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].

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. Type-Driven Random Generation

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

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

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

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

$$\begin{aligned}&eval :: Exp \to Int \\ &eval\ (Val\ n) \quad = n\end{aligned}$$

$$eval\ (Add\ x\ y) = eval\ x + eval\ y$$
$$eval\ (Mul\ x\ y) = eval\ x * eval\ y$$

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.

Type-Driven Generation of Random Values

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

$$\begin{aligned}
&gen_{Exp} = sized\ (\lambda size \rightarrow \\
&\quad \textbf{if}\ size \equiv 0 \\
&\quad \textbf{then}\ Val\ \langle\$\rangle\ arbitrary \\
&\quad \textbf{else}\ frequency \\
&\qquad [(1,\ Val\ \langle\$\rangle\ arbitrary) \\
&\qquad ,(2,\ Add\ \langle\$\rangle\ smaller\ gen_{Exp}\ \langle*\rangle\ smaller\ gen_{Exp}) \\
&\qquad ,(1,\ Mul\ \langle\$\rangle\ smaller\ gen_{Exp}\ \langle*\rangle\ smaller\ gen_{Exp})])
\end{aligned}$$

This random generator is defined using *QuickCheck*'s combinator $sized$ 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 will pick among any data constructor of Exp with a explicitly given frequency—two times more Adds than Val or Mul. 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 ($resize\ (n-1)\ gen_{Exp}$), decreasing the generation size 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 **CITE**, 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 **CITE**.

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.

Presence of Complex Pattern Matchings

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

$$\begin{aligned}
&foo :: Exp \rightarrow Exp \\
&foo\ (Add\ (Add\ x\ (Val\ 50))\ (Add\ (Val\ 25)\ y)) \\
&\quad = error\ \texttt{"pattern \#1"}
\end{aligned}$$

$$
\begin{aligned}
&\mathit{foo}\ (\mathit{Mul}\ (\mathit{Val}\ 50)\ (\mathit{Mul}\ (\mathit{Val}\ x)\ y)) \\
&\quad = \mathit{error}\ \texttt{"pattern \#2"} \\
&\mathit{foo}\ x = x
\end{aligned}
$$

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 Ints 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:

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

If we use ***MegaDeTH*** to automatically derive a random generator for *Exp*, we obtain a uniform generation probability distribution over constructors, i.e., $P(Val) = P(Add) = P(Mul) = 1/3$. 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 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.

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:

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

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 **CITE**.

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:

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

In the previous definition we state that $head$ always returns a valid $Html$ value ($valid\ \ (head\ \ x)$), provided that we are given a valid $Html$ as input ($valid\ \ x$). 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 section we present a compositional representation 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 sources of structured value information of a data type in a homogeneous way that we call a "higher-level representation" (HRep from now on).[1] 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.

We wil reuse the previously defined data type *Exp* and the function *foo* 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 *Exp* as follows:

$$\begin{aligned}&\mathbf{data}\ HRep_{Val}\ \ r = Mk_{Val}\ Int\\&\mathbf{data}\ HRep_{Add}\ \ r = Mk_{Add}\ r\ r\\&\mathbf{data}\ HRep_{Mul}\ r = Mk_{Mul}\ r\ r\end{aligned}$$

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 (Mk_{Add} 10 20) has type $HRep_{Add}\ Int$, while the value ($Mk_{Mul}\ True\ False$) has type $HRep_{Mul}\ Bool$. In practice, this parametricity let us instatiate r with the type of the chosen generation specification (which might be composed of several HReps), without having to modify anything in the underlying machinery.

Having the HRep of each data constructor, we can define an evaluation relation ($[\![\_]\!]_t : \text{HRep}_f \to t$) 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:

$$\begin{aligned}&[\![Mk_{Val}\ n\ \ ]\!]_{Exp} = Val\ n\\&[\![Mk_{Add}\ x\ y]\!]_{Exp} = Add\ [\![x]\!]_{Exp}\ [\![y]\!]_{Exp}\\&[\![Mk_{Mul}\ x\ y]\!]_{Exp} = Mul\ [\![x]\!]_{Exp}\ [\![y]\!]_{Exp}\end{aligned}$$

The last missing piece is to automatically derive a random generators for each constructor HRep. 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 each recursive fields:

$$\begin{aligned}&gen_{Val}\ gen_r = Mk_{Val}\ \langle\$\rangle\ arbitrary\\&gen_{Add}\ gen_r = Mk_{Add}\ \langle\$\rangle\ smaller\ gen_r\ \langle*\rangle\ smaller\ gen_r\\&gen_{Mul}\ gen_r = Mk_{Mul}\ \langle\$\rangle\ smaller\ gen_r\ \langle*\rangle\ smaller\ gen_r\end{aligned}$$

