Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 31, Num. 2 (2024) 20-29

RESEARCH ARTICLE

Random Generation of Haskell Programs Applied to Optimization Testing in Compilers

Geração Aleatória de Programas Haskell Aplicada a Testes de Otimizações em Compiladores

Guilherme Rafael Graeff[1]*, Braulio Adriano de Mello[1], Denio Duarte[1], Andrei de Almeida Sampaio Braga[1], Samuel da Silva Feitosa[1]

Abstract : Compilers and interpreters are essential for developing any system or application, so validating their functionality and properties is crucial. These tools are susceptible to failures, like any software artifact, which can introduce errors into programs developed using them. For example, a compiler or interpreter error that alters a program's behavior can compromise a critical system, and the system impact can be costly. The literature reports several problems found in compilers and interpreters of different languages. When bugs are detected in early releases, they can be reported to the development team, who can fix them before the end user notices the problem. This work describes the procedure used to generate random Haskell programs, which serve as input for property-based tests. More specifically, this work aims to test the *compilation* and *behavior* properties of a program, comparing the compilation and execution results of programs generated with different levels of optimization of the Glasgow Haskell Compiler (GHC). From developing a tool that automates the tests, 10,000 random programs were generated, compiled, and executed, of which 57 presented compilation errors with different optimization levels. This result demonstrates that the used approach is promising for error detection and can be improved in further studies.
Keywords: Programming Languages — Program Generation — Compiler Testing — Property-based Testing

Resumo: Como compiladores e interpretadores são ferramentas essenciais para o desenvolvimento de qualquer sistema ou aplicação, validar suas funcionalidades e propriedades é crucial. Assim como qualquer artefato de software, estes também são suscetíveis a falhas, e essas falhas podem introduzir erros nos programas desenvolvidos através dessas ferramentas. Por exemplo, uma falha no compilador que altera o comportamento de um programa pode comprometer um sistema crítico e o impacto no sistema pode ser custoso. A literatura relata vários problemas encontrados em compiladores e interpretadores de diferentes linguagens. Quando os *bugs* são detectados em versões iniciais, eles podem ser relatados à equipe de desenvolvimento da ferramenta, que pode corrigir os erros antes mesmo que o usuário final perceba o problema. Nesse contexto, este trabalho tem como objetivo descrever o procedimento utilizado para a geração de programas Haskell aleatórios, os quais servem de entrada para a realização de testes baseados em propriedades. Mais especificamente, este trabalho objetiva testar as propriedades de *compilação* e *comportamento* de um programa, comparando os resultados da compilação e execução dos programas gerados com diferentes níveis de otimização do Glasgow Haskell Compiler (GHC). A partir do desenvolvimento de uma ferramenta que automatiza os testes, foram gerados, compilados e executados 10 mil programas aleatórios, dos quais 57 apresentaram erros de compilação com diferentes níveis de otimização. Esse resultado demonstra que a abordagem utilizada é promissora para detecção de erros, podendo ser aprimorada em novos estudos.
Palavras-Chave: Linguagens de Programação — Geração de Programas — Testes de Compiladores — Testes Baseados em Propriedades

[1] *Ciência da Computação, Universidade Federal da Fronteira Sul (UFFS), Chapecó, Santa Catarina, Brasil*
***Corresponding author**: guilherme.rafael.graeff@gmail.com
DOI: https://doi.org/10.22456/2175-2745.134925 • **Received:** 21/10/2023 • **Accepted:** 10/06/2024

1. Introdução

A confirmação da confiabilidade de ferramentas no desenvolvimento de software é uma etapa crucial para garantir a qualidade e a evolução contínua das tecnologias que englobam o desenvolvimento de software [1, 2, 3]. Uma abordagem comum para aprimorar a qualidade de software nas mais diversas áreas se dá a partir de testes. Entretanto, a criação de casos de teste capazes de detectar a maioria dos erros de um sistema não é uma tarefa trivial. Essa situação se agrava em sistemas grandes, em que os dados de entrada possuem um número exponencial de combinações. Nesta categoria estão incluídos os compiladores ou ferramentas de desenvolvimento de software, uma vez que os mesmos devem ser capazes de compilar ou lidar com um número expressivo de programas com diferentes níveis de complexidade. Garantir a confiabilidade e a corretude desses compiladores é essencial para evitar erros e comportamentos inesperados nos programas resultantes [1].

Para o teste de compiladores ou ferramentas de desenvolvimento, uma área que vem ganhando atenção é a de geração aleatória de programas. Neste método são gerados diferentes casos de teste de forma automática, abrangendo uma ampla variedade de construções sintáticas e semânticas da linguagem-alvo. Estes programas são usados como casos de teste para avaliar o desempenho, a robustez e a conformidade do compilador com as especificações da linguagem. Neste sentido, diversas abordagens para gerar código têm sido propostas [2, 4, 5], corroborando com a premissa de que testes gerados automaticamente podem auxiliar a comunidade responsável pelo desenvolvimento de compiladores e ferramentas relacionadas.

Embora seja possível gerar programas, desenvolver bons geradores não é uma tarefa trivial. Os programas gerados devem conter uma estrutura que seja aceita pelo compilador, o que inclui restrições como uma sintaxe correta ou instruções com tipagem correta [4]. Além disso, é importante que esses programas abranjam uma ampla variedade de construções e cenários, a fim de explorar diferentes partes do compilador e identificar possíveis falhas ou comportamentos indesejados [6].

Com o intuito de contribuir para as áreas de geração aleatória de programas e testar o desempenho e robustez de compiladores, este artigo tem como objetivo propor uma ferramenta de geração aleatória de casos de teste válidos e, a partir da formalização do sistema de tipos do Cálculo Lambda com extensões, encontrar erros no compilador GHC da linguagem Haskell. Pretende-se, com esta contribuição, aumentar a confiabilidade do compilador-alvo, colaborando para evitar que programadores se deparem com problemas que não foram inseridos pelos mesmos durante a fase de desenvolvimento dos seus sistemas.

Ao testar funcionalidades como a de otimização de código feita automaticamente pelo compilador, espera-se que as propriedades de compilação e de comportamento sejam preservadas. Isto significa que, na propriedade de compilação, se um programa é sintaticamente válido e bem tipado, o mesmo programa deve compilar com e sem otimização. Caso contrário, seria um indicativo de uma falha no compilador. E na propriedade de comportamento, se o mesmo programa é executado após ser compilado com e sem otimização, o resultado deve ser o mesmo em ambas as execuções. Este trabalho considera essas propriedades, tendo o intuito de evidenciar a ausência ou detectar a presença de comportamentos inesperados durante o processo de compilação e execução de diferentes programas gerados aleatoriamente, auxiliando assim na detecção de possíveis *bugs*.

