Type-directed Generation and Shrinking for Imperative Programming Languages

Samuel Bodin
sbn15005@student.mdh.se
Supervisor
Daniel Hedin

Joel Söderman
jsn14012@student.mdh.se
Examiner
Björn Lisper

Västerås
Mälardalen University
Academy of Innovation, Design and Engineering
Abstract

Optimizing compilers are large and complex systems, potentially consisting of millions of lines of code. As a result, the implementation of an industry standard compiler can contain several serious bugs. This is alarming given the critical part that software plays in modern society. Therefore, we investigate type-directed program generation and shrinking for the testing of compilers for imperative programming languages. We implement a type-directed generator and shrinker for a subset of C in Haskell. The generator is capable of generating type-correct programs that contain variable mutations and challenging program constructs, such as while-statements and if statements. We evaluate the quality of the generated programs by measuring statement coverage of a reference interpreter and by comparing the results to reference test suite. The results show that the generated programs can surpass the test suite in terms of quality by one percentage point of coverage. To test the bug finding capabilities of the generator and the shrinker, we use them to test seven interpreters and manage to identify a total of seven bugs. Furthermore, we test the same interpreters with the reference test suite and find only three bugs. The implications of the results lead us to question the reliability of code coverage as a measure of quality for compiler test suites. Finally, we conclude that type-directed generation and shrinking is a powerful method that can be used for the testing of compilers for imperative programming languages.
## Contents

1 Introduction ................................................................. 4

2 Background ................................................................. 6
   2.1 Embedded domain-specific languages ................................. 6
   2.2 Testing .................................................................. 6
   2.3 QuickCheck and property-based testing .............................. 6
   2.4 Differential testing ..................................................... 7
   2.5 The compiler pipeline .................................................. 7
   2.6 Testing compilers ....................................................... 8
   2.7 Differential testing of compilers ...................................... 9

3 Language ............................................................................. 10
   3.1 Grammar .................................................................. 10
   3.2 Type system ................................................................ 11
      3.2.1 Programs and functions ........................................... 11
      3.2.2 Expressions ......................................................... 11
      3.2.3 Statements .......................................................... 12

4 Generation ............................................................................. 14
   4.1 The generator .............................................................. 15
   4.2 Program output .............................................................. 16

5 Templates ............................................................................. 17

6 Evaluation: statement coverage ............................................ 19

7 Shrinking ............................................................................. 20
   7.1 The shrinker ................................................................. 20
   7.2 Optimization ................................................................. 21

8 Evaluation: differential testing ................................................ 22
   8.1 The tests .................................................................. 22

9 Discussion ............................................................................. 26
   9.1 Statement coverage ........................................................ 26
   9.2 Differential testing ........................................................ 26
   9.3 Statement coverage for compilers ...................................... 27
   9.4 Type-directed generation and shrinking vs. test suites ........... 27
   9.5 Limitations: the generator ................................................. 28
   9.6 Limitations: the shrinker ................................................. 29
   9.7 Controlled error generation .............................................. 31

10 Related work ....................................................................... 33
   10.1 Testing C compilers ....................................................... 33
   10.2 Testing functional compilers ........................................... 34

11 Conclusions ....................................................................... 35
The grammar above defines the abstract syntax for our language.

Type rules for expressions.

Type rules for statements.

The default template which sprung out of the development phase of the generator.

A generated program mainly consisting of nested binary operations.

A program generated using a template where the paramcount interval was set to \([7, 7]\).

A generated program with nested while-loops and if-statements.

Statement coverage of the reference suite and the generated suite.

The output of the shrinker when given the expression \((1 + 2) \times (3 - 4)\).

Results of testing the interpreters with the generator and the shrinker.

Testing the first interpreter: command-line output showing the bug-triggering program.

Testing the first interpreter: command-line output showing the shrinking phase.

Results of testing the interpreters with the reference test suite.

A bug-triggering program.

f inlined at the call site.

A minimal bug-triggering program.

A chain of function calls.

Function that takes several parameters and consists of several statements.

