# Towards Random Elixir Code Generation

Bernardo Beltrame Facchi
bernardobf[at]outlook.com.br
Universidade Federal da Fronteira Sul
Chapecó, SC, Brazil

Andrei de Almeida Sampaio Braga
andrei.braga[at]uffs.edu.br
Universidade Federal da Fronteira Sul
Chapecó, SC, Brazil

André Rauber Du Bois
dubois[at]inf.ufpel.edu.br
Universidade Federal de Pelotas
Pelotas, RS, Brazil

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

## ABSTRACT

Developers expect compilers to be correct. Unfortunately, these tools are not entirely bug-free. A failure introduced by the compiler could compromise a critical system and consequently have catastrophic consequences, specially in applications of great complexity, affecting both end users and developers. Such failures can lead to significant financial losses, security vulnerabilities, and a loss of trust in the software's reliability. Therefore, testing and validating all the compiler functionalities to assure its correctness is essential given their importance in software development. In light of the given context, this paper describes a random code generation tool using Haskell that generates well-typed Elixir code by adhering to a specified syntax and typing rules, which serves as input for property-based tests, striving to contribute to the overall quality and dependability of software systems built using Elixir.

## KEYWORDS

Code generation, Elixir Compiler, Property-based Testing

## 1 INTRODUCTION

The Elixir programming language has quickly become a powerful tool in modern software development, known for its scalability, concurrency, and functional programming capabilities [2]. Its Ruby-influenced syntax offers a user-friendly experience while maintaining efficiency and reliability for complex, real-time applications. This combination has led to its increasing adoption in various domains, from web development to embedded systems.

As Elixir's popularity grows, ensuring the compiler's correctness is crucial. Manual testing of compilers is often inefficient and can overlook edge cases, leading to potential failures in software. To address this, random code generation automates the creation of diverse test cases, allowing for more embrancing testing of the compiler's functionality [7]. This approach helps cover a larger subset of the language and test various functionalities more systematically.

Developing a random code generation tool is challenging due to the need to adhere to the language's syntactical and type system constraints. The paper investigates a bottom-up, goal-oriented approach [1, 5, 7] to generate randomized programs using Haskell, a language well-suited for code generation. The quality of the generator is measured through code coverage and property-based testing with the QuickCheck library, ensuring the generated code is well-typed and covers a wide range of scenarios. The paper also outlines the steps to generate Elixir programs, presents a prototype implementation, and discusses related works and future directions.

$$\frac{}{c : c}\ (\text{cst}) \qquad \frac{\Gamma \dashv x : t}{x : t}\ (\text{var})$$

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}\ (+) \qquad \frac{\overline{x} : \overline{s} \vdash e : t}{\lambda(\overline{x}.e) : \overline{s} \to t}\ (\lambda)$$

$$\frac{f : \overline{s} \to t \quad \overline{e} : \overline{s}}{f(\overline{e}) : t}\ (\text{app}) \qquad \frac{\overline{e} : \overline{t}}{\{\overline{e}\} : \{\overline{t}\}}\ (\text{tuple})$$

$$\frac{f : \text{int} \quad e : \{t_0, \ldots, t_n\}}{\pi_f(e) : t_i}\ (\text{proj}) \qquad \frac{f : s \quad x : s \vdash e : t}{\text{let } x : s = f \text{ in } e : t}\ (\text{let})$$

$$\frac{e_1 : s \quad \Gamma, \overline{\text{vars}(p, s) \vdash e_2} : t}{\Gamma \vdash \text{case } e_1 \text{ do } \overline{pg \to e_2} : t}\ (\text{case})$$

**Figure 1: Type system considered for expressions.**

## 2 ELIXIR LANGUAGE

The Elixir functional programming language, created by José Valim, had its first version released to the public in 2014, and runs on top of the Erlang Virtual Machine (BEAM). Elixir offers productive programming for secure and maintainable distributed applications by leveraging the virtual machine resources on which it is based [8]. Elixir is functional, process-oriented, scalable, concurrent, and fault-tolerant [3].

### 2.1 Syntax and Type System

Most Elixir constructors are syntactic sugar based on function application and pattern matching. Types in Elixir are polymorphic, set-theoretic, and recursively defined. The syntax this project was based upon is described in Castagna et al. [2], including types, expressions, patterns, guards, and selectors.

Both statically and dynamically typed programming languages have a type system. Its purpose is to define how the language constructions can be used besides its grammar. This process is carried out through a set of rules. Figure 1 presents a formal subset of the Elixir type system [2].

Each rule determines how the type of a specific term should be verified by the compiler. A term is valid if its premises are in accordance with the type system restrictions. Considering the approach of generating code by following each rule, we guarantee that the randomly generated code will be correct, allowing the code to be compiled and executed.

## 3 CODE GENERATION

The process of generating random, well-typed Elixir code is divided into three key steps: (*i*) randomly generating a valid Elixir type, (*ii*) randomly generating an expression in an abstract representation, and (*iii*) compiling the generated expression into Elixir concrete syntax. On this basis, the generated type is used as input to the expression generation process, which uses the typing rules as constraints to create valid expressions based on a bottom-up approach, where to satisfy the conclusion of a rule, it is necessary to respect their premises. This process gives rise to an expression generation judgment, as follows.

