Differential Testing using Random Well-Typed Haskell Programs

Samuel da Silva Feitosa
samuel.feitosa[at]uffs.edu.br
Universidade Federal da Fronteira Sul
Chapecó, SC, Brazil

Rodrigo Geraldo Ribeiro
rodrigo.ribeiro[at]ufop.edu.br
Universidade Federal de Ouro Preto
Ouro Preto, MG, Brazil

ABSTRACT

Haskell is a statically-typed, purely functional programming language with a strong academic foundation, widely recognized for its application in both research and industry for developing robust, high-assurance software. Nowadays, it is being adopted for a variety of projects, where applications reach a level of complexity where manual testing and human inspection are insufficient to ensure quality in software development. Even in the presence of automated unit testing, such methodologies infrequently encompass the entirety of significant code scenarios, which means that certain defects may remain undetected, particularly when the code is subjected to an identical set of validation criteria repetitively. Considering this context, this paper describes a type system guided algorithm to generate random and well-typed Haskell programs, which can be used for testing the compiler, development tools, and libraries.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Code generation, Fuzzing, Haskell compiler, Differential Testing.

1 INTRODUCTION

Nowadays, Haskell stands as a distinguished language, renowned for its emphasis on pure functional programming. It is a statically typed, purely functional language with type inference and lazy evaluation. Developed in 1990, Haskell has been subject to continuous development, with the Haskell community actively enhancing its capabilities through successive versions [14]. A notable release introduced advanced features such as type classes, pattern matching, and list comprehensions, which enrich Haskell's functional paradigm by allowing more expressive and concise code [7].

With the adoption of the Haskell language in large-scale projects [24], it is important to have mechanisms to verify that the compiler is working as expected. Therefore, due to its complexity, checking whether a compiler is working properly often surpasses the effectiveness of traditional testing, code reviews, and manual inspection. This trend suggests the need for tools to exhaustively examine all potential actions, assisting to find and fix the compiler on the presence of unexpected behaviors. Given Haskell's strong static type system and emphasis on pure functions, these tools are particularly adept at verifying correctness in Haskell code [23]. However, the task of formalizing and proving properties for a whole language is a difficult and time-consuming activity. Considering this fact, a lightweight approach which combines properties and testing can be applied to improve the quality of programming language tools.

The construction of test suites for programming languages and compilers presents a significant challenge, since several criteria must be adhered to in order to forge a test case that is both valid and useful [1, 3]. The human factor in this process often results in a constrained scope of creativity, with the potential for presuppositions regarding the implementation that may affect the quality of the test cases. Furthermore, the ongoing evolution of the language brings additional complexities in the maintenance of these tests. Consequently, there is a growing research community interested within the domain of random test generation, a.k.a. programming language fuzzing. This area of research, however, has its own set of challenges, as it necessitates the generation of programs that comply with the intricate constraints imposed by the programming language's compiler, such as the adherence to syntactical correctness, the fulfillment of type-system requirements inherent to statically-typed languages, and that avoid non-termination and undefined behaviors.

In this context, we investigate the type-directed method [2, 8, 21] to fuzz the programming language, which operates by applying rules from the type system in a bottom-up, goal-oriented way. We implement a generator capable of generating type-correct programs containing a considerable subset of the Haskell language, including functions, lambda expressions, algebraic data types, and pattern matching, as well as basic operations. This means that the generator respects the rules of the type system and considers the context in which individual expressions are generated. To measure the quality of our generator, we check the code coverage. Furthermore, to evaluate the capabilities of the generator, we perform differential testing with a Haskell compiler.

More specifically, this paper presents the following contributions:

- We provided a type-directed algorithm for constructing random programs for a large subset of Haskell, including ADTs and pattern matching. We argue that the algorithm is sound with respect to the Haskell type system, i.e., it generates only well-typed programs.
- We used a lightweight manner to check the Haskell compiler by applying differential testing using the generated programs.
- We analyzed the execution of 10,000 tests against the compilation and behavior preservation properties of Haskell programs using the GHC compiler with different optimization levels.

The remainder of this text is organized as follows: Section 2 summarizes the Haskell subset considered by the generation method. Section 3 presents the process of generating well-typed random programs in the context of the subset of Haskell. Section 4 shows the results of differential testing to check the compilation and behavior preservation properties for each generated program to attest

conformance. Section 5 discusses related works. Finally, we present the final remarks in Section 6.

2 THE SUBSET OF HASKELL

Haskell is as a language dedicated to Functional Programming (FP), which was created aiming to be suitable for industry and academic purposes, where its features could be extended for research. It was named after Haskell B. Curry, a logician known by his contributions to the field of combinatorial logic and lambda calculus [13]. During the last years, Haskell has been used as a type-system laboratory, assisting on the study of many advanced ideas regarding programming language features.

The standard implementation for Haskell is the Glasgow Haskell Compiler (GHC) [15]. The type system within GHC is referred to as System FC, which is a typed lambda calculus developed from the foundational System F lambda calculus [12]. However, there exist several other active projects that implement the Haskell language.

Next we present the Haskell subset considered for our generation algorithm, showing first the syntax that should be respected and then the typing rules needed for guiding the generation process.

