Generating Random Structurally Rich Algebraic Data Type Values

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

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

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

***Index Terms*—random testing, algebraic data types, Haskell**

I. Introduction

Random testing is a promising approach for finding bugs [1]–[3]. *QuickCheck* [4] is the dominant tool of this sort used by the Haskell community. It requires developers to specify (i) *testing properties* describing programs' expected behavior and (ii) *random data generators* based on the *types* of the expected inputs (e.g., integers, strings, etc.). *QuickCheck* then generates random test cases and reports violating testing properties.

QuickCheck comes equipped with random generators for built-in types, while it requires to manually write generators for user-defined ADTs. Recently, there has been a proliferation of tools to automatically derive *QuickCheck* generators for ADTs [5]–[9]. The main difference about these tools lies on the guarantees provided to ensure *the termination of the generation process* and the *distribution of random values*. Despite their differences, these tools guarantee that generated values are *well-typed*. In other words, generated values follow the structure described by ADT definitions.

Well-typed ADT values are specially useful when testing programs which expect highly structured inputs like compilers [10]–[12]. Generating ADT values also proves fruitful when looking for vulnerabilities in combination with fuzzers [8], [13]. Despite these success stories, ADT type-definitions do not often capture all the invariants expected from the data that they are intended to model. As a result, even if random values are well-typed, they might not be built with enough structure to penetrate into deep software layers.

In this work, we identify two different sources of structural information that can be statically exploited to improve the generation process of ADT values (Section III). Then, we show how to capture this information into our (automatically) derived random generators. More specifically, we propose a generation process that is capable of considering how programs branch on input ADTs values as well as how they get manipulated by abstract interfaces (Section IV). Furthermore, we show how to predict the *expected* distribution of the ADT constructors, values fitting certain branching patterns, and calls to interfaces that our random generators produce. For that, we extend some recent results on applying *branching processes* [14]—a simple stochastic model conceived to study population growth (Section V). We implement our ideas as an extension of the already existing derivation tool called DRAGEN [9]. We call our extension as DRAGEN2[1] to make it easy the distinction for the reader. DRAGEN2 is capable of automatically synthesizing *QuickCheck* generators which produce *rich ADT values*, where the distributions of random values can be adjusted at compile-time to what developers might want. Finally, we provide empirical evaluations showing that including static information from the user codebase improves the code coverage of two external applications when tested using random values generated following our ideas (Section VI).

We remark that, although this work focuses on Haskell algebraic data types, this technique is general enough to be applied to most programming languages.

II. Background

In this section, we briefly introduce the common approach for automatically deriving random data generators for ADTs in QuickCheck. To exemplify this, and for illustrative purposes, let us consider the following ADT definition to encode simple Html pages:

```
data Html = Text   String
          | Single String
          | Tag    String Html
          | Join   Html   Html
```

[1] DRAGEN2 is available at http://github.com/OctopiChalmers/dragen2

The type Html allows to build pages via four possible constructions: Text—which represents plain text values—, Single and Tag—which represent singular and paired HTML tags, respectively—, and Join—which concatenates two HTML pages one after another. In Haskell, Text, Single, Tag, and Join are known as *data constructors* (or constructors for short) and are used to distinguish which variant of the ADT we are constructing. Each data constructor is defined as a product of zero or more types known as *fields*. For instance, Text has a field of type String, whereas Join has two recursive fields of type Html. In general, we will say that a data constructor with no recursive fields is *terminal*, and *non-terminal* or *recursive* if it has at least one field of such nature. With this representation, the example page `<html>hello<hr>bye</html>` can be encoded as:

```
Tag "html" (Join (Join
  (Text "hello") (Single "hr")) (Text "bye"))
```

