# Deriving Compositional Random Generators

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

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

## ABSTRACT

Generating good random values described by algebraic data types is often quite intricate. State-of-the-art tools for synthesizing random generators serve the valuable purpose of helping with this task, while providing different levels of invariants imposed over the generated values. However, they are often not built for composability nor extensibility, a useful feature when the shape of our random data needs to be adapted while testing different properties or sub-systems.

In this work, we develop an extensible framework for deriving compositional generators, which can be easily combined in different ways in order to fit developers' demands using a simple type-level description language. Our framework relies on familiar ideas from the à la Carte technique for writing composable interpreters in Haskell. In particular, we adapt this technique with the machinery required in the scope of random generation, showing how concepts like generation frequency or terminal constructions can also be expresed in the same type-level fashion. We provide an implementation of our ideas, and evaluate its performance using real-world examples.

## CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; *Software maintenance tools.*

## KEYWORDS

random testing, type-level programming, Haskell

**ACM Reference Format:**
Agustín Mista and Alejandro Russo. 2020. Deriving Compositional Random Generators. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, New York, NY, USA, 12 pages.

## 1 INTRODUCTION

Random property-based testing is a powerful technique for finding bugs [1, 10, 11, 16]. In Haskell, *QuickCheck* is the predominant tool for this task [2]. The developers specify (i) the testing properties their systems must fulfill, and (ii) random data generators (or generators for short) for the data types involved at their properties. Then, *QuickCheck* generates random values, and uses them to evaluate the testing properties in search of possible counterexamples, which always indicate the presence of bugs, either in the program or in the specification of our properties.

Although *QuickCheck* provides default generators for the common base types, like `Int` or `String`, it requires implementing generators for any user-defined data type we want to generate. This process is cumbersome and error prone, and commonly follows closely the shape of our data types. Fortunately, there exists a variety of tools helping with this task, providing different levels of invariants on the generated values as well as automation [6, 8, 14, 18]. We divide the different approaches in two kinds: those which are *manual*, where generators are often able to enforce a wide-range of invariants on the generated data, and those which are *automatic* where the generators can only guarantee lightweight invariants like generating well-typed values.

On the manual side, *Luck* [14] is a domain-specific language for manually writing testing properties and random generators in tandem. It allows obtaining generators specialized to produce random data which is proven to satisfy the preconditions of their corresponding properties. In contrast, on the automatic side, tools like *MegaDeTH* [8, 9], *DRAGEN* [18] and *Feat* [6] allow obtaining random generators automatically at compile time. *MegaDeTH* and *DRAGEN* derive random generators following a simple recipe: to generate a value, they simply pick a random data constructor from our data type with a given probability, and proceed to generate the required sub-terms recursively. *MegaDeTH* pays no attention to the generation frequencies, nor the distribution induced by the derived generator—it just picks among data constructors with uniform probability. Differently, *DRAGEN* analyzes type definitions, and tunes the generation frequencies to match the desired distribution of random values specified by developers. Finally, *Feat* relies on functional enumerations, deriving random generators which sample random values uniformly across the whole search space of values of up to a given size of the data type under consideration. In this work, we focus on automatic approaches to derive generators.

While *MegaDeTH*, *DRAGEN*, and *Feat* provide a useful mechanism for automating the task of writing random generators by hand, they implement a derivation procedure which is often too generic to synthesize useful generators in common scenarios, mostly because *they only consider the structural information encoded in type definitions.* To illustrate this point, consider the following type definition encoding basic HTML pages—inspired by the widely used *html* package:[1]

```
data Html =
    Text String
  | Sing String
  | Tag  String Html
  | Html :+: Html
```

This type allows building HTML pages via four possible data constructors: `Text` is used for plain text values; `Sing` and `Tag` represent singular and paired HTML tags, respectively; whereas the infix `(:+:)` constructor simply concatenates two HTML pages one

[1]http://hackage.haskell.org/package/html

after another. Note that the constructors Tag and (:+:) are recursive, as they have at least one field of type Html. Then, the example page <html>hi<br><b>bye</b></html> can be encoded with the following Html value:

```
Tag "html" (Text "hi" :+: Sing "br" :+: Tag "b" (Text "bye"))
```

In this work, we focus on two scenarios where deriving generators following only the information extracted from type definitions does not work well. The first case is when type definitions are too general (like the case of Html) where, as consequence, the generation process leaves a large room for ill-formed values, e.g., invalid HTML pages. For instance, when generating an Html value using the Sing constructor, it is very likely that an automatically derived generator will choose a random string not corresponding to any valid HTML singular tag. In such situations, a common practice is to rely on existing abstract interfaces to generate random values—such interfaces are often designed to preserve our desired invariants. As an example, consider that our Html data type comes equipped with the following abstract interface:

```
br :: Html
bold :: Html → Html
list :: [Html] → Html
(⟨+⟩) :: Html → Html → Html
```

These high-level combinators let us represent structured HTML constructions like line breaks (br), bold blocks (bold), unordered lists (list) and concatenation of values one below another (⟨+⟩). This methodology of generating random data employing high-level combinators has shown to be particularly useful in the presence of monadic code [3, 9].

The second scenario that we consider is that where derived generators fails at producing very specific patterns of values which might be needed to trigger bugs. For instance, a function for simplifying Html values might be defined to branch differently over complex sequences of Text and (:+:) constructors:

```
simplify :: Html → Html
simplify (Text t_1 :+: Text t_2) = ···
simplify (Text t  :+: x :+: y)   = ···
simplify ···                     = ···
```

(Symbol ··· denote code that is not relevant for the point being made.) Generating values that match, for instance, the pattern Text t :+: x :+: y using *DRAGEN* under an uniform distribution will only occur 6% of the times! Clearly, these input pattern matchings should be included as well into our generators, allowing them to produce random values satisfying such inputs. This structural information can help increasing the chances of reaching portions of our code which otherwise would be very difficult to test. Functions pattern matchings often expose interesting relationships between multiple data constructors, a valuable asset for testing complex systems expecting highly structured inputs [13].

Our previous work [17] focuses on extending *DRAGEN*'s generators as well as its predictive approach to include all these extra sources of structural information, namely high-level combinators and functions' input patterns, while allowing tuning the generation parameters based on the developers' demands. In turn, this work focuses on an orthogonal problem: that of *modularity*. In essence, all

**(a) Machinery derivation**

```
derive [constructors ''Html
       ,interface    ''Html
       ,patterns     'simplify]
```

**(b) Generators specification**

```
type Html_valid =                   type Html_simplify =
     Con "Text"  ⊗ 2                     Con "Text"         ⊗ 2
   ⊕ Con ":+:"   ⊗ 4                   ⊕ Con "Sing"         ⊗ 1
   ⊕ Fun "hr"    ⊗ 3                   ⊕ Con "Tag"          ⊗ 3
   ⊕ Fun "bold"  ⊗ 2                   ⊕ Con ":+:"          ⊗ 4
   ⊕ Fun "list"  ⊗ 3                   ⊕ Pat "simplify" 1   ⊗ 3
   ⊕ Fun "<+>"   ⊗ 5                   ⊕ Pat "simplify" 2   ⊗ 5

genHtml_valid    = genRep @Html_valid
genHtml_simplify = genRep @Html_simplify
```

**Figure 1: Usage example of our framework. Two random generators obtained from the same underlying machinery.**

the automatic tools cited above work by synthesizing *rigid* monolithic generator definitions. Once derived, these generators have almost no parameters available for adjusting the shape of our random data. Sadly, this is something we might want to do if we need to test different properties or sub-systems using random values generated in slightly different ways. As the reader might appreciate, it can become handy to cherry pick, for each situation, which data constructors, abstract interfaces functions, or functions' input patterns to consider when generating random values.