2.1 Syntax

The abstract syntax for the considered subset of Haskell is given in Figure 1, where T represents type declarations, M and I denote modules and imports, D defines declarations, P stands for patterns, l is used for literal values, and e refers to the allowed expressions. The meta-variables N and K are used to represent type and constructor names, x and f are used for variable and function names. Throughout this paper, we write $\overline{T}$ as shorthand for a possibly empty sequence $T_1, ..., T_n$ (similarly for $\overline{P}$, $\overline{e}$, etc.). Following the common practice, we let the meta-variable Γ denote an arbitrary typing environment, which consists of a finite mapping between variables and functions, and their respective types. We also let the meta-variable Δ represent an environment to store user-defined types. Sequences of declarations and parameter names are assumed to contain no duplicate names. Similarly, for simplicity, we assume that there are no duplicate variable names in the same scope to avoid shadowing and variable capture.

On this subset, we have primitive types ($\mathbb{I}$, $\mathbb{R}$, $\mathbb{B}$, $\mathbb{S}$, $\mathbb{C}$ standing for integer, float, boolean, string and char), function, tuple and list types, and user-defined types, which can be build by Algebraic Data Types (ADTs) or by aliases. Note that the syntax doesn't include polymorphic types. A module is defined by a name, a list of imported modules and a list of declarations. If the module contains a *main* function, it can be executed directly. A declaration allows the user to create ADTs with a name N, a sequence of constructors K, and a list of types, type aliases providing a new name to an existing type, and (typed) functions with a sequence of equations having patterns and expressions. A pattern consists of variables, wildcards, ADT constructors, literal values, tuples, empty and cons lists. An expression allows the use of variables, functions, lambda abstractions, function applications, let bindings, pattern matching, conditionals, tuples, lists and ADT constructors. The presented syntax represents a substantial subset of the Haskell language.

Syntax

T ::=	types
$\mathbb{I} \mid \mathbb{R} \mid \mathbb{B} \mid \mathbb{S} \mid \mathbb{C}$	primitive
$\overline{T} \to T$	function
$(\overline{T})$	tuple
$[T]$	list
$N :: \overline{K\ \overline{T}}$	algebraic
N	synonym
M ::=	modules
module $n\ \overline{I}\ \overline{D}$	
I ::=	imports
import n	
D ::=	declarations
data $N = \overline{K\ \overline{T}}$	ADTs
type $N = T$	aliases
$f\ ::\ \overline{T} \to T \triangleright \overline{f\ \overline{P}\ =\ e}$	functions
P ::=	patterns
$x \mid \_ \mid K\ \overline{P} \mid l \mid (\overline{P}) \mid [] \mid [P : P]$	
e ::=	expressions
x	variable
f	function
$\lambda P :: \overline{T} \to e$	abstraction
$e\ \overline{e}$	application
let $x\ =\ e$ in e	binding
case e of $\overline{P\ \leftarrow\ e}$	pattern matching
if e then e *else* e	conditional
$(\overline{e})$	tuple
$[\overline{e}]$	list
$K\ \overline{e}$	constructor

Figure 1: Syntax considered for the selected subset of Haskell.

2.2 Type System

Since our generation approach is guided by the type system, we present the typing rules for declarations in Figure 2 and for expressions in Figure 3. For short, we omit the rules for trivial syntactical constructors. In our definitions, for simplicity, we consider that the rules for declarations D extend the Δ context, storing user-defined types (ADTs and aliases), and also extend the Γ context, inserting typing information about the functions. Rule T-Adt extends the Δ context adding each constructor K with its corresponding type $\overline{T} \to N$. Rule T-Alias just adds a new name N for a given type T in the context Δ. And the rule T-Function extends the Γ context adding the function f with its defined type $\overline{T} \to T$. Since we are considering that an equation can be defined using pattern matching constructors, we use a function *var-types*, which extracts the variables and types from a given pattern.

The typing rules for expressions are mostly standard [22]. We use the typing judgment with the form $\Delta; \Gamma \vdash e : T$, meaning that in the environments Δ and Γ, expression e has type T. The rule T-Var and T-Fun obtain the respective type from the Γ context. Rule T-Abs is adapted to resemble Haskell's type system, where pattern matching can be applied to parameters. Here we also use

$$\Delta \vdash \text{data } N = \overline{K\ \overline{T}} \mapsto \Delta, \overline{K : \overline{T} \to N} \quad \text{[T-Adt]}$$

$$\Delta \vdash \text{type } N = T \mapsto \Delta, N : T \quad \text{[T-Alias]}$$

$$\frac{\Gamma, \overline{\textit{var-types}(\overline{P}, \overline{T}) \vdash e : T}}{\Gamma \vdash f :: \overline{T} \to T \triangleright \overline{f\ \overline{P} = e} \mapsto \Gamma, f : \overline{T} \to T} \quad \text{[T-Function]}$$

Figure 2: Type system considered for declarations.

the function *var-types* to extract variables and types from a given pattern. The resulting type is a function type $\overline{T} \to T$. Rule T-App is similar to the regular simply-typed lambda calculus, however one can consider multi-parameter function invocation. Rule T-Let is standard. Rule T-Case enforces that each pattern matching $\overline{P}$ is refering to type T_1, and that all expressions $\overline{e}$ return type T_2. Rules T-If, T-Tuple and T-List are also standard. And finally, rule T-Constr defines that the constructor K exists in the Δ context, and that expressions $\overline{e}$ are respecting its type $\overline{T}$.