Once f has been shrunken to the rightmost form, it can be inlined at the call site.
1 Introduction

To generate high-performance programs, industry standard compilers for statically typed programming languages [1] have to optimize the input source code. The optimization stage of modern compilers typically consists of a large set of transformations; the online documentation for the GCC compiler suite [2] states 72 different optimization passes. Several of these passes are platform specific, with the aim of fine-tuning the output of the compiler for the instruction set of the targeted machine’s hardware architecture. Given the size of some modern instruction sets, such as Intel’s x86 [3], which consists of hundreds of instructions, it is not hard to see that engineering an optimizing compiler can be a challenging task.

Because of the challenges of source code optimization, modern compiler implementations are often large and complex. In early 2015 the code base of the previously mentioned GCC compiler suite was estimated to consist of over 14 million lines of code [4]. Like in all other areas of software, having a large and complex code base makes it difficult for developers to maintain the correctness of the implementation. Given the critical function that software plays in our society, a bug in a compiler that results in erroneous code generation could have catastrophic effects. It is therefore important to find and eliminate any bugs in compilers used in the software industry today.

A conventional software engineering method used to discover and identify bugs in large code bases is testing. Typically, testing is performed by exposing an implementation to a set of input test cases, a test suite. If the result of a test case is something other than expected, a bug has been found. This method of testing is problematic for several reasons:

- Collecting a set of test programs that is large enough is a time-consuming process.
- It is difficult for developers to write programs that cover all corner cases.
- The implementation will be subject to more or less static test data, which makes it possible for bugs to remain undetected for long periods of time.

A method that does not suffer from these drawbacks is randomized property-based testing, where a tool is used to automatically generate test cases from a specification of the expected behavior of the program. How the program is supposed to behave is communicated to the tool by specifying a set of properties that always should hold true. Given a set of properties and a program to test, the testing tool will generate random input data and try to find counterexamples that expose inconsistencies in the implementation.

However, using randomized property-based testing to test compilers is challenging. Firstly, the generation of randomized input data is non-trivial, since programs have a precise structure for the compiler to consider them valid. When testing a compiler for a statically typed programming language, this means that the randomly generated programs have to be type correct. Moreover, the generation process has to avoid creating programs that either are non-terminating or trigger undefined behaviors. Secondly, a compiler is a program that translates a source program into an executable program. The property we would like the compiler to satisfy is that the source program and the resulting executable are semantically equivalent. There are two issues with this formulation:

1. Defining semantic equivalence between two potentially very different execution models is challenging.
2. Semantic equivalence, in general, is undecidable.

The standard solution to (1) is to test the compiler against a reference implementation, i.e., an oracle. This allows us to test the correctness of the compiler by checking semantic equivalence on the compilation product, i.e., on the produced executables, to verify that the compiler and the reference agree. Even though this simplifies the formulation of semantic equivalence, it remains undecidable and must be approximated. There are several ways to do this, e.g., via abstract interpretation or via testing using observable equivalence on terminating runs. A
problem with this approach is that it relies on the fact that an oracle is available and also that it is correct. For most compilers, there are no perfect implementations available, and we have to relax the requirement for correctness. Still, the method is proven effective.

Pałka et al. [5] have developed a method for generating type-correct randomized programs. Their generation algorithm operates by successively applying rules from the type system for simply-typed lambda calculus in a bottom-up, goal-oriented fashion. The generated programs are compiled with and without optimization and compared by observable equivalence on terminating runs. If the runs yield different outputs, an error in the implementation of the optimization stage has been found. When an error is found, they apply a shrinking scheme to minimize the source code - simplifying the program, while still making sure that an error is triggered. This process, also known as shrinking, is repeated until a minimal test case is found. The method of [5] is based on a type system for simply-typed lambda calculus, which means that they do not handle mutable state. While this allows for testing of functional languages, most languages are imperative in the sense that they allow for mutable state.

Our contributions are as follows. We adopt the method of [5] and implement a type-directed generator and shrinker for a subset of C# in Haskell. The generator is capable of generating type-correct programs containing the following constructs: while-loops, if-statements, function calls, variable declarations, and assignments, return statements, as well as basic arithmetic and comparison operations. Thus, the generator respects the rules of the type system and considers the context in which individual statements and expressions are generated. The generation algorithm itself operates in a mostly top-down manner, meaning that the program generation starts in the outer layers and works its way down to individual statements and expressions. As seen in [5], randomized program generation can result in large and complex programs that are hard to reason about. We address this by implementing and applying a shrinking procedure to minimize any bug-triggering programs. The shrinker is capable of shrinking any bug-triggering programs that the generator produces without introducing type-errors, because, like the generator, the shrinker considers the rules of the type system as well the current type environment. The shrinker is optimized for a fast shrinking: it employs a greedy heuristic, essentially prioritizing large reductions over small reductions, and removes whole functions when they are no longer used.