As principais contribuições deste trabalho são:

- Um procedimento para gerar programas Haskell aleatórios direcionado pelas regras formais do sistema de tipos do Cálculo Lambda;
- Uma ferramenta que automatiza o processo de testes a partir dos programas gerados aleatoriamente;
- Uma análise com 10000 casos de teste que verificam as propriedades de compilação e execução de programas Haskell submetidos ao compilador GHC, considerando diferentes níveis de otimização; e
- A definição das propriedades de compilação e de manutenção de comportamento através das funcionalidades providas pelo QuickCheck para validar a abordagem proposta.

O restante do texto está organizado da seguinte forma: a próxima seção apresenta, brevemente, os conceitos de Cálculo Lambda, a linguagem Haskell e a biblioteca QuickCheck. Na sequência são apresentados os trabalhos relacionados na Seção 3. Os passos para a geração aleatória de programas em Haskell são apresentados na Seção 4. A Seção 5 detalha o processo de automatização da compilação e execução dos códigos gerados considerando os diferentes níveis de otimização, coletando as informações estatísticas durante o processo. Em seguida, a Seção 6, apresenta os resultados obtidos a partir da compilação e execução dos testes. Por fim, a Seção 7 apresenta as conclusões deste trabalho e algumas sugestões de desenvolvimento de trabalhos futuros.

2. Conceitos Preliminares

Esta seção apresenta brevemente alguns conceitos essenciais para auxiliar no entendimento deste trabalho. Inicialmente, é apresentado o Cálculo Lambda, em seguida a linguagem Haskell e o compilador GHC e, para finalizar, a biblioteca Quickcheck.

2.1 Cálculo Lambda

O Cálculo Lambda (Cálculo-λ) é um modelo formal para representar e manipular funções matemáticas. Ele foi desenvolvido por Alonzo Church na década de 1930 como uma ferramenta para investigar a computabilidade e os fundamentos da matemática [7]. O Cálculo-λ é considerado uma base

teórica fundamental para a ciência da computação, sendo usado para estudar a computabilidade, a complexidade computacional e a semântica das linguagens de programação [8]. Além disso, ele serviu de inspiração para o desenvolvimento de linguagens de programação funcionais, como Lisp, Haskell e Scheme, as quais incorporam os princípios do Cálculo Lambda em suas sintaxes e semânticas.

No Cálculo-λ, as funções são tratadas como objetos de primeira classe, *i.e.*, funções podem ser passadas como argumentos para outras funções, retornadas como resultados e atribuídas a variáveis. O Cálculo Lambda original é baseado em apenas três principais construções: variáveis, abstrações e aplicações [8]. A Figura 1 apresenta a formalização da sintaxe e do sistema de tipos do Cálculo Lambda estendido com valores numéricos.

Sintaxe:

$$
\begin{aligned}
e &::= n \\
&\mid x \\
&\mid \lambda x : \mathrm{T} \,.\, e \\
&\mid e \;\; e \\
\mathrm{T} &::= \mathrm{T} \to \mathrm{T} \\
&\mid \mathrm{Num} \\
\Gamma &::= \emptyset \\
&\mid \Gamma, x : \mathrm{T}
\end{aligned}
$$

Sistema de Tipos:

$$\Gamma \vdash n : \mathrm{Num} \quad \text{[T-Num]}$$

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

$$\frac{\Gamma, x : \mathrm{T}_1 \vdash e : \mathrm{T}_2}{\Gamma \vdash \lambda x : \mathrm{T}_1 \,.\, e : \mathrm{T}_1 \to \mathrm{T}_2} \quad \text{[T-Abs]}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{T}_1 \to \mathrm{T}_2 \quad \Gamma \vdash e_2 : \mathrm{T}_1}{\Gamma \vdash e_1 \; e_2 : \mathrm{T}_2} \quad \text{[T-App]}$$

Figura 1. Sintaxe e Sistema de Tipos do Cálculo Lambda.

A sintaxe apresentada na Figura 1 demonstra as possíveis expressões e do Cálculo Lambda. As construções sintáticas incluem valores numéricos n, variáveis x, abstrações $\lambda x : \mathrm{T} \,.\, e$, as quais representam funções anônimas, e as aplicações $(e\ e)$, que permitem aplicar um valor a determinada função. Além disso, são apresentados os tipos válidos T da linguagem, que incluem os tipos de função ($\mathrm{T} \to \mathrm{T}$), o tipo numérico (Num), e o contexto Γ, que é representado indutivamente por uma lista (possivelmente vazia $\emptyset$) contendo variáveis com seus respectivos tipos ($\Gamma, x : \mathrm{T}$).

O sistema de tipos apresentado na Figura 1 inclui 4 regras, sendo uma para cada construtor sintático. A definição de cada regra é descrita a seguir [8]:

- T-Num: é um axioma que indica que um número n possui um tipo Num.
- T-Var: indica que o tipo de uma variável x é definido a partir da existência dessa variável com seu respectivo tipo no contexto Γ.
- T-Abs: indica que o tipo de uma expressão lambda é dado a partir do tipo do argumento T_1 que é associado explicitamente à variável x, e o tipo de retorno T_2 que é definido indutivamente através do julgamento do sistema de tipos da linguagem.
- T-App: indica que o tipo de uma aplicação de função é T_2 o qual depende da definição indutiva dos tipos das expressões e_1 e e_2, onde e_1 deve ser de um tipo de função ($\mathrm{T}_1 \to \mathrm{T}_2$) e e_2 deve possuir o tipo T_1 esperado como argumento para a função.

Estas regras de tipo apresentadas são importantes para compreender o funcionamento do mecanismo de geração apresentado na Seção 4, uma vez que servem como um direcionador para garantir a construção de programas bem-tipados.

2.2 Haskell e GHC

Haskell é uma linguagem de programação funcional pura, ou seja, não há interferência de fatores externos ou efeitos colaterais, conhecida por sua forte tipagem estática e pela ênfase na expressão matemática de problemas. Além disso, possui alta expressividade, permitindo a solução de problemas complexos de maneira sucinta, *i.e.*, com poucas construções sintáticas. Ela foi nomeada em homenagem ao lógico e matemático Haskell Curry e foi desenvolvida no final da década de 1980 [9].