A. Type-driven generation of random values

In order to generate random ADTs values, most approaches require users to provide a random data generator for each ADT definition. This is a cumbersome and error prone task that usually *follows closely the structure of the ADTs*. For instance, consider the following definition of a *QuickCheck* random generator for the type Html:

```
gen_Html = sized (λsize →
  if size == 0
  then frequency
  [ (2, Text   ⟨$⟩ gen_String)
  , (1, Single ⟨$⟩ gen_String)]
  else frequency
  [ (2, Text   ⟨$⟩ gen_String)
  , (1, Single ⟨$⟩ gen_String)
  , (4, Tag    ⟨$⟩ gen_String ⟨*⟩ smaller gen_Html)
  , (3, Join   ⟨$⟩ smaller gen_Html ⟨*⟩ smaller gen_Html)])
```

We use the Haskell syntax [] and (,) for denoting lists and pairs of elements, respectively (e.g., $[(1,2),(3,4)]$ is a list of pairs of numbers.) The random generator gen_{Html} is defined using *QuickCheck*'s function sized to parameterize the generation process up to an external natural number known as the *generation size*—captured in the code with variable size. This parameter is chosen by the user, and it is used to limit the maximum amount of recursive calls that this random generator can perform and thus ensuring the termination of the generation process. When called with a positive generation size, this generator can pick to generate among any Html data constructor with an explicitly *generation frequency* that can be chosen by the user—in this example, 2, 1, 4 and 3 for Text, Single, Tag, and Join, respectively. When it picks to generate a Text or a Single data constructor, it also generates a random String value using the standard *QuickCheck* generator $\text{gen}_{\text{String}}$.[2] On the other hand, when it picks to generate a Join constructor, it also generates two independent random sub-expressions recursively, decreasing the generation size by a unit on each recursive invocation (smaller gen_{Html}). The case of random generation of Tag constructors follows analogously. This random process keeps calling itself recursively until the generation size reaches zero, where the generator is constrained to pick among terminal data constructors, being Text and Single the only possible choices in our particular case.

The previous definition is rather mechanical, except perhaps for the chosen generation frequencies. DRAGEN [9] is a tool conceived to mitigate the problem of finding the appropriated generation frequencies. It uses the theory of branching processes [14] to model and predict analytically the expected number of generated data constructors. This prediction mechanism is used to feedback a simulation-based optimization process that adjusts the generation frequency of each data constructor in order to obtain a particular distribution of values that can be specified by the user—thus providing a flexible testing environment while still being mostly automated.

As many other tools for automatic derivation of generators (e.g., [5]–[7], [13]), DRAGEN synthesizes random generators similar to the one shown before, where the generation process is limited to pick *a single data constructor at the time and then recursively generate each required sub-expression independently*. In practice, this procedure is often too generic to generate random data with enough structural complexity required for testing certain applications.

III. Sources of structural information

In this section, we describe the motivation for considering two additional sources of structural information which lead us to obtain better random data generators. We proceed to exemplify the need to consider such sources with examples.

A. Branching on input data

To exemplify the first source of structural information, consider that we want to use randomly generated Html values to test a function simplify :: Html → Html. In Haskell, the notation f::T means that program f has type T. In our example, function simplify takes an Html as an input and produces an Html value as an output—thus its type Html → Html. Intuitively, the purpose of this function is to assemble sequences of Text constructors into a single big one. More specifically, the code of simplify is as follows:

```
simplify :: Html → Html
simplify (Join (Text t1) (Text t2))
   = Text (concat t1 t2)
simplify (Join (Join (Text t1) x) y)
   = simplify (Join (Text t1) (simplify (Join x y)))
simplify (Join x y) = Join (simplify x) (simplify y)
simplify (Tag t x) = Tag t (simplify x)
simplify x = x
```

Function concat just concatenates two strings. The body of simplify is described using *pattern matching* over possible kinds of Html values. Pattern matching allows to define

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

functions idiomatically by defining different function clauses for each input pattern we are interested in. In other words, pattern matching is a mechanism that functions have to branch on input arguments. In the code above, we can see that simplify patterns match against sequences of Text constructors combined by a Join constructor—see first and second clauses. Generally speaking, patterns can be defined to match specific constructors, literal values or variable sub-expressions (like x in the last clause of simplify). Patterns can also be nested in order to match very specific values.

