Forging a differential tester for Haskell compilers using Xsmith

EVERARD DE VREE, University of Twente, The Netherlands

Testing and verification are integral parts of software development that happen at every level of abstraction. When testing software written on one level, software engineers might consider the layer below as intransparent and assume it to not produce any unspecified behaviour. However, this is not always the case, and low-level compiler bugs can go unnoticed for years. One powerful strategy to investigate output of compilers is differential testing, or differential fuzzing. Recently, a general approach to differential testing was introduced to the field in the form of Xsmith: a potent tool that can be used to write fuzz testers in arbitrary languages. Xsmith has been implemented for a variety of programming languages, and its effectiveness has been demonstrated by the discovery of bugs in implementations of Racket, Dafny and WebAssembly. We extend that list by investigating the effectiveness and limitations of Xsmith when used to implement a differential tester for the Haskell programming language. These limitations are investigated by evaluating program generation speed and compiler coverage for the generated Haskell programs.

Additional Key Words and Phrases: differential testing, differential fuzzing, compiler validation, GHC.

1 INTRODUCTION

For software engineers building on the abstraction provided by a high-level programming language, compilers and interpreters are important tools. By providing semantic and syntactical analysis, and generating (optimized) machine code, they form a large part of the bridge between human mind and computer processor. However, compilers are not always free from imperfections, and cracks in the bridge can be both impactful and hard to find. This has made compiler validation an active research topic, where researchers explore a variety of testing strategies. One powerful strategy is differential fuzzing or differential testing, where the same programs are compiled with different compilers, to find differences in execution [17].

A recent development in differential testing is the publication of Xsmith. Hatch et al. [11] introduce their contribution as "a Racket library and domain-specific language that provides mechanisms for implementing a fuzz tester in only a few hundred lines of code." In order to generate programs that only fail differential tests in case of implementation errors, Xsmith limits itself to generating "conforming" programs. This term was coined by Hatch et al. [11] to describe inputs (programs) "that conform strictly to the language specification, as well as avoiding undefined, implementation-defined, or nondeterministic behavior."

We explore the use of Xsmith as a way to generate conforming programs in a functional programming language. Specifically, we implement a fuzz tester for Haskell and use cross-optimization testing strategies on inputs generated by Xsmith. For all compiler testing in this research, the Glasgow Haskell Compiler (GHC) is used to compile randomly generated programs using different optimization levels. This provides insight on the performance and limitations of Xsmith for functional programming implementations. Additionally, a thorough fuzzing campaign may find previously undiscovered bugs in implementations of the Haskell programming language.

The problem statement is summarized in the following research question: How can Xsmith be used to implement an effective fuzz tester for evaluating Haskell compilers? The answer to this question can be constructed by first answering the following sub-questions:

- RQ1 How effective can we make an Xsmith-based Haskell fuzz tester?
- RQ2 What limitations does Xsmith impose on the development of a Haskell fuzz tester?

2 RELATED WORK

This section provides an overview of related work in differential testing, GHC testing, and Xsmith-based fuzz testers.

2.1 Differential testing

Differential testing in software is a test oracle: a mechanism that decides whether a test is passed or failed. Finding or generating the right test oracle for the right circumstances is difficult and includes balancing effectiveness and computation cost, and balancing the evaluation load between human and computer.

2.1.1 History. Differential testing as a test oracle was formally introduced in 1988 by McKeeman [17] who introduced a prototype differential tester for testing C compilers. To illustrate the advantage of differential testing, he states the following: "Differential testing addresses a specific problem — the cost of evaluating test results." [17] This refers to the problems of evaluating possibly "millions of tests" to find bugs in software. Differential testing is an effective way to automate the process of finding interesting test results. However, different results for the same input programs might still not indicate an implementation fault, notably in cases where input programs execute instructions that are specified to produce non-deterministic or undefined behaviour.