The contribution of this work is an automated framework for synthesizing compositional random generators, which can be naturally extended to include the extra sources of structural information mentioned above. Using our approach, a user can obtain random generators following different *generation specifications* whenever necessary, all of them built upon the *same* underlying machinery which only needs to be derived *once*.

Figure 1 illustrates a possible usage scenario of our approach. We first invoke a derivation procedure (1a) to extract the structural information of the type Html encoded on (i) its data constructors, (ii) its abstract interface, and (iii) the patterns from the function simplify. Then, two different generation specifications, namely Html$_{\text{valid}}$ and Html$_{\text{simplify}}$ can be defined using a simple type-level idiom (1b). Each specification mentions the different sources of structural information to consider, along with (perhaps) their respective generation frequency. Intuitively, Html$_{\text{valid}}$ chooses among the constructors Text and :+:, as well as functions from Html's abstract interface; while Html$_{\text{simplify}}$ chooses among all Html's constructors and the patterns of the first and second clauses in the function simplify. The syntax used there will be addressed in detail in Sections 3 to 5. Finally, we obtain two concrete random generators following such specifications by writing genRep @Html$_{\text{valid}}$ and genRep @Html$_{\text{simplify}}$, respectively.

The main contribution of this paper are:

- We present an extensible mechanism for representing random values built upon different sources of structural information, adopting ideas from *Data Types à la Carte* [24] (Section 3).

- We develop a modular generation scheme, extending our representation to encode information relevant to the generation process at the type level (Section 4).
- We propose a simple type-level idiom for describing extensible generators, based on the types used to represent the desired shape of our random data (Section 5).
- We provide a Template Haskell tool[2] for automatically deriving all the required machinery presented throughout this paper, and evaluate its generation performance with three real-world case studies and a type-level runtime optimization (Section 6).

Overall, we present a novel technique for reusing automatically derived generators in a composable fashion, in contrast to the usual paradigm of synthesizing rigid, monolithic generators.

## 2 RANDOM GENERATORS IN HASKELL

In this section, we introduce the common approach for writing random generators in Haskell using *QuickCheck*, along with the motivation for including extra information into our generators, discussing how this could be naively implemented in practice.

In order to provide a common interface for writing generators, *QuickCheck* uses Haskell's overloading mechanism known as *type classes* [26], defining the Arbitrary class for random generators as:

```
class Arbitrary a where
  arbitrary :: Gen a
```

where the overloaded symbol arbitrary :: Gen a denotes a monadic generator for values of type a. Using this mechanism, a user can define a sensible random generator for our Html data type as follows:

```
instance Arbitrary Html where
  arbitrary = sized gen
    where gen 0 = frequency
                [(2, Text ⟨$⟩ arbitrary)
                ,(1, Sing ⟨$⟩ arbitrary)]
          gen d = frequency
                [(2, Text ⟨$⟩ arbitrary)
                ,(1, Sing ⟨$⟩ arbitrary)
                ,(4, Tag  ⟨$⟩ arbitrary ⟨∗⟩ gen (d-1))
                ,(3, (:+:) ⟨$⟩ gen (d-1)  ⟨∗⟩ gen (d-1))]
```

At the top level, this definition parameterizes the generation process using *QuickCheck*'s sized combinator, which lets us build our generator via an auxiliary, locally defined function gen :: Int → Gen Html. The Int passed to gen is known as the *generation size*, and is threaded seamlessly by *QuickCheck* on each call to arbitrary. We use this parameter to limit the maximum amount of recursive calls that our generator can perform, and thus the maximum depth of the generated values. If the generation size is positive (case gen d), our generator picks a random Html constructor with a given generation frequency (denoted here by the arbitrarily chosen numbers 2, 1, 4 and 3) using *QuickCheck*'s frequency combinator. Then, our generator proceeds to fill its fields using randomly generated sub-terms—here using Haskell's applicative notation [15] and the default Arbitrary instance for Strings. For the case of the recursive sub-terms, this generator simply calls the function gen recursively with a smaller depth limit (gen (d-1)). This process repeats until we reach the base case (gen 0) on each recursive sub-term. At this point, our generator is limited to pick only among terminal Html constructors, hence ending the generation process.

As one can observe, the previous definition is quite mechanical, and depends only on the generation frequencies we choose for each constructor. This simple generation procedure is the one used by tools like *MegaDeTH* or *DRAGEN* when synthesizing generators.

### 2.1 Abstract Interfaces

A common choice when implementing abstract data types is to transfer the responsibility of preserving their invariants to the functions on their abstract interface. Take for example our Html data type. Instead of defining a different constructor for each possible HTML construction, we opted for a small generic representation that can be extended with a set of high-level combinators:

```
br :: Html
br = Sing "br"

bold :: Html → Html
bold = Tag "b"

list :: [Html] → Html
list [ ] = Text "empty list"
list xs = Tag "ul" (foldl1 (:+:) (Tag "li" ⟨$⟩ xs))

(⟨+⟩) :: Html → Html → Html
(⟨+⟩) x y = x :+: br :+: y
```

Note how difficult it would be to generate random values containing, for example, structurally valid HTML lists, if we only consider the structural information encoded in our Html type definition. After all, much of the valid structure of HTML has been encoded on its abstract interface.

A synthesized generator could easily contemplate this structural information by creating random values arising from applying such functions to randomly generated inputs:

```
instance Arbitrary Html where
  arbitrary = · · ·
    frequency
    [...
    ,(1, pure br)
    ,(5, bold ⟨$⟩ gen (d-1))
    ,(2, list ⟨$⟩ listOf (gen (d-1)))
    ,(3, (⟨+⟩) ⟨$⟩ gen (d-1) ⟨∗⟩ gen (d-1))]
```

where (...) represents the rest of the code of the random generator introduced before. From now on, we will refer to each choice given to the frequency combinator as a different *random construction*, since we are not considering generating only single data constructors anymore, but more general value fragments.

### 2.2 Functions' Pattern Matchings

A different challenge appears when we try to test functions involving complex pattern matchings. Consider, for instance, the full definition of the function simplify introduced in Section 1:

[2]Available at https://github.com/OctopiChalmers/dragen2

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

This function traverses Html values, joining together every contiguous pair of Text constructors. Ideally, we would like to put approximately the same testing effort into each clause of simplify, or perhaps even more to the first two ones, since those are the ones performing actual simplifications. However, these two clauses are the most difficult ones to test in practice! The probability of generating a random value satisfying nested patterns decreases multiplicatively with the number of constructors we simultaneously pattern match against. In our tests, we were not able to exercise any of these two patterns more than 6% of the overall testing time, using random generators derived using both *MegaDeTH* and *DRAGEN*. As expected, most of the random test cases were exercising the simplest (and rather uninteresting) patterns of this function.

To solve this issue, we could opt to consider each complex pattern as a new kind of random construction. In this light, we can simply generate values satisfying patterns directly by returning their corresponding expressions, where each variable or wildcard pattern is filled using a random sub-expression:

```
instance Arbitrary Html where
  arbitrary = ···
    frequency
    [...
    , (2, do t₁ ← arbitrary;  t₂ ← arbitrary;
            return (Text t₁ :+: Text t₂))
    , (4, do t ← arbitrary;  x ← gen (d-1);  y ← gen (d-1);
            return (Text t :+: x :+: y))]
```

While the ideas presented in this section are plausible, accumulating cruft from different sources of structural information into a single, global Arbitrary instance is unwieldy, especially if we consider that some random constructions might not be relevant or desired in many cases, e.g., generating the patterns of the function simplify might only be useful when testing properties involving such function, and nowhere else.

In contrast, the following sections of this paper present our extensible approach for deriving generators, where the required machinery is derived once, and each variant of our random generators is expressed on a per-case basis.