Atualmente, Haskell está inserido tanto no meio acadêmico [10, 9] como no meio comercial [11], tendo inspirado linguagens de outros paradigmas a implementarem funcionalidades similares, como por exemplo as expressões lambda, funções de alta ordem e *pattern matching*.

O *script* a seguir exemplifica um código em Haskell para calcular o fatorial de um número:

A primeira linha indica a anotação de tipo da função, a qual pode ser omitida, ficando o compilador responsável por realizar a inferência de tipos. As duas próximas linhas representam a implementação da operação fatorial de forma recursiva, sendo que na primeira é realizado *pattern matching* para o caso base (fatorial de zero) e na segunda é definida a expressão do caso recursivo que realiza a multiplicação do número n pelo resultado do fatorial de $n-1$.

Um programa em Haskell pode ser interpretado ou compilado. O interesse deste trabalho é em programas Haskell compilados, sendo Glasgow Haskell Compiler (GHC) o compilador alvo. O GHC é um compilador de código aberto, conhecido por sua velocidade e eficiência na geração de código, sendo atualmente um dos compiladores mais populares para a linguagem Haskell. Ele oferece suporte a várias extensões,

bibliotecas e recursos avançados, sendo adotado em diversos projetos acadêmicos e sistemas desenvolvidos por empresas [10].

O GHC permite ao desenvolvedor escolher o nível de otimização durante a compilação de um programa. Existem quatro formas (níveis) de indicar o tipo de otimização desejada durante a compilação, sendo definidas pela *flag* -O seguida pelo nível de otimização desejada [12]:

- -O0: nenhuma otimização deve ser realizada.
- -O: representa uma combinação básica das *flags* individuais de otimização.
- -O1: aplica otimizações que geram código de boa qualidade sem demandar muito tempo de compilação.
- -O2: aplica otimizações que não devem correr o risco de ficarem piores em questão ao tempo de execução ou em espaço em relação às outras, mesmo que isso tome mais tempo de compilação

No contexto deste trabalho, a linguagem Haskell é utilizada em dois momentos: (*i*) como ferramenta para desenvolver o gerador de código aleatório e aplicar os testes baseados em propriedades, e (*ii*) como linguagem alvo para a geração de código. Já o compilador GHC é o alvo dos testes, *i.e.*, os códigos gerados são compilados e executados pelo GHC utilizando os parâmetros de otimização citados anteriormente, *i.e.*, O0, O, O1 e O2, para identificar se as propriedades de compilação e comportamento são preservadas.

2.3 Biblioteca Quickcheck

A biblioteca QuickCheck [13] é uma ferramenta popular para testes automatizados em Haskell. Ela foi produzida para auxiliar desenvolvedores na criação de casos de teste, buscando aumentar a confiança na corretude dos programas desenvolvidos. Esta biblioteca foi precursora na metodologia de testes baseados em propriedades, tendo inspirado outras linguagens na construção de bibliotecas similares. Teste baseado em propriedades representa uma abordagem para testes de software que se concentra na verificação de propriedades gerais do sistema, em vez de casos específicos de entrada e saída. Ao invés de testar se o sistema retorna um resultado específico para um conjunto limitado de entradas, o teste baseado em propriedades visa verificar se uma propriedade mais geral é verdadeira, a qual é verificada através de entradas geradas aleatoriamente [13].

O QuickCheck e os testes baseados em propriedades oferecem uma abordagem promissora para a identificação de erros no contexto de ferramentas e linguagens de programação. Como o conjunto de programas passíveis de compilação é infinito, a possibilidade de gerar um grande conjunto de códigos aleatórios permite aprofundar os casos de teste na maioria das construções sintáticas da linguagem, permitindo que os testes sejam mais completos do que aqueles escritos manualmente. Isso é ainda mais importante no contexto de linguagens funcionais como o Haskell, pois em programas funcionais puros não há a existência de efeitos colaterais[1], o que garante que o código gerado e executado não depende de fatores externos. Este fato é explorado por este trabalho por meio da realização da geração de código Haskell de forma aleatória para a aplicação de testes baseados em propriedades para verificar possíveis problemas de compilação e execução de programas com diferentes níveis de otimização.

Assim, utilizando a sintaxe do Cálculo-λ para gerar programas em Haskell e o gerador de caso de testes do Quickchek, o GHC é testado para identificar se as as propriedades de compilação e comportamento são preservadas.

3. Trabalhos Relacionados

A seguir são apresentados alguns trabalhos que auxiliaram no processo de definição do tema do presente projeto. Estes trabalhos dissertam sobre a geração de programas com o fim de testar alguma ferramenta existente. Alguns autores utilizaram ferramentas similares às desta proposta, outros fizeram uso de outras ferramentas, entretanto, todos serviram de inspiração para algumas das partes desenvolvidas. Os estudos citados se utilizam de diferentes formas para gerar programas, sendo que dois destes utilizam linguagem e biblioteca específica (Haskell e QuickCheck), da mesma forma que o presente projeto, e dessa forma, possuem maior relação com o mesmo. Porém, os outros também foram importantes, pois discutiram maneiras diferentes de gerar casos de teste, apresentando uma abordagem diversa para o mesmo problema e sendo relevante para compreender o tema de maneira geral.

Pałka et al. [4], assim como este trabalho, buscam ajudar a confirmar o bom funcionamento dos compiladores. Utilizam ferramentas similares às utilizadas no desenvolvimento deste artigo (Haskell e QuickCheck), porém implementam menos construções do que foram propostas nesta pesquisa. Considerando as resultados apresentados pelo próprio autor, uma vez que seu estudo considerou apenas um pequeno *subset* da linguagem Haskell, notou-se que a abordagem é válida, permitindo a realização de geração de código aleatório para muitas outras construções sintáticas da linguagem. O estudo descreve que a partir apenas dele não foi possível exaurir todo o potencial de encontrar *bugs* no compilador GHC, o que serviu de motivação para a presente proposta.