Definition 1. *Expression generation judgment.* $\Gamma; T \overset{exp}{\Longrightarrow} e$

*Given a* $\Gamma$ *context containing the free variables, and a type* $T$*, a new expression* $e$ *is generated by selecting a syntactical constructor at random respecting the typing rules.*

The generation technique we use is derived by interpreting the typing rules from Figure 1 backwards. In essence, to produce an expression that appears as the result of a rule, one must initially create expressions that form the rule's premises and then merge them. Consequently, the process of generating a term may recursively need the creation of sub-goals. Employing the typing rules guarantees that the generated terms are correctly typed.

All the code produced for this article was developed in Haskell (version 8.6.5), using the QuickCheck library (version 2.12.6.1) for property-based testing[1].

### 3.1 Type Generation

The first step was to define all the valid Elixir types according to the language syntax in Haskell using Algebraic Data Type (ADT) constructors. We implemented the primitive types (*int* and *atom*), tuple type and function type. The process of generating types is entirely based on syntax, which implies that it's unrestricted and any valid type can be generated at random. The type generation follows a recursive process. If the selected type is final, i.e., a primitive type, the return is immediate. Otherwise, we recursively continue the generation. To avoid non-termination due to the recursive approach, we decrement a fuel on each recursive call so that when it reaches zero, only terminal types can be created, forcing the generation to stop.

### 3.2 Expression Generation

The expression generation process is similar to the type generation process. The difference is that it must be guided by the typing rules presented in Figure 1 to generate type-correct expressions.

The objective of the expression generation is to create a well-typed expression at random that should be evaluated to a specific type. To guarantee that the expression is well-typed, we respect the restrictions imposed by the typing rules during the generation process. For this reason, to generate an expression of a specific type, only a subset of the typing rules can be considered. For example, the tuple rule can only be used when the expected type is a tuple, the lambda rule can only be used when the expected type is a function type, the arithmetic addition rule can only by used when an integer type is expected, and so on.

As shown by the process generation judgment (Definition 1), for the generation of each expression, it is expected the use of two inputs: (1) a context (initially empty) containing variables that might be used during the generation of sub-expressions[2] that is fed during the generation process with new variables when they are created; and (2) a type, that defines what the expression should be evaluated to. Note that during the generation of sub-expressions their return type might not be the same as the return type of the initial expression.

The generation process uses a bottom-up approach, where to generate an expression, we must satisfy the expression typing rule's premises, which might require the generation of sub-expressions. Hence, the generation method explored in this paper is recursively defined and guided by the type system. This approach guarantees that the expressions created are well-typed.

To understand how the typing rules are used to guarantee the generation of type-correct expressions, let's consider the following example.

Example 1. *Using the expression generation judgment to create a new expression of type* int. $\Gamma; int \overset{exp}{\Longrightarrow} e$

*A typing rule can be formatted using the question mark ? as a placeholder for that expression, representing the first generation step, as follows:*

$$\Gamma \vdash ? : int \tag{1}$$

*Suppose the rule for arithmetic addition (+) was selected at random. Then, the second generation step would look like:*

$$\frac{\Gamma \vdash ?_1 : int \qquad \Gamma \vdash ?_2 : int}{\Gamma \vdash ?_1 + ?_2 : int}\ (+) \tag{2}$$

*The question marks (*$?_1$ *and* $?_2$*) represent the sub-expressions that will be generated as sub-goals.*

*To generate each sub-goal, the generation judgment should be used recursively for each placeholder. It means that other typing rules can be selected.*

*Suppose that, for short, each sub-goal selected the rule for constants (cst). Then, the third generation step would be:*

$$\frac{\dfrac{}{2 : int}\ (\text{cst}) \qquad \dfrac{}{5 : int}\ (\text{cst})}{\Gamma \vdash\ 2\ +\ 5 : int}\ (+) \tag{3}$$

*As can be noted, since only terminal rules (without premises) were selected, the generation process finished producing a new expression* $(2 + 5)$ *of type* $int$*.*

The approach presented in Example 1 was used to generate correctly typed literals, variables, arithmetic operators, lambda expressions and applications, tuples and projections, and pattern matching through case expressions with guards and selectors, in accordance with the typing rules specified by Castagna, Duboc, and Valim [2]. Given that the code produced conforms to a valid Elixir program, we believe that they can be further studied for testing purposes.

[1] https://anonymous.4open.science/r/Elixir-Generator-840F

[2] A sub-expression is part of an expression that is by itself an expression.

## 4 PROPERTY-BASED TESTS

To provide a proof-of-concept of our work, we implemented an Elixir random code generator tool using Haskell based upon the ideas presented in Section 3 and a simple test suite also using Haskell and QuickCheck. To ensure the validity of our generated Elixir code, we implemented a property designed to verify that all generated test cases compile successfully. This property establishes that the code generated by our tool is both syntactically valid and correctly typed. For each generated program, the process involves writing the code to disk and subsequently invoking the Elixir compiler to compile the program and report back whether the compilation was successful or if any error was found.

