Deriving Random Generators via Higher-Level 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

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].

We remark that, altough this work focuses on Haskell data types, this technique is general enough to be applied to most programming languages with some level of support for composite types.

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 type 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. Random Data Generation in Haskell

In this section we briefly introduce the common approach for automatically deriving random data generators in Haskell using a type-driven approach, along with its main drawbacks.

Haskell is a strongly typed programming language with a powerful type system. It lets programmers encode a considerable amount of information about the structure 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 terms of **sums** (also known as *variant types*) and **products** (or tuples) of other data types. To exemplify this, consider the following type definition to encoding *Html* expressions:

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

In the previous definition, we declare *Html* as the sum of four possible constructions: *Text* represents plain text values. *Sing* and *Tag* represent singular and paired Html tags, and *Join* concats two expression one after another. We only encode a very small subset of the actual Html specification for illustrative reasons. In Haskell, *Text*, *Sing* and *Join* and *Tag* are known as *data constructors* (or constructors for short) and are used to distiguish which variant of the data type 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* two recusive 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 reprentation, the expression "*<html>hello<hr>bye</html>*" can be encoded as:

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

Additionally, we can define a function *render* to serialize *Html* values as follows:

```
render :: Html → String
render (Text t) = t
```

$render\ (Sing\ t) = \texttt{"<"} \mathbin{+\!\!+} t \mathbin{+\!\!+} \texttt{">"}$
$render\ (Join\ x\ y) = render\ x \mathbin{+\!\!+} render\ y$
$render\ (Tag\ t\ x)$
$\quad = \texttt{"<"} \mathbin{+\!\!+} t \mathbin{+\!\!+} \texttt{">"} \mathbin{+\!\!+} render\ x \mathbin{+\!\!+} \texttt{"</"} \mathbin{+\!\!+} t \mathbin{+\!\!+} \texttt{">"}$

In the previous definition, *render* is described using *pattern matching* over each possible kind of value. Using pattern matching we can define functions idiomatically by defining different function clauses for each input pattern we are interested on. Patterns can be defined to match specific constructors, literal values or variable subexpressions (like t, x and y in the definition of *render*). They can also be nested in order to match very specific patterns of values.

Type-Driven Generation of Random Values

In order to generate random value of types involving user defined ADTs, most approaches require the user to provide a random data generator for each one of them. This is 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 *Html*:

```
$gen_{Html} = sized\ (\lambda size \rightarrow$
  $\mathbf{if}\ size \equiv 0$
  $\mathbf{then}\ frequency$
    $[(2, Text \langle\$\rangle\ arbitrary)$
    $, (1, Sing \langle\$\rangle\ arbitrary)]$
  $\mathbf{else}\ frequency$
    $[(2, Text \langle\$\rangle\ arbitrary)$
    $, (1, Sing \langle\$\rangle\ arbitrary)$
    $, (3, Join \langle\$\rangle\ smaller\ gen_{Html} \langle*\rangle\ smaller\ gen_{Html})$
    $, (4, Tag\ \ \langle\$\rangle\ arbitrary \langle*\rangle\ smaller\ gen_{Html})])$
```

This random generator is defined using *QuickCheck*'s combinator *sized* to paremetrize the generation process up to an external natural number known as the *generation size*. This parameter is chosen by the user, and we use to limit the maximum amount of recursive calls that this random generator can perform. When called with a positive generation size, this generator can pick to generate among any *Html* data constructor of with a explicitly given frequency that can be chosen by the user (2, 1, 3 and 4 for *Text*, *Sing*, *Join* and *Tag* constructors, respectively). When it picks to generate a *Val* or a *Sing* data constructor, it also generates a random *String* value using the standard overloaded generator *arbitrary* (*QuickCheck* provides standard random generators for most base types like *String*, *Int*, *Bool*, etc.). [1] On the other hand, when it picks to generate either an *Join* constructor, it also generates two independent random subexpression 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.

[1]The operators $\langle\$\rangle$ and $\langle*\rangle$ are used in Haskell to combine values obtained from effectful computations (like calling to a random generator) and they are not particularly relevant for the point being made in this work.

This random process keeps calling itself recursively until the generation size reaches zero, where the generator is constrained to pick only among terminal data constructors, being *Text* and *Sing* the only possible choices in our particular case.