## 3 MODULAR RANDOM CONSTRUCTIONS

This section introduces a unified representation for the different constructions we might want to consider when generating random values. The key idea of this work is to lift each different source of structural information to the type level. In this light, the shape of our random data is determined entirely by the types we use to represent it during the generation process.

For this purpose, we will use a set of simple "open" *representation types*, each one encoding a single random construction from our *target* data type, i.e., the actual data type we want to randomly generate. These types can be (i) combined in several ways depending on the desired shape of our test data (applying the familiar à la Carte technique); (ii) randomly generated (see Section 4); and finally, (iii) transformed to the corresponding values of our target data type automatically. This representation can be automatically derived from our source code at compile time, relieving programmers of the burden of manually implementing the required machinery.

### 3.1 Representing Data Constructors

When generating values of algebraic data types, the simplest piece of meaningful information we ought to consider is the one given by each one of its constructors. In this light, each constructor of our target type can be represented using a single-constructor data type. Recalling our Html example, its constructors can be represented as:

```
data Con_Text r = Mk_Text String
data Con_Sing r = Mk_Sing String
data Con_Tag  r = Mk_Tag  String r
data Con_(:+:) r = Mk_(:+:) r r
```

Each representation type has the same fields as its corresponding constructor, except for the recursive ones which are abstracted away using a type parameter r. This parametricity lets us leave the type of recursive sub-terms unspecified until we have decided the final shape of our random data. Then, for instance, the value Mk$_{Tag}$ "div" x :: Con$_{Tag}$ r represents the Html value Tag "div" x, for some sub-term x :: r that can be transformed to Html as well. Note how these representations types encode the minimum amount of information they need, leaving everything else unspecified.

An important property of these parametric representations is that, in most cases, they form a functor over its type parameter, thus we can use Haskell's **deriving** mechanism to obtain suitable Functor instances for free—this will be useful for the next steps.

The next building block of our approach consists of providing a mapping from each constructor representation to its corresponding target value, provided that each recursive sub-term has already been translated to its corresponding target value. This notion is often referred as an *F-Algebra* over the functor used to represent each different construction. Accordingly, to represent this mapping, we will define a type class Algebra with a single method alg as follows:

```
class Functor f ⇒ Algebra f a | f → a where
  alg :: f a → a
```

where f is the functor type used to represent a construction of the target type a. The functional dependency f → a helps the type system to solve type of the type variable a, which appears free on the right hand side of the ⇒. This means that, every representation type f will uniquely determine its target type a. Then, we need to instantiate this type class for each data constructor representation we are considering, providing an appropriate implementation for the overloaded alg function:

```
instance Algebra Con_Text Html where
  alg (Mk_Text x) = Text x
instance Algebra Con_Sing Html where
  alg (Mk_Sing x) = Sing x
```

```
instance Algebra Con_Tag Html where
  alg (Mk_Tag t x) = Tag t x
instance Algebra Con_(:+:) Html where
  alg (Mk_(:+:) x y) = x :+: y
```

There, we simply transform each constructor representation into its corresponding data constructor, piping its fields unchanged.

## 3.2 Composing Representations

So far we have seen how to represent each data constructor of our Html data type independently. In order to represent interesting values, we need to be able to combine single representations into (possibly complex) composite ones. For this purpose, we will define a functor type ⊕ to encode the choice between two given representations:

```
data ((f :: ∗ → ∗) ⊕ (g :: ∗ → ∗)) r = In_L (f r) | In_R (g r)
```

This infix type-level operator let us combine two representations f and g into a composite one f ⊕ g, encoding either a value drawn from f (via the $\mathtt{In}_L$ constructor) or a value drawn from g (via the $\mathtt{In}_R$ constructor). This operator works pretty much in the same way as Haskell's Either data type, except that, instead of combining two base types, it works combining two *parametric type constructors*, hence the kind signature $* \to *$ in both f and g. For instance, the type $\mathtt{Con}_{\mathtt{Text}} \oplus \mathtt{Con}_{\mathtt{Tag}}$ encodes values representing either plain text HTMLs or paired tags. Such values can be constructed using the injections $\mathtt{In}_L$ and $\mathtt{In}_R$ on each case, respectively.

The next step consists of providing a mapping from composite representations to target types, provided that each component of can be translated to the same target type:

```
instance (Algebra f a, Algebra g a) ⇒ Algebra (f ⊕ g) a where
  alg (In_L fa) = alg fa
  alg (In_R ga) = alg ga
```

There, we use the appropriate Algebra instance of the inner representation, based on the injection used to create the composite value.

Worth remarking, the order in which we associate each operand of ⊕ results semantically irrelevant. However, in practice, associativity takes as dramatic role when it comes to generation speed. This phenomenon is addressed in detail in Section 6.

## 3.3 Tying the Knot

Even though we have already seen how to encode single and composite representations for our target data types, there is a piece of machinery still missing: our representations are not recursive, but parametric on its recursive fields. We can think of them as a encoding a *single layer* of our target data. In order to represent recursive values, we need to close them *tying the knot* recursively, i.e., once we have fixed a suitable representation type for our target data, each one of its recursive fields has to be instantiated with itself. This can be easily achieved by using a type-level fixed point operator:

```
data Fix (f :: ∗ → ∗) = Fix {unFix :: f (Fix f)}
```

Given a representation type f of kind $* \to *$, the type Fix f instantiates each recursive field of f with Fix f, closing the definition of f into itself—thus the kind of Fix f results $*$.

In general, if a type f is used to represent a given target type, we will refer to Fix f as a *final representation*, since it cannot be further combined or extended—the ⊕ operator has to be applied *within* the Fix type constructor.

The effect of a fixed point combinator is easier to interpret with an example. Let us imagine we want to represent our Html data type using all of its data constructors, employing the following type:

```
type Html' = Con_Text ⊕ Con_Sing ⊕ Con_Tag ⊕ Con_(:+:)
```

Then, for instance, the value x = Text "hi" :+: Sing "hr" :: Html can be represented with a value x' :: Fix Html' as:

```
x' = Fix (In_R (In_R (In_R (Mk_(:+:)
        (Fix (In_L (Mk_Text "hi")))
        (Fix (In_R (In_L (Mk_Sing "hr"))))))))
```

where the sequences of $\mathtt{In}_L$ and $\mathtt{In}_R$ data constructors *inject* each value from an individual representation into the appropriate position of our composite representation Html'.

Finally, we can define a generic function eval to evaluate any value of a final representation type Fix f into its corresponding value of the target type a as follows:

```
eval :: Algebra f a ⇒ Fix f → a
eval = alg ∘ fmap eval ∘ unFix
```

This function exploits the Functor structure of our representations, unwrapping the fixed points and mapping their algebras to the result of evaluating recursively each recursive sub-term.

In our particular example, this function satisfies eval x' == x. More specifically, the types Html and Fix Html' are in fact isomorphic, with eval as the witness of one side of this isomorphism—though this is not the case for any arbitrary representation.

## 3.4 Representing Additional Constructions

The representation mechanism we have developed so far let us determine the shape of our target data based on the type we use to represent its constructors. However, it is hardly useful for random testing, as the values we can represent are still quite unstructured. It is not until we start considering more complex constructions that this approach becomes particularly appealing.

*3.4.1 Abstract Interfaces.* Let us consider the case of generating values obtained by abstract interface functions. If we recall our Html example, the functions on its abstract interface can be used to obtain Html values based on different input arguments. Fortunately, it is easy to extend our approach to incorporate the interesting structure arising from these functions into our framework. As before, we start by defining a set of open data types to encode each function as a random construction:

```
data Fun_br   r = Mk_br
data Fun_bold r = Mk_bold r
data Fun_list r = Mk_list [r]
data Fun_⟨+⟩  r = Mk_⟨+⟩ r r
```

Each data type represents a value resulting from evaluating its corresponding function, using the values encoded on its fields as input arguments. Once again, we replace each recursive field (representing a recursive input argument) with a type parameter r in order to leave the type of the recursive sub-terms unspecified until we have decided the final shape of our data.