$$\frac{x : T \in \Gamma}{\Delta; \Gamma \vdash x : T} \quad \text{[T-Var]}$$

$$\frac{f : \overline{T} \to T \in \Gamma}{\Delta; \Gamma \vdash f : \overline{T} \to T} \quad \text{[T-Fun]}$$

$$\frac{\Delta; \Gamma, \overline{\textit{var-types}(\overline{P}, \overline{T})} \vdash e : T}{\Delta; \Gamma \vdash \lambda \overline{P} :: \overline{T} \to e : \overline{T} \to T} \quad \text{[T-Abs]}$$

$$\frac{\Delta; \Gamma \vdash e : \overline{T} \to T \quad \Delta; \Gamma \vdash \overline{e} : \overline{T}}{\Delta; \Gamma \vdash e\ \overline{e} : T} \quad \text{[T-App]}$$

$$\frac{\Delta; \Gamma \vdash e_1 : T_1 \quad \Delta; \Gamma, x : T_1 \vdash e_2 : T_2}{\Delta; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2} \quad \text{[T-Let]}$$

$$\frac{\Delta; \Gamma \vdash e : T_1 \quad \Delta; \Gamma, \overline{\textit{var-types}(P, T_1) \vdash e} : T_2}{\Delta; \Gamma \vdash \text{case } e \text{ of } \overline{P \leftarrow e} : T_2} \quad \text{[T-Case]}$$

$$\frac{\Delta; \Gamma \vdash e : \mathbb{B} \quad \Delta; \Gamma \vdash e_1 : T \quad \Delta; \Gamma \vdash e_2 : T}{\Delta; \Gamma \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : T} \quad \text{[T-If]}$$

$$\frac{\Delta; \Gamma \vdash \overline{e} : \overline{T}}{\Delta; \Gamma \vdash (\overline{e}) : (\overline{T})} \quad \text{[T-Tuple]}$$

$$\frac{\text{for each } i \quad \Delta; \Gamma \vdash e_i : T}{\Delta; \Gamma \vdash [\overline{e}] : [T]} \quad \text{[T-List]}$$

$$\frac{K : \overline{T} \to N \in \Delta \quad \Delta; \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash K\ \overline{e} : T} \quad \text{[T-Constr]}$$

Figure 3: Type system considered for expressions.

With these rules in mind, we propose an algorithm to generate well-typed Haskell programs, which will be presented in the next section.

3 PROGRAM GENERATION

We split the generation of valid (well-typed) Haskell programs in four parts. The first part considers the generation of valid types, which include the primitive and user-defined types. The second shows the generation of patterns, which are used in expressions that allow pattern matching. The third consists in generating declarations, i.e., a set of ADTs, aliases and functions, respecting the language type system, similarly to what a developer would do. For each of the generated declaration signature, we use the Δ and Γ context to store its information. That way, we can use this information when generating new declarations and expressions. The fourth part defines how we generate another expression to be used as the *main* function. Likewise, we can use the previously generated declarations during the expression generation to produce more complex expressions. To allow us to compare the compilation and execution properties of different executions, we compile, run and print the value of this expression on the standard output, and collect the results to be further analyzed. The analysis of differential testing is presented in Section 4.

The following subsections present in detail how types, patterns, declarations, and expressions are randomly generated considering the language constraints, i.e., syntax and typing rules, to produce an executable from the presented subset of Haskell, which consider programs with ADTs, aliases and functions.

3.1 Type Generation

On the presented subset, the type generation is purely syntactical, i.e., there is no extra constraint for generating a valid type. This is valid for this subset, but not for the standard Haskell language, because we are not considering kinds and higher-order types. This means that, to generate a valid type, we can only select any at random. The following definition describes the judgment to generate types.

Definition 1. *Type generation judgment.* $\Delta; \Gamma \overset{t}{\Rightarrow} T$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, a new type T is generated by selecting a valid type at random.

Considering the syntax for types, when a terminal value (primitive or used-defined type) is selected, the result is immediate. However, if a function, tuple, or list type is selected, the judgment should be applied recursively to generate the required syntax.

Example 1. *To generate a function type, the type generation judgment should be used three times.*

$$\frac{\overline{\Delta; \Gamma \overset{t}{\Rightarrow} T_p} \ (2) \qquad \Delta; \Gamma \overset{t}{\Rightarrow} T_r \ (3)}{\Delta; \Gamma \overset{t}{\Rightarrow} \overline{T_p} \to T_r \ (1)}$$

Note that the invocation (1) selected a function type. Then, the invocation (2) was used recursively to generate the types for each parameter for that function. We use a threshold to define the maximum number of parameters allowed by the generator. And for last, the invocation (3) was used once more to generate the return type. Since the generation is recursive, our procedure is not guaranteed

to terminate. To avoid non-termination we decrement a fuel parameter on each recursive call. When this parameter reaches zero, only terminal symbols can be generated, forcing the generation to stop. This approach is applied similarly on the next sections.

3.2 Pattern Generation

The considered subset of Haskell allows us to use pattern matching with expressions according to their types. It allows the use of irrefutable patterns, such as variables and wildcards, and refutable patterns, such as ADT data constructors, literals, tuples, empty and non-empty lists. The following definition describes the judgment to generate patterns.

Definition 2. *Pattern generation judgment.* $\Delta; \Gamma; T \overset{pat}{\Longrightarrow} P$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, a new pattern* P *is generated by selecting a valid pattern at random.*

Taking into account the syntax for patterns, when the irrefutable patterns (x and _), the literal (l), or the empty list ([]) pattern is selected, the result is immediate. But, if the ADT data constructor, tuple, or non-empty list pattern is selected, the judgment has to be applied recursively to generate a valid pattern.

When implementing a pattern generator, we need to consider that there is no guarantee that the generated patterns will be exhaustive. A simple solution is to generate an irrefutable pattern (variable or wildcard) on the last equation. This way, we can be certain that any pattern will match with it.