In line with the approach of [5], our generator operates according to a set of heuristics governing the quality of the generated programs. To measure the quality of our generator, we compare code coverage [7] of a generated test suite to that of a reference test suite. More specifically, we compare their statement coverage on an interpreter written in C# with JetBrains’ tool dotCover [8]. The results of this comparison show that the generated suite reaches a higher coverage than the reference suite, but only by one percentage point. Furthermore, to evaluate the bug-finding capabilities of the generator and the shrinker, we perform differential testing of seven anonymized interpreters from a compiler course at MDH. For comparison, we expose the same interpreters to the previously mentioned reference suite. The results of the tests show that the reference suite only triggers three errors in two of the interpreters, whereas the generator triggers seven errors in five of the interpreters. Moreover, the shrinker turns out to be a powerful tool in identifying the cause of these errors. The implications of these results lead us to question the reliability of code coverage as a measure of quality for compiler test suites. After a thorough discussion, we conclude that type-directed generation and shrinking can be used successfully for the testing of compilers for imperative programming languages.
2 Background

In this section, we present terms and concepts that we believe readers should grasp to understand our work. We begin by introducing testing concepts which are fundamental, followed by a brief presentation of the compiler pipeline. Finally, we present testing in the context of compilers.

2.1 Embedded domain-specific languages

Domain-specific programming languages, or DSLs, are languages designed for solving problems in a certain domain. An effective domain-specific language gives users the ability to express their algorithms elegantly within the problem domain. Typically, domain-specific languages are constrained to their domain, and not suitable for general purpose programming; increased expressiveness in one domain comes at the expense expressiveness in other domains. Thus, domain-specific languages are seldom Turing complete. Some examples of popular domain specific languages are SQL, used for handling relational databases; LaTeX, a typesetting language; and HTML, a markup language for writing web pages. Domain-specific languages are divided into two categories: domain specific languages and embedded domain-specific languages, or EDSLs. What separates EDSLs from DSLs, is that while DSLs require separate compilation or interpretation, EDSLs can be embedded in a host programming language. In the functional programming community, it is common to design libraries in terms of small EDSLs. Features like higher-order functions, make functional programming languages particularly suitable for writing EDSLs [9].

2.2 Testing

Testing is a conventional software engineering method used to discover and identify bugs in large code bases. Typically, testing is performed by exposing an implementation to a test suite, i.e., a set of input test cases. If the result of any of the test cases is something other than expected, a bug has been found. This method of testing is problematic for several reasons:

• Collecting a set of test programs that is large enough is a time-consuming process.

• It is difficult for developers to write programs that cover all corner cases.

• The implementation will be subject to more or less static test data, which makes it possible for bugs to remain undetected for long periods of time.

2.3 QuickCheck and property-based testing

A testing method that does not suffer from these drawbacks is property-based testing. It revolves around specifying the behavior of a program, and then randomly generating input, to find counterexamples for which the behavior is not consistent with the specification. The specification of the behavior consists of a set of properties for which the output of a program always should hold true, hence the name property-based testing. The specification for a program that sorts a list of numbers can be defined by two properties: the output list should be sorted, and the output list has to contain the same elements as the input list. Since test cases are generated automatically, property-based testing is not as time-consuming for developers as writing individual tests by hand. Moreover, the non-determinism and dynamism of pseudo-random generators [10] is likely to produce test data that would never have been written by a human. Because of this, property-based testing can sometimes find bugs that hand-written tests would never have found.

Property-based testing was introduced in 2000 with QuickCheck [11], testing library for Haskell [12], written by Koen Claessen and John Hughes. The library is lightweight; the initial release consisted of about 300 lines of Haskell code. Despite its humble size, users of the library are given control of all the nuances of test case generation, property specification, and other test parameters. Both the generators and the test harness are written
using composable EDSLs, providing users a simple and concise way of writing automated randomized tests.

With QuickCheck, a property-based test for a program can be constructed as follows:

1. Define pseudo-random generators for the set of data types that act as input for the program to be tested.
2. Define a set of properties that should be tested.
3. Run the test by providing QuickCheck the program to be tested, the generators and the properties.