By representing values obtained from function application this way, we are not performing any actual computation—we simply store the functions' input arguments. Instead, these functions are evaluated when transforming each representation into its target type, by the means of an `Algebra`:

```
instance Algebra Fun_br Html where
  alg Mk_br = br
instance Algebra Fun_bold Html where
  alg (Mk_bold x) = bold x
instance Algebra Fun_list Html where
  alg (Mk_list xs) = list xs
instance Algebra Fun_⟨+⟩ Html where
  alg (Mk_⟨+⟩ x y) = x⟨+⟩y
```

Where we simply return the result of evaluating each corresponding function, using its representation fields as an input arguments.

It is important to remark that this approach inherits any possible downside from the functions we use to represent our target data. In particular, representing non-terminating functions might produce a non-terminating behavior when calling to the `eval` function.

*3.4.2 Functions' Pattern Matchings.* The second source of structural information that we consider in this work is the one present in functions' pattern matchings. If we recall to our `simplify` function, we can observe it has two complex, non-trivial patterns that we might want to satisfy when generating random values. We can extend our approach in order to represent these patterns as well. We start by defining data types for each one of them, this time using the fields of each single data constructor to encode the free pattern variables (or wildcards) appearing on its corresponding pattern:

```
data Pat_simplify#1 r = Mk_simplify#1 String String
data Pat_simplify#2 r = Mk_simplify#2 String r r
```

where the number after the `#` distinguishes the different patterns from the function `simplify` by the index of the clause they belong to. As before, we abstract away every recursive field (corresponding to a recursive pattern variable or wildcard) with a type variable `r`.

Then, the `Algebra` instance of each pattern will expand each representation into the corresponding target value resembling such pattern, where each pattern variable gets instantiated using the values stored in its representation field:

```
instance Algebra Pat_simplify#1 Html where
  alg (Mk_simplify#1 t_1 t_2) = Text t_1 :+: Text t_2
instance Algebra Pat_simplify#2 Html where
  alg (Mk_simplify#1 t x y) = Text t :+: x :+: y
```

## 3.5 Lightweight Invariants for Free!

Using the machinery presented so far, we can represent values of our target data coming from different sources of structural information in a compositional way.

Using this simple mechanism we can obtain values exposing lightweight invariants very easily. For instance, a value of type `Html` might encode invalid HTML pages if we construct them using invalid tags in the process (via the `Sing` or `Tag` constructors). To avoid this, we can explicitly disallow the direct use of the `Sing` and `Tag` constructors, replacing them with safe constructions from its abstract interface. In this light, a value of type:

$$\mathtt{Con}_{\mathtt{Text}} \oplus \mathtt{Con}_{(:+:)} \oplus \mathtt{Fun}_{\mathtt{br}} \oplus \mathtt{Fun}_{\mathtt{bold}} \oplus \mathtt{Fun}_{\mathtt{list}} \oplus \mathtt{Fun}_{\langle + \rangle}$$

always represents a valid HTML page.

Similarly, we can enforce that every `Text` constructor within a value will always appear in pairs of two, by using the following type:

$$\mathtt{Con}_{\mathtt{Sing}} \oplus \mathtt{Con}_{\mathtt{Tag}} \oplus \mathtt{Con}_{(:+:)} \oplus \mathtt{Pat}_{\mathtt{simplify\#1}}$$

Since the only way to place a `Text` constructor within a value of this type is via the construction $\mathtt{Pat}_{\mathtt{simplify\#1}}$, which always contains two consecutive `Text`s.

As a consequence, generating random data exposing such invariants will simply become using an appropriate representation type while generating random values, without having to rely on runtime reinforcements of any sort. The next section introduces a generic way to generate random values from our different representations, extending them with a set of combinators to encode information relevant to the generation process directly at the type level.

## 4 GENERATING RANDOM CONSTRUCTIONS

So far we have seen how to encode different random constructions representing interesting values from our target data types. Such representations follow a modular approach, where each construction is independent from the rest. This modularity allows us to derive each different construction representation individually, as well to specify the shape of our target data in simple and extensible manner.

In this section, we introduce the machinery required to randomly generate the values encoded using our representations. This step also follows the modular fashion, resulting in a random generation process entirely compositional. In this light, our generators are built from simpler ones (each one representing a single random construction), and are solely based on the types we use to represent the shape of our random data.

Ideally, our aim is to be able to obtain random generators with a behavior similar to the one presented for `Html` in Section 2. If we take a closer look at its definition, there we can observe three factors happening simultaneously:

- We use *QuickCheck*'s generation size to limit the depth of the generated values, reducing it by one on each recursive call of the local auxiliary function `gen`.
- We differentiate between *terminal* and *non-terminal (i.e. recursive) constructors*, picking only among terminal ones when we have reached the maximum depth (case `gen 0`).
- We generate different constructions in a different frequency.

For the rest of this section, we will focus on modeling these aspects in our modular framework, in such a way that does not compromise the compositionality obtained so far.

## 4.1 Depth-Bounded Modular Generators

The first obstacle that arises when trying to generate random values with a limited depth using our approach is related to modularity. If we recall the random generator for `Html` from Section 2 we can observe that the depth parameter `d` is threaded to the different recursive calls of our generator, always within the scope of the local function `gen`. Since each construction will have an specialized random generator, we cannot group them as we did before using an internal `gen` function. Instead, we will define a new type for depth-bounded generators, wrapping *QuickCheck*'s `Gen` type with an external parameter representing the maximum recursive depth:

```
type BGen a = Int → Gen a
```

A `BGen` is, essentially, a normal *QuickCheck* `Gen` with the maximum recursive depth as an input parameter. Using this definition, we can generalize *QuickCheck*'s `Arbitrary` class to work with depth-bounded generators simply as follows:

```
class BArbitrary (a :: ∗) where
  barbitrary :: BGen a
```

From now on, we will use this type class as a more flexible substitute of `Arbitrary`, given that now we have two parameters to tune: the maximum recursive depth, and the *QuickCheck* generation size. The former is useful for tuning the overall size of our random data, whereas the latter can be used for tuning the values of the *leaf types*, such as the maximum length of the random strings or the biggest/smallest random integers.

Here we want to remark that, even though we could have used *QuickCheck*'s generation size to simultaneously model the maximum recursive depth and the maximum size of the leaf types, doing so would imply generating random values with a decreasing size as we move deeper within a random value, obtaining for instance, random trees with all zeroes on its leaves, or random lists skewed to be ordered in decreasing order. In addition, one can always obtain a trivial `Arbitrary` instance from a `BArbitrary` one, by setting the maximum depth to be equal to *QuickCheck*'s generation size:

```
instance BArbitrary a ⇒ Arbitrary a where
  arbitrary = sized barbitrary
```

Even though this extension allows *QuickCheck* generators to be depth-aware, here we also need to consider the parametric nature of our representations. In the previous section, we defined each construction representation as being parametric on the type of its recursive sub-terms, as a way to defer this choice until we have specified the final shape of our target data. Hence, each construction representation is of kind $* \to *$. If we want to define our generators in a modular way, we also need to parameterize somehow the generation of the recursive sub-terms! If we look at *QuickCheck*, this library already defines a type class `Arbitrary1` for parametric types of kind $* \to *$, which solves this issue by receiving the generator for the parametric sub-terms as an argument:

```
class Arbitrary1 (f :: ∗ → ∗) where
  liftArbitrary :: Gen a → Gen (f a)
```