Feitosa et al. [14], em um contexto similar de geração de programas, tratam de desenvolver a especificação de um gerador de programas bem-tipados (*well-typed*) usando o formalismo do Featherweight Java (FJ). Os autores buscam desenvolver a área de testes automatizados para FJ, o qual realiza a geração de código para testes no contexto da linguagem Java. A ferramenta, assim como aquela proposta por Pałka et al. [4], também foi desenvolvida na linguagem Haskell. Ainda de acordo com Feitosa et al. [14], a especificação desenvolvida seria uma generalização do que foi realizado na proposta de Pałka et al., considerando o contexto do Featherweight Java.

[1] Uma função possui um "efeito colateral" quando modifica estado não-local ou quando seu retorno é não determinístico.

Então os métodos utilizados pelos autores estão bem relacionados, uma vez que foi possível gerar programas aleatórios a partir das regras de tipo de uma linguagem de programação, com o uso da ferramenta QuickCheck, e com a verificação de propriedades que permitem testar o processo de compilação e ganhar confiança de que os códigos gerados eram todos bem-tipados.

Britto et. al. [2] utilizam uma abordagem um pouco diferente dos outros dois trabalhos citados bem como a linguagem alvo: TypeScript. TypeScript é uma linguagem de código aberto e comercial voltada majoritariamente para o desenvolvimento web. As falhas encontradas podem poupar tempo de desenvolvimento para os desenvolvedores web, e ainda agilizar o processo de desenvolvimento de novas aplicações para toda a comunidade, já que os erros que não são capturados na etapa de refatoração (ferramenta-alvo da pesquisa) podem introduzir *bugs* de difícil rastreio. Os autores utilizaram o Alloy Analyzer, uma API desenvolvida em Java que gera instâncias que satisfazem especificações de um programa, para desenvolver a ferramenta TSDolly, a qual transforma modelos em árvores de sintaxe abstrata (Abstract Syntax Tree - AST) de programas TypeScript, que então podem ser compilados utilizando TypeScript Compiler (TSC). A pesquisa busca testar ferramentas de refatoração para TypeScript de diferentes IDE's, compilando antes e depois da refatoração e coletando as métricas para verificar se os programas mantêm a propriedade de compilação. A partir de diversos casos de teste, com diferentes especificações na etapa de geração das instâncias pelo Alloy Analyzer, os autores conseguiram identificar uma falha nas etapas de refatoração de código de uma das diferentes IDE's. O *bug* foi então reportado à Microsoft, permitindo aos desenvolvedores da ferramenta corrigirem a falha rapidamente, garantindo um ambiente de desenvolvimento mais seguro do que era antes do *bug* ser descoberto.

Yang et. al. [1] concentraram seus esforços em utilizar testes automatizados para fazer com que os compiladores da linguagem C evoluíssem. É importante ressaltar que este estudo foi feito em um momento em que o teste de compiladores se dava muitas vezes pelo *feedback* dos usuários. Os resultados deste trabalho são referência na gerações de programas, sendo citado por diversos outros artigos [14, 4, 2]. O mesmo descreve um gerador de programas C aleatórios capaz de criar códigos que consideram um grande *subset* da linguagem, o qual cooperou para a melhoraria da qualidade de vários compiladores. Esta ferramenta, que também gera casos de teste randômicos, foi útil para reportar centenas de *bugs* aos desenvolvedores de diferentes compiladores de C ao longo dos anos. Na pesquisa, foram realizados diferentes experimentos utilizando o Csmith, que é um gerador de programas aleatórios resultante dos esforços dedicados pelos autores. Os resultados deste projeto foram de grande impacto na comunidade de desenvolvedores da linguagem C. Diferentes compiladores como o GCC e o LLVM puderam ser testados por esta ferramenta. Os autores afirmam que esta ferramenta é capaz de encontrar *bugs* em partes difíceis de se atingir, pois consegue gerar cenários atípicos utilizando combinações de diferentes recursos da linguagem. Mesmo este estudo tendo sido realizado no ano de 2011, os conceitos são de grande similaridade ao que se buscou chegar com esta pesquisa.

4. Gerando Programas Haskell de Forma Aleatória

O processo de geração de código aleatório adotado neste projeto considera a descrição formal do sistema de tipos do Cálculo-λ com diversas extensões, incluindo valores numéricos e *booleanos*, expressões lógicas e aritméticas, expressões condicionais, ligações locais (*let*) e listas.

O mecanismo de geração desenvolvido é definido em dois passos: (*i*) a geração aleatória de um tipo válido aceito pela linguagem, e (*ii*) a geração aleatória de uma expressão cuja avaliação resulta no tipo gerado no passo *i*. Neste sentido, o tipo é usado como entrada para o mecanismo de geração da expressão, o qual utiliza-se das regras de tipo da linguagem para verificar quais termos podem ser gerados respeitando o tipo esperado através da conclusão da regra de inferência. Isto significa que a regra é lida de baixo para cima (*bottom-up*), uma vez que para satisfazer a conclusão é necessário gerar uma determinada expressão que respeite o tipo de entrada. Entretanto, a conclusão pode depender da geração de sub-expressões[2], as quais são definidas nas premissas dessas mesmas regras. Sendo assim, o mecanismo de geração é um processo recursivo que utiliza o sistema de tipos para direcionar o processo de geração de código.

No que tange à implementação da ferramenta, foram executadas as seguintes etapas: a definição e representação da Árvore de Sintaxe Abstrata (AST) para os tipos e expressões através de Tipos de Dados Algébricos (ADTs) da linguagem Haskell; a implementação de geradores para os tipos e expressões, considerando a sintaxe da linguagem e respeitando as regras do sistema de tipos; a exportação dos códigos gerados nas ASTs para a sintaxe concreta do Haskell; e a execução dos testes baseados em propriedades.

Mais detalhes sobre o processo formal de geração, bem como a implementação da ferramenta são descritos nas seções que seguem.

4.1 Código-fonte

O código-fonte apresentado neste artigo foi desenvolvido em Haskell (versão 8.8.4), com o uso da biblioteca QuickCheck (versão 2.14.2) para a realização dos testes baseados em propriedades. No decorrer do texto são apresentados apenas trechos do código que são importantes para a compreensão do mecanismo de geração, sendo omitidos trechos que podem distrair o leitor do entendimento da implementação em alto nível. Todo o código produzido para este artigo pode ser consultado no repositório do GitHub do trabalho[3].

[2] Uma sub-expressão é uma parte de uma expressão que é, por si mesma, uma expressão correta.