2.1.2 Csmith. In 2011, the publication of Csmith provided an influential demonstration of the effectiveness of compiler fuzzing. By implementing overflow checks and using static analysis to generate programs free from undefined and unspecified behavoiur, Yang et al. [22] discovered over 325 previously unkown C compiler bugs. Per ACM citation metrics, their publication has been cited over 600 times, and Csmith has inspired numerous studies and publications. A recent example is the development of CsmithEdge, which relaxes the conservative static analysis of Csmith to find new bugs in C compilers and achieve greater code coverage in the compiler codebases [6].

2.1.3 Effectiveness. In a 2016 comparison, Randomized Differential Testing (RDT) was measured against Different Optimization Level

TScIT 40, February 2, 2024, Enschede, The Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(DOL) and Equivalence Modulo Input (EMI) using programs generated by Csmith. By keeping track of the bugs found by each testing strategy, Chen et al. [2] draw some conclusions that are relevant for this research:

- "RDT oracle is the strongest, whereas EMI oracle is the weakest."
- "Efficiency has the most significant impact on the effectiveness of compiler testing, and the effectiveness of generated test programs has the least significant impact."

Here, the strength of test oracles refers to which testing oracle finds the most bugs per program, efficiency refers to the number of programs that can be tested per time interval, and effectiveness of generated programs refers to the number of bugs found per tested program. The first conclusion supports the choice to use cross-compiler and cross-optimization testing for this research. The second conclusion provides insight for the evaluation of 'effectiveness' when answering sub-question RQ1: efficiency must be weighed heavily when balanced against other parameters.

2.1.4 Greybox fuzzing. In an effort to combat compiler immunization against fuzzers, some fuzzers have taken a more dynamic approach in the form of greybox fuzzing. Greybox fuzzers use the compiler codebase coverage from previously generated tests as feedback. This feedback is then used to maximize the coverage of newly generated test programs, by guiding the mutation of previously generated programs. Notable examples are the AFL and AFL++ frameworks [8, 23], and the libFuzzer library [19]. In a recent publication, the greybox fuzzer GrayC pioneers the use of "semantics-aware mutation operators" to avoid generating syntactically invalid programs with naive mutations; Even-Mendoza et al. [7] used GrayC to discover 25 previously unknown compiler and code analyser bugs. Additionally, they report achieving more middle- and back-end coverage for the LLVM compiler than other greybox and blackbox fuzzers.

2.2 GHC testing

Differential testing for GHC is not new. In 2011, Palka et al. [18] take a goal-directed approach to generating simply-typed lambda calculus programs in Haskell. The goal is a "target type", which is provided as input together with an "environment containing all variables and constants that can be used in the terms." Then, the syntax rules of lambda calculus are traversed backwards to obtain an expression that results in a term of the specified target type. The generated programs are then used as input for cross-optimization GHC tests, at which point a bug was discovered in GHC's strictness analyzer. Though the generated expressions can achieve high depths, they cover only a limited subset of possible Haskell programs due to the simple nature of simply-typed lambda calculus and lack of effect analysis. For example, values are limited to a small set of constants, and functions to list operations from the Haskell Prelude, while Hatch et al. [11] claim Xsmith fuzzers "can be featureful, generating correct code for conditionals, rich types, variable references, and so on."

2.3 Xsmith generators

Hatch et al. [11] provide Xsmith fuzzers for Racket, Dafny, Standard ML, WebAssembly as well as Python, Lua, and Javascript. However, the last three fuzzers were "not used in substantial fuzzing campaigns." Besides these out-of-the-box fuzzers, a thorough internet search only reveals the Xsmith-based R7RS fuzzer 'rattle-smith' but it is unclear whether it has been used succesfully [16]. The collection of existing fuzzers provides the possibility for replication studies, but leaves ample space for researching new implementations of Xsmith-based fuzzers. The implementation of a fuzzer for a functional programming language like Haskell will likely give rise to problems that have not been explored within the Xsmith domain.

3 GENERATING HASKELL PROGRAMS

In order to evaluate Xsmith in a functional context, we provide a simple Haskell fuzzer [4]. It is implemented in Racket, to make use of the Xsmith library.