Then, we can use this same mechanism for our modular generators, extending `Arbitrary1` to be depth-aware as follows:

```
class BArbitrary1 (f :: ∗ → ∗) where
  liftBGen :: BGen a → BGen (f a)
```

Note the similarities between `Arbitrary1` and `BArbitrary1`. We will use this type class to implement random generators for each construction we are automatically deriving. Recalling our `Html` example, we can define modular random generators for the constructions representing its data constructors as follows:

```
instance BArbitrary1 Con_Text where
  liftBGen bgen d = Mk_Text ⟨$⟩ arbitrary
instance BArbitrary1 Con_Sing where
  liftBGen bgen d = Mk_Sing ⟨$⟩ arbitrary
instance BArbitrary1 Con_Tag where
  liftBGen bgen d = Mk_Tag ⟨$⟩ arbitrary ⟨∗⟩ bgen (d-1)
instance BArbitrary1 Con_(:+:) where
  liftBGen bgen d = Mk_(:+:) ⟨$⟩ bgen (d-1) ⟨∗⟩ bgen (d-1)
```

Note how each instance is defined to be parametric of the maximum depth (using the input integer `d`) and of the random generator used for the recursive sub-terms (using the input generator `bgen`). Every other non-recursive sub-term can be generated using a normal `Arbitrary` instance—we use this to generate random `Strings` in the previous definitions.

The rest of our representations can be generated analogously. For example, the `BArbitrary1` instances for $\texttt{Fun}_{\texttt{bold}}$ and $\texttt{Pat}_{\texttt{simplify\#2}}$ are as follows:

```
instance BArbitrary1 Fun_bold where
  liftBGen bgen d = Mk_bold ⟨$⟩ bgen (d-1)
instance BArbitrary1 Pat_simplify#2 where
  liftBGen bgen d =
    Mk_simplify#2 ⟨$⟩ arbitrary ⟨∗⟩ bgen (d-1) ⟨∗⟩ bgen (d-1)
```

Then, having the modular generators for each random construction in place, we can obtain a concrete depth-aware generator (of kind $*$) for any final representation `Fix f` as follows:

```
instance BArbitrary1 f ⇒ BArbitrary (Fix f) where
  barbitrary d = Fix ⟨$⟩ liftBGen barbitrary d
```

There, we use the `BArbitrary1` instance of our representation `f` to generate sub-terms recursively by lifting itself as the parameterized input generator (`liftBGen barbitrary`), wrapping each recursive sub-term with a `Fix` data constructor.

The machinery developed so far lets us generate single random constructions in a modular fashion. However, we still need to develop our generation mechanism a bit further in order to generate composite representations built using the ⊕ operator. This is the objective of the next sub-section.

## 4.2 Encoding Generation Behavior Using Types

As we have seen so far, generating each representation is rather straightforward: there is only one data constructor to pick, and every field is generated using a mechanical recipe. In our approach, most of the generation complexity is encoded in the random generator for composite representations, built upon the ⊕ operator. Before introducing it, we need to define some additional machinery to encode the notions of terminal construction and generation frequency.

Recalling the random generator for `Html` presented in Section 2, we can observe that the last generation level (see `gen 0`) is constrained to generate values only from the subset of terminal constructions. In order to model this behavior, we will first define a data type `Term` to tag every terminal construction explicitly:

```
data Term (f :: ∗ → ∗) r = Term (f r)
```

Then, if `f` is a terminal construction, the type `Term f ⊕ g` can be interpreted as representing data generated using values drawn both from `f` and `g`, but closed using only values from `f`. Since this data type will not add any semantic information to the represented values, we can define suitable `Algebra` and `BArbitrary1` instances for it simply by delegating the work to the inner type:

```
instance Algebra f a ⇒ Algebra (Term f) a where
  alg (Term f) = alg f
instance BArbitrary1 f ⇒ BArbitrary1 (Term f) where
  liftBGen bgen d = Term ⟨$⟩ liftBGen bgen d
```

Worth mentioning, our approach does not requires the final user to manually specify terminal constructions—a repetitive task which might lead to obscure non-termination errors if a recursive construction is wrongly tagged as terminal. In turn, this information can be easily extracted at derivation time and included implicitly in our refined type-level idiom, described in detail in Section 5.

The next building block of our framework consists in a way of specifying the generation frequency of each construction. For this purpose, we can follow the same reasoning as before, defining a type-level operator ⊗ to explicitly tag the generation frequency of a given representation:

```
data ((f :: ∗ → ∗) ⊗ (n :: Nat)) r = Freq (f r)
```

This operator is parameterized by a type-level natural number `n` (of kind `Nat`) representing the desired generation frequency. In this light, the type `(f⊗3)⊕(g⊗1)` represents data generated using values from both `f` and `g`, where `f` is randomly chosen three times more frequently than `g`. In practice, we defined ⊗ such that it associates more strongly than ⊕, thus avoiding the need of parenthesis in types like the previous one. Analogously as `Term`, the operator ⊗ does not add any semantic information to the values it represents, so we can define its `Algebra` and `BAbitrary1` instance by delegating the work to the inner type as before:

```
instance Algebra f a ⇒ Algebra (f ⊗ n) a where
  alg (Freq f) = alg f
instance BArbitrary1 f ⇒ BArbitrary1 (f ⊗ n) where
  liftBGen bgen d = Freq ⟨$⟩ liftBGen bgen d
```

With these two new type level combinators, `Term` and ⊗, we are now able to express the behavior of our entire generation process based solely on the type we are generating.

In addition to these combinators, we will need to perform some type-level computations based on them in order to define our random generator for composite representations. Consider for instance the following type—expressed using parenthesis for clarity:

```
(f ⊗ 2) ⊕ ((g ⊗ 3) ⊕ (Term h ⊗ 5))
```

Our generation process will traverse this type one combinator at a time, processing each occurrence of ⊕ independently. This means that, in order to select the appropriate generation frequency of each operand we need to calculate the overall sum of frequencies on each side of the ⊕. For this purpose, we rely on Haskell's type-level programming feature known as *type families* [23]. In this light, we can implement a type-level function `FreqOf` to compute the overall sum of frequencies of a given representation type:

```
type family FreqOf (f :: ∗ → ∗) :: Nat where
  FreqOf (f ⊕ g)    = FreqOf f + FreqOf g
  FreqOf (f ⊗ n)    = n ∗ FreqOf f
  FreqOf (Term f)   = FreqOf f
  FreqOf _          = 1
```

This type-level function takes a representation type as an input and traverses it recursively, adding up each frequency tag found in the process, and returning a type-level natural number. Note how in the second equation we multiply the frequency encoded in the ⊗ tag with the frequency of the type it is wrapping. This way, the type `((f ⊗ 2) ⊕ g) ⊗ 3` is equivalent to `(f ⊗ 6) ⊕ (g ⊗ 3)`, following the natural intuition for the addition and multiplication operations over natural numbers. Moreover, if a type does not have an explicit frequency, then its generation frequency is defaulted to one.

Furthermore, the last step of our generation process, which only generates terminal constructions, could be seen as considering the non-terminal ones as having generation frequency zero. This way, we can introduce another type-level computation to calculate the *terminal generation frequency* $\texttt{FreqOf}^{\mathsf{T}}$ of a given representation:

```
type family FreqOfᵀ (f :: ∗ → ∗) :: Nat where
  FreqOfᵀ (f ⊕ g)    = FreqOfᵀ f + FreqOfᵀ g
  FreqOfᵀ (f ⊗ n)    = n ∗ FreqOfᵀ f
  FreqOfᵀ (Term f)   = FreqOf f
  FreqOfᵀ _          = 0
```

Similar to `FreqOf`, the type family above traverses its input type adding the terminal frequency of each sub-type. However, $\texttt{FreqOf}^{\mathsf{T}}$ only considers the frequency of those representation sub-types that are explicitly tagged as terminal, returning zero in any other case.

