST.*

[17] J. Hughes, U. Norell, N. Smallbone, and T. Arts. 2016. Find more bugs with QuickCheck!. In *Proc. of the International Workshop on Automation of Software Test, AST@ICSE.*

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

[19] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. 2017. Generating Good Generators for Inductive Relations. *Proc. ACM Programming Languages* 2, POPL, Article 45 (2017).

[20] M. Zalewski. 2010. American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/. (2010).

[21] C. Mcbride and R. Paterson. 2008. Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (Jan. 2008).

[22] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson. 2017. Effect-driven QuickChecking of compilers. *Proceedings of the ACM on Programming Languages, Volume 1* ICFP (2017).

[23] N. Mitchell. 2007. Deriving Generic Functions by Example. In *Proc. of the 1st York Doctoral Syposium.* Tech. Report YCS-2007-421, Department of Computer Science, University of York, UK, 55–62.

[24] M. Palka, K. Claessen, A. Russo, and J. Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *The IEEE/ACM International Workshop on Automation of Software Test (AST 2011).*

[25] S. M. Poulding and R. Feldt. 2017. Automated Random Testing in Multiple Dispatch Languages. *IEEE International Conference on Software Testing, Verification and Validation (ICST)* (2017).

[26] C. Runciman, M. Naylor, and F. Lindblad. 2008. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Proc. of the ACM SIGPLAN Symposium on Haskell.*

[27] T. Sheard and Simon L. Peyton Jones. 2002. Template meta-programming for Haskell. *SIGPLAN Notices* 37, 12 (2002), 60–75.

A Demonstrations

In this appendix, we provide the formal development to show that the mean matrix of types can be used to predict the distribution of constructors. We start by defining some terminology.

Definition. 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.

Proposition A.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. 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:

$$X_k^{C_i^u C_j^v} : cons(T_v) \longrightarrow \mathbb{N}$$

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

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

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

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

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

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

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

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

■

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

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

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

Proof. Let $m_{T_u T_v}$ be an element of M_T, we know that $m_{T_u T_v}$ represents the expected number of placeholders of type T_v generated whenever a placeholder of type T_u is generated, i.e. by any of its constructors. Therefore, we need to average the number of 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.

$$Y^{uv} : cons(T_u) \longrightarrow \mathbb{N}$$

$$Y^{uv}(C_k^u) = \beta(T_v, C_k^u)$$

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

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

■

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

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

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

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

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

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

Now, we can rewrite (17) as follows:

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

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

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

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

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

■

Now, we proceed to prove our main result.

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

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

■

B 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.

B.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 of termination ensuring mechanisms. 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.* Aditionally, 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 probabilty of generating a i constructor. In this case $d = 1/2$.

Figure 14 provides an empirical verification of this non-terminating behavior. It shows the distribution (in terms of amount of constructors) of 100000 randomly generated T values obtained using the *derive* generator shown above. The black bar of 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.

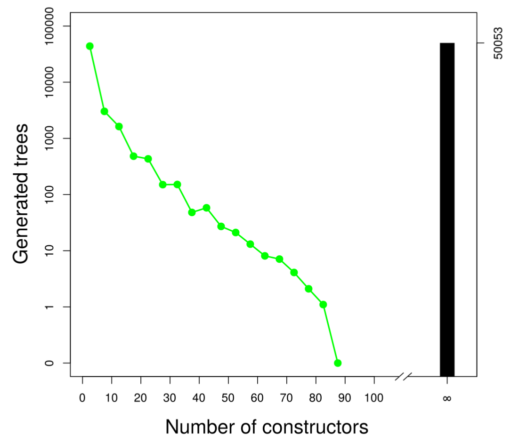

Figure 14. Distribution of (the amount of) T constructors induced by *derive*.

B.2 Multi-type Branching Processes

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

$$E[G^j_n | G_{n-1}] = E\left[\sum_{i=1}^{d} \sum_{p=1}^{G_{n-1}} \xi^p_{ij}\right]$$

Where symbol ξ^p_{ij} 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 $\xi^p_{ij} = 0$. 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:

$$E[G^j_n | G_{n-1}] = \sum_{i=1}^{d} \sum_{p=1}^{G_{n-1}} E\left[\xi^p_{ij}\right]$$

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:

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

B.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.

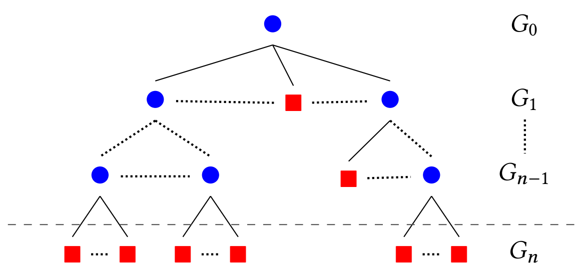

Figure 15. Generation processes of non-terminal (●) and terminal (■) constructors.

B.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 mechanishm and the probabilities optimizator.

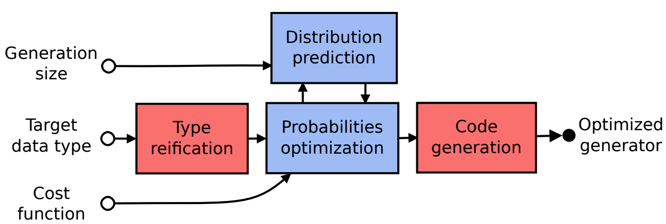

Figure 16. Generation schema.

Cost functions The probabilities optimizer that our tool implements essentially works minimizing a provided cost function that encodes the desired distribution of constructors at the optimized generator. As shown in Section 7, *DRaGeN* comes with a minimal set of useful cost functions. Such functions are built around the *Chi-Square Goodness of Fit Test* [3], 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 funcions 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)^2 / 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 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.

```
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.

B.5 Case studies

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

Case Study	Number of involved types	Number of involved constructors	Composite types	Mutually recursive types
Lisp	7	14	Yes	Yes
Bash	31	136	Yes	Yes
Gif	16	30	Yes	No

Table 2. Type information for ADTs used in the 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.

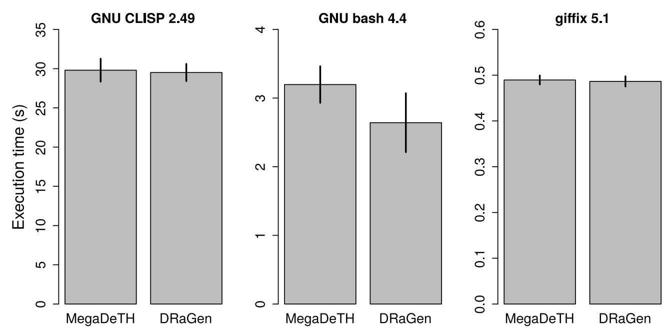

Figure 19. Execution time required to test the biggest randomly generated corpora consisting of 1000 files.