Type Level Combinators

In this work we define type level combinators which let the user combine the automatically derived HReps 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:

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

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

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

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:

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

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:

$$\begin{aligned}&gen_{f^{\bigstar}}\ \ gen_r = Term\ \langle\$\rangle\ gen_f\ gen_r\\&gen_{f\otimes n}\ gen_r = Freq\ \ \langle\$\rangle\ gen_f\ gen_r\end{aligned}$$

Perhaps more interesting, we define a combinator ($\oplus$) to compose two HReps into a single one using a sum type and representing a random choice between them:

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

[1] The notion of higher-level comes from that the generation process of given target data type is entirely determined by the type of the chosen representation, instead of by a concrete generator defined at the term level.

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

$$[\![L\ x : f \oplus g]\!]_t = [\![x : f]\!]_t$$
$$[\![R\ x : f \oplus g]\!]_t = [\![y : g]\!]_t$$

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:

$$\begin{aligned}
&gen_{f\oplus g}\ gen_r = sized\ (\lambda size \rightarrow \\
&\quad \textbf{if}\ size \equiv 0 \\
&\quad \textbf{then}\ frequency \\
&\qquad [(freq0\ \ @f, L \langle\$\rangle gen_f\ gen_r) \\
&\qquad ,(freq0\ \ @g, R \langle\$\rangle gen_g\ gen_r)] \\
&\quad \textbf{else}\ frequency \\
&\qquad [(freq\ \ @f, L \langle\$\rangle gen_f\ gen_r) \\
&\qquad ,(freq\ \ @g, R \langle\$\rangle gen_g\ gen_r)])
\end{aligned}$$

The previous definition we reflect the type level frequencies of f and g (*freq* @f and *freq* @g). This frequency reflection defaults to 1 if no frequency tag $(\otimes)$ is used. 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.

With the introduced combinators, we can easily create a type synonym $HRep_{Exp}$ to specify a generation schema equivalent to the one seen in the concrete random generator of type *Exp* presented in Section II:

$$\begin{aligned}
\textbf{type}\ HRep_{Exp} = &\ HRep^{\bigstar}_{Val} \\
&\oplus\ HRep_{Add} \otimes 2 \\
&\oplus\ HRep_{Mul}
\end{aligned}$$

However, a value of type $HRep_{Exp}$ still has its recursive calls abstracted away—the type parameter r is implicit in the definition of $HRep_{Exp}$. If we want to generate recursive values using this representation we need to introduce a last type level combinator to "tie the knot" recursively:

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

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. To translate fixed points back to our target data type we simply need translate the inner representation:

$$[\![Fix\ x : Fix\ f]\!]_t = [\![x : f]\!]_t$$

Categorically, this generic transformation of fixed points is called a *catamorphism*, where our evaluation relation is known as its *algebra*.

Unlike the other combinators, to randomly generate a fixed point of a certain representation f, we do not parametrize the generation of the recursive fields of the inner represention over an external generator gen_r. Instead, we can tie the recursive knot in our random generator by piping the generation of the inner HRep, and calling itself to generate recursive values:

$$genFix_f = Fix \langle\$\rangle gen_f\ genFix_f$$

This way we obtain a concrete recursive generator for each representation f that we define. With this generator, we can define a random generator for our target data type simply by evaluating a random value of the representation:

$$\begin{aligned}
gen_{Exp} = \textbf{do}\ &x \leftarrow genFix_{\,\mathrm{HRep}_{Exp}} \\
&return\ [\![x]\!]_{Exp}
\end{aligned}$$

So far we have introduced the representation machinery required to represent the random generation of a target data type considering only the structure encoded on its definition. This approach has the advantage of being extendable with 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 function *foo* defined in the previous section. We can derive a new data type for each one of its pattern matchings, whose fields represent each pattern variable present on the given pattern, abstracting away every pattern variable of type *Exp*. Concretely, we define the following data types:

$$\begin{aligned}
&\textbf{data}\ HRep_{foo\#1}\ r = Mk_{foo\#1}\ r\ r \\
&\textbf{data}\ HRep_{foo\#2}\ r = Mk_{foo\#2}\ Int\ r
\end{aligned}$$

The first pattern of *foo* conteins two pattern variables (x and y) of type *Exp* that are abstracted in the fields of $Mk_{foo\#1}$. Similarly, the second pattern of *foo* contains the pattern variable x of type *Int* represented by the first field of $Mk_{foo\#2}$, along with the pattern variable y of type *Exp* that also is abstracted in the second field of $Mk_{foo\#2}$.