Then, using the `Term` and ⊗ combinators introduced at the beginning of this sub-section, along with the previous type-level computations over frequencies, we are finally in position of defining our random generator for composite representations:

```
instance (BArbitrary1 f, BArbitrary1 g)
   ⇒ BArbitrary1 (f ⊕ g) where
  liftBGen bgen d =
    if d > 0
    then frequency
      [(freqVal @(FreqOf f), Inₗ ⟨$⟩ liftBGen bgen d)
      ,(freqVal @(FreqOf g), Inᵣ ⟨$⟩ liftBGen bgen d)]
    else frequency
      [(freqVal @(FreqOfᵀ f), Inₗ ⟨$⟩ liftBGen bgen d)
      ,(freqVal @(FreqOfᵀ g), Inᵣ ⟨$⟩ liftBGen bgen d)]
```

Like the generator for `Html` introduced in Section 2, this generator branches over the current depth `d`. In the case we can still generate values from any construction (`d > 0`), we will use *QuickCheck*'s `frequency` operation to randomly choose between generating a value of each side of the ⊕, i.e., either a value of `f` or a value of `g`, following the generation frequencies specified for both of them, and wrapping the values with the appropriate injection $\texttt{In}_{\mathsf{L}}$ or $\texttt{In}_{\mathsf{R}}$ on each case. Such frequencies are obtained by *reflecting* the type-level natural values obtained from applying `FreqOf` to both `f` and `g`, using a type-dependent function `freqVal` that returns the number corresponding to the type-level natural value we apply to it:

```
freqVal :: ∀n . KnownNat n ⇒ Int
```

Note that the type of `freqVal` is ambiguous, since it quantifies over every possible known type-level natural value `n`. We use a *visible type application* [7] (employing the `@(...)` syntax) to disambiguate to which natural value we are actually referring to. Then, for instance, the value `freqVal @(FreqOf (f ⊗ 5 ⊕ g ⊗ 4))` evaluates to the concrete value `9 :: Int`.

The **else** clause of our random generator works analogously, except that, this time we only want to generate terminal constructions, hence we use the $\texttt{FreqOf}^{\mathsf{T}}$ type family to compute the terminal generation frequency of each operand. If any of $\texttt{FreqOf}^{\mathsf{T}}$ `f` or $\texttt{FreqOf}^{\mathsf{T}}$ `g` evaluates to zero, it means that such operand does not contain any terminal constructions, and `frequency` will not consider it when generating terminal values.

Moreover, if it happens that both $\texttt{FreqOf}^{\mathsf{T}}$ `f` and $\texttt{FreqOf}^{\mathsf{T}}$ `g` compute to zero simultaneously, then this will produce a runtime error triggered by the function `frequency`, as it does not have anything with a positive frequency to generate. This kind of exceptions will arise, for example, if we forget to include at least one terminal construction in our final representation—thus leaving the door open for potential infinite generation loops. Fortunately, such runtime exceptions can be caught at compile time. We can define a type constraint `Safe` that ensures we are trying to generate values using a representation with a strictly positive terminal generation frequency—thus containing at least a single terminal construction:

```
type family Safe (f :: * → *) :: Constraint where
  Safe f = IsPositive (FreqOfᵀ f)

type family IsPositive (n :: Nat) :: Constraint where
  IsPositive 0 = TypeError "No terminals"
  IsPositive _ = ()
```

These type families compute the terminal generation frequency of a representation type `f`, returning either a type error, if its result is zero; or, alternatively, an empty constraint () that is always trivially satisfied. Finally, we can use this constraint to define a safe generation primitive `genRep` to obtain a concrete depth-bounded generator for every target type `a`, specified using a "safe" representation `f`:

```
genRep :: ∀f a . (BArbitrary1 f, Safe f, Algebra f a) ⇒ BGen a
genRep d = eval ⟨$⟩ barbitrary @(Fix f) d
```

Note how this primitive is also ambiguous in the type used for the representation. This allows us to use a visible type application to obtain values from the same target type but generated using different underlying representations. For instance, we can obtain two different concrete generators of our `Html` type simply by changing its generation representation type as follows:

```
genHtml_valid :: BGen Html
genHtml_valid = genRep @Html_valid

genHtml_simplify :: BGen Html
genHtml_simplify = genRep @Html_simplify
```

where $\texttt{Html}_{\texttt{valid}}$ and $\texttt{Html}_{\texttt{simplify}}$ are the representations types introduced in Figure 1b—the syntax used to define them is completed in the next section.

So far we have seen how to represent and generate values for our target data type by combining different random constructions, as well as a series of type-level combinators to encode the desired generation behavior. The next section refines our type-level machinery in order to provide a simple idiom for defining composable random generators.

## 5 TYPE-LEVEL GENERATION SPECIFICATIONS

This section introduces refinements to our basic language for describing random generators, making it more flexible and robust in order to fit real-world usage scenarios.

The first problem we face is that of naming conventions. In practice, the actual name used when deriving the representation for each random construction needs to be generated such that it complies with Haskell's syntax, and also that it is *unique within our namespace.* This means that, type names like $\texttt{Fun}_{\langle + \rangle}$ or $\texttt{Pat}_{\texttt{simplify\#1}}$ are, technically, not valid Haskell data type names, thus they will have to be synthesized as something like `Fun_lt_plus_gt_543` and `Pat_simplify_1_325`, where the last sequence of numbers is inserted by Template Haskell to ensure uniqueness.

This naming convention results hard to use, specially if we consider that we do not know the actual type names until they are synthesized during compilation, due to their unique suffixes. Fortunately, it is easy to solve this problem using some type-level machinery. Instead of imposing a naming convention in our derivation tool, we define a set of open type families to hide each kind of construction behind meaningful names:

```
type family Con (c :: Symbol)
type family Fun (f :: Symbol)
type family Pat (p :: Symbol) (n :: Nat)
```

where `Symbol` is the kind of type-level strings in Haskell. Then, our derivation process will synthesize each representation using unique names, along with a type instance of the corresponding type family, i.e., `Con` for data constructors, `Fun` for interface functions, and `Pat` for functions' patterns. For instance, along with the constructions representations $\texttt{Con}_{\texttt{Text}}$, $\texttt{Fun}_{\langle + \rangle}$ and $\texttt{Pat}_{\texttt{simplify\#1}}$, we will automatically derive the following type instances:

```
type instance Con "Text"         = Term Con_Text_123
type instance Fun "<+>"          = Fun_lt_plus_gt_543
type instance Pat "simplify" 1 = Term Pat_simplify_1_325
```

As a result, the end user can simply refer to each particular construction by using these synonyms, e.g., with representation types like `Con "Text" ⊕ Fun "<+>"`. The additional `Nat` type parameter on `Pat` simply identifies each pattern number uniquely.

Moreover, notice how we include the appropriate `Term` tags for each terminal construction automatically—namely `Con "Text"` and `Pat "simplify" 1` in the example above. Since this information is statically available, we can easily extract it during derivation time. This relieves us of the burden of manually identifying and declaring the terminal constructions for every generation specification Additionally, it helps ensuring the static termination guarantees provided by our `Safe` constraint mechanism.

Using the type-level extension presented so far, we are now able to write the generation specifications presented in Figure 1b in a clear and concise way.