[3] ⟨https://github.com/GuilhermeGraeff/lambda-calculus⟩

4.2 Geração Aleatória de Tipos

O primeiro passo realizado foi a geração de um tipo válido e aceito pela linguagem. A Figura 2 apresenta os tipos que foram considerados no *subset* utilizado para geração de código Haskell, o qual inclui tipos numéricos, booleanos, expressões lambda e listas.

$$
\begin{aligned}
\mathrm{T} ::= \ & \mathrm{Num} \\
\mid \ & \mathrm{Bool} \\
\mid \ & \mathrm{T} \rightarrow \mathrm{T} \\
\mid \ & \mathrm{List\ T}
\end{aligned}
$$

Figura 2. Tipos considerados no mecanismo de geração.

A partir da especificação na gramática acima, foi desenvolvido um módulo Haskell, que implementa um ADT para representar a sintaxe dos tipos. Esse tipo de dado representa a AST que é percorrida e utilizada pelo mecanismo de geração de tipos. A seguir, é apresentado o código Haskell que define a sintaxe dos tipos.

A partir desta definição sintática, já é possível definir o mecanismo de geração de tipos. Neste caso específico, não há restrições, além da sintaxe válida, para a geração, sendo assim, um dos tipos disponíveis é selecionado aleatoriamente. Caso um valor terminal seja selecionado, como é o caso do tipo ou , o retorno é imediato. Entretanto, caso sejam selecionados os tipos ou , é necessário utilizar o processo de geração de tipo recursivamente, para atender a definição sintática dos mesmos.

A biblioteca QuickCheck torna simples a implementação desta funcionalidade, oferecendo a função que permite selecionar as funções que geram cada tipo com pesos determinados pelo programador. O código a seguir implementa este processo.

A função é a principal operação recursiva que gera a AST de um tipo válido, sendo chamada sempre que um tipo é necessário. É possível notar que esta função recebe o parâmetro , o qual é usado para evitar recursão infinita. Isso significa que quando este parâmetro estiver com um valor menor ou igual a zero, apenas tipos terminais poderão ser selecionados, evitando novas chamadas recursivas e possibilitando a terminação do algoritmo. Sendo assim, realiza a seleção aleatória de um dos tipos válidos, onde os tipos recursivos são gerados a partir das funções e , e os tipos terminais são gerados pela função .

É importante ressaltar que a técnica utilizada é um padrão da área e não existe restrição para o valor definido para o parâmetro . Normalmente, este parâmetro é utilizado em conjunto com a função do QuickCheck, que utiliza valores aleatórios para cada chamada da função de geração. Neste artigo, para viabilizar um experimento mais controlado, foram utilizados valores fixos para este parâmetro, possibilitando a análise do comportamento do algoritmo para as métricas de avaliação aplicadas.

A Figura 3 ilustra um exemplo de uma AST de um tipo recursivo gerado a partir da função , o qual representa o tipo $(\mathrm{Num} \rightarrow \mathrm{Bool}) \rightarrow \mathrm{Bool} \rightarrow (\mathrm{Bool} \rightarrow \mathrm{Bool})$. É possível notar que, aleatoriamente, foram gerados tipos de função compostos e tipos terminais, como é o caso dos tipos numéricos e *booleanos*.

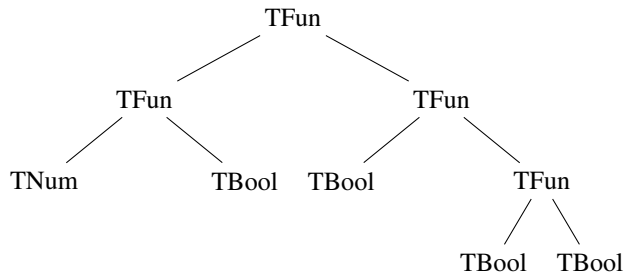

Figura 3. AST gerada que representa um tipo.

Este tipo gerado é utilizado como entrada para o próximo passo do mecanismo de geração, que é responsável por gerar aleatoriamente uma expressão. Este processo será detalhado na próxima subseção.

4.3 Geração Aleatória de Expressões

Esta seção apresenta a implementação da AST que representa as expressões válidas da linguagem e o processo de geração de expressões a partir da gramática BNF, que define a estrutura das expressões sintaticamente válidas e consideradas pelo mecanismo de geração proposto. A Figura 4 mostra parte das construções sintáticas de Haskell utilizadas[4].

$$
\begin{aligned}
e ::= \ & \mathrm{True} \\
\mid \ & \mathrm{False} \\
\mid \ & n \\
\mid \ & x \\
\mid \ & \lambda x : \mathrm{T} \ \rightarrow \ e \\
\mid \ & e \ \ e \\
\mid \ & \mathrm{let}\ x \ = \ e\ \mathrm{in}\ e \\
\mid \ & \mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e \\
\mid \ & [] \\
\mid \ & e : e \\
\mid \ & \mathit{head}\ e \\
\mid \ & \mathit{tail}\ e \\
\mid \ & \ldots \ \text{Outras construções sintáticas}
\end{aligned}
$$

Figura 4. *Subset* da sintaxe considerada pelo gerador.

[4]Todas as construções sintáticas utilizadas são bem conhecidas e aparecem na maioria dos livros-texto da área, como por exemplo em [8].

De forma similar à seção anterior, a partir da gramática acima foi adicionado outro ADT para representar a AST das expressões. Durante o processo de geração, as expressões geradas aleatoriamente são representadas através deste tipo de dado. A seguir é apresentado parte do ADT em Haskell que define a sintaxe de expressões.

A partir das definições sintáticas e o tipo gerado aleatoriamente, é possível iniciar o processo de geração de uma expressão que deve resultar no tipo gerado, respeitando o sistema de tipos da linguagem Haskell.

