Synthesizing Random Generators via Higher-Order Representations

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

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

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

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

I. Introduction

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

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

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

The main contribuitions of this paper are:

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

II. Type-Driven Random Generation

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

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

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

In the previous definition, we declare Exp as the sum of three possible classes of values: Val represents literal values, whereas Add and Mul represent the addition and multiplication of two integer expressions, respectively. In Haskell, Val, Add and Mul are called *data constructors* and are used to distiguish which variant of the data type we are reffering to. Each data constructor is then defined as a product of zero or more types known as *fields*. In particular, *Val* contains a field of type *Int*, while *Add* and *Mul* contain two fields of type *Exp* representing the operands of each operation. Note that *Exp* is used as a field of at least one of its data constructors (case *Add* and *Mul*), making it a recursively defined ADT. In general, we will say that a data constructor with no recursive fields is *terminal*, and *non-terminal* or *recursive* if it has at least one field of such nature. With this reprentation, the expression "2 + (5 * 6)" can be encoded simply by using *Exp* data constructors as $(Add\ (Val\ 2)\ (Mul\ (Val\ 5)\ (Val\ 6)))$. Furthermore, an evaluation function from *Exp*s to integer values can be defined very idiomatically:

```
eval :: Exp → Int
eval (Val n)     = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x ∗ eval y
```

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

A. Type-Driven Generation of Random Values

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

```
genExp :: Gen Exp
genExp = sized gen
  where
    gen 0 = Val ⟨$⟩ arbitrary
    gen n = oneof
      [ Val ⟨$⟩ arbitrary
      , Add ⟨$⟩ gen (n−1) ⟨∗⟩ gen (n−1)
      , Mul ⟨$⟩ gen (n−1) ⟨∗⟩ gen (n−1)]
```

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

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

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

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

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

In this work we identify two sources of additional structure information which are not considered by the aforementioned automatic derivation tools to obtain random data generators:

1) The target code behaves differently over inputs matching very specific patterns.
2) The target code yields part of the responsability of preserving its invariants to the functions of its abstract interface.

This information can be reified in compile time and used to synthesize richer random generators automatically. We proceed to exemplify the previous points in detail.

B. *Presence of Complex Pattern Matchings*

To exemplify the firts problematic scenarion that may arise, suppose we want to use randomly generated *Exp*s to test a property comprising the following function:

$$\begin{array}{l} \mathit{foo} :: \mathit{Exp} \to \mathit{Exp} \\ \mathit{foo}\ (\mathit{Add}\ (\mathit{Add}\ x\ (\mathit{Val}\ 50))\ (\mathit{Add}\ (\mathit{Val}\ 25)\ y)) \\ \quad = \mathit{error}\ \texttt{"pattern \#1"} \\ \mathit{foo}\ (\mathit{Mul}\ (\mathit{Val}\ 50)\ (\mathit{Mul}\ (\mathit{Val}\ x)\ y)) \\ \quad = \mathit{error}\ \texttt{"pattern \#2"} \\ \mathit{foo}\ x = x \end{array}$$

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

QuickCheck's default implementation for random *Int*s pick a number in the interval $[-n, n]$ with uniform distribution. Hence, in principle we need to recognize what is a suitable generation size, which should be big enough for our generator be able to generate the numbers we pattern match against to, otherwise we will not be able to generate any value to match against *foo* clauses, leaving fragments of code completely untested. Suppose then that we pick the a generation size 50, i.e. the minimum size that is big enough to produce an *Int* number equal to 50. Under this consideration, the probability of generating a value matching the first clause of *foo* (and hence triggering the first error) results as follows:

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

If we use *MegaDeTH* to automatically derive a random generator for *Exp*, we obtain a uniform generation probability distribution over constructors, i.e., $P(\mathit{Val}) = P(\mathit{Add}) = P(\mathit{Mul}) = 1/3$. In this setting, $P(match(foo\#1))$ results $1/2430000$, meaning that, in average, we will need to generate over than two million test cases in order to be able to test the first clause of *foo* only once. This situation can be somewhat improved if we decide to use **DRAGEN** to obtain a random generator. Using this tool, we can optimize the generation probabilities in order to benefit the generation of some data constructors over the rest. Considering that the first pattern matching of *foo* involves the data constructor Add as the only recursive one, we can set an target generation proportion of Add data constructors of, for instance, $20 : 1$ with respect to the rest of the generated data constructors. By doing so, the obtained distribution of values results such that $P(match(foo\#1)) \approx 1/300000$. Although this certainly improves the probability of generating a matching value, this probability is not substantial enough to become practical.

Additionally, by favoring the generation probabilities towards the Add constructor, we found that the probability of generating a value matching the second clause of *foo* (which does not matches against it) also gets diminished into an impractival value.

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

C. *Data Invariants Encoded on Abstract Interfaces*

A common choice when implementing a data structure is to transfer the responsability of preserving its invariants to the functions that manipulates its values. Consider for instance the following possible implementation of a basic $HTML$ manipulation library:

$$\begin{array}{l} \mathbf{data}\ \mathit{Html} = \mathit{Html}\ \mathit{String} \\ \mathit{head} :: \mathit{Html} \to \mathit{Html} \\ \mathit{head}\ (\mathit{Html}\ \mathit{inner}) \\ \quad = \mathit{Html}\ (\texttt{"<head>"} +\!\!+\ \mathit{inner} +\!\!+ \texttt{"</head>"}) \\ \mathit{body} :: \mathit{Html} \to \mathit{Html} \\ \mathit{body}\ (\mathit{Html}\ \mathit{inner}) \\ \quad = \mathit{Html}\ (\texttt{"<body>"} +\!\!+\ t +\!\!+ \texttt{"</body>"}) \\ \mathit{br} :: \mathit{Html} \\ \mathit{br} = \mathit{Html}\ \texttt{"<br />"} \\ \mathit{concat} :: \mathit{Html} \to \mathit{Html} \to \mathit{Html} \\ \mathit{concat}\ (\mathit{Html}\ x)\ (\mathit{Html}\ y) = \mathit{Html}\ (x +\!\!+ y) \end{array}$$

In the previous definition, the *Html* data type is defined using a single data constructor that contains the textual representation of the Html code it represents in plain text. Later, the combinator functions defined over $HTML$ are the ones in charge of transforming this plain text representation with the invariant that, given valid *Html* parameters, they always return a valid *Html* value. This is a very commmon programming pattern that can be found in a variety of Haskell libraries **CITATION NEEDED**.

If we use a standard type-driven approach to derive a random generator for *Html*, the only "structural" information that we can use in the derivation process is that *Html* values are composed of strings. As a consequence, we essentially end up generating random strings, which rarely represents a valid (or at least well structured) Html value. Additionally, if our test suite contains properties constrained by certain preconditions, this lack of domain knowleadge may lead in an impractically high discard ratio of randomly generated test. For instance, suposse we write a property to verify that *head* preserves the invariant previously mentioned:

$$\begin{aligned}
&\mathit{prop\_head\_inv} :: \mathit{Html} \rightarrow \mathit{Bool} \\
&\mathit{prop\_head\_inv}\ x = \mathit{valid}\ x \Longrightarrow \mathit{valid}\ (\mathit{head}\ x)
\end{aligned}$$

In the previous definition we state that *head* always returns a valid *Html* value ($\mathit{valid}\ (\mathit{head}\ x)$), provided that we are given a valid *Html* as input ($\mathit{valid}\ x$). Then, while testing this property we rarely satisfy its precondition, in which case *QuickCheck* discards the whole test, retrying with a diferent random input, degradating this way the testing performance.

We believe that, if certain patterns of values are relevant enough to appear in the codebase being tested, a practical random testing methodology should be able to produce values satisfying such patterns in a substantial proportion. The next section introduces a reprentation model that let us encode the structure information presented here into our automatically derived random generators in a modular and flexible way.