3.3 Expression Generation

The generation of expressions is goal-oriented, i.e., the generation selects valid syntactical constructors according to a given context, which includes information about user-defined types, free-variables, and a valid type. It is important to mention that, to generate valid expressions, respecting only the syntax is not enough. The generator should be able to generate type-correct expressions. Because of that, we guide the generation process by using the typing rules presented in Figure 3.

The aim of the expression generator is to produce a well-typed expression of the desired type. Note that for generating an expression of a given type, only a subset of the typing rules can be used. For example, the rule T-Var can only be used when generating the body of a function or lambda expression, since the formal parameters give rise to free-variables, the rules T-Fun and T-Abs can only be used when the expected type is a function type, and so on.

The method we use to generate expressions involves reading the typing rules backwards. In other words, to produce an expression that appears as the consequence of a rule, we must first create the expressions that form the rule's premises and then merge them. This process may require recursively generating sub-goals to ultimately generate the desired expression. Employing the typing rules in this manner guarantees that the expressions we create are correctly typed.

Considering this, we propose an expression generation judgment as follows.

Definition 3. *Expression generation judgment.* $\Delta; \Gamma; T \overset{e}{\Rightarrow} e$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, a new expression* e *is generated by selecting a syntactical constructor at random respecting the typing rules.*

It is worth to remember that an expression can contain sub-expressions, and because of that, the expression generation judgment should be invoked recursively to fulfill the sub-expressions accordingly. Since the subset of Haskell we are considering allows expressions to have variables, application of lambda expressions and functions, *let* bindings, pattern matching with *case* expressions, conditionals, tuples, lists, ADT data constructors, and other simple operators[1], we present next the definitions to create each valid expression individually.

Definition 4. *Variable generation.* $\Delta; \Gamma; T \overset{var}{\Longrightarrow} x$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, a variable* x *of type* T *is selected from the* Γ *context at random, if and only if there exists one or more variables of type* T*.*

Following the typing rule T-Var, a variable expression can only be created if there is some variable of the expected type in the Γ environment. So, it is important to mention that we don't use a greedy algorithm to avoid the need for backtracking. Instead, for each type, the algorithm presents a list of candidate expressions that can be generated, which allows a random selection, excluding syntactical constructors that cannot be used for that type.

Definition 5. *Function access generation.* $\Delta; \Gamma; \overline{T} \to T \overset{fun}{\Longrightarrow} f$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* $\overline{T} \to T$*, a function* f *of type* $\overline{T} \to T$ *is selected from the* Γ *context at random, if and only if there exists one or more functions of type* $\overline{T} \to T$*.*

The function access generation is similar to the variable generation, except that it searches the Γ context for function types.

Definition 6. *Abstraction generation.* $\Delta; \Gamma; \overline{T} \to T \overset{abs}{\Longrightarrow} \lambda \overline{P} :: \overline{T} \to e$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* $\overline{T} \to T$*, a lambda expression is created by generating a sequence of patterns* $\overline{P}$ *for each type in* $\overline{T}$*, and an expression* e *which result type should be* T*, considering an extended* Γ *environment with variables extracted from the patterns* $\overline{P}$*.*

The abstraction generation follows the typing rule T-Abs to generate a lambda expression. The input type $\overline{T} \to T$ defines that the algorithm needs to generate a function with arguments of type $\overline{T}$ and return type T. It means that we need to use the pattern generation judgment to generate patterns for each argument type, and apply recursively the expression generation judgment to generate the body of the required function, following the premise of rule

[1] We omit the rules for mathematical and relational operator for space reasons, however, their semantics and typing rules are standard.

T-Abs, where the body expression can use the variables extracted from the generated patterns.

Example 2. *To generate an abstraction expression, we use the judgments to generate patterns and expressions.*

$$\frac{\overline{\Delta;\Gamma;T_p \overset{pat}{\Longrightarrow} P}\ (2) \qquad \Delta;\Gamma,\ var\text{-}types(\overline{P},\overline{T_p});T_r \overset{e}{\Rightarrow} e\ (3)}{\Delta;\Gamma;\overline{T_p} \to T_r \overset{abs}{\Longrightarrow} \lambda\overline{P} :: \overline{T_p} \to e\ (1)}$$

In this example we show how the judgments are used to generate an abstraction expression. Invocation (1) uses the abstraction generation judgment to generate the lambda expression. Then, invocation (2) uses the pattern generation judgment to generate a pattern P for each type in $\overline{T_p}$. And last, invocation (3) uses the expression generation judgment to generate an expression which should respect the function return type T_r considering the variables extracted from the generated patterns through the function *var-types*[2].

Definition 7. *Application generation.* $\Delta;\Gamma;T \overset{app}{\Longrightarrow} e\ \overline{e}$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, an application is created by generating an expression* e *with a function type* $\overline{T} \to T$*, and a sequence of expressions* $\overline{e}$ *of type* $\overline{T}$*.*

To generate a function application, there is two valid options. The first considers using a function declaration in which the return type is T, if that function exists. The second considers the generation of a lambda expression with return type T. In any case, the algorithm have to generate the expressions to represent the actual parameters $\overline{e}$ respecting the types in $\overline{T}$.

Example 3. *To generate an application expression, the expression generation judgment should be used recursively twice.*

$$\frac{\Delta;\Gamma;\overline{T} \to T \overset{e}{\Rightarrow} e\ (2) \qquad \overline{\Delta;\Gamma;T \overset{e}{\Rightarrow} e}\ (3)}{\Delta;\Gamma;T \overset{app}{\Longrightarrow} e\ \overline{e}\ (1)}$$

The example above shows that to generate an application expression, we need to generate the required sub-expressions using the expressions generation judgment recursively. Invocation (1) uses the application generation judgment, which from a type T generates the function invocation. The invocation (2) generates an expression of a function type $\overline{T} \to T$. Then, invocation (3) calls the expression generation judgment for each type in $\overline{T}$, generating a sequence of expressions $\overline{e}$ to represent the actual parameters.