3.1 Haskell

Haskell is a purely functional programming language, specified in the Haskell 98 Language report and the Haskell 2010 Language Report [13, 14]. The latter describes it as providing "higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes." Haskell's strong, static typing system guarantees that no type errors can occur at runtime. A conforming Haskell program can therefore only crash in a limited number of ways, like pattern-matching failure, unsafe arithmetic, or the use of other unsafe language features like partial functions or the Foreign Function Interface. Xsmith provides useful infrastructure for generating type-correct programs, meaning that generating error-free Haskell programs is relatively straightforward, provided that unsafe features are omitted from the Xsmith specification.

3.1.1 Syntax. In order to facilitate the many features mentioned at the start of this section, the Haskell 2010 Language report specifies a complex context-free syntax with more than 70 variables and over 150 rules [14]. However, the developed fuzzer does not implement all the specified grammar rules: it is limited to generating single argument lambda abstractions and function applications, as well as literals and one- and two-argument functions for certain data types. This code example shows the Haskell syntax for a simple, one-argument lambda function application: `(\x → x + 1) 5`
Here, `x` is the parameter and `x + 1` is the expression that defines the function body. In this case, the lambda abstraction simply increments its argument by 1, so this expression evaluates to 6. Lambda abstractions are powerful and can be nested indefinitely; In fact, the the GHC compiler desugars all multi-argument functions into nested

single-argument lambda abstractions. Listing 1 contains an example:

```
-- Haskell syntax
f x y = x + y
-- Desugared version
f =
  \x →
    \y → (+) x y
```

Listing 1. Example of Haskell syntax desugaring into lambda functions

3.2 Xsmith

Xsmith uses the Scheme library RACR to define a reference attribute grammar specification. Attribute grammars allow the addition of attributes to their rules and nonterminals. RACR allows referential attributes, which can be used for the "controlled rewriting" of an Abstract Syntax Tree (AST) after evaluation [1]. Xsmith makes use of this feature, together with choice objects to guide the generation of a random AST [9]. Choice objects have choice methods which are used to guide generation by filtering the grammar productions. Finally, Xsmith provides properties, which are "compile-time macros for generating attributes and choice methods and so are not themselves used during generation." [9] We do not directly use choice methods and attributes in the provided fuzzer: since Haskell eliminates the need of an effect analysis through its lack of side effects, the main area of interest for the development of the Haskell fuzzer is satisfying Haskell's type system. We use the `type-info` property provided by Xsmith for this. It is used to describe the relation between a node's type, and the types of its children.

3.3 Program generation

This section provides a detailed description of certain key features of the Haskell fuzzer like typing, unsafe functions, and structure of the generated programs. Additionally, this section includes an overview of Xsmith-related obstacles encountered throughout development.

3.3.1 Typing. The fuzzer emulates Haskell's typeclasses `Numeric`, `Real`, `Integral`, and `Fractional`. To do this, we first define each typeclass as a nonleaf Xsmith `base-type`. This means that these these types must eventually be resolved to a leaf subtype. Additionally, the `Real` typeclass is defined as a `real-type`, subtype of `numeric-type`. `Integral`, and `Fractional` become subtypes of `real-type` in order to emulate their `Numeric` and `Real` class constraints. Finally, we add `int-type` and `float-type` as subtypes of `integral-type` and `fractional-type`, respectively. These types are leaf types, meaning that no further subtypes can be created. Listing 2 contains the Racket code for these definitions. Each type is defined as a product of the `base-type` function, which takes a name, an optional supertype (default false) and leaf property (default true).

This hierarchy was designed to work with the Xsmith function `fresh-subtype-of`, which takes a type *t* and returns a type variable constrained to a subtype of *t*. This was assumed to return the first nonleaf subtype of *t*, which would make calling `(fresh-subtype-of number-type)` an elegant way to constrain all children nodes of a Numeric function to either Ints or Floats. This prevents the generation of expressions constrained `(Integral a, Fractional a)` ⇒a, which are not printable. However, in practice `fresh-subtype-of` *t* does not seem to constrain *t* to anything. Initially, the Xsmith development team identified this as a potential bug [21]. However, upon further review, it was clarified that this was intended behaviour, although implementing the described behavior as an optional feature could be beneficial. We now implement types of numeric functions by explicitly defining their type variables as integral or fractional: `(fresh-type-variable integral-type fractional-type.`

```
(define number-type
  (base-type 'number #:leaf? #f))
(define real-type
  (base-type 'real number-type #:leaf? #f))
(define integral-type
  (base-type 'integral real-type #:leaf? #f))
(define int-type
  (base-type 'int integral-type))
(define integer-type
  (base-type 'integer integral-type))
(define fractional-type
  (base-type 'fractional real-type #:leaf? #f))
(define float-type
  (base-type 'float fractional-type))
```