### 5.1 Parametric Target Data Types

So far we have seen how to specify random generators for our simple self-contained `Html` data type. In practice, however, we are often required to write random generators for parametric target data types as well. Consider, for example, the following `Tree` data type definition encoding binary trees with generic information of type `a` in the leaves:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

In order to represent its data constructors, we can follow the same recipe presented in Section 3, but also parameterizing our representations over the type variable `a` as well:

```
data Con_Leaf a r = Mk_Leaf a
data Con_Node a r = Mk_Node r r
```

The rest of the machinery can be derived in the same way as before, carrying this type parameter and including the appropriate `Arbitrary` constraints all along the way:

```
instance Algebra (Con_Leaf a) (Tree a) where ···
instance Algebra (Con_Node a) (Tree a) where ···
instance Arbitrary a ⇒ BArbitrary1 (Con_Leaf a) where ···
instance Arbitrary a ⇒ BArbitrary1 (Con_Node a) where ···
```

Then, instead of carrying this type parameter in our generation specifications, we can avoid it by hiding it behind an existential type:

```
data Some (f :: ∗ → ∗ → ∗) (r :: ∗) = ∀(a :: ∗) . Some (f a r)
```

The type constructor `Some` is a wrapper for a 2-parametric type that hides the first type variable using an explicit existential quantifier. Note thus that the type parameter `a` does not appears at the left hand side of `Some` on its definition. In this light, when deriving any `Con`, `Fun` or `Pat` type instance, we can use this type wrapper it to hide the additional type parameters of each construction representation:

```
type instance Con "Leaf" = Term (Some Con_Leaf)
type instance Con "Node" = Some Con_Node
```

As a consequence, we can write generation specifications for our `Tree` data type without having to refer to its type parameter anywhere. For instance:

```
type Tree_Spec =   Con "Leaf" ⊗ 2
                 ⊕ Con "Node" ⊗ 3
```

Instead, we defer handling this type parameter until we actually use it to define a concrete generator. For instance, we can write a concrete generator of `Tree Int` as follows:

```
genIntTree :: BGen (Tree Int)
genIntTree = genRep @(Tree_Spec ◁ Int)
```

Where ◁ is a type family that simply traverses our generation specification, applying the `Int` type to each occurrence of `Some`, thus eliminating the existential type:

```
type family (f :: ∗ → ∗) ◁ (a :: ∗) :: ∗ → ∗ where
  (Some f) ◁ a = f a
  (f ⊕ g)  ◁ a = (f ◁ a) ⊕ (g ◁ a)
  (f ⊗ n)  ◁ a = (f ◁ a) ⊗ n
  (Term f) ◁ a = Term (t ◁ a)
  f        ◁ a = f
```

As a result, in `genIntTree`, the ◁ operator will reduce the type (`Tree_Spec ◁ Int`) to the following concrete type:

```
(Term (Con_Leaf Int) ⊗ 2) ⊕ ((Con_Node Int) ⊗ 3)
```

Worth mentioning, this approach for handling parametric types can be extended to multi-parametric data types with minor effort.

Along with our automated constructions derivation mechanism, the machinery introduced in this section allows us to specify random generators using a simple type-level specification language.

The next section evaluates our approach in terms of performance using a set of case studies extracted from real-world Haskell implementations, along with an interesting runtime optimization.

## 6 BENCHMARKS AND OPTIMIZATIONS

The random generation framework presented throughout this paper allows us to write extensible generators in a very concise way. However, this expressiveness comes attached to a perceptible runtime overhead, primarily inherited from the use of Data Types à la Carte—a technique which is not often scrutinized for performance. In this section, we evaluate the implicit cost of composing generators using three real-world case studies, along with a type-level optimization that helps avoiding much of the runtime bureaucracy.

*Balanced Representations.* As we have shown in Section 4, the random generation process we propose in this paper can be seen as having two phases. First, we generate random values from the representation types used to specify the shape of our data; and then we use their algebras to translate them to the corresponding values of our target data types. In particular, this last step is expected to pattern match repeatedly against the $\mathtt{In}_L$ and $\mathtt{In}_R$ constructors of the ⊕ operators when traversing each construction injection. Because of this, in general, we expect a performance impact with respect to manually-written concrete generators.

As recently analyzed by Kiriyama et al., this slowdown is expected to be linear in the depth of our representation type [12]. In this light, one can drastically reduce the runtime overhead by associating each ⊕ operator in a balanced fashion. So, for instance, instead of writing (`f⊕g⊕h⊕i`), which is implicitly parsed as (`f⊕(g⊕(h⊕i))`); we can associate constructions as ((`f` ⊕ `g`) ⊕ (`h` ⊕ `i`)), thus reducing the depth of our representation from four to three levels and, in general, from a $O(n)$ to a $O(log(n))$ complexity in the runtime overhead, where $n$ is the amount of constructions under consideration.

Worth mentioning, this balancing optimization cannot be applied to the original fashion of Data Types à la Carte by Swierstra. This limitation comes from that the linearity of the representation types is required in order to define *smart injections*, allowing users to construct values of such types in an easy way, injecting the appropriate sequences of $\mathtt{In}_L$ and $\mathtt{In}_R$ constructors automatically. There, a naïve attempt to use smart injections in a balanced representation may fail due to the nature of Haskell's type checker, and in particular on the lack of backtracking when solving type-class constraints. Fortunately, smart injections are not required for our purposes, as users are not expected to construct values by hand at any point—they are randomly constructed by our generators.

*Benchmarks.* We analyzed the performance of generating random values using three case studies: (i) Red-Black Trees (RBT), inspired by Okasaki's formulation [19], (ii) Lisp S-expressions (SExp),

| Case Study | `#Con` | `#Fun` | `#Pat` | Total Constructions |
|---|---|---|---|---|
| RBT | 2 | 5 | 6 | 13 |
| SExp | 6 | - | 9 | 15 |
| HTML | 4 | 132 | - | 136 |

**Table 1: Overview of the size of our case studies.**

inspired by the package *hs-zuramaru*[3], and (iii) HTML expressions (HTML), inspired by the *html* package, which follows the same structure as our motivating `Html` example. The magnitude of each case study can be outlined as shown in Table 1.

These case studies provide a good combination of data constructors, interface functions and patterns, and cover from smaller to larger numbers of constructions.

Then, we benchmarked the execution time required to generate and fully evaluate 10000 random values corresponding to each case study, comparing both manually-written concrete generators, and those obtained using our modular approach. For this purpose, we used the *Criterion* [20] benchmarking tool for Haskell, and limited the maximum depth of the generated values to five levels. Additionally, our modular generators were tested using both linear and balanced generation specifications. Figure 2 illustrates the relative execution time of each case study, normalized to their corresponding manually-written counterpart—we encourage the reader to obtain a colored version of this work.

As it can be observed, our approach suffers from a noticeable runtime overhead when using linearly defined representations, specially when considering the HTML case study, involving a large number of constructions in the generation process. However, we found that, by balancing our representation types, the generation performance improves dramatically. At the light of these improvements, *our tool includes an additional type-level computation that automatically balances our representations* in order to reduce the generation overhead as much as possible.

On the other hand, it has been argued that the generation time is often not substantial with respect to the rest of the testing process, especially when testing complex properties over monadic code, as well as using random values for penetration testing [9, 18].

[3]http://hackage.haskell.org/package/zuramaru

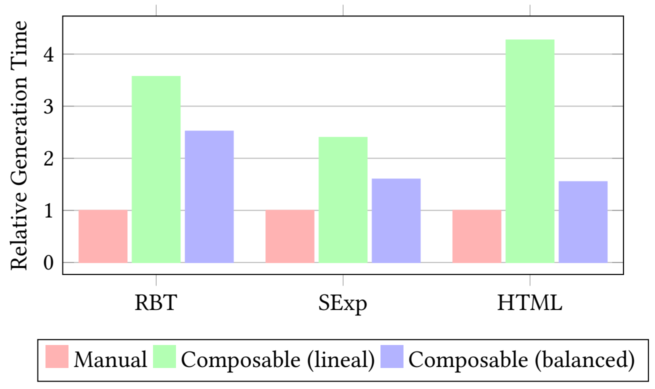

**Figure 2: Generation time comparison between manually written and automatically derived composable generators.**

All in all, we consider that these results are fairly encouraging, given that the flexibility obtained from using our compositional approach does not produce severe slowdowns when generating random values in practice.