QuickCheck will then perform some tests, generating random input data, trying to find a counterexample that will prove the tested program to be incorrect. If a counterexample is found, it will be reported to the user.

Apart from property-based testing, QuickCheck provides test-case shrinking. When shrinking is enabled, QuickCheck will try to minimize the found counterexample to find the smallest input data possible that will trigger the same bug. This functionality is helpful for developers to find the cause of an error in a counterexample. For many domains, where counterexamples can be large and complex, this functionality is necessary. This is especially true for compilers: randomly generated programs are not only large and complex but because they are produced without concerns of expressing a specific intent, they are inherently challenging for developers to read. Pallka et al. [5] and Midtgaard et al. [13] have successfully applied shrinking for the testing of compilers.

The success of QuickCheck has led to the development of other libraries for property-based testing. For Haskell, there are now several alternatives such as SmartCheck [14] and SmallCheck [15]. One of the authors of QuickCheck, John Hughes, has developed a proprietary version of the library for Erlang [16], which he and others have used to test large and complex systems written in other languages [17]. Finally, there exist implementations of libraries for property-based testing for other functional programming languages, such as ScalaCheck [18] for Scala [19], as well as for imperative programming languages, such as JUnit QuickCheck [20] for Java [21].

2.4 Differential testing

Differential testing [22] is a conventional software testing method where the output of the tested program is compared to that of an oracle. Given that the test is correctly implemented, any difference in the output of the oracle and that of the tested program should indicate the existence of a bug in the tested program. If the oracle itself is not correct, then there is a possibility that a false positive or negative could occur.

Having one oracle and one tested implementation is not essential for differential testing, we can also use it for testing multiple implementations of the same specification. By comparing the outputs of all of the instances when given the same input, we can regard the results as votes. The implementations that win the majority are considered to be correct. A problem with this approach is that if there is no majority, we cannot tell which implementations are correct. Chen et al. [23] employ this method successfully in the context of compiler testing.

2.5 The compiler pipeline

The compilation process of modern optimizing compilers can be visualized as a pipeline consisting of several stages, where each stage is responsible for one type of transformation. These stages can be many or few depending on the chosen granularity. However, we have chosen to divide the pipeline into five components:

**Lexing stage.** This initial stage transforms a string of characters into a list of tokens that can be forwarded to the parser. Typically, a token is a pair consisting of a part of the input string and a token type. If the lexical analysis is unable to transform the entire string, we consider it an error. [24]
**Parsing stage.** The parsing stage takes a list of tokens as input and transforms it into an abstract syntax tree, or AST, that represents the program. Abstract in this context means that the syntax tree does not have a one-to-one correspondence with the lextmes of the programming language’s syntax. For example, parentheses used to dictate evaluation order need not be represented explicitly, as the evaluation order of expression is inferred by the structure of the tree. [24]

**Type-checking stage.** In this stage, abstract syntax trees are analyzed for correctness according to a set of rules commonly referred to as the static semantics. If the program is correct, the type-checking stage outputs a tree that is isomorphic to the input tree but often augmented with additional semantic information. [24]

**Optimization stage.** This stage can consist of several transformations, all with the aim of translating the program into a more efficient representation while preserving the semantic meaning of the original program. Some optimizations, such as constant folding, can be performed directly on the abstract syntax tree. However, many kinds of optimizations are often performed after translating programs into a simpler language, an intermediate representation. [24]

**Code generation stage.** The final stage in our decomposition of the compiler pipeline takes a program in the form of an intermediate representation and translates it to a platform-specific assembly language. [24]

### 2.6 Testing compilers

Testing the lexing stage of a compiler can be done by feeding a string of characters to the lexer and looking at the output. If the list of tokens that are generated by the lexer is not equal to the expected result, an error has occurred. This process can be automated using a reference implementation by generating random strings comparing the output of the lexer being tested to that of the reference implementation. Writing a lexer for a complete programming language by hand is often time-consuming and can be difficult. Most languages contain tokens that have identical prefixes, such as : : and : : in Haskell. To guarantee that the correct token is identified, the lexer needs to be able to look ahead in the string. Typically, the lexer will look at the next \( k \) characters, identifying tokens and choosing the one that consumes the longest string. Because of these subtleties, lexers are commonly written using lexer generator DSLs in the style of Lex [25], where tokens are specified using regular expressions.