Ideally, we would like to put approximately the same amount of effort into testing each clause of the function simplify. However, each data constructor is generated independently by those generators automatically derived by just considering ADT definitions. Observe that the probability of generating a value satisfying a nested pattern (like Join (Text t_1) (Text t_2)) decreases multiplicatively with the number of constructors we simultaneously pattern against. As an evidence of that, in our tests, we found at the first two clauses of simplify get exercised only approximately between 1.5% and 6% of the time when using the state-of-the-art tools for automatically deriving *QuickCheck* generators *MegaDeTH* [13] and DRAGEN [9]. Most of the generated values were exercising the simplest clauses of our function, i.e, simplify (Join x y), simplify (Tag t x), and simplify x.

Although the previous example might seem rather simple, branching against specific patterns of the input data is not an uncommon task. In that light, and in order to obtain interesting test cases, it is desirable to conceive generators able to produce random values capable of exercising patterns with certain frequency—Section IV shows how to do so.

B. Abstract interfaces

A common choice when implementing ADTs is to transfer the responsibility of preserving structural invariants to the interfaces that manipulate values of such types. To illustrate this point, let us consider three new primitives responsible to handle Html data as shown in Fig 1. These functions encode additional information about the structure of Html values in the form of specific HTML tags. Primitive hr represents the tag `<hr>` used to separate content in an HTML page. Function div and bold place an Html value within the tags `div` and `b` in order to introduce divisions and activate bold fonts, respectively. For instance, the page `<html><b>hello</b><hr>bye</html>` can be encoded as:

```
hr :: Html
hr = Single "hr"
div :: Html → Html
div x = Tag "div" x
bold :: Html → Html
bold x = Tag "b" x
```

Fig. 1: Abstract interface of the type Html.

```
Tag "html" (Join (Join
  (bold (Text "hello")) hr) (Text "bye"))
```

Observe that, instead of including a new data constructor for each possible HTML tag in the Html definition (recall Section II), we defined a minimal general representation with a set of high-level primitives to build valid Html tags. This programming pattern is often found in a variety of Haskell libraries. As a consequence of this practice, generators derived by only looking into ADT definitions often fail to synthesize useful random values, e.g., random HTML pages with valid tags. After all, most of the *valid structure* of values has been encoded into the primitives of the ADT abstract interface. When considering the generator described in Section II, the chances of generating a Tag value representing a commonly used HTML tag such as div or b are extremely low.

So far, we have introduced two scenarios where derivation approaches based only on ADT definitions are unable to capture all the available structural information from the user codebase. Fortunately, this information can be automatically exploited and used to generate interesting and more structured random values. The next section introduces a model capable of encoding structural information presented in this section into our automatically derived random generators in a modular and flexible way.

IV. Capturing ADTs Structure

In this section, we show how to augment the automatic process of deriving random data generators with the structural information expressed by pattern matchings and abstract interfaces. The key idea of this work is to represent the different sources in an homogeneous way.

Figure 2 shows the workflow of our approach for the Html ADT. Based on the codebase, the user of DRAGEN2 specifies: (i) the ADT definition to consider (noted as Html_{ADT}), (ii) its patterns of interest (noted $\text{Html}_{\text{Patterns}}$), and (iii) the primitives from abstract interfaces to involve in the generation process (noted as $\text{Html}_{\text{Interface}}$). Our tool then *automatically derives generators* for each source of structural information. These generators produce random *partial ADT values* in a way that it is easier to combine them in order to create structurally richer ones. For instance, the generator obtained from Html_{ADT} only generates constructors of the ADT but leaves the generation at the recursive fields incomplete, e.g., it generates values of the form (Text "xA2sx"), (Single "xj32da"), (Tag "divx234jx" •) and (Join • •), where • is a placeholder denoting a "yet-to-complete" value. Similarly, the generator obtained from $\text{Html}_{\text{Patterns}}$ generates values satisfying the expected patterns where recursive fields are also left uncompleted, e.g., it generates values of the form (Join (Text "xxa34") (Text "yxa123")) and (Join (Join (Text "xd32sa") •) •). Finally, the generator derived from $\text{Html}_{\text{Interface}}$ generates calls to the interface's primitives, where each argument of type Html is left uncompleted, e.g., (div •) and (bold •).