## 7 RELATED WORK

*Extensible Data Types.* Swierstra proposed Data Types à la Carte [24], a technique for building extensible data types, as a solution for the *expression problem* coined by Wadler [25]. This technique has been successfully applied in a variety of scenarios, from extensible compilers, to composable machine-mechanized proofs [4, 5, 21, 27]. In this work, we take ideas from this approach and extend them to work in the scope of random data generation, where other parameters come into play apart from just combining constructions, e.g., generation frequency and terminal constructions.

From the practical point of view, Kiriyama et al. propose an optimization mechanism for Data Types à la Carte, where a concrete data type has to be derived for each different composition of constructions defined by the user [12]. This solution avoids much of the runtime overhead introduced when internally pattern matching against sequences of $\texttt{In}_{\texttt{L}}$ and $\texttt{In}_{\texttt{R}}$ data constructors. However, this approach is not entirely compositional, as we still need to rely on Template Haskell to derive the machinery for *each* specialized instance of our data type. In our particular setting, we found that our solution has a fairly acceptable overhead, achieved by automatically balancing our representation types.

*Domain Specific Languages.* Testing properties using small values first is a good practice, both for performance and for obtaining small counterexamples. In this light, *SmallCheck* [22] is a library for defining *exhaustive* generators inspired by *QuickCheck*. Such generators can be used to test properties against *all* possible values of a data type of up to a given depth. The authors also present *Lazy SmallCheck*, a variation of *SmallCheck* prepared to use partially defined inputs to explore large parts of the search space at once.

*Luck* [14] is a domain-specific language for describing testing properties and random generators in parallel. It allows obtaining random generators producing highly constrained random data by using a mixture of backtracking and constraint solving while generating values. While this approach can lead to quite good testing results, it still requires users to manually think about how to generate their random data. Moreover, the generators obtained are not compiled, but interpreted. In consequence, *Luck*'s generators are rather slow, typically around 20 times slower than compiled ones.

In contrast to these tools, this work lies on the automated side, where we are able to provide lightweight invariants over our random data by following the structural information extracted from the users' codebase.

*Automatic Derivation Tools.* In the past few years, there has been a bloom of automated tools for helping the process of writing random generators.

*MegaDeTH* [8, 9] is a simple derivation tool that synthesizes generators solely based on their types, paying no attention whatsoever to the generation frequency of each data constructor. As a result, it has been shown that its synthesized generators are biased towards generating very small values [18].

*Feat* [6] provides a mechanism to uniformly generating values from a given data type of up to a given size. It works by enumerating all the possible values of such type, so that sampling uniformly from it simply becomes sampling uniformly from a finite prefix of natural numbers—something easy to do. This tool has been shown to be useful for generating unbiased random values, as they are drawn uniformly from their value space. However, sampling uniformly may not be ideal in some scenarios, specially when our data types are too general, e.g., using *Feat* to generate valid HTML values as in our previous examples would be quite ineffective, as values drawn uniformly from the value space of our `Html` data type represent, in most cases, invalid HTML values.

On the other hand, *DRAGEN* is a tool that synthesizes optimized generators, tuning their generation frequencies using a simulation-based optimization process, which is parameterized by the distribution of values desired by the user [18]. This simulation is based on the theory of *branching processes*, which models the growth and extinction of populations across successive generations. In this setting, populations consist of randomly generated data constructors, where generations correspond to each level of the generated values. This tool has shown to improve the code coverage over complex systems, when compared to other automated generators derivation tools.

In a recent work, we extended this approach to generate random values considering also the other sources of structural information covered here, namely abstract interfaces and function pattern matchings [17]. There, we focus on the generation model problem, extending the theory of branching processes to obtain sound predictions about distributions of random values considering these new kinds of constructions. Using this extension, we shown that using extra information when generating random values can be extremely valuable, in particular under situations like the ones described in Section 2, where the usual derivation approaches fail to synthesize useful generators due to a lack of structural information. In turn, this paper tackles the representation problem, exploring how a compositional generation process can be effectively implemented and automated in Haskell using advanced type-level features.

In the light of that none of the aforementioned automated derivation tools are designed for composability, we consider that the ideas presented in this paper could perhaps be applied to improve the state-of-the-art in automatic derivation of random generators in the future.

## 8 CONCLUSIONS

We presented a novel approach for automatically deriving flexible composable random generators inspired by the seminal work on Data Types à la Carte. In addition, we incorporate valuable structural information into our generation process by considering not only data constructors, but also the structural information statically available in abstract interfaces and functions' pattern matchings.

In the future, we aim to extend our mechanism for obtaining random generators with the ability of performing stateful generation. In this light, a user could indicate which random constructions interact with their environment, obtaining random generators ensuring strong invariants like well scopedness or type correctness, all this while keeping the derivation process as automatic as possible.

## ACKNOWLEDGMENTS

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

## REFERENCES

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

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

[3] Koen Claessen and John Hughes. 2002. Testing Monadic Code with QuickCheck. *SIGPLAN Not.* 37, 12 (2002), 47–59. https://doi.org/10.1145/636517.636527

[4] L. E. Day and G. Hutton. 2014. Compilation À La Carte. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages (IFL '13)*.

[5] Benjamin Delaware, Bruno C d S Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 207–218.

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

[7] Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. 2016. Visible type application. In *European Symposium on Programming*. Springer, 229–254.

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

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

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

[11] J. Hughes, U. Norell, N. Smallbone, and T. Arts. 2016. Find more bugs with QuickCheck!. In *The IEEE/ACM International Workshop on Automation of Software Test (AST)*.

[12] H. Kiriyama, H. Aotani, and H. Masuhara. 2016. A Lightweight Optimization Technique for Data Types a La Carte. In *Companion Proceedings of the 15th Int. Conference on Modularity (MODULARITY 2016)*. ACM, New York, NY, USA.

[13] Casey Klein and Robert Bruce Findler. 2009. Randomized testing in PLT Redex. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*.

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

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

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

[17] Agustín Mista and Alejandro Russo. 2019. Generating Random Structurally Rich Algebraic Data Type Values. In *Proceedings of the 14th International Workshop on Automation of Software Test*.

[18] Agustín Mista, Alejandro Russo, and John Hughes. 2018. Branching processes for QuickCheck generators. In *Proc. of the ACM SIGPLAN Int. Symp. on Haskell*.

[19] Chris Okasaki. 1999. Red-black trees in a functional setting. *Journal of functional programming* 9, 4 (1999), 471–477.

[20] Bryan O'Sullivan. 2014. Criterion: a Haskell microbenchmarking library. http://www.serpentine.com/criterion/

[21] Anders Persson, Emil Axelsson, and Josef Svenningsson. 2011. Generic monadic constructs for embedded languages. In *International Symposium on Implementation and Application of Functional Languages*. Springer, 85–99.

[22] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, Vol. 44. ACM, 37–48.

[23] T. Schrijvers, M. Sulzmann, S. P. Jones, and M. Chakravarty. 2007. Towards open type functions for Haskell. In *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*.

[24] Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008), 423–436. https://doi.org/10.1017/S0956796808006758

[25] Philip Wadler. 1998. The expression problem. https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt

[26] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. 60–76.

[27] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. (2014).