A test for the parsing stage can be performed by feeding the parser an input list of tokens and comparing it to the expected result. Just like with lexers, this process can be automated using a generator and a reference implementation. To be able to randomly generate lists of tokens that result in a well-formed AST, we can base the generation process on the grammar of the language. Purdom [26] did this in 1972 when he presented a random sentence generator for the testing of compilers. The grammar of most programming languages are complex enough to discourage hand-written parsers; C++ [27] is an example of a language that is notoriously tricky to parse. Similarly to the DSLs for generating lexers, there exist DSLs for generating parsers as well, the two most notable implementations being Yacc [25] and Bison [25].

Although automated random generation of input data has been shown to be useful for testing the lexing and the parsing stages [28, 29], using it to test the later stages is challenging. Firstly, the generation of randomized input data is non-trivial, since programs have a precise structure for the compiler to consider them valid. When testing a compiler for a statically typed programming language, this means that the randomly generated programs have to be type correct. Moreover, to successfully test the optimization and code generation stages, the program generator has to avoid generating programs that are either non-terminating or trigger undefined behaviors.

Secondly, a compiler is a program that translates a source program into an executable program. The property we would like the compiler to satisfy is that the source program and the resulting executable are semantically
equivalent. There are two issues with this formulation:

1. Defining semantic equivalence between two potentially very different execution models is challenging.
2. Semantic equivalence, in general, is undecidable.

The standard solution to (1) is to test the compiler against a reference implementation. This allows us to test the correctness of the compiler by checking semantic equivalence on the compilation product, i.e., on the produced executables to verify that the compiler and the reference agree. Even though this simplifies the formulation of semantic equivalence, it remains undecidable and must be approximated. There are several ways to do this, e.g., via abstract interpretation or via testing using observable equivalence on terminating runs, i.e., differential testing.

### 2.7 Differential testing of compilers

In the context of automatic random testing for compilers, differential testing can be explained with the help of the following algorithm:

1. Generate a random program.
2. Compile the program with the compiler to be tested, as well as with a compiler that will be used as a reference. Alternatively, when testing only the optimization stage, the reference can be the same compiler but with optimization disabled.
3. Run the two compiled programs and compare their output. If the outputs are equal the test case was successful; otherwise, a bug has been found.

By comparing the outputs of the compilation products, the equivalence of two programs is defined practically. For testing using random program generation, we have to ensure that a program provides a sensible output that can be compared. Typically, this is done by serializing values to character strings and outputting them to a file or console.

To successfully perform differential testing of compilers, the output of the tested implementation has to be compared to something else. This does not necessarily mean a reference that is guaranteed to be correct. We can also use a reference that we trust without knowing whether or not it is 100% correct, as long as we make sure to check the validity of the test output. The works of Pałka et al. [5], Midtgaard et al. [13], Yang et al. [30] and Zhang et al. [31] show that it is possible to define differential testing for compilers in several ways:

- Instead of a reference implementation, we can use the compiler itself but without optimization enabled.
- We can implement an interpreter that acts as an oracle.
- We can compare the compiler to one or more implementations of the same programming language specification if there are any.
3 Language

The language we have chosen to work with is a subset of C [6]. Among the constructs in the language, we find the following: variable declarations, functions and function calls, while-loops, if-statements, return statements, as well as basic arithmetic and comparison operations. The following sections list the grammar and a type system [1] for the language, inspired by a compiler course at MDH [32]. More experienced readers will find no surprises.

3.1 Grammar

Figure 1 shows the grammar of our language. We let \( n, b \) and \( x \) range over intergers, booleans and identifiers respectively. \( \odot \) ranges over the arithmetic binary operators, i.e., \( \odot : ::= + | - | * | / \). \( \odot \) ranges over the binary comparison operators, i.e., \( \odot : ::= < | > | \leq | \geq \). Finally, \( X \) denotes a sequence of \( X \), e.g., \( \bar{S} \) is a sequence of statement \( S \).