Definition 8. *Let generation.* $\Delta;\Gamma;T \overset{let}{\Longrightarrow} let\ x = e_1\ in\ e_2$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, a let binding is created by generating an unused variable name* x*, an expression* e_1 *with type* T_1*, and an expression* e_2 *which should have type* T*, considering an extended* Γ *environment with the variable* x *of type* T_1*.*

[2]When generating a lambda expression, only one equation is allowed in the standard Haskell implementation.

The *let* binding generation first selects an unused name for the variable x from a predefined list of names. Then it uses the type generation judgment to decide which type expression e_1 should have, and generates this expression using the expression generation judgment recursively. The last step generates expression e_2 augmenting the Γ context with variable x.

Definition 9. *Case generation.* $\Delta;\Gamma;T \overset{case}{\Longrightarrow} case\ e_1\ of\ \overline{P \leftarrow e}$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, a case expression is created by generating an expression* e_1 *of type* T_1*, and a sequence of alternatives in the form* $\overline{P \leftarrow e}$*, where each of them have a pattern* P *for the type* T_1 *and an expression* e *which should have type* T*, considering an extended* Γ *environment with variables extracted from the pattern* P*.*

To generate a *case* expression, the algorithm uses first the type generation judgment to generate a type T_1 that allows pattern matching[3]. Then an expression e_1 is generated through the expression generation judgment using as input the type T_1. The last step is to generate the sequence of *case* alternatives, where each alternative is composed by a pair of patterns and expressions. Here we define a threshold to limit the number of alternatives. To generate these items the algorithm uses the pattern generation judgment and the expression generation judgment, following the premises of rule T-Case, where each expression can use the variables extracted from the generated patterns.

Example 4. *To generate a case expression, we need to use the judgments to generate types, patterns and expressions.*

$$\frac{\overline{\Delta;\Gamma;T_1 \overset{pat}{\Longrightarrow} P}\ (4) \qquad \overline{\Delta;\Gamma,\ var\text{-}types(P,T_1);T \overset{e}{\Rightarrow} e}\ (5) \qquad \Delta;\Gamma \overset{t}{\Rightarrow} T_1\ (2) \qquad \Delta;\Gamma;T_1 \overset{e}{\Rightarrow} e_1\ (3)}{\Delta;\Gamma;T \overset{case}{\Longrightarrow} case\ e_1\ of\ \overline{P \leftarrow e}\ (1)}$$

This example shows us how the judgments are used to generate a *case* expression. Invocation (1) uses the *case* generation judgment to start the process. Then invocation (2) uses the type generation judgment to select a valid type T_1 that can be used in the pattern matching process. After that, invocation (3) uses the expression generation judgment to generate an expression e_1 of type T_1. Then invocation (4) and (5) are used with a threshold limit to generate the alternatives, where the first uses the pattern generation judgment and the second uses once more the expression generation judgment. That way, we guarantee that the expression is well-typed.

Definition 10. *Conditional generation.* $\Delta;\Gamma;T \overset{if}{\Longrightarrow} if\ e\ then\ e_1\ else\ e_2$

Given a Δ *context containing the user-defined types, a* Γ *context containing the free variables and functions, and a type* T*, a conditional expression is created by generating an expression* e *of type* $\mathbb{B}$*, and expressions* e_1 *and* e_2 *which should be of type* T*.*

Generation of conditional expressions should respect the T-If rule. The algorithm uses the expression generation judgment 3

[3]Here we need that distinction, because the function type doesn't allow pattern matching.

times, where the first should generate an expression of type *boolean*, and the others should have the same type T, according to Haskell semantics.

Definition 11. *Tuple generation.* $\Delta; \Gamma; (\overline{T}) \overset{tup}{\Longrightarrow} (\overline{e})$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, and a tuple type $(\overline{T})$, a tuple is created by generating an expression e for each type in $\overline{T}$.

The tuple generation is simple. It should use the expression generation judgment recursively to generate expressions respecting the tuple type.

Definition 12. *List generation.* $\Delta; \Gamma; [T] \overset{list}{\Longrightarrow} [\overline{e}]$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, and a list type $[T]$, a list is created by generating a sequence of expressions $\overline{e}$ of type T.

Similarly, the list generation uses the expression generation judgment recursively to generate the list elements of a given type. Different from the tuple generation, the list has only one type for all of its elements. For simplicity, the algorithm creates lists only which the square brackets notation, not using the list cons operator.

Definition 13. *ADT data constructor generation.* $\Delta; \Gamma; N \overset{constr}{\Longrightarrow} K\ \overline{e}$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, and a user-defined type N, an ADT data constructor is created by selecting a valid data constructor K with its list of types $\overline{T}$ from type N at random, and generating an expression e for each expected type in $\overline{T}$.

To generate an ADT data constructor from a valid user-defined type T, the algorithm selects one of them from the Δ context, and generates an expression for each expected type of that constructor using the expression generation judgment recursively. Since this procedure respects the T-Constr typing rule, it is guaranteed that only valid data constructor expressions will be generated.