Observe that *partial ADT values* can be combined easily and the result is still a well-formed value of type Html. For instance, if we want to combine the following random generated ADT value (Text "xx34s"), pattern

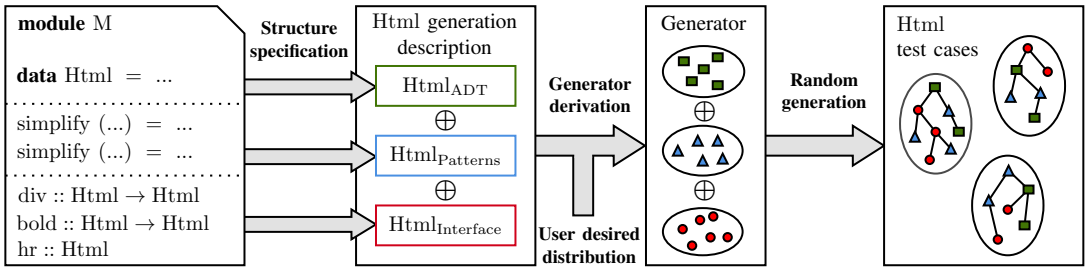

Fig. 2: Deriving a generator for the ADT Html with the structural information found in module M.

(Join (Join (Text "xd32sa") •), and interface call (div •), we can obtain the following well-typed Html value:

```
Join (Join (Text "xd32sa") (div (Text "xx34s")))
```

Finally, our tool puts all these three generators together into one that combines partial ADT values into fully formed ones. Importantly, the user can specify the desire distribution of the expected number of constructors, pattern matching values, and interface calls that the generator will produce. All in all, our approach offers the following advantages over usual derivation of random generators based only on ADT definitions:

► **Composability:** our tool can combine different partial ADT values arising from different structural information sources depending on what property or sub-system becomes necessary to test using randomly generated values.

► **Extensibility:** the developer can specify new sources of structural information and combine them with the existing ones simply by adding them to the existing specification of the target ADT.

► **Predictability:** the tool is capable of synthesizing generators with adjustable distributions based on developers' demands. For instance, a uniform distribution of pattern matching values, or a distribution where some constructors are generated twice as often as others. We explain the prediction of distributions in the next section.

We remark that, for space reasons, we were only able to introduce the specification of a rather simple target ADT like Html. In practice, this reasoning can be extended to mutually recursive and parametric ADT definitions as well.

V. Predicting Distributions

Characterizing the distribution of values of an arbitrary random generator is a hard task. It requires modeling every random choice that a generator could possibly make to generate a value. In a recent work [9], we have shown that it is possible to *analytically* predict the average distribution of data constructors produced by random generators automatically derived considering only ADT definitions—like the one presented on Section II. For this purpose, we found that random generation of ADT values can be characterized using the theory of *branching processes* [14]. This probabilistic theory was originally conceived to predict the growth and extinction of royal family trees the Victorian Era, later being applied to a wide variety of research areas. In this work, we adapt this model to predict the average distribution of values of random generators derived considering structural information coming from functions' pattern matchings and abstract interfaces.

Essentially, a branching process is a special kind of Markov process that models the evolution of a population of *individuals of different kinds* across discrete time steps known as *generations*. Each kind of individual is expected to produce an average number of offspring of (possibly) different kinds from one generation to the next one. Mista el at. [9] show that branching processes can be adapted to predict the generation of ADT values by simply considering each data constructor as a kind of its own. In fact, any ADT value can be seen as a tree where each node represents a root data constructor and has its sub-expressions as sub-trees—hence note the similarity with family trees. In this light, each tree level of a random value can be seen as a generation of individuals in this model.