The previous definition is rather mechanical, except perhaps for the chosen generation frequencies. In this light, it easy to extend this 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* **CITE**. Given a target data type, it synthesizes a random generator for it that behaves similarly to the one presented above, where the generation frequencies are defined to be uniform across constructors. However, picking among different data constructors with uniform frequency can lead to a generation process biased towards generating (in average) very small values, regardless of the generation sized set by the user **CITE**.

DRAGEN is meta-programming tool conceived to mitigate this problem. Instead of setting a uniform generation probability of data constructors, this tool this tool uses the theory of branching processes to modelize and predict analitically 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 frequency of each data constructor in order to obtain a particular distribution of values that can be specified by the user, providing this way a more flexible testing environment while still being mostly automated.

Althogh both *MegaDeTH* and **DRAGEN** synthesize random generators that are theoretically capable to generate the whole space of values of the target data type. 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 often too generic to generate random data with enough structural complexity to be used for testing purposes.

In this work we identify two patological situations which are not properly handled by the aforementioned derivation tools:

1) The target code behaves differently over inputs matching specific patterns of nested values.
2) The target code encodes a significant portion of its structure on its abstract interface.

In Section III we show how this structural information can be used to synthesize richer random generators automatically. We proceed to exemplify the previous points in detail.

Presence of Complex Pattern Matchings

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

$simpl :: Html \rightarrow Html$
$simpl\ (Join\ (Text\ t1)\ (Text\ t2))$
$\quad = Text\ (t1 \mathbin{+\!\!+} t2)$

```
simpl (Join (Join (Text t1) x) y)
   = simpl (Join (Text t1) (simpl (Join x y)))
simpl (Join x y) = Join (simpl x) (simpl y)
simpl (Tag t x) = Tag t (simpl x)
simpl x = x
```

This function simplifies sequences of *Text* constructors into a single big *Text* constructor. To do so, it has to pattern match against sequences of *Text* constructors combined by a *Join* constructor using nested patterns (see *simpl* fist and second clauses). The remaining clauses are only meant to propagate this simplification within nested expressions.

Ideally, we would like to test each clause of the function *simpl* approximately the same amount of time each. However, each data constructor is generated independently when using either *MegaDeTH* or **DRAGEN**, thus the probability of generating a value satisfying a nested pattern decreases multiplicatively with the number of constructors we pattern against to simultaneously. In our tests, we found that the first two clauses of *simpl* get exercised only approximately 1.5% of the time when using *MegaDeTH* to derive a random generator for *HTML*. On the other hand, the best result we could achieve with **DRAGEN** was only able to exercise the first and second clauses of *simpl* approximately 3% and 6% of the time, respectively. With both derivation tools, the most of the time was spent testing the trivial clauses of our function, in view of they pattern match against simpler patterns of input values.

Altough the previous example might seem rather simple, branching against specific patterns of the input data is a common task. For instance, balancing a Red-Black tree requires to consider specific combinations of color, left and right subtrees and sub-subtrees in order to preserve the height invariant **CITE**. 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 **CITE**.

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 which manipulates its values. For this purspose, suppose we extend our *Html* data type with the following basic combinators:

```
module M where
  div :: Html → Html
  div inner = Tag "div" inner
  bold :: Html → Html
  bold inner = Tag "b" inner
  hr :: Html
  hr = Sing "hr"
```

These functions encode additional information about the structure of our *Html* data type in the form of specific Html tags.

Instead of including a new data constructor for each possible Html tag in out type definition, we defined a mininal representation and then extended it with an set of high level combinators. This programming pattern is often called a "shallow embedding", and can be found in a variety of Haskell libraries, being *html* **CITE**, *svg-builder* **CITE**some examples of this.

As a consequence of this practice, type-driven derivation techniques often fail to synthesize useful random generators due to that most of the data type structure has been encoded into its abstract interface of combinators. In our particular case, the chances of generating a *Tag* value representing a commonly used Html tag such as *div* are extremely low.

So far we have introduced two testing scenarios where type-driven derivation approaches are unable to capture all the available structure 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 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 section we present a compositional representation of values to express the random generation of values following the internal structure of their data types along with the structure present on patterns matchings and abstract interfaces.

The key idea of this work is to represent different structured constructions of data in a homogeneous way that we call a "higher-level representation" (*HRep* from now on). Instead of generating each data constructor independently, a random generator derived from this representation might generate composite structured values on each random choice it performs. For this purpose, we use a series of automatically derived data types, each one representing an atomic unit of information that can be randomly generated and then reflected back to the corresponding value of the original data type. Later, the user can compose these atomic representations using the provided type level combinators in different ways into a "generation specification" that completely determines the generation process behavior.