III. Extracting Structure

In this work, we ...

$$\begin{aligned}
&\mathbf{data}\ \mathrm{HRep}_{Val}\ \ a = \mathrm{Mk}_{Val}\ \mathit{Int} \\
&\mathbf{data}\ \mathrm{HRep}_{Add}\ a = \mathrm{Mk}_{Add}\ a\ a \\
&\mathbf{data}\ \mathrm{HRep}_{Mul}\ a = \mathrm{Mk}_{Mul}\ a\ a
\end{aligned}$$

For this purpose, we employ Haskell's type classes mechanism **CITATION NEEDED**. We define an evaluation type class ($\Downarrow$) which specifies how to perform a single transformation *step* from the higher-order representation back into a concrete value of the target:

$$\begin{aligned}
&\mathbf{class}\ (\mathit{rep}\ \Downarrow\ \mathit{target})\ \mathbf{where} \\
&\quad \mathit{step} :: \mathit{rep}\ \mathit{target} \rightarrow \mathit{target}
\end{aligned}$$

Then, we can define the following overloaded instances of the *step* operation for the canonical representations of data constructors simply by mapping each lifted constructor back into its corresponding one.

$$\begin{aligned}
&\mathbf{instance}\ (\mathrm{HRep}_{Val}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{Val}\ n) = \mathit{Val}\ n \\
&\mathbf{instance}\ (\mathrm{HRep}_{Add}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{Add}\ x\ y) = \mathit{Add}\ x\ y \\
&\mathbf{instance}\ (\mathrm{HRep}_{Mul}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{Mul}\ x\ y) = \mathit{Mul}\ x\ y
\end{aligned}$$

With these individual representations, we can define a type combinator ($\oplus$) to compose two representations into a single one:

$$\mathbf{data}\ (f\ \oplus\ g)\ a = L\ (f\ a)\ |\ R\ (g\ a)$$

Then, a composite representation can be transformed back into the concrete target type by pattern matching on the data type variant and applying the *step* tranformation to the inner representation:

$$\begin{aligned}
&\mathbf{instance}\ (f\ \oplus\ g\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (L\ f) = \mathit{step}\ f \\
&\quad \mathit{step}\ (R\ g) = \mathit{step}\ g
\end{aligned}$$

Furthermore, we can define a type combinator ($\otimes$) to tag every representation with an explicit generation frequency:

$$\mathbf{data}\ (f\ \otimes\ n)\ a = \mathit{Freq}\ (f\ a)$$

This combinator is evaluated back to our target data type simply by piping the result from the inner representation. It does not change the evaluation semantics, as it is only considered at generation time:

$$\begin{aligned}
&\mathbf{instance}\ (f\ \otimes\ n\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathit{Freq}\ f) = \mathit{step}\ f
\end{aligned}$$

With the introduced combinators, we can easily create a type synonym HRep_{Exp} to refer to the canonical representation of our original data type *Exp*, tagging for instance the representation of the constuctor *Add* to be generated in double the proportion of rest of the representations:

$$\begin{aligned}
\mathbf{type}\ \mathrm{HRep}_{Exp} = {}&\ \mathrm{HRep}_{Val} \\
&\oplus\ \mathrm{HRep}_{Add}\ \otimes\ 2 \\
&\oplus\ \mathrm{HRep}_{Mul}
\end{aligned}$$

A. Representing Pattern Matchings

$$\begin{aligned}
&\mathbf{data}\ \mathrm{HRep}_{foo\#1}\ a = \mathrm{Mk}_{foo\#1}\ a\ a \\
&\mathbf{data}\ \mathrm{HRep}_{foo\#2}\ a = \mathrm{Mk}_{foo\#2}\ \mathit{Int}\ a
\end{aligned}$$