We characterize the numbers of constructors that a random generator produces in the n-th generation as a vector G_n, a vector that groups the number of constructors of each kind produced in that generation—in our Html example, this vector has four components, i.e., one for each constructor. From branching processes theory, the following equation captures the expected distribution of constructors at the generation n, noted $E[G_n]$, as follows:

$$E[G_n]^T = E[G_0]^T \cdot M^n \tag{1}$$

Vector $E[G_0]$ represents the initial distribution of constructors that our generator produces, which simply consists of the generation probability of each one. The interesting aspect of the prediction mechanism is encoded in the matrix M, known as a the *mean matrix* of this stochastic process. M is a squared matrix with as rows and columns as different data constructors involved in the generation process. Each element $M_{i,j}$ of this matrix encodes the average number of data constructors of kind j that gets generated in a given generation, provided that we generated a constructor of kind i at the previous one. In this sense, this matrix encodes the "branching" behavior of our random generation from one generation to the next one. Each element of the matrix can be automatically calculated by exploiting ADT definitions, as well as the individual probability of generating each constructor. For instance, the average number of Text data constructors that we will generate provided that we generated a Join constructor on the previous level results:

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

where 2 is the number of holes present when generating a partial ADT value Join (i.e., Join • •) and p_{Text} is the

probability of individually generating the constructor Text. This reasoning can be used to build the rest of the mean matrix analogously.

A. Extending predictions for structural information

In this work, we show how to naturally fit structural information beyond ADT definitions into the prediction mechanism of branching processes. Our realization is that it suffices to consider *each different pattern matching and function call as a kind of individual on its own*. In that manner, we can extend our mean matrix M adding a row and a column for each different pattern matching and function call as shown in Figure 3. Symbol $C_1 \cdots C_i$ denotes constructors, $P_1 \cdots P_j$ pattern matchings, and $F_1 \cdots F_k$ function calls. The light-red colored matrix is what we had before, whereas the light-blue colored cells are new—we encourage readers to obtain a colored copy of this work.

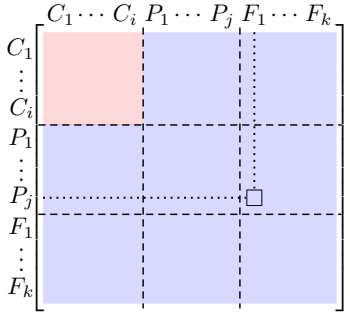

Fig. 3: Mean matrix M including pattern matching and function calls information.

The new cells are filled as before: we need to consider the amount of holes when generating partial pattern matching values and function calls as well as their individual probabilities. For instance, if we consider P_j as the second pattern of function simplify and F_1 as function div, then the marked cell above has the value $2 \cdot p_{div}$, i.e., the amount of holes in the partially generated pattern (Join (Join (Text s) •) •), where s is some random string, times the probability to generate a call to function div. The rest of this matrix can be computed analogously.

As another contribution, we found that the whole prediction process can be factored in terms of two vectors β and $\mathcal{P}$, such that β represents the number of holes in each partial ADT value that we generate, whereas $\mathcal{P}$ simply represents the probability of generating that partial ADT value. Then, the equation (1) can be rewritten as:

$$E[G_n]^T = \beta^T \cdot (\beta \cdot \mathcal{P}^T)^n$$

For instance, β and $\mathcal{P}$ for our generation specification of HTML values are as shown in Figure 4. We note simplify#1 and simplify#2 to the patterns occurring in the first and second clauses of simplify, respectively.

Note that by varying the shape of the vector $\mathcal{P}$ we can tune the distribution of our random generator in a way that can be always characterized and predicted. DRAGEN2 follows a similar approach as DRAGEN and uses an heuristic to tune the generation probabilities of each source of structural information. This is done by running a simulation-based optimization process at compile-time. This process is parameterized by the desired distribution of values set by the user. In this manner, developers can specify, for instance, a uniform distribution of data constructors, pattern matching values and function calls or, alternatively, a distribution of values with some constructions appearing in a different proportion as others, e.g., two times more functions calls to div than Join constructors.

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

Fig. 4: Prediction vectors of our Html generation specification.