We wil reuse the previously defined data type *Html* and the functions defined in the previous section to explain the different concepts involved all across this section.

Representing Data Constructors

We begin by introducing the simplest data type representation that we can extract from our codebase: the representation of single data constructor. Each data constructor can be represented by an automatically derived data type consisting of a single constructor with the same fields as the original, except for the recursive ones that are abstracted away. In this light, we represent each constructor of the data type *Html* as follows:

$$\mathbf{data}\ HRep_{Text}\ r = Con_{Text}\ String$$
$$\mathbf{data}\ HRep_{Sing}\ r = Con_{Sing}\ String$$

$$\textbf{data}\ \mathit{HRep}_{\mathit{Join}}\ r = \mathit{Con}_{\mathit{Join}}\ r\ r$$
$$\textbf{data}\ \mathit{HRep}_{\mathit{Tag}}\ r = \mathit{Con}_{\mathit{Tag}}\ \mathit{String}\ r$$

Note that the previous definitions are type parametric over the type parameter r. This allow us to replace r with any concrete data type, obtaining different possible values on each case. For instance, the value $(\mathit{Con}_{\mathit{Join}}\ 10\ 20)$ has type $\mathit{HRep}_{\mathit{Join}}\ \mathit{Int}$, while the value $(\mathit{Con}_{\mathit{Join}}\ \mathit{True}\ \mathit{False})$ has type $\mathit{HRep}_{\mathit{Join}}\ \mathit{Bool}$.

In practice, this parametricity let us instatiate r with the type of the chosen generation specification (which might be composed of several *HRep*s), without having to modify anything in the underlying machinery.

Having the *HRep* of each data constructor, we can define an evaluation relation ($[\![\_]\!]_t : \mathit{HRep}_f \to t$) that maps a value from each representation f back to the target data type t. Then, we simply need to translate each constructor representation back into its corresponding one, translating the abstracted fields recursively:

$$\begin{aligned}
[\![\mathit{Con}_{\mathit{Text}}\ s\ ]\!]_{\mathit{Html}} &= \mathit{Text}\ s \\
[\![\mathit{Con}_{\mathit{Sing}}\ s\ ]\!]_{\mathit{Html}} &= \mathit{Sing}\ s \\
[\![\mathit{Con}_{\mathit{Join}}\ x\ y]\!]_{\mathit{Html}} &= \mathit{Join}\ [\![x]\!]_{\mathit{Html}}\ [\![y]\!]_{\mathit{Html}} \\
[\![\mathit{Con}_{\mathit{Tag}}\ s\ x\ ]\!]_{\mathit{Html}} &= \mathit{Tag}\ s\ [\![x]\!]_{\mathit{Html}}
\end{aligned}$$

The missing piece is to automatically synthesize a random generators for each constructor representation. For this purpose, it is important to consider that each constructor *HRep* has its recursive fields abstracted away with a type parameter that will be later instantiated with the generation specification type. Given that this specification is unknown at the derivation time, we parametrize each *HRep* generator with a random generator gen_r that is used to generate random values for each recursive field:

$$\begin{aligned}
\mathit{gen}_{\mathit{Text}}\ \mathit{gen}_r &= \mathit{Con}_{\mathit{Text}}\ \langle\$\rangle\ \mathit{arbitrary} \\
\mathit{gen}_{\mathit{Sing}}\ \mathit{gen}_r &= \mathit{Con}_{\mathit{Sing}}\ \langle\$\rangle\ \mathit{arbitrary} \\
\mathit{gen}_{\mathit{Join}}\ \mathit{gen}_r &= \mathit{Con}_{\mathit{Join}}\ \langle\$\rangle\ \mathit{smaller}\ \mathit{gen}_r\ \langle*\rangle\ \mathit{smaller}\ \mathit{gen}_r \\
\mathit{gen}_{\mathit{Tag}}\ \mathit{gen}_r &= \mathit{Con}_{\mathit{Tag}}\ \langle\$\rangle\ \mathit{arbitrary}\ \langle*\rangle\ \mathit{smaller}\ \mathit{gen}_r
\end{aligned}$$

Type Level Combinators

The next step is to define a series of type level combinators to enable us combining the automatically derived *HRep*s in several ways.