Example 5. *To generate an ADT data constructor expression, we use the expression generation judgment recursively.*

$$\frac{K : \overline{T} \to N \in \Delta \quad (2) \qquad \overline{\Delta; \Gamma; T \overset{e}{\Rightarrow} e} \quad (3)}{\Delta; \Gamma; N \overset{constr}{\Longrightarrow} K\ \overline{e} \quad (1)}$$

In this example, we can see that the ADT data constructor judgment is used to start the generation process in invocation (1). Then, in the invocation (2), a constructor K of type N is selected from the Δ context. Last, we use the expression generation judgment to generate expressions $\overline{e}$ for each type in $\overline{T}$ in invocation (3).

3.4 Declaration Generation

In the presented subset of Haskell we have three different constructors for declarations. We can generate ADTs, type aliases, and functions. All declarations need to create new names, for ADTs, constructors, aliases and functions. These names are selected from a predefined list to avoid repetitions and to generate programs with human readable names.

Similar to the generation of types, generation of ADTs and aliases doesn't depend on the typing rules, having only to respect the syntax and the use of valid types.

Haskell allows one developer to make compound types by using ADTs, i.e., creating a new type where one can specify the shape of each of the elements, which can represent sum or product types. An ADT is composed by a type constructor, a non-empty list of data constructors, where each data constructor can have a possibly empty list of valid types[4]. The following definition presents how an ADT is generated by our algorithm.

Definition 14. *ADT generation judgment.* $\Delta; \Gamma \overset{adt}{\Longrightarrow} data\ N = \overline{K\ \overline{T}}$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, a new ADT is created by generating a new name N, and a sequence of constructor names K with their list of valid types $\overline{T}$.

We can note that to create a new ADT, besides generating names for type and data constructors, we need to use the judgment to generate types for each constructor. This way, it is guaranteed that only valid types will be selected during the generation process. We use a threshold to define the maximum number of constructors and types that can be generated.

A type alias or type synonym is a new name for an existing type, where values of different synonyms of the same type are entirely compatible. The generation is simple, and is presented in the following definition.

Definition 15. *Alias generation judgment.* $\Delta; \Gamma \overset{alias}{\Longrightarrow} type\ N = T$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, a new alias is created by generating a new name N for a valid type T.

In a similar manner, we use the type generation judgment to allow only valid types.

Since Haskell is a functional language, functions play a major role when using this language. Mathematically, a function relates all values in a set A to values in a set B. In Haskell, functions can be written with an optional type specification, and a set of equations, where each of them can use pattern matching to deal with parameters and expressions to be evaluated. The following definition presents the function declaration generation judgment.

Definition 16. *Function declaration generation judgment.* $\Delta; \Gamma \overset{f}{\Rightarrow} f :: \overline{T} \to T \triangleright \overline{f\ \overline{P} = e}$

Given a Δ context containing the user-defined types, a Γ context containing the free variables and functions, a new function is created by generating a new name f, a function type $\overline{T} \to T$, and a sequence of equations of the form $\triangleright\ \overline{f\ \overline{P} = e}$, where each of them have a list of patterns $\overline{P}$ for the types $\overline{T}$, and an expression e which result type should be T, considering an extended Γ environment with variables extracted from the patterns $\overline{P}$.

[4] In this subset, we are not considering the record syntax for the ADT generation.

The function declaration generation judgment is a bit complex. It combines all the generation judgments presented in the last subsections, since we need to generate the function name and type, patterns and expressions. Furthermore, it should respect the T-Function rule, to guarantee that all generated expressions have the expected type. Considering that all the previous judgments are guaranteed to be well-typed, the generated function should also be well-typed.

Example 6. *To generate a function, we use the judgments to generate types, patterns and expressions.*

$$\frac{\overline{\Gamma; T_p \overset{pat}{\Longrightarrow} P} \ (4) \qquad \overline{\Gamma, \textit{var-types}(\overline{P}, \overline{T}); T_r \overset{e}{\Rightarrow} e} \ (5) \qquad \overline{\Gamma \overset{t}{\Rightarrow} T_p} \ (2) \qquad \Gamma \overset{t}{\Rightarrow} T_r \ (3)}{\Gamma \overset{f}{\Rightarrow} f :: \overline{T_p} \to T_r \triangleright \overline{f\ \overline{P} = e} \ (1)}$$

In this example we show how the judgments are used to generate a function declaration. Invocation (1) uses the function generation judgment where a name f is selected from a predefined list of unused names. Then, to generate the function type we use the type generation judgment twice. Invocation (2) generates the types for each parameter and invocation (3) generates the function return type. We use a threshold to limit the maximum number of parameters. After that, to complete the function, the process needs to generate a non-empty sequence of equations starting with the function name already selected. For each equation, the pattern generation judgment is used to generate patterns for each parameter type, as we can see in invocation (4). Similarly, for each equation, the expression generation judgment is used to generate an expression which should respect the function return type T_r considering the variables extracted from the generated patterns through the function *var-types*, as shown in invocation (5).

4 DIFFERENTIAL TESTING

To provide a proof-of-concept, we implemented a Haskell code generator based upon the definitions presented in Section 3, and a test suite using the Haskell language[5]. Then, we used the test suite to run 10 batches of 1000 tests. Each batch took less than 5 seconds to generate 1000 programs, however to generate and compile each program on a batch with the GHC compiler, it was necessary around 20 minutes on average to conclude. The experiments were performed on an ASUS laptop with an Intel(R) Core i5-7200U CPU (2,50 GHz × 4), with 8GB RAM, running Linux 6.5.0-35 (Ubuntu 22.04.4 LTS). The experiments with the prototype were performed with a weak laptop configuration with an acceptable running time, what suggests that using a powerful hardware configuration could improve drastically the performance. It is worth mentioning that all generated programs were compiled successfully by the GHC compiler, which indicates that we are indeed generating only well-typed programs, as stated by our definitions.