$$\begin{aligned}
&\mathbf{instance}\ (\mathrm{HRep}_{foo\#1}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{foo\#1}\ x\ y) \\
&\qquad = \mathit{Add}\ (\mathit{Add}\ x\ (\mathit{Val}\ 50))\ (\mathit{Add}\ (\mathit{Val}\ 25)\ y) \\
&\mathbf{instance}\ (\mathrm{HRep}_{foo\#2}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{foo\#2}\ x\ y) \\
&\qquad = \mathit{Mul}\ (\mathit{Val}\ 50)\ (\mathit{Mul}\ (\mathit{Val}\ x)\ y)
\end{aligned}$$

$$\begin{aligned}
\mathbf{type}\ \mathrm{HRep}_{foo} = {}&\ \mathrm{HRep}_{foo\#1} \\
&\oplus\ \mathrm{HRep}_{foo\#2}
\end{aligned}$$

B. Representing Abstract Interfaces

$$\begin{aligned}
&\mathbf{data}\ \mathrm{HRep}_{ten}\ \ \ \ \ a = \mathrm{Mk}_{ten} \\
&\mathbf{data}\ \mathrm{HRep}_{square}\ a = \mathrm{Mk}_{square}\ a \\
&\mathbf{data}\ \mathrm{HRep}_{minus}\ a = \mathrm{Mk}_{minus}\ a\ a
\end{aligned}$$

$$\begin{aligned}
&\mathbf{instance}\ (\mathrm{HRep}_{ten}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ \mathrm{Mk}_{ten} = \mathit{ten} \\
&\mathbf{instance}\ (\mathrm{HRep}_{square}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{square}\ x) = \mathit{square}\ x \\
&\mathbf{instance}\ (\mathrm{HRep}_{minus}\ \Downarrow\ \mathit{Exp})\ \mathbf{where} \\
&\quad \mathit{step}\ (\mathrm{Mk}_{minus}\ x\ y) = \mathit{minus}\ x\ y
\end{aligned}$$

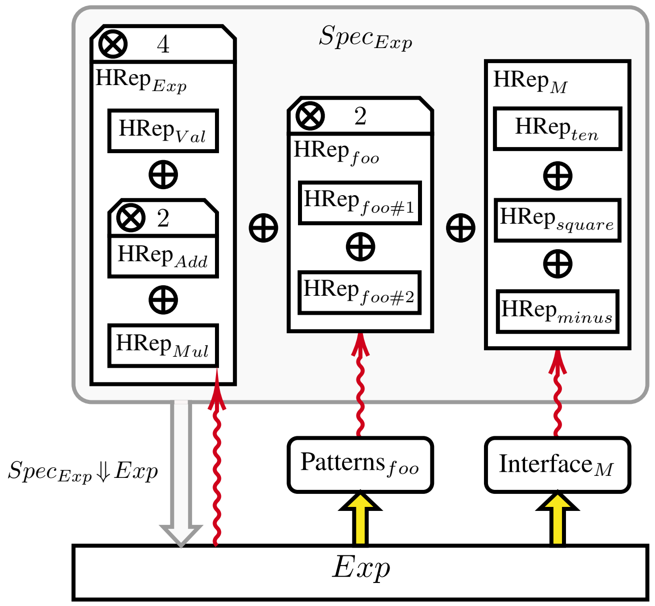

Fig. 1: Higher order representation of the data type *Exp*, using structural information from the function *foo* and the abstract interface of the module M.

$$
\begin{aligned}
\textbf{type}\ \text{HRep}_M = {} & \text{HRep}_{ten} \\
\oplus\ & \text{HRep}_{square} \\
\oplus\ & \text{HRep}_{minus}
\end{aligned}
$$

$$
\begin{aligned}
\textbf{type}\ \text{Spec}_{Exp} = {} & \text{HRep}_{Exp} \otimes 4 \\
\oplus\ & \text{HRep}_{foo} \otimes 2 \\
\oplus\ & \text{HRep}_{M}
\end{aligned}
$$

This previous definition can be interpreted graphically as it is shown in the Figure 1. Curly arrows represent the structural information extracted using meta-programming.

IV. Random Generators Synthesis

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

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

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

V. Related Work

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

VI. Final Remarks

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

References

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