In first place, we define a type combinator ($\_^{\bigstar}$) to tag a *HRep* to be terminal, i.e., a representation that is allowed be generated when the generation size gets exausted:

$$\textbf{data}\ (f^{\bigstar})\ a = \mathit{Term}\ (f\ a)$$

Additionally, we define a combinator ($\otimes$) to tag a *HRep* with an explicit generation frequency n:

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

The previous combinators only include information relevant to the generation process, in a sense that neither one adds new structure to the final representation. In this light, they do not alter the evaluation semantics, and we translate them back to our target data type by evaluating the inner representation:

$$\begin{aligned}
[\![\mathit{Term}\ x : f^{\bigstar}\ ]\!]_t &= [\![x : f]\!]_t \\
[\![\mathit{Freq}\ x\ : f \otimes n]\!]_t &= [\![x : f]\!]_t
\end{aligned}$$

In the previous equations we explicitly annotate (using a colon) the type of the evaluated term for clarity. Later, to generate these combinators is enough to wrap a generated value from the inner representation with the apropriate tag:

$$\begin{aligned}
\mathit{gen}_{f}^{\bigstar}\ \ \mathit{gen}_r &= \mathit{Term}\ \langle\$\rangle\ \mathit{gen}_f\ \mathit{gen}_r \\
\mathit{gen}_{f\otimes n}\ \mathit{gen}_r &= \mathit{Freq}\ \ \langle\$\rangle\ \mathit{gen}_f\ \mathit{gen}_r
\end{aligned}$$

Perhaps more interesting, we define a combinator ($\oplus$) to compose two *HRep*s into a single one using a sum type to represent a random choice between them:

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

A composite representation built using ($\oplus$) is transformed back into the target data type by pattern matching on the data constructor variant and evaluating the inner *HRep* accodingly:

$$\begin{aligned}
[\![L\ x : f \oplus g]\!]_t &= [\![x : f]\!]_t \\
[\![R\ x : f \oplus g]\!]_t &= [\![y : g]\!]_t
\end{aligned}$$

Generating a composite *HRep* is slightly more complicated than before, as we need to perform a random choice based on the generation size and the given frequencies for each sub-represention:

```
$\mathit{gen}_{f\oplus g}\ \mathit{gen}_r = \mathit{sized}\ (\lambda \mathit{size} \to$
  $\textbf{if}\ \mathit{size} \equiv 0$
  $\textbf{then}\ \mathit{frequency}$
    $[(\mathit{freq0}\ \ @f, L\ \langle\$\rangle\ \mathit{gen}_f\ \mathit{gen}_r)$
    $,(\mathit{freq0}\ \ @g, R\ \langle\$\rangle\ \mathit{gen}_g\ \mathit{gen}_r)]$
  $\textbf{else}\ \mathit{frequency}$
    $[(\mathit{freq}\ \ \ @f, L\ \langle\$\rangle\ \mathit{gen}_f\ \mathit{gen}_r)$
    $,(\mathit{freq}\ \ \ @g, R\ \langle\$\rangle\ \mathit{gen}_g\ \mathit{gen}_r)])$
```

In the previous definition, we reflect the type level frequencies of the types f and g (*freq @f* and *freq @g*) into term-level values. This reflection defaults to 1 if the frequency tag ($\otimes$) is not present. Then we use these frequencies to generate each inner *HRep* in the apropriate proportion. When the generation size gets exhausted, we reflect the terminal generation frequency of each inner *HRep* in the same way as before (*freq0 @f* and *freq0 @g*). This time, however, we default the frequency reflection to 0 for any inner that not tagged as terminal, avoiding to generate non-terminal constructions in the last step.

Later, we can use these combinators to create a type synonym $\mathit{HRep}_{\mathit{Html}}$ that specifies a generation schema equivalent to the one seen in the concrete random generator of type *Html* presented in Section II:

$$\begin{aligned}
\textbf{type}\ \mathit{HRep}_{\mathit{Html}} = \ & \mathit{HRep}_{\mathit{Text}}^{\bigstar}\ \otimes\ 2 \\
\oplus\ & \mathit{HRep}_{\mathit{Sing}}^{\bigstar}
\end{aligned}$$

$$\begin{aligned} &\oplus\ \mathit{HRep}_{Join}\ \otimes\ 3 \\ &\oplus\ \mathit{HRep}_{Tag}\ \otimes\ 4 \end{aligned}$$