O processo de geração adota um procedimento orientado a um objetivo, o qual recebe como entrada um determinado contexto Γ (inicialmente vazio) e o tipo desejado para a expressão. O objetivo do gerador de expressões é produzir um termo bem-tipado do tipo desejado, o qual pode conter variáveis vindas de um dado contexto[5]. Para garantir que as expressões sejam bem-tipadas, as regras de tipo são utilizadas no processo de geração. O sistemas de tipo apresentado na Figura 1 é estendido com novas regras consideradas neste trabalho. A Figura 5 apresenta esta extensão.

$$\Gamma \vdash \text{True} : \text{Bool} \ \text{[T-True]}$$

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

$$\frac{\Gamma \vdash e_1 : \text{T}_1 \quad \Gamma, x : \text{T}_1 \vdash e_2 : \text{T}_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \text{T}_2} \ \text{[T-Let]}$$

$$\frac{\Gamma \vdash e_1 : \text{T}_1 \quad \Gamma \vdash e_2 : \text{List T}_1}{\Gamma \vdash e_1 : e_2 : \text{List T}_1} \ \text{[T-Cons]}$$

$$\frac{\Gamma \vdash e : \text{List T}}{\Gamma \vdash \text{head } e : \text{T}} \ \text{[T-Head]}$$

$$\frac{\Gamma \vdash e : \text{List T}}{\Gamma \vdash \text{tail } e : \text{List T}} \ \text{[T-Tail]}$$

Figura 5. Outras regras do sistema de tipos considerado pelo gerador.

É importante mencionar que, para gerar uma expressão de um dado tipo, apenas algumas regras de tipo podem ser usadas. Por exemplo, das regras de tipo mostradas na Figura 1, a regra T-Num só pode ser utilizada quando deseja-se gerar uma expressão de tipo numérico. Já a regra T-Var só pode ser usada quando gerando o corpo de uma expressão lambda ou de um *let*, uma vez que depende da existência de uma variável no contexto Γ. A regra T-Abs, por sua vez, só pode ser usada caso a expressão sendo gerada seja do tipo função, e por fim, a regra T-App pode ser usada em qualquer contexto, desde que respeite os tipos esperados. Já nas regras apresentadas na Figura 5, a regra T-True só pode ser usada quando for necessário gerar uma expressão que resulte em um tipo *booleano*. As regras T-Cons e T-Tail só podem ser usadas quando uma expressão que resulte em um tipo de lista seja esperado. Finalmente, as regras T-If, T-Let e T-Head podem ser usadas em qualquer contexto, desde que respeitem os tipos esperados durante a geração.

O método adotado neste trabalho para a geração é obtido a partir da leitura das regras de tipo apresentadas nas Figuras 1 e 5 de baixo para cima, *i.e.*, para gerar uma expressão que é consequência de uma regra, primeiro é necessário gerar (recursivamente) as sub-expressões descritas em suas premissas, e então combiná-las para gerar uma expressão final. Ao utilizar as regras de tipo fica assegurado que as expressões resultantes serão bem-tipadas.

Para exemplificar o processo de geração, será considerado a geração de um tipo numérico. É possível definir uma regra de tipo parcialmente, utilizando um ponto de interrogação ? como um espaço reservado (*placeholder*) para a expressão a ser gerada, representando o primeiro passo de geração:

$$\Gamma \vdash ? \ : \ \text{Num}$$

Dentre as regras apresentadas na Figura 5 é possível utilizar a T-Num e a T-App para gerar um tipo numérico[6]. Será considerada a regra T-App para o segundo passo de geração:

$$\frac{\Gamma \vdash ?_1 : \text{T}_{11} \to \text{Num} \quad \Gamma \vdash ?_2 : \text{T}_{11}}{\Gamma \vdash ?_1 \ ?_2 : \text{Num}}$$

Os pontos de interrogação $?_1$ e $?_2$ representam sub-expressões que serão geradas indutivamente como sub-objetivos. Neste sentido, é possível notar que a sub-expressão $?_1$ deve possuir um tipo de função, cujo retorno seja numérico Num, e a expressão $?_2$ deve ser do mesmo tipo T_{11} esperado pelo argumento da função definida por $?_1$. Para continuar o processo de geração, é necessário observar novamente as regras de tipo disponíveis, ou seja, o gerador é aplicado recursivamente para cada espaço reservado pelos pontos de interrogação.

O leitor pode notar que, usando esta estratégia de geração a partir do sistema de tipos, seria possível gerar programas recursivos, os quais poderiam ser não terminantes. Entretanto, a implementação do algoritmo evita explicitamente a criação deste tipo de programa, como forma de simplificar os testes de comportamento, uma vez que, se os programas não finalizassem, seria preciso definir mecanismos de interrupção abrupta dos casos de teste.

De forma similar ao processo de geração de tipos, foi definido um algoritmo que gera expressões recursivamente. Entretanto, na geração de expressões, além da sintaxe da linguagem, também são consideradas as regras de tipo, para garantir que o código gerado seja compilável pelo GHC. Também de forma similar à subseção anterior, para evitar a não terminação do

[5]Este contexto é alimentado a medida que expressões lambda ou expressões *let* são geradas

[6]Neste exemplo em específico, diversas outras regras consideradas pelo gerador proposto poderiam ser utilizadas, porém as mesmas foram suprimidas da Figura 5 por simplicidade.

mecanismo de geração, cada invocação recursiva é parametrizada por um parâmetro que indica a profundidade da recursão, a qual é decrementada a cada invocação. Quando o valor da profundidade chega a zero, somente regras terminais podem ser usadas, evitando assim novas chamadas recursivas. O algoritmo a seguir implementa a entrada do processo de geração de uma expressão.

Como pode ser visto, a função invoca a função quando deve interromper a recursão, e invoca a função para situações onde ainda é possível gerar termos recursivamente.

O exemplo apresentado através das regras de tipos com os espaços reservados pelos pontos de interrogação foi implementado através da função , cujo código pode ser visto a seguir.

Como pode ser notado, para gerar uma aplicação de função, primeiramente é preciso gerar um tipo , o qual é usado para gerar a expressão lambda em conjunto com o tipo , para então gerar um parâmetro para ser aplicado na função . Note a natureza recursiva dessa definição, através das invocações das funções e . O leitor pode perceber que para cada chamada recursiva, o parâmetro está sendo dividido ao meio, o que segue as convenções apresentadas nos exemplos de uso da biblioteca QuickCheck. Qualquer outra estratégia de redução do valor deste parâmetro poderia ser adotada, atuando apenas como forma de garantir a terminação do algoritmo recursivo.

No código a seguir é possível visualizar um programa gerado através do mecanismo proposto em que é aplicado o número 5 a uma expressão lambda.

A AST correspondente é apresentada na Figura 6.

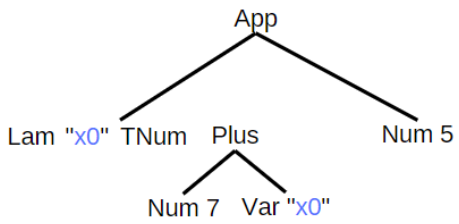

Figura 6. AST que representa a aplicação de um número à uma expressão lambda.