\[
P ::= D_1 D_2 \ldots D_n
\]

\[
D ::= \tau f (F) \bar{S}
\]

\[
F ::= \tau x
\]

\[
\tau ::= \text{int} | \text{bool} | \text{void}
\]

\[
B ::= \{ \bar{S} \}
\]

\[
S ::= B
| \text{if} \ E B_1 B_2
| \text{while} \ E \ B
| E
| \tau x
| \text{return}
| \text{return} \ E
\]

\[
E ::= n
| b
| x
| x = E
| E_1 \oplus E_2
| E_1 \otimes E_2
| E_1 \parallel E_2
| E_1 \&\& E_2
| !E
| -E
| f (\bar{B})
| E_1 == E_2
| E_1 != E_2
\]

Figure 1: The grammar above defines the abstract syntax for our language.
3.2 Type system

The types, \( \tau \), of the language are integers, \( \text{int} \), and booleans \( \text{bool} \), \( \tau ::= \text{int}|\text{bool} \). The variable type environments, \( y \), are mappings from variable names to types. The type environments, \( \Gamma \), are stacks of variable type environments, \( \Gamma \cdot y \) to handle syntactic scoping. We define lookup environments to find the first binding of a variable as follows:

\[
\begin{align*}
  \forall \tau, u. \ y[x] = \tau & \quad \Rightarrow \quad y \cdot \Gamma[x] = u \\
  \Gamma[x] = \text{undefined} & \quad \Rightarrow \quad (y \cdot \Gamma)[x] = \tau
\end{align*}
\]

3.2.1 Programs and functions

A program is type correct if all functions are type correct.

\[
\begin{align*}
  \Pi \vdash \tau_1 f_1(F_1) \bar{S}_1 : t_{o k} \\
  \Pi \vdash \tau_2 f_2(F_2) \bar{S}_2 : t_{o k} \\
  \quad \vdots \\
  \Pi \vdash \tau_n f_n(F_n) \bar{S}_n : t_{o k}
\end{align*}
\]

Program

\[
\begin{align*}
  \Pi \vdash \tau_1 f_1(F_1) \bar{S}_1, \ \tau_2 f_2(F_2) \bar{S}_2, \ \ldots, \ \tau_n f_n(F_n) \bar{S}_n
\end{align*}
\]

A function is type correct if its body is type correct in the environment induced by the formal arguments and their types, and all return statements respect the declared return type.

\[
\begin{align*}
  \text{tfunc}_1 \quad & \Pi, \tau \vdash \langle \{ x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \}, \bar{S} \rangle : \Gamma \\
  \Pi \vdash \tau f(\tau_1 x_1, \tau_2 x_2, \ldots, \tau_n x_n) \bar{S} : t_{o k}
\end{align*}
\]

\[
\begin{align*}
  \text{tfunc}_2 \quad & \Pi, \tau \vdash \langle \{ \}, \bar{S} \rangle : \Gamma \\
  \Pi \vdash \tau f(\epsilon) \bar{S} : t_{o k}
\end{align*}
\]

3.2.2 Expressions

Figure 2 shows the inference rules for expressions. The expressions contain both the assignment operation, \( t_{a sn} \), and function call, \( t_{c all} \). The rule for assignment, \( t_{a sn} \), checks that the type of the new value corresponds to the declared type. The boolean expressions, \( t_{o r} \) and \( t_{a nd} \), require that the operands are booleans, and result in booleans. The equality expressions, \( t_{e q} \) and \( t_{n eq} \), require that the operands have the same type, and result in booleans. The arithmetic expressions, \( t_{a op} \), require that the operands are integers, and result in integers. The inequality expressions, \( t_{n eq} \), require that the operands are integers, and result in booleans. A variable has the declared type, \( t_{v a r} \).
The sequence rules thread the type environment, since statements contain variable declarations. The new type is $\Pi_1$, and

$$\Pi \vdash \langle \Gamma, n \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, x \rangle : \tau$$

$$\Pi \vdash \langle \Gamma, E_1 \rangle : \text{bool}$$

$$\Pi \vdash \langle \Gamma, E_2 \rangle : \text{bool}$$

$$\Pi \vdash \langle \Gamma, E_1 \parallel E_2 \rangle : \text{bool}$$

$$\Pi \vdash \langle \Gamma, E_1 == E_2 \rangle : \text{bool}$$

$$\Pi \vdash \langle \Gamma, E_1 \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, E_2 \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, E_1 == E_2 \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, E_1 \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, E_2 \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, E_1 \oplus E_2 \rangle : \text{int}$$