Listing 2. Hierarchy of numeric types used by the fuzzer

3.3.2 Program structure. A crucial step in differential testing is comparing program output across different systems under test. In order to print output that contains information about the program, we simply prepend the generated program with `main = print`. The AST is then rendered starting from a single argument lambda function application whose body can contain a range of expressions, including more lambda function applications. It is important that lambda functions are always applied to enough arguments, since partially applied functions are not valid arguments for `print`. In order to avoid generating partially applied functions, we restrict the generation of lambda expressions to child nodes of procedure applications. This ensures that lambda functions are always applied to an argument, guaranteeing a printable result.

Listing 3 contains a simplified code example showing how the `add-to-grammar` form is used to add the productions that form the root nodes of our AST to the Haskell fuzzer `raskell`. For every production, we specify its name, its supertype, and a list of child nodes. Assuming program generation is specified to start at the `Program` node, it is impossible for a `LambdaExpression` to be generated without being provided an argument from a parent `ProcedureApplication` node. This is achieved by not marking `LambdaExpression` as a subtype of `Expression`. This means that `LambdaExpressions` can only be used to fill the corresponding `LambdaExpression` hole nodes, which are only created by `ProcedureApplication` nodes. Additionally, the `print []` expression is normally ambiguous since the `Show` class constraint for the use of the `print` function is not satisfied by an empty list. This happens because the `Show` instance of a list is normally derived from the `Show` instance of a list's element type. However, the empty list has no element type unless explicitly specified, resulting in an ambiguous type variable error. We solve this problem

```
(add-to-grammar
 raskell
 [Program #f ([λ : ProcedureApplication])]
 [ProcedureApplication Expression
        ([procedure : LambdaExpression]
         [argument : Expression])]
 [LambdaExpression #f
        ([parameter : FormalParameter] [body : Expression])])
```

Listing 3. Overview of the structure at the root of every AST

by enabling GHC's extended type defaulting rules through the `-XExtendedDefaultRules` flag. This allows GHC to default the empty list's element type to the unit type `()`, resolving the error.

3.3.3 Unsafe functions. Generating programs that result in runtime exceptions complicates differential testing. Halting the program early prevents code, that was possibly expensive to generate, from being evaluated. In order to avoid this bottleneck, we avoid certain unsafe functions by providing our own safe versions or importing them from the Safe library. For example, the `div` function is unsafe because it throws a `divide by zero` exception for any x in `x `div` 0`. We provide a custom function to replace it that returns zero when the divisor is zero.

```
safeDiv :: Integral a ⇒ a → a → a
safeDiv x 0 = 0
safeDiv x y = div x y
```

Listing 4. safeDiv definition

Another set of unsafe functions is the partial functions included in Haskell's standard module Prelude. It includes list operations like the `head` function, which returns the first element of a list. If the provided list is empty, `head` throws an exception. Likewise, the list index function `!!` returns an `index too large` exception when the list it is applied to is shorter than the provided index. For these cases we use alternatives from the Safe module that take an extra fallback argument which is only returned when the function would otherwise return an exception.

3.3.4 Canned components. To simplify fuzzer development, Xsmith provides "canned components". These are implemented as library which provides functions to quickly add common expressions to a language component. The canned-components library adds grammar productions to their relevant type-info attributes, leaving essentially only defining the render-info attribute to the user. Besides its use as a source of valuable examples, this library was used to implement basic boolean logic, list literals, and safe list reference. To do this, we simply enable the `Boolean` and `MutableArray` flags, and provide the correct Haskell syntax through the render-info property. For other applications, the 'canned' grammar specification proved to be too inflexible to implement the simple, custom program structure explained in section 3.3.2. For example, the provided LambdaExpression can be generated outside of being a child node of a ProcedureApplication node. This leads to the generation of partially applied lambda functions, which are difficult to evaluate during testing.