However, a value of type HRep_{Html} still has its recursive calls abstracted away—the type parameter r is implicit at the definition of HRep_{Html}. We can think of it as a "single layer" of representation. To make it able to represent recursive values we need to define a last type level combinator to "tie the knot":

$$\textbf{data}\ \mathit{Fix}\ f = \mathit{Fix}\ (f\ (\mathit{Fix}\ f))$$

This datatype represents the *fixed point* of a parametric data type f, i.e., a data type where each recursive call gets instantiated with iself. Then, to translate fixed points back to our target data type we simply need translate the inner representation:

$$[\![\mathit{Fix}\ x : \mathit{Fix}\ f]\!]_t = [\![x : f]\!]_t$$

Unlike the other combinators, to define a random generator for the fixed point of a certain representation f, we do not to parametrize the generation of the recursive fields of f over an external generator gen_r. Instead, we replace it with our fixed point generator, calling itself recursively on any recursive field of f that might appear inside:

$$\mathit{genFix}_f = \mathit{Fix}\ \langle\$\rangle\ \mathit{gen}_f\ \mathit{genFix}_f$$

This way we obtain a concrete recursive generator for each representation f that we define. Then, we can define a random generator for our target data type simply by generating a random value of our chosen representation, and transforming it back to our target by the means of the evaluation relation:

$$\begin{aligned} \mathit{gen}_{Html} = \textbf{do}\ & x \leftarrow \mathit{genFix}_{\mathit{HRep}_{Html}} \\ & \mathit{return}\ [\![x]\!]_{Html} \end{aligned}$$

So far we have introduced the machinery required to represent the random generation of a target data type considering only the structure encoded on its definition. However, this approach can now be easily extended to encode different sources of structured information. We proceed to introduce two extensions that help to address the problematic testing scenarios presented in the previous section.

Representing Pattern Matchings

We can follow a similar reasoning as before to represent the pattern matching structure from a given function. Consider the nested pattern matchings of function *simpl* defined in the previous section:

$$\begin{aligned} &\mathit{simpl}\ (\mathit{Join}\ (\mathit{Text}\ t1)\ (\mathit{Text}\ t2))\ = ... \\ &\mathit{simpl}\ (\mathit{Join}\ (\mathit{Join}\ (\mathit{Text}\ t1)\ x)\ y) = ... \end{aligned}$$

To represent these patterns, we derive a new data for each one of them, whose fields represent the pattern variables that occur inside, and abstracting away every pattern variable of type *Html*. Concretely, we define the following data types:

$$\begin{aligned} &\textbf{data}\ \mathit{HRep}_{simpl\#1}\ r = \mathit{Pat}_{simpl\#1}\ \mathit{String}\ \mathit{String} \\ &\textbf{data}\ \mathit{HRep}_{simpl\#2}\ r = \mathit{Pat}_{simpl\#2}\ \mathit{String}\ r\ r \end{aligned}$$

The first pattern of *simpl* contains two pattern variables ($t1$ and $t2$) of type *String* that are included as fields of $\mathit{Pat}_{simpl\#1}$. Similarly, the second pattern of *simpl* contains a pattern variable $t1$ of type *String* represented by the first field of $\mathit{Pat}_{simpl\#2}$, along with two pattern variables (x and y) of type *Html* that are abstracted in the second and third fields of $\mathit{Pat}_{simpl\#2}$.

To transform these representations back to our target data type, we simply need to expand them it into the concrete value represented by the original pattern, evaluating its fields back to *Html* as well:

$$\begin{aligned} &[\![\mathit{Pat}_{simpl\#1}\ t1\ t2]\!]_{Html} \\ &\quad = \mathit{Join}\ (\mathit{Text}\ [\![t1]\!]_{Html})\ (\mathit{Text}\ [\![t2]\!]_{Html}) \\ &[\![\mathit{Pat}_{simpl\#2}\ t1\ x\ y]\!]_{Html} = \\ &\quad = \mathit{Join}\ (\mathit{Join}\ (\mathit{Text}\ [\![t1]\!]_{Html}))\ [\![x]\!]_{Html})\ [\![y]\!]_{Html}) \end{aligned}$$

This way, any value of type $\mathit{HRep}_{simpl\#1}$ is guaranteed to satisfy the first pattern matching of *simpl*—the same property follows for $\mathit{HRep}_{simpl\#2}$.