B. Overall prediction

It is possible to provide an overall prediction of the expected number of constructors when restricting the generation process to only bare data constructors and pattern matching values. To achieve that, we should stop considering pattern matching values as atomic constructions and start seeing them as compositions of several data constructors. In that manner, it is possible to obtain the expected *total* number of generated data constructors that our generators will produce—regardless if they are generated on their own, or as part of a pattern matching value. We note this number as $E^{\downarrow}[\_]$ and, to calculate it, we only need to add the expected number of bare constructors that are included within each pattern matching. For instance, we can calculate the total expected number of constructors Text and Join that we will generate by simply expanding the expected number of generated pattern matching values simplify#1 and simplify#2 into their corresponding data constructors:

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

Observe that each time we generate a value satisfying the first pattern matching of the function simplify, we add two Text and one Join data constructors to our random value. The case of the second pattern matching of simplify follows analogously. Note that the overall prediction cannot be applied if we also generate random values containing function calls, as we cannot predict the output of an arbitrary function.

VI. Case Studies

This section describes two case studies showing that considering additional structural information when deriving generators can consistently produce better testing results in terms of code coverage. Instead of restricting our scope to Haskell, in this work we follow a broader evaluation approach taken previously to compare state-of-the-art techniques to derive random data generators based on ADT definitions [8], [9].

We evaluate how including additional structural information when generating a set of random test cases (often referred as a *corpus*) affects the code coverage obtained when testing a given target program. For that, we considered two external programs which expect highly structured inputs, namely *GNU CLISP* [3]—the GNU Common Lisp compiler, and *HTML Tidy* [4]—a well

[3] https://www.gnu.org/software/gcl/ [4] http://www.html-tidy.org

known HTML refactoring and correction utility. We remark that these applications are not written in Haskell. However, there exist Haskell libraries defining ADTs encoding their input structure, i.e., Lisp and HTML values respectively. These libraries are: *hs-zuramaru*[5], implementing an embedded Lisp interpreter for a small subset of this programming language, and *html*[6], defining a combinator library for constructing HTML values. These libraries also come with serialization functions to map Haskell values into corresponding test case files.

We firstly compiled instrumented versions of the target programs in a way that they also return the execution path followed in the source code every time we run them with a given input test case. This let us distinguish the amount of different execution paths that a randomly generated corpus can trigger. We then used the ADTs defined on the chosen libraries to derive random generators using DRAGEN and DRAGEN2, including structural information extracted from the library's codebase in the case of the latter. Then, we proceeded to evaluate the code coverage triggered by independent, randomly generated corpora of different sizes varying from 100 to 1000 test cases each. In order to remove any external bias, we derived generators optimized to follow *a uniform distribution of constructors (and pattern matchings or function calls in the case* DRAGEN2*), and carefully adjusted their generation sizes to match the average test case size in bytes*. This way, any noticeable difference in the code coverage can be attributed to the presence (or lack thereof) structural information when generating the test cases. Additionally, to achieve statistical significance we repeated each experiment 30 times with independently generated sets of random test cases.

Figure 5 illustrates the mean number of different execution paths triggered for different combinations of corpus size and derivation tool, including error bars indicating the standard error of the mean on each case. We proceed to describe each case study and our findings in detail as follows.

A. Branching on input data

In this first case study we wanted to evaluate the observed code coverage differences when considering structural information present on functions pattern matchings.

Our chosen library encodes Lisp S-expressions essentially as lists of symbols, represented as plain strings; and literal values like booleans or integers. In order to interpret Lisp programs, this unified representation of data and code requires this library to pattern match against common patterns like let-bindings, if-then-else expressions and arithmetic operators among others. In particular, each one of these patterns match a against special symbol of the Lisp syntax like `"let"`, `"if"` or `"+"`; and their corresponding sub-expressions. We extracted this structural information and included it into the generation specification of our random Lisp values—which were generated by randomly picking from a total of 6 data constructors and 8 different pattern matchings. By doing this, we obtained a code coverage improvement of approximately 4% using DRAGEN2 with respect to the one obtained with DRAGEN (see Figure 5 (a)). While it seems an small improvement, we argue that an improvement of 4% is not negligible considering (a) the little effort that took us to specify the pattern matchings and (b) that we are testing a full-fledged compiler.