3.4 State of the fuzzer

The Fuzzer is designed to generate type-correct Haskell programs conforming to the Haskell 2010 language specification. It generates basic expressions and a selection of functions from the Prelude module [3], a standard Haskell module available to all Haskell programs by default.

- Single argument lambda functions and function application
- Boolean literals (True, False) and functions (||, &&, not)
- Numeric literals (Int, Float)
- Numeric functions (+, -, *, negate, abs, signum, fromInteger, subtract, even, odd, gcd, lcm, ^, ^^, fromIntegral, realToFrac)
- Safe implementations of integral functions (quot, rem, div, mod)
- Other integral functions (toInteger)
- Fractional division: (/)
- List literals
- Safe implementations of partial list functions (head, last, tail, init, !!)

4 MEASUREMENT TOOLS

This section gives a brief overview of the tools used, clarifying their role in the research, as well as any possible technical limitations.

4.1 Test driver

The Flux research group provides a harness to run a fuzzing campaign on their distributed testbed EMULAB [10]. At its core, it contains a Python script for running the fuzzer, feeding the generated program to different compilers, executing the binary and then comparing the output. This script was used with custom configuration files for each of the experiments in section 5.

4.2 Glasgow Haskell Compiler

We limit the evaluation of compiler codebase coverage and distribution to GHC, because of its flexible build system and extensive documentation. It is a mature open source compiler and interactive interpreter, in compliance with the full Haskell 2010 language specification.

Attempts at alternative implementations of the Haskell 2010 standard include the Utrecht Haskell Compiler (UHC) [5] and the LLVM Haskell Compiler (LHC) [12]. These compilers were initially considered for cross-compiler testing, but ultimately rejected for several reasons. Even though UHC supports large parts of the Haskell 98 and 2010 standards, we do not use it for tests because official documentation appears to be offline and it lacks recent contributions: UHC's last commit was made on the 1st of January 2018. Additionally, we do not consider LHC a viable option as it has been limited to compiling "very simple programs for now" since 2014 [12].

4.2.1 Structure of the compiler. In order to reason about coverage, it is important to understand GHC's general structure. Based on an article on GHC's general structure by lead developers Peyton-Jones and Marlow [15], we divide the compiler into three main parts:

- The front end, which parses, renames and type-checks the Haskell source code before desugaring it into a simplified core language called GHC Core. The Core language is a small,

expressive, and strongly typed intermediate representation that GHC uses for analyses and optimizations.
- The Simplifier, which applies a series of transformations to the Core language, such as inlining, strictness analysis, and simplification, to improve the performance and correctness of the generated code.
- The back end, which converts the optimized Core code into STG, before being translated into C–. Finally, the C– code is translated to native machine code, C code, or LLVM code which is passed to the LLVM compiler. Additionally, since version 9.6.1, GHC can be built to compile to WebAssembly and JavaScript. However, these features should be regarded as technology previews and are not available in standard builds of GHC 9.6.1 [20]

4.3 Haskell Program Coverage

GHC ships with the Haskell Program Coverage (HPC) toolkit, which provides code coverage information for Haskell code. It can be used to obtain detailed information of Haskell programs compiled with GHC's `-fhpc` flag. Since GHC is bootstrapped with an older version of GHC, it is possible to instrument the compiler source code with the code coverage flag enabled. GHC's build system Hadrian was used to target the the Haskell files for the GHC modules and binary with this flag. The resulting GHC binary outputs a .tix file every time it compiles a Haskell program. This file details how many of the compiler's declarations, alternatives, and expressions were reached during execution.

The SUM and COMBINE tools provided by HPC can then be used to combine multiple .tix files into one, in order to obtain cumulative coverage information. These tools are used to calculate the total coverage of the Haskell fuzzer over multiple test runs. The outputs of these tests can then be used as input for the REPORT and MARKUP tools to provide detailed, human-readable output in text or HTML, respectively.