O trecho de código a seguir apresenta outro programa gerado pelo mecanismo proposto, o qual utiliza ainda outras regras de tipo não apresentadas na Figura 5[7]. Neste exemplo, é possível notar que um trecho maior dos construtores sintáticos é coberto por este caso de teste gerado.

O exemplo anterior mostra a geração de uma chamada de função em uma lista construída através do operador de construção de listas, além do uso de operadores aritméticos, condicionais e de ligação de variáveis.

[7]As regras de tipo triviais, como as dos operadores aritméticos (soma, multiplicação, etc.) foram omitidas deste artigo por questões de simplicidade.

5. Aplicação dos Testes Baseados em Propriedades

Uma função de exportação foi criada para aplicar os testes baseados em propriedades. Esta função transforma as ASTs em códigos Haskell, a qual é responsável por transformar as representações definidas como um ADT em uma string com a sintaxe concreta do Haskell. Esse passo é necessário, uma vez que os códigos serão testados diretamente pelo compilador GHC.

Com todo o aparato do gerador de código desenvolvido, foi possível automatizar o processo de realização dos testes, objetivando testar a compilação e execução dos códigos Haskell gerados, usando o compilador GHC com diferentes níveis de otimizações.

O automatizador de testes fica responsável por invocar o mecanismo de geração de tipos e expressões, gerando o código de um programa Haskell que é então gravado no sistema de arquivos do computador, para posteriormente realizar os testes das propriedades desejadas. Um exemplo de código final, gerado pela ferramenta desenvolvida e gravado no sistema de arquivos pode ser visto a seguir.

Como pode ser notado, o programa gerado faz uso de diferentes construções sintáticas, incluindo o uso do operador *let*, expressões aritméticas, expressões lambda, expressões condicionais e constantes numéricas e *booleanas*.

O programa gerado é, então, utilizado para fazer o teste das duas propriedades propostas neste trabalho:

1. Propriedade de *compilação*: cada programa gerado é compilado quatro vezes, uma sem usar *flags* de otimização, e outras três que incluem as *flags* de -O0 até -O2, comparando o resultado das compilações.

2. Propriedade de *manutenção de comportamento*: cada binário gerado após a compilação é executado, e o resultado final da avaliação da expressão gerada é comparada entre as diferentes execuções.

Para facilitar o acompanhamento dos testes, a ferramenta apresenta resultados no console, mostrando as estatísticas e retornos do GHC para cada nível de otimização. Além disso, as estatísticas são escritas em um arquivo de *log* para fins de análise posterior. Isso permite a verificação dos resultados de compilação e execução dos programas gerados nos diferentes cenários, sendo útil para detectar quais construções geraram os erros, tanto para o mecanismo de geração, quando para a ferramenta alvo dos testes.

A Figura 7 apresenta um exemplo parcial do arquivo que armazena as estatísticas resultantes da realização dos testes. Cada programa gerado é utilizado como entrada para os testes das duas propriedades citadas, observando os quatro cenários de níveis de otimização: -O0, -O, -O1 e -O2.

Através do uso da ferramenta gerada foi possível aplicar o gerador de códigos desenvolvido em conjunto com os testes baseados em propriedades através do QuickCheck. Os

```
1   Teste 1: Passed    Resultado: False
2   Teste 1: Passed    Resultado: False
3   Teste 1: Passed    Resultado: False
4   Teste 1: Passed    Resultado: False
5
6
7   Teste 2: Passed    Resultado: [9]
8   Teste 2: Passed    Resultado: [9]
9   Teste 2: Passed    Resultado: [9]
10  Teste 2: Passed    Resultado: [9]
```

Figura 7. Arquivo parcial de estatísticas de execução dos programas gerados aleatoriamente.

resultados da execução dos testes serão discutidos na próxima seção.

6. Análise dos Resultados

A partir das estatísticas coletadas foi possível sumarizar os resultados obtidos a partir da aplicação dos testes automatizados. Foram testados um total de 10.000 programas gerados aleatoriamente, dos quais a profundidade da recursão (parâmetro 'd' nos códigos anteriores) foi configurado para 5 em 5.000 dos casos de teste e para 15 nos outros 5.000.

Foi possível perceber claramente que a frequência de erros de compilação dos programas gerados com profundidade 15 é bem maior que aqueles gerados com profundidade 5. Dos 10.000 testes realizados, as falhas foram de 2 e 55 para as profundidades 5 e 15, respectivamente. As falhas apenas estão presentes quando utilizadas as *flags* O0 e O.

Com base nos resultados, foi realizada uma análise pontual do arquivo de estatísticas de compilação e execução, considerando apenas os programas que apresentaram falhas de compilação. Analisando detalhadamente cada programa que apresentou erro de compilação, foi notada uma característica em comum entre eles. Todos os programas que apresentaram erro possuem entre 400 mil e 8 milhões de caracteres, *i.e.*, são arquivos de tamanho substancial, o que atualmente não é uma prática comum no desenvolvimento de software, devido as técnicas de modularização. A Figura 8 apresenta a mensagem gerada pelo GHC ao tentar compilar um dos programas que apresentou erro.

```
[1 of 1] Compiling Main          ( 11_expressions_test.hs, 11_expressions_test.o )
ghc: sorry! (unimplemented feature or known bug)
  (GHC version 8.8.4 for x86_64-unknown-linux):
       Trying to allocate more than 129024 bytes.

This is currently not possible due to a limitation of GHC's code generator.
See http://ghc.haskell.org/trac/ghc/ticket/4505 for details.
Suggestion: read data from a file instead of having large static data
structures in code.
```

Figura 8. Mensagem de erro do GHC para um dos programas gerados e que apresentou falha de compilação.

Esse erro é um *bug* já conhecido pelos desenvolvedores desde 2010, com discussões ativas até o ano de 2022, o qual é desencadeado a partir da tentativa de alocação de grande quantidade de memória para armazenar dados estáticos presentes no código-fonte. Como indicado na mensagem, este erro é visto como uma limitação do GHC[8].

Perceba que este erro ocorre exclusivamente quando não é aplicado nenhum nível de otimização durante o processo de compilação. Esta observação é essencial, pois evidencia que a propriedade de compilação em questão, isto é, a ocorrência do erro, não se mantém quando as otimizações estão habilitadas.