The random generation of pattern *HRep*s is defined in the same way as we did before for representations of data constructors:

$$\begin{aligned} &\mathit{gen}_{simpl\#1}\ \mathit{gen}_r \\ &\quad = \mathit{Pat}_{simpl\#1}\ \langle\$\rangle\ \mathit{arbitrary}\ \langle * \rangle\ \mathit{arbitrary} \\ &\mathit{gen}_{simpl\#2}\ \mathit{gen}_r \\ &\quad = \mathit{Pat}_{simpl\#2}\ \langle\$\rangle\ \mathit{arbitrary}\ \langle * \rangle\ \mathit{smaller}\ \mathit{gen}_r\ \langle * \rangle\ \mathit{smaller}\ \mathit{gen}_r \end{aligned}$$

Finally, we can join the pattern matching representation of each clause of *simpl* into a single one:

$$\textbf{type}\ \mathit{HRep}_{simpl} = \mathit{HRep}_{simpl\#1} \oplus \mathit{HRep}_{simpl\#2}$$

Representing Abstract Interfaces

To introduce the higher level representation of module abstract interfaces, consider module M defininf *Html* combinators introduced in the previous section.

We can represent each function of M that returns a value of type *Html* following the same idea as before, deriving a data type with a single data constructor for each one of them:

$$\begin{aligned} &\textbf{data}\ \mathit{HRep}_{div}\ \ r = \mathit{Fun}_{div}\ r \\ &\textbf{data}\ \mathit{HRep}_{bold}\ r = \mathit{Fun}_{bold}\ r \\ &\textbf{data}\ \mathit{HRep}_{hr}\ \ \ r = \mathit{Fun}_{hr} \end{aligned}$$

In this case, each single constructor will have as fields the types of the inputs of the function that they represent. As before, we abstract away any field representing and input of type *Html* with a type parameter r.

To evaluate these representations, we simply use the values on its fields as input parameter of each original function, returning its result.

$$\begin{aligned} &[\![\mathit{Fun}_{div}\ \ x]\!]_{Html} = \mathit{div}\ \mathit{eval\_x\_html} \\ &[\![\mathit{Fun}_{bold}\ x]\!]_{Html} = \mathit{bold}\ \mathit{eval\_x\_html} \\ &[\![\mathit{Fun}_{hr}\ \ \ \ ]\!]_{Html} = \mathit{hr} \end{aligned}$$

Note that, by doing this, the generation process inherits any patology that the functions we use to generate values might have. For instance, if the function div would happen to be non-terminating for some inputs, our generation process could suffer from this as well.

Furthermore, the generation procedure for abstract interface representations follows the same pattern as before:

$$\begin{aligned} &gen_{div}\ \ gen_r = Fun_{div} \langle \$ \rangle\ smaller\ gen_r \\ &gen_{bold}\ gen_r = Fun_{bold} \langle \$ \rangle\ smaller\ gen_r \\ &gen_{hr}\ \ gen_r = return\ Fun_{hr} \end{aligned}$$

And we will also define a type synonym to join all the representations of the module M into a single one.

$$\textbf{type}\ HRep_M = HRep_{div} \oplus HRep_{bold} \oplus HRep_{hr}$$

Finally, we can put all the derived machinery together into a generation specification $Html_S$, assigning (possibly) different generation frequencies to each individual $HRep$ we combine:

$$\begin{aligned} \textbf{type}\ Html_S = &\ HRep_{Html} \otimes 4 \\ \oplus &\ HRep_{simpl} \otimes 2 \\ \oplus &\ HRep_M \end{aligned}$$

We want to remark that, for space reasons, we were only able to introduce the representation of a rather simple target data type. In practice, this reasoning can be extended to mutually recursive and parametric types as well.

Overall, this approach offers significant advantages over the usual type-driven derivation of random generators:

- **Composability:** we can combine different atomic representations using different structure information sources depending on what property or sub-system we need to verify using randomly generated values.
- **Extensibility:** the developer can derive representations for new sources of structure information and combine them with the existing ones simply by adding them to the existing generation specification of the target data type.
- **Predictability:** using branching processes theory, it is possible to predict the average distribution of generated values in terms of number of constructors. This prediction is completely modular, and can be obtained for any composite representation obtained using the automatically derived $HRep$s and the provided combinators.

Maybe we can show an example random value from the representation and its corresponding target value

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.

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.

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.