5 EXPERIMENTAL SETUP

We take an empirical approach to measuring the performance and effectiveness of Xsmith and the Haskell fuzzer.

5.1 Parameters

In order to evaluate the fuzzer, it is important to select a fitting metric for fuzzer effectiveness. In some related work, fuzzers are categorized by the amount of bugs they can find in a certain time frame [2]. However, this method is not a reasonable metric for evaluating the developed Haskell fuzzer, as its scope is likely too small too discover any compiler bugs at the time of publication. Therefore, we evaluate the effectiveness of the Haskell compiler according to the following parameters:

(1) efficiency (at different levels of AST depth)
(2) compiler codebase coverage
(3) compiler coverage distribution

A small cross-optimization testing campaign will still be conducted: randomly generated Haskell programs will be compiled using GHC's 3 optimization levels. The 3 resulting binaries will be executed and tested for different output, since varied outcomes for the same input program indicate a compiler bug. Each experiment will be conducted in the same testing environment: A Linux Mint 21.3 computer with an AMD Ryzen 5 2600X Six-Core Processor and 16GB of 3200 MHz DDR4 memory installed.

5.2 Efficiency

To investigate the relationship between program generation speed and generated program complexity, the Haskell fuzzer will be executed with different values for the Xsmith command line argument `--max-depth`. The fuzzer will run for 2 hours each at levels 1, 3, 5, and 7 to obtain a varied performance profile. Together with information about compiler coverage achieved at different AST depths, this can provide insight into the ratio between computational cost and effectiveness.

5.3 Compiler coverage

To investigate the effect of including different expressions in the fuzzer, we run the Haskell fuzzer for 2 hours at every AST max depths 1, 3, 5, and 7, using the HPC-enabled build of GHC 9.8.1. The coverage file of every run is aggregated and compared across AST depths. Additionally, the effect of including expressions will be evaluated by comparing the coverage results of an old, simple version of the fuzzer against the latest one.

5.4 Coverage distribution

We convert the aggregated .tix file from the `--max-depth 7` coverage experiment to XML format with the HPC REPORT tool. In order to gain a high-level overview of the coverage data, all the data for subdirectories of GHC is aggregated into a single entry. For example, the coverage information for `ghc-9.8.1/GHC/Hs.Binds` and `ghc-9.8.1/GHC/Hs.Decls` is summed and then stored under the `Hs` component. These components can then be assigned to the front-, middle-, or back-end of the GHC compiler on a best-effort basis. The difference in coverage between the compiler parts can be used to reason about the validity of Xsmith's claim to fame: finding deep semantic bugs [11].

6 RESULTS

This section presents an examination of the results obtained by executing the specified experiments.

6.1 Program generation speed

The results of the experiment exploring program generation speed at different AST depths are detailed in table 1. The 'AST depth' column corresponds to the max-depth option provided to the Xsmith Command Line Interface. 'Programs generated' provides the number of programs that were generated over a 2 hour testing period, while 'Xsmith errors' indicate the amount of times Xsmith crashed or timed out during the test. 'GHC errors' indicates the amount of generated programs that failed to compile.

There is no significant difference in generation speed between depths 1 and 3. This indicates that the cost of increasing the AST complexity at these levels is negligible compared to the overhead associated with program generation in general. At depth 5, generations speed is reduced approximately twofold. At depth 7, program

AST depth	Programs	Xsmith errors	GHC errors
1	1054	0	0
3	1049	0	0
5	470	19	10
7	168	12	3

Table 1. Number of programs generated in 2 hours at different AST depths

generation speed further decreases with a factor of approximately 3. Xsmith and GHC errors start to occur at depth 5 and higher. All GHC errors are associated with two-argument Numeric functions (e.g. `subtract`) generating one Fractional argument (e.g. a Fractional literal) and one Integral argument (e.g. application of the `div` function). These errors are likely a result of the rewriting of the type-info property for numeric functions as discussed in section 3.3.1.