As a way to check the diversity of the generated programs, we used the Haskell Program Coverage (HPC) [11] tool. We proceed with that verification because the algorithm decides the branches to generate at random, i.e., some programs should contain only parts of the considered syntax. However, by the statistics reports of HPC, we could notice that 100% of the syntactical constructors by the generated programs, and 97% of the equations and alternatives from the function that exports the AST to the concrete syntax of Haskell were reached, when considering a batch of 1000 generated test cases.

Having made the basic tests, and collected some statistics, we turned our attention for differential testing the compiler. Each of the generated programs was compiled and executed without optimization, and with different levels of optimization on the GHC compiler (-O0, -O, -O1, and -O2) aiming to check two properties:

- *Compilation preservation.* If a program is with a valid syntax and well-typed, it should compile with and without optimization. Otherwise, it could indicate a *bug* on the compilation step.
- *Behavior preservation.* If the same program is executed with and without optimization, the result should be the same on both executions. Otherwise, it could also indicate a bug at runtime.

Each generated program goes through the compilation step 4 times. The first uses the GHC flag -O0, to compile the program without optimization. The other three compile the generated program with optimization flags -O, -O1, and -O2, which apply from basic to more advanced optimizations. Then, we compare the GHC output, checking whether the program was successfully compiled each time with the different optimization levels, to test the compilation preservation property.

The compilation phase generates 4 binary files for each program, each of them was compiled with different levels of optimization. To test the behavior property, we run each of these binaries, collecting and comparing their output. As an extra test, we also execute the same program using the Haskell interpreter.

We used QuickCheck [4] as a lightweight manner to define these properties, for which the generated random Haskell programs are passed as input. QuickCheck helped us by providing a library with basic building blocks and combinators for generating random data and running the properties, and by allowing us to experiment with different design and implementations of our generator. This was essential to find bugs and fix them during the development phase.

After running 10,000 tests, we analyzed the statistics provided by QuickCheck as a way to summarize the results. During our tests, we were able to identify a potential bug on GHC when testing the compilation preservation property. The same source-code was compiled by GHC with the flag -O (standard optimizations) successfully, but it was a compilation error with flag -O0 (without optimization). Next we can see part of the GHC error message.

```
[1 of 1] Compiling Main
ghc: sorry! (uninplemented feature or known bug)
 (GHC version 8.8.4 for x86_64-unknown-linux):
     Trying to allocate more then 132879 bytes.
```

Upon analyzing the error message and the generated source code, we observed that the generated file was extremely large. The generator was creating static data to populate a list of elements, which turned out to be of an enormous size. While searching online,

[5]The complete source-code of our implementation is available online for access at https://github.com/sfeitosa/hsgenerator.

we discovered a bug report for this issue, which had been reported in 2010[6]. The root cause of this bug is the allocation of a significant amount of memory to store static data within a single source file. Currently, this is recognized as a limitation of GHC.

To prevent the recurrence of the same error, we made strategic modifications to our program generator. These changes were specifically aimed at reducing the amount of static data produced. Previously, the generator was creating extensive static data to populate large lists within the programs, which led to the error. By refining the generator's algorithm, we minimized the creation of such data, thereby avoiding the generation of excessively large programs that could trigger the identified error. This proactive approach has significantly improved the reliability and efficiency of our code generation process.

5 RELATED WORK

The use of random testing for the field of programming languages dates back to the early 1960s. Even so, the generation of random programs continues to be a challenge by itself. As a consequence of this, random testing for finding bugs in compilers and programming language tools received some attention in recent years.

Generation of random programs for imperative languages. YARPGen [18] is a random test-case generator for C and C++, which was used to find and report several bugs in GCC, LLVM and the Intel C++ Compiler. The testing tool Csmith [25] is a generator of programs for the C language, supporting a large number of language features, which was used to find a number of bugs in compilers such as GCC, LLVM, etc. These works differ by ours, first by generating code for a different programming language and paradigm, and second because most of them rely on informal approaches to describe the generation process. We believe that several ideas presented on these papers can be incorporated in our formal generation method. The work of Klein et al. [16] presents a new algorithm for randomly testing programs and libraries combining contracts and environment bindings to guide the test-case generator in the context of a Java-like class system. Feitosa, Ribeiro and Du Bois [9]) also used Featherweight Java as a basis to formalize a type-directed procedure to generate random well-typed programs, and an extension to generate programs considering Java 8 constructors. Although these papers use a similar approach to ours on the generation of programs, they consider the generation of an object-oriented language. However, their generator is also implemented is Haskell.

Generation of random programs for functional languages. The work of Palka, Claessen and Hughes [21] used the QuickCheck library in their work aiming to generate λ-terms to test the GHC compiler. Our approach was somewhat inspired by theirs, since we also generate well-typed terms following the formal typing rules to test the Haskell compiler. Our work differs from theirs by providing the a type-directed algorithm to generate a larger subset of Haskell. The work of Mista and Russo [20], develops an extensive framework for deriving compositional generators, which can be combined in different ways to fit the developers' demands. This work approaches a different and more general problem than ours, however, we believe that their ideas could be explored further explored by us in future works. Another example is the work of Drienyovszky, Horpácsi and Thompson [6] which presents an automated testing framework based on QuickCheck for testing refactoring tools written for the Erlang programming language. The intersection point with our work is the generation of source-code. We believe that our results can also be applied for testing refactoring tools, although it was out of scope on this paper. More recently, Frank, Quiring, and Lampropoulos [10] provided a novel algorithm to generate functions, where they delay the generation of types for sub-expressions to allow the use of arguments more efficiently.