Como resultado da verificação da propriedade de *manutenção de comportamento* não foi detectado nenhum erro ou comportamento distinto entre os diferentes níveis de otimização, ou seja, para aqueles programas que compilaram com todos os níveis de otimização, os resultados da execução apresentaram os mesmos resultados entre eles.

7. Conclusão

Neste trabalho foi apresentado um gerador de código aleatório baseado no sistema de tipos do Cálculo-λ, capaz de gerar códigos da linguagem Haskell para serem utilizados como entrada para testes baseados em propriedades. A abordagem leve[9] oferecida pelo QuickCheck permitiu experimentar diferentes designs e implementações do gerador, além de possibilitar os testes das propriedades de compilação e manutenção de comportamento após compilar os programas gerados com diferentes níveis de otimização do compilador GHC. É importante mencionar que durante o desenvolvimento do trabalho, foram realizadas diversas modificações no código base da ferramenta proposta, e para assegurar que as modificações estavam consistentes era necessário apenas a reexecução do conjunto de testes, ou seja, os programas gerados foram úteis para encontrar *bugs* no próprio gerador de código desenvolvido.

A partir das ferramentas desenvolvidas, *i.e.*, de geração de código e de testes, foi possível atingir os objetivos propostos neste trabalho: identificar falha no compilador em relação à propriedade de compilação e confirmar que o GHC manteve a propriedade de manutenção de comportamento. Apesar do erro encontrado já estar documentado para comunidade de desenvolvedores Haskell, a identificação foi importante, pois indica que a abordagem é promissora para o contexto de teste de propriedades em que foi executada.

Como direções futuras de pesquisa, pode-se citar:

- A utilização de expressões geradas para testar propriedades mais gerais de programas ou bibliotecas, permitindo encontrar *bugs* sem a necessidade de criar casos de teste de forma manual.

[8] Este erro já foi discutido pela comunidade de desenvolvedores do compilador GHC e esta discussão pode ser encontrada em ⟨https://gitlab.haskell.org/ghc/ghc/-/issues/4505⟩.

[9] A abordagem de testes utilizada é considerada leve em comparação com abordagens de verificação formal de compiladores.

- A aplicação de teste diferencial em compiladores ou ferramentas que possuem a mesma semântica. Além dessas, a própria extensão do gerador para contemplar mais construções sintáticas do Haskell, como uma forma de cobrir mais código alvo no compilador GHC poderia ser útil para descobrir novos *bugs*.
- A criação de testes baseados em mutações de código existente, combinando com os códigos gerados pela presente ferramenta.
- A formalização mecanizada do algoritmo de geração poderia desencadear também novas pesquisas nesta área.

Author contributions

Authors Guilherme Graeff, Samuel Feitosa and Andrei Braga contributed to the development of the code and writing the article. Authors Bráulio Mello and Denio Duarte contributed to testing, implementation adjustments and text refinement.

References

[1] YANG, X. et al. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 46, n. 6, p. 283–294, jun 2011. Disponível em: ⟨https://doi.org/10.1145/1993316.1993532⟩.

[2] BRITTO, G. A.; TEIXEIRA, L.; GHEYI, R. TSDolly: A program generator for TypeScript. In: *Proceedings of the 25th Brazilian Symposium on Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2021. (SBLP '21), p. 66–74. Disponível em: ⟨https://doi.org/10.1145/3475061.3475079⟩.

[3] CHEN, J. et al. A survey of compiler testing. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 53, n. 1, p. 1–36, feb 2020. Disponível em: ⟨https://doi.org/10.1145/3363562⟩.

[4] PAŁKA, M. H. et al. Testing an optimising compiler by generating random lambda terms. In: *Proceedings of the 6th International Workshop on Automation of Software Test*. New York, NY, USA: Association for Computing Machinery, 2011. (AST '11), p. 91–97. Disponível em: ⟨https://doi.org/10.1145/1982595.1982615⟩.

[5] FEITOSA, S. da S. *Strategies for testing and formalizing properties of modern programming languages*. Tese (PhD thesis) — Universidade Federal de Pelotas, Pelotas, RS, Brazil, 2019. Disponível em: ⟨https://guaiaca.ufpel.edu.br/handle/prefix/6280⟩.

[6] KRAUS, L. F. et al. Synthesis of random real-world java programs from preexisting libraries. New York, NY, USA, p. 108–115, 2021. Disponível em: ⟨https://doi.org/10.1145/3475061.3475087⟩.

[7] BARENDREGT, H. *The Lambda Calculus. Its syntax and semantics*. Amsterdam: North Holland, 1984. Disponível em: ⟨https://www.sciencedirect.com/bookseries/studies-in-logic-and-the-foundations-of-mathematics/vol/103/suppl/C⟩.

[8] PIERCE, B. C. *Types and Programming Languages*. Cambridge, MA: The MIT Press, 2002. Disponível em: ⟨https://mitpress.mit.edu/9780262303828/types-and-programming-languages/⟩.

[9] O'SULLIVAN, B.; GOERZEN, J.; STEWART, D. B. *Real World Haskell*. [S.l.]: O'Reilly Media, 2008.

[10] HUDAK, P. et al. A history of Haskell: Being lazy with class. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. New York, NY, USA: ACM, 2007. (HOPL III, 12), p. 12–1–12–55. Disponível em: ⟨https://doi.org/10.1145/1238844.1238856⟩.

[11] GITHUB. github.com, 2019. Disponível em: ⟨https://github.com/github/semantic/blob/main/docs/why-haskell.md⟩.

[12] GHC Optimizations. haskell.org, 2020. Disponível em: ⟨https://downloads.haskell.org/ghc/latest/docs/users_guide/using-optimisation.html⟩.

[13] CLAESSEN, K.; HUGHES, J. Quickcheck: A lightweight tool for random testing of Haskell programs. *SIGPLAN Not.*, Association for Computing Machinery, New York, NY, USA, v. 35, n. 9, p. 268–279, sep 2000. Disponível em: ⟨https://doi.org/10.1145/357766.351266⟩.

[14] FEITOSA, S. da S.; RIBEIRO, R. G.; BOIS, A. R. D. Generating random well-typed Featherweight Java programs using QuickCheck. *Electronic Notes in Theoretical Computer Science*, v. 342, p. 3–20, 2019. The proceedings of CLEI 2018, the XLIV Latin American Computing Conference. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/S1571066119300027⟩.