6.2 Compiler coverage

The results of the experiment exploring GHC compiler coverage are available in table 2. There is a clear division between depths 3 and 5, but the coverage information is otherwise similar. This indicates that some compiler code paths are impossible to reach for smaller programs, even if they are generated quicker. The increase in depth from 5 to 7 does not increase compiler coverage as dramatically: it appears that the fuzzer can exhaust most possible code paths in 2 hours at AST depth 5.

The cross-optimization fuzzing campaign did not uncover any compiler bugs. In 4 hours, 403 conforming programs were generated, all of which maintained consistent results across different compiler optimization levels.

6.3 Compiler coverage at different fuzzer complexity

A comparison between the results of a simple version and more complex version of the fuzzer is provided in table 3. The simple fuzzer supports integer and boolean literals, and the following functions: `+`, `-`, `*`, `safeDiv`, `<`, and `>`. There is slight increase in all coverage metrics for the complex version of the fuzzer. This means that extending the fuzzer to cover more of Haskell's language specification has a positive effect on compiler coverage. However, the small size of the effect coupled with the fact that the the compiler codebase is not nearly fully covered indicates that there is still significant room for improvement in the fuzzer's features.

6.4 Coverage distribution

Table 3 contains a selection of the results of aggregating and simplifying coverage information after 2 hours of generating programs at depth 7. Front-end components like Parser, Rename, and Tc (Typechecker) do not show large differences in coverage compared to the middle-end Core module or back-end modules like the Cmm and Stg modules. This could offer support to the claim that Xsmith-generated programs can reach "deep" code paths, but this would require a more extensive analysis of the functionalities and scope of GHC's submodules.

7 CONCLUSION

We conclude that Xsmith provides useful infrastructure for writing a differential tester for a functional programming language like Haskell. The use of attribute grammars provides a straightforward way of satisfying the type checker. In a statically typed and purely functional context, this greatly simplifies generating conforming programs. Xsmith's canned-components library provides a practical and educational interface, but can be limited in flexibility. Additionally, we find that, for the provided fuzzer, limiting the max AST depth speeds up program generation and slightly decreases compiler coverage. Further research with a more complete fuzzer is required to draw conclusions about the nature of this effect. Finally, considering the demonstrated importance of developing a featureful fuzzer and Xsmith's previous successes [11], we believe that with adequate future research, Xsmith can play an important role in increasing the reliability of the Glasgow Haskell Compiler.

8 DISCUSSION

The conducted experiments indicate that the fuzzer built for this study was likely too small to conclusively determine whether the depth of the AST has meaningful effects beyond enabling all grammar productions to be used. Coupled with the fact that compiler coverage increases with the amount of fuzzer features, we emphasize the need for further research to fully understand the impact of Xsmith's AST depth on fuzz testing. Future works could focus on expanding the feature set of the Haskell fuzzer. This would significantly improve its effectiveness in verifying the correctness of Haskell compilers by either reporting bugs or confidently verifying that they are rare.

REFERENCES

[1] Christoff Bürger. 2015. GitHub repository - RACR documentation. https://github.com/christoff-buerger/racr last commit d6f678d1639d1dc771c49720aec2db657f5f3d19.

[2] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin Texas, 180–190. https://doi.org/10.1145/2884781.2884878

[3] Core Libraries Committee. 2023. *Hackage documentation - Prelude.* https://hackage.haskell.org/package/base-4.19.0.0/docs/Prelude.html version 4.19.0.0.

[4] Everard de Vree. 2024. GitLab repository - Xsmith research project. https://gitlab.utwente.nl/s2516268/research-project-2023-1B

[5] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. [n. d.]. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell* (Edinburgh Scotland, 2009-09-03). ACM, 93–104. https://doi.org/10.1145/1596638.1596650

[6] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering* 27, 6 (July 2022), 129. https://doi.org/10.1007/s10664-022-10146-1

[7] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 1219–1231. https://doi.org/10.1145/3597926.3598130

[8] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[9] William Hatch, Pierce Darragh, and Eric Eide. 2023. Xsmith Reference. https://docs.racket-lang.org/xsmith/Xsmith_Reference.html Xsmith version 2.0.6.