[5] http://hackage.haskell.org/package/zuramaru
[6] http://hackage.haskell.org/package/html

B. Abstract interfaces

For our second case study, we wanted to evaluate how including structural information coming from abstract interfaces when generating random HTML values might improve the testing performance.

The library we used for this purpose represents HTML values very much in the same way as we exemplify in Section II, i.e., defining a small set of general constructions representing plain text and tags—although this library also supports HTML tag attributes as well. Then, this representation is extended with a large abstract interface consisting of combinators representing common HTML tags and tag attributes—equivalent to the combinators div, bold and hr illustrated in Section III.

In this case study we included the structural information present on the abstract interface of this library into the generation specification of random HTML values, resulting in a generation process that randomly picked among 4 data constructors and 163 abstract functions. With this large amount of additional structural information, we observed an increase of up to 83% in the code coverage obtained with DRAGEN2 with respect to the one observed with DRAGEN (see Figure 5 (b)). A manual inspection of the corpora generated with each tool revealed us that, in general terms, the test cases generated with DRAGEN rarely represent syntactically correct HTML values, consisting to a large extent of random strings within and between HTML tag delimiters (`"<"`, `">"` and `"/>"`). On the other hand, test cases generated with DRAGEN2 encode much more interesting structural information, being mostly syntactically correct. We found that, in many cases, the test cases generated with DRAGEN2 were parsed, analyzed and reported as valid HTML values by the target application.

With these results we are confident that including the structural information present on the user codebase improves the overall testing performance.

VII. Related Work

Boltzmann models [15] are a general approach to randomly generating combinatorial structures such as trees and graphs, closed simply-typed lambda terms, etc. A random generator built around such models uniformly generates values of a target size with a certain size tolerance. However, it has been argued that this approach has theoretical and practical limitations in the context of software testing [16]. In a recent work, Bendkowski et al. provides a framework called *boltzmann-brain* to specify and synthesize standalone Haskell random generators based on Boltzmann models [17]. This framework mixes parameter tuning and rejection of samples of unwanted sizes to approximate the desired distribution of values according to user demands. The overall discard ratio then depends on how constrained the desired sizes of values are. On the other

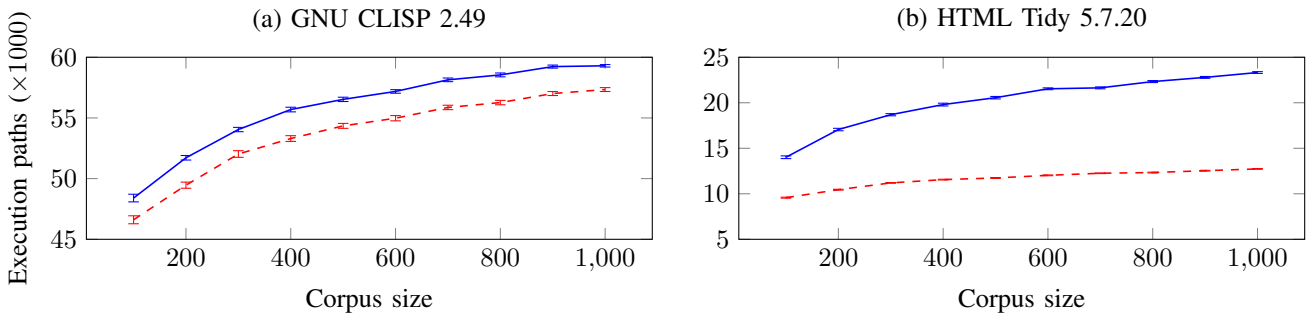

Fig. 5: Path coverage comparison between DRAGEN (- - -) and DRAGEN2 (——).

hand, our work is focused on approximating the desired distribution as much as possible via parameter optimization, without discarding any generated value at runtime. Although promising, we found difficulties to compare both approaches in practice due that *boltzmann-brain* is considered a conceptual standalone utility that produces self-contained samplers. In this light, data specifications have to be manually written using a special syntax, and cannot include Haskell ground types like String or Int, difficulting the integration of this tool to existing Haskell codebases like the ones we consider in this work.