This property-based approach left us confident that each generated test case is valid, and in addition it also provided us with a mechanism for continuously validating the overall effectiveness and reliability of our code generator. The tested property ensures that our tool consistently produces valid, compilable code, which is a fundamental prerequisite for any further semantic testing or execution of the generated programs.

We used the test suite to run ten batches of 1000 tests. Each batch generated, compiled and executed all programs in roughly 6 minutes on a computer with an Intel(R) Core i7-10700k CPU (5,00 GHz x 8) running Ubuntu 20.04.6 LTS on Windows 11 through WSL. It is worth noting that all generated programs compiled successfully, confirming that we are indeed generating only well-typed programs, meaning that our code generation and testing framework were reliable in validating the generated Elixir code. It indicates that the generated programs can be used in other test scenarios.

We also generated programs and compared them across different versions of Elixir by looking at their outputs. We conducted ten sets of 1000 tests, each taking around 18 minutes. The versions we used were 1.15.0, 1.16.3, and 1.17.1, all of which utilized Erlang OTP/25. We employed asdf to manage multiple Elixir versions. During test execution, no differences in output were found between the versions we tested. Some tests, however, could not be completed due to the compilation process taking more than 10 seconds, which forced us to reduce the generated code size.

Additionally, we employed the Haskell Program Coverage (HPC) tool to assess the diversity of the generated programs and provide detailed insights into the execution paths taken by our code generation logic. Given that our approach to code generation is randomized, we have no control over what branches the algorithm will take during execution, it is crucial to ensure that this randomness adequately explores all syntactical constructs and does not inadvertently miss any critical paths or edge cases.

Upon analyzing the statistics report provided by HPC, we observed that 100 percent of the syntactical constructors were covered in a batch of 1,000 test cases. This result is significant as it demonstrates that our random code generation method effectively explores all possible code patterns within defined constraints. Achieving full coverage means that our tool can generate a wide variety of valid Elixir programs, ensuring that no syntactical edge cases are overlooked, which implies that our program generation approach can be trusted to test the full spectrum of valid Elixir syntax.

By ensuring that all syntactical constructs are represented in the generated programs, we can state that our tool provides good inputs for testing purposes, which is important for further tests, increasing the probability of identifying and addressing potential bugs.

## 5 RELATED WORKS

Although the concept of random code generation originated in the early 1960s [9], it continues to be a challenge to this day. Palka; Claessen; Russo; Hughes [7] tested the Haskell compiler (GHC) by generating lambda terms to locate errors in the target compiler. Livingskii; Babokin; Regehr [6] developed YARPGen for C and C++, which tested the GCC, LLVM, and Intel® C++ Compiler. Feitosa; Ribeiro; Bois [4] provided a Java program test generator specification using the Featherweight Java (FJ) formalism to generate well-typed programs. Yang; Chen; Eide; Regehr [10] developed CSmith, a random code generation tool for the C language targeting the GCC and LLVM compilers. These studies employ varied code generation techniques, which could be further investigated within the Elixir compiler testing framework.

## 6 FINAL REMARKS

In this paper, a random code generator was presented based on a formalization of Elixir's syntax and type system to generate type-correct Elixir programs. We reasoned that the generation method is sound concerning a subset of Elixir's type system, which includes primitive types and operators, several expressions, and pattern matching with guards. Furthermore, we used the QuickCheck library to perform a preliminar evaluation, and the HPC tool to produce an analysis of the Elixir code coverage through the generation and execution of the generated programs.

## ACKNOWLEDGMENTS

This work was partially funded by Universidade Federal da Fronteria Sul under process number PES-2023-0183.

## REFERENCES

[1] Elton M. Cardoso, Daniel F. Pereira, Regina M. A. De Paula, Leonardo V. S. Reis, and Rodrigo G. Ribeiro. 2022. A Type-Directed Algorithm to Generate Random Well-Formed Parsing Expression Grammars. In *Proc. of SBLP'22*. ACM, New York, NY, USA, 8–14. https://doi.org/10.1145/3561320.3561326

[2] Giuseppe Castagna, Guillaume Duboc, and José Valim. 2023. The Design Principles of the Elixir Type System. arXiv:2306.06391 [cs.PL]

[3] Elixir. 2023. Elixir. https://elixir-lang.org/

[4] Samuel Feitosa, Rodrigo Ribeiro, and Andre Du Bois. 2019. Generating Random Well-Typed Featherweight Java Programs Using QuickCheck. *ENTCS* (apr 2019), 3–20. https://doi.org/10.1016/j.entcs.2019.04.002

[5] 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

[6] 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

[7] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms *(AST '11)*. 91–97. https://doi.org/10.1145/1982595.1982615

[8] PetSI. 2018. ELIXIR: uma linguagem de programação brasileira em sistemas distribuídos do mundo. http://www.each.usp.br/petsi/jornal/?p=2459

[9] Richard L. Sauder. 1962. A general test data generator for COBOL. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE '62 (Spring))*. 317–323. https://doi.org/10.1145/1460833.1460869

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

Received 24 June 2024; revised 3 August 2024; accepted 3 August 2024