[10] William Hatch, Pierce Darragh, and Eric Eide. 2024. GitLab repository - Xsmith harness. https://gitlab.flux.utah.edu/xsmith/harness last commit 68cc082b01d4758d0688be9d789daef2e03823f7.

[11] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. 2023. Generating Conforming Programs with Xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, Cascais Portugal, 86–99. https://doi.org/10.1145/3624007.3624056
[12] David Himmelstrup. 2014. *Notes on the LHC: The New LHC*. http://lhc-compiler.blogspot.com/2014/11/the-new-lhc.html
[13] Simon Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries The Revised Report*.
[14] Simon Marlow (Ed.). 2010. *Haskell 2010 Language Report*.
[15] Simon Marlow and Simon Peyton Jones. 2012. *The Architecture of Open Source Applications, Vol 2*. Chapter 5: The Glasgow Haskell Compiler.
[16] Paulo Matos. 2020. GitHub repository - pmatos/rattle-smith. Retrieved September 2023 from https://github.com/pmatos/rattle-smith
[17] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.
[18] Michał H. Pałka, 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*. ACM, Waikiki, Honolulu HI USA, 91–97. https://doi.org/10.1145/1982595.1982615
[19] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. 157–157. https://doi.org/10.1109/SecDev.2016.043
[20] GHC Team. 2020. *Gitlab repository — Glasgow Haskell Compiler User's Guide*. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/9.6.1-notes.html
[21] Guy Watson. 2024. *Xsmith-dev mailing list archives*. http://www.flux.utah.edu/listarchives/xsmith-dev/msg00109.html
[22] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532
[23] Michael Zelowski. 2020. *American fuzzy lop*. https://lcamtuf.coredump.cx/afl/ version 2.52b.

AST depth	Programs	Errors	Expressions (%)	Boolean (%)	Alternatives (%)	Local decl. (%)	Top decl. (%)
1	753	1	14.68	2.97	10.32	18.09	21.09
3	752	0	14.78	3.05	10.42	18.13	21.24
5	269	28	15.48	3.62	11.22	19.27	21.85
7	202	34	15.49	3.60	11.23	19.27	21.86

Table 2. Compiler coverage results for generating Haskell programs at different maximum AST depths.

Fuzzer	Programs	Errors	Expressions (%)	Boolean (%)	Alternatives (%)	Local decl. (%)	Top decl. (%)
Simple	241	0	15.06	3.32	10.84	18.89	21.52
Complex	269	28	15.48	3.62	11.22	19.27	21.85

Table 3. Compiler coverage results for generating Haskell programs with fuzzers of different complexity

Module	Exprs (%)	Booleans (%)	Guards (%)	Conds (%)	Quals (%)	Alts (%)	Local (%)	Top (%)
Cmm	18.41	29.34	27.78	30.51	77.78	16.50	35.14	20.02
CmmToAsm	10.07	16.40	15.12	21.52	50.00	8.75	16.74	19.30
CmmToLlvm	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Core	15.71	25.06	25.50	17.20	10.00	16.85	21.71	26.17
Hs	3.55	8.33	8.00	10.00	0.00	2.74	2.14	1.53
HsToCore	10.71	18.86	19.86	15.09	0.00	8.96	11.30	20.14
Parser	15.43	15.57	17.43	12.28	0.00	8.54	23.18	14.96
Rename	13.83	17.55	17.11	26.32	0.00	9.90	18.21	28.29
Runtime	2.59	4.58	5.00	3.85	0.00	4.63	2.04	9.02
Stg	13.87	18.97	19.11	21.43	0.00	11.53	16.08	19.92
StgToCmm	16.57	33.97	34.17	35.29	14.29	11.24	29.88	40.72
Tc	14.42	23.18	23.86	21.19	9.52	10.76	14.95	28.76
Types	22.66	19.54	19.63	19.35	0.00	14.33	21.34	22.69

Table 4. Simplified overview of compiler coverage per GHC component after 2 hours of fuzzing at max-depth 7