From the practical point of view, Feldt and Poulding propose *GödelTest* [16], a search-based framework for generating biased data. Similar to our approach, *GödelTest* works by optimizing the parameters governing the desired biases on the generated data. However, the optimization mechanism uses meta-heuristic search to find the best parameters at runtime. DRAGEN2 on the other hand implements an analytic and composable prediction mechanism that is only used at compile time to optimize the generation parameters, thus avoiding performing any kind of runtime reinforcement.

Directed Automated Random Testing (DART) is a technique that combines random testing with symbolic execution for C programs [18]. It requires instrumenting the target programs in order to introduce testing assertions and obtain feedback from previous testing executions, which is used to explore new paths in the source code. This technique has been shown to be remarkably useful, although it forces a strong coupling between the testing suite and the target code. Our tool intends to provide better random generation of values following an undirected fashion, without having to instrument the target code, but still extracting useful structural information from it.

VIII. Final Remarks

We extended the standard approach for automatically deriving random generators in Haskell. Our generators are capable of producing complex and interesting random values by exploiting static structural information found in the user codebase. Based on the theory of branching processes, we adapt our previous prediction mechanism to characterize the distribution of random values representing the different sources of structural information that our generators might produce. This predictions let us optimize the generation parameters in compile time, resulting in an improved testing performance according to our experiments.

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

References

[1] J. Hughes, U. Norell, N. Smallbone, and T. Arts, "Find more bugs with QuickCheck!" in *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2016.
[2] J. Hughes, C. P. B, T. Arts, and U. Norell, "Mysteries of DropBox: Property-based testing of a distributed synchronization service," in *Proc. of the Int. Conf. on Software Testing, Verification and Validation*, 2016.
[3] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing AUTOSAR software with QuickCheck," in *In Proc. of IEEE International Conference on Software Testing, Verification and Validation, ICST Workshops*, 2015.
[4] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000.
[5] N. Mitchell, "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, 2007, pp. 55–62.
[6] C. Runciman, M. Naylor, and F. Lindblad, "Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values," in *Proc. of the ACM SIGPLAN Symposium on Haskell*, 2008.
[7] J. Duregård, P. Jansson, and M. Wang, "Feat: Functional enumeration of algebraic types," in *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2012.
[8] G. Grieco, M. Ceresa, A. Mista, and P. Buiras, "QuickFuzz testing for fun and profit," *Journal of Systems and Software*, vol. 134, 2017.
[9] A. Mista, A. Russo, and J. Hughes, "Branching processes for quickcheck generators," in *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*, 2018.
[10] M. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing and optimising compiler by generating random lambda terms," in *The IEEE/ACM International Workshop on Automation of Software Test (AST)*, 2011.
[11] J. Midtgaard, M. N. Justesen, P. Kasting, F. Nielson, and H. R. Nielson, "Effect-driven QuickChecking of compilers," *In Proceedings of the ACM on Programming Languages, Volume 1*, no. ICFP, 2017.
[12] C. Klein and R. B. Findler, "Randomized testing in PLT Redex," in *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2009.
[13] G. Grieco, M. Ceresa, and P. Buiras, "QuickFuzz: An automatic random fuzzer for common file formats," in *Proc. of the ACM SIGPLAN International Symposium on Haskell*, 2016.
[14] H. W. Watson and F. Galton, "On the probability of the extinction of families," *The Journal of the Anthropological Institute of Great Britain and Ireland*, 1875.
[15] P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer, "Boltzmann samplers for the random generation of combinatorial structures," *Combinatorics, Probability and Computing.*, vol. 13, 2004.
[16] R. Feldt and S. Poulding, "Finding test data with specific properties via metaheuristic search," in *Proc. of International Symp. on Software Reliability Engineering (ISSRE)*. IEEE, 2013.
[17] M. Bendkowski, O. Bodini, and S. Dovgal, "Polynomial tuning of multiparametric combinatorial samplers," in *Proc. of the Fifteenth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, 2018.
[18] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, 2005.