Other sorts of program generation. Recently, the field of generative AI and machine learning based test-case generation has gained a lot of attention. For example, the work of Liu et al. [17] uses a machine learning approach targeting C compilers, where 82% of the generated code can be compiled and optimized, aiming finding compiler crash bugs. Similarly, Cummins et al. [5] approaches the problem using the same technique, mainly targeting OpenCL compilers. Lyu et. al. [19] proposed a coverage-guided fuzzer for prompt fuzzing that iteratively generates fuzz drivers to explore undiscovered library code. These works use different techniques to generate code, which can also be explored in the context of testing the Haskell compiler.

6 CONCLUSION

In this paper, we described a syntax directed judgment for generating random type-correct Haskell programs. We argued that the generation method is sound with respect to a subset of Haskell's type system, which includes primitive types and operators, ADT constructors, several expressions, conditional and pattern matching. Furthermore, we presented an implementation of a test suite in Haskell, used to apply differential testing with the properties of compilation and behavior preservation on the GHC compiler.

As future work, we can expand the algorithm to cover more complex syntactical constructors of Haskell, such as polymorphic and higher-order types. The differential testing can also be applied to other Haskell compilers. Furthermore, the use of AI to generate programs can be explored to compare with the approach presented in this paper.

[6] https://gitlab.haskell.org/ghc/ghc/-/issues/4505.

REFERENCES

[1] Franco Bazzichi and Ippolito Spadafora. 1982. An automatic generator for compiler testing. *IEEE Transactions on Software Engineering* 4 (1982), 343–353.

[2] Elton Maximo Cardoso, Daniel Freitas Pereira, Regina Sarah Monferrari Amorim De Paula, Leonardo Vieira Dos Santos Reis, and Rodrigo Geraldo Ribeiro. 2022. A Type-Directed Algorithm to Generate Random Well-Formed Parsing Expression Grammars. In *Proceedings of the XXVI Brazilian Symposium on Programming Languages* (<conf-loc>, <city>Virtual Event</city>, <country>Brazil</country>, </conf-loc>) *(SBLP '22)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3561320.3561326

[3] Augusto Celentano, S Crespi Reghizzi, P Della Vigna, Carlo Ghezzi, G Granata, and Florencia Savoretti. 1980. Compiler testing using a sentence generator. *Software: Practice and Experience* 10, 11 (1980), 897–918.

[4] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266

[5] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[6] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. 2010. Quickchecking Refactoring Tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on*

Erlang (Baltimore, Maryland, USA) *(Erlang '10)*. ACM, New York, NY, USA, 75–80. https://doi.org/10.1145/1863509.1863521

[7] Simon Marlow (Editor). 2010. Haskell 2010 Language Report. http://www.haskell.org/onlinereport/haskell2010/ Accessed: 2024-04-10.

[8] Samuel Feitosa, Rodrigo Ribeiro, and Andre Du Bois. 2020. A type-directed algorithm to generate random well-typed Java 8 programs. *Science of Computer Programming* 196 (2020), 102494. https://doi.org/10.1016/j.scico.2020.102494

[9] Samuel Feitosa, Rodrigo Ribeiro, and Andre Du Bois. 2020. A type-directed algorithm to generate random well-typed Java 8 programs. *Science of Computer Programming* 196 (2020), 102494. https://doi.org/10.1016/j.scico.2020.102494

[10] Justin Frank, Benjamin Quiring, and Leonidas Lampropoulos. 2024. Generating Well-Typed Terms That Are Not "Useless". *Proc. ACM Program. Lang.* 8, POPL, Article 77 (jan 2024), 22 pages. https://doi.org/10.1145/3632919

[11] Andy Gill and Colin Runciman. 2007. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop* (Freiburg, Germany) *(Haskell '07)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1291201.1291203

[12] Jean-Yves Girard. 1986. The system F of variable types, fifteen years later. *Theoretical Computer Science* 45 (1986), 159–192. https://doi.org/10.1016/0304-3975(86)90044-7

[13] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. 2007. A history of Haskell: being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (San Diego, California) *(HOPL III)*. Association for Computing Machinery, New York, NY, USA, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[14] Graham Hutton. 2016. *Programming in Haskell* (2nd ed.). Cambridge University Press, USA.

[15] SL Peyton Jones, K Hammond, WD Partain, PL Wadler, CV Hall, and Simon Peyton Jones. 1993. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele* (proceedings of joint framework for information technology technical conference, keele ed.). DTI/SERC, 249–257. https://www.microsoft.com/en-us/research/publication/the-glasgow-haskell-compiler-a-technical-overview/

[16] Casey Klein, Matthew Flatt, and Robert Bruce Findler. 2010. Random Testing for Higher-order, Stateful Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 555–566. https://doi.org/10.1145/1869459.1869505

[17] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[18] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (Nov. 2020), 25 pages. https://doi.org/10.1145/3428264

[19] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. arXiv:2312.17677

[20] Claudio Agustin Mista and Alejandro Russo. 2019. Deriving Compositional Random Generators. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL '19)*.

[21] Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (Waikiki, Honolulu, HI, USA) *(AST '11)*. 91–97. http://doi.acm.org/10.1145/1982595.1982615

[22] Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

[23] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3167092

[24] Haskell Wiki. 2020. Haskell in industry. http://wiki.haskell.org/Haskell_in_industry Accessed: 2024-05-14.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.* 46, 6 (June 2011), 283–294. https://doi.org/10.1145/1993316.1993532