To evaluate these representations back to our target data type, we simply expand them it into the concrete value represented by the original pattern:

$$\begin{aligned}
&[\![Mk_{foo\#1}\ x\ y]\!]_{Exp} = \\
&\quad Add\ (Add\ [\![x]\!]_{Exp}\ (Val\ 50))\ (Add\ (Val\ 25)\ [\![y]\!]_{Exp}) \\
&[\![Mk_{foo\#2}\ x\ y]\!]_{Exp} = \\
&\quad Mul\ (Val\ 50)\ (Mul\ (Val\ x)\ [\![y]\!]_{Exp})
\end{aligned}$$

And the random generation of pattern HReps is defined in the same way as we did before with data constructors HReps:

$$\begin{aligned}
&gen_{foo\#1}\ gen_r = Mk_{foo\#1} \langle\$\rangle\ smaller\ gen_r \langle*\rangle\ smaller\ gen_r \\
&gen_{foo\#2}\ gen_r = Mk_{foo\#2} \langle\$\rangle\ arbitrary \qquad \langle*\rangle\ smaller\ gen_r
\end{aligned}$$

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

$$\textbf{type}\ HRep_{foo} = HRep_{foo\#1} \oplus HRep_{foo\#2}$$

Representing Abstract Interfaces

To introduce the higher level representation of module abstract interfaces, suppose we have a module M defining Exp combinators as follows:

$$\textbf{module } M \textbf{ where}$$
$$ten :: Exp$$
$$ten = Val\ 10$$
$$square :: Exp \to Exp$$
$$square\ x = Mul\ x\ x$$
$$minus :: Exp \to Exp \to Exp$$
$$minus\ x\ y = Add\ x\ (Mul\ y\ (Val\ (-1)))$$

We will represent each function of M returning a value of type Exp following the same idea as before, deriving a data type with a single data constructor for each one of them:

$$\textbf{data } HRep_{ten}\ \ r = Mk_{ten}$$
$$\textbf{data } HRep_{square}\ r = Mk_{square}\ r$$
$$\textbf{data } HRep_{minus}\ r = Mk_{minus}\ r\ r$$

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 Exp with a type parameter r.

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

$$[\![Mk_{ten}\ \ ]\!]_{Exp} = ten$$
$$[\![Mk_{square}\ x\ ]\!]_{Exp} = square\ [\![x]\!]_{Exp}$$
$$[\![Mk_{minus}\ x\ y]\!]_{Exp} = minus\ [\![x]\!]_{Exp}\ [\![y]\!]_{Exp}$$

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

$$gen_{ten}\ \ gen_r = Mk_{ten}$$
$$gen_{square}\ gen_r = Mk_{square}\ \langle\$\rangle\ smaller\ gen_r$$
$$gen_{minus}\ gen_r = Mk_{minus}\ \langle\$\rangle\ smaller\ gen_r\ \langle*\rangle\ smaller\ gen_r$$

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

$$\textbf{type } HRep_M = HRep_{ten} \oplus HRep_{square} \oplus HRep_{minus}$$

Finally, we can put all the derived machinery together into a generation specification $Spec$ for the type Exp, assigning (possibly) different generation frequencies to each individual HRep that constitutes it:

$$\begin{aligned}\textbf{type } Spec = \ & HRep_{Exp} \otimes 4 \\ \oplus\ & HRep_{foo} \otimes 2 \\ \oplus\ & HRep_M\end{aligned}$$

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

This representation can be extended to mutually recursive and parametric types ...

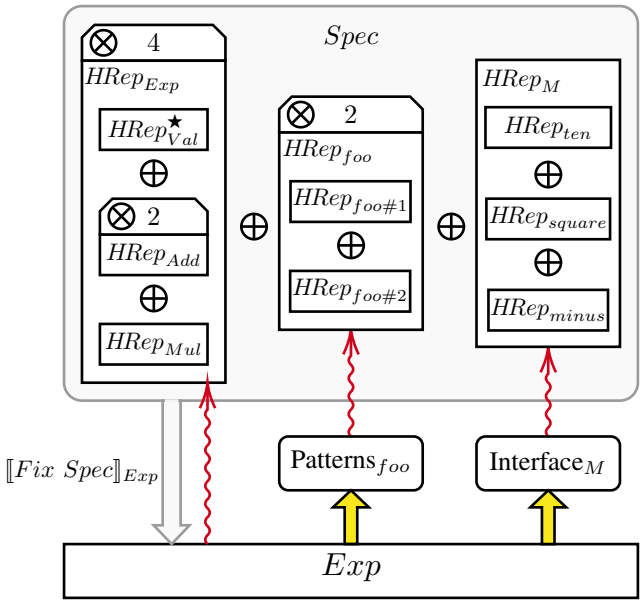

Fig. 1: Higher level representation of the data type Exp, defined using structural information from the function foo and the abstract interface of the module M.

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.