$$\Pi \vdash \langle \Gamma, f \rangle = (\tau, (\tau_1 x_1, \tau_2 x_2, \ldots, \tau_n x_n), \bar{S})$$

$$\Pi \vdash \langle \Gamma, E_1 \rangle : \tau_1$$

$$\Pi \vdash \langle \Gamma, E_2 \rangle : \tau_2$$

$$\ldots$$

$$\Pi \vdash \langle \Gamma, E_n \rangle : \tau_n$$

$$\Pi \vdash \langle \Gamma_1, f(E_1, E_2, \ldots, E_n) \rangle : \tau$$

$$\Pi \vdash \langle \Gamma, \text{println} (E_1, E_2, \ldots, E_n) \rangle : \text{void}$$

Figure 2: Type rules for expressions.

3.2.3 Statements

The sequence rules thread the type environment, since statements contain variable declarations.

$$\Pi, \tau \vdash \langle \Gamma, \epsilon \rangle : \Gamma$$

$$\Pi \vdash \langle \Gamma_1, S \rangle : \Gamma_2$$

$$\Pi \vdash \langle \Gamma_2, \bar{S} \rangle : \Gamma_3$$

$$\Pi \vdash \langle \Gamma_1, S \cdot \bar{S} \rangle : \Gamma_3$$

Figure 3 shows the inference rules for expressions. All statements are typed in a function type environment, $\Pi$, and a return type $\tau$. The return type is checked in the type rules for return, where $\text{return}_{n_1}$ checks that the return type is $\text{void}$, and $\text{return}_{n_2}$ checks that the return type corresponds to the type of the return value. A block is type correct if all statements of the block are type correct in a new variable type environment. The new type environment makes declarations local, since $\text{declare}_1$ and $\text{declare}_2$ add declared variables to the top-most variable type environment if they have not been previously declared. Following C#, the type system does not allow shadowing.
Figure 3: Type rules for statements.
4 Generation

Generating arbitrary strings will produce a sequence of characters that is unlikely to be interpreted by the lexing stage as anything but nonsense. A more sophisticated approach is grammar-based generation, where the production rules of the grammar are used to guide the generation, recursively building a program with the right structure. Such a program will make it through the lexing and the parsing stages, but not necessarily be type-correct. To generate programs that can be compiled, we have to generate programs that are type-correct. Examples of errors that can occur in programs generated from the grammar are: references to undefined variables, variable assignments that do not have the correct type, and applications of functions with the wrong number of arguments.

To reliably generate type-correct programs the generator must consider the rules of the type system and the current context of the program. We refer to this as type-directed generation. Intuitively, we can view a type-directed generator as a grammar-based generator with additional constraints. When the generator is about to generate an expression, it first needs to ensure that the expression is correct with respect to its type in the current context. We can imagine that the generator has a list of expressions that can be generated, and then filters the list by removing expressions that do not satisfy the expected type. For example, if the generator wants to generate a reference to a variable, it will first filter the list to only contain variables that have the correct type and are accessible according to the scope rules.

When it comes to defining the generation algorithm itself, there are two main approaches: top-down or bottom-up. In a top-down approach, the program generation starts from the outer layers, i.e., functions, and works its way down to individual statements and expressions. For example, the algorithm could start by defining $n$ functions and registering all of the names in a dictionary and then generate each function. Generating a function amounts to generating $m$ statements and for each of them registering their effects on the variable environment. For each entity we generate, we only have access to those entities which we have previously created. Inside any given statement in a function, we can refer to all functions as well as all variables that were created previously in the current block, or in the parent and ancestor blocks. This idea is simple, and the algorithm can easily be implemented to work in one pass using recursion.

While top-down generation is a series of steps that accumulate previous actions from the outside in, a purely bottom-up approach would generate a program starting from the inside. Instead of starting at the top with function definitions, we start with individual statements and expressions. A helpful analogy is to think of this approach as demand-driven: in the generation of any entity, the algorithm can create a demand that needs to be satisfied. Let’s say that we are generating an addition operation and we decide that we want the operand to be the integer constant $1$ and the variable reference $a$, and $a$ does not exist. To do this, we generate our reference and log our demand: there should be a declaration of a variable of type `int` with the name $a$ sometime before the current statement. There are several ways to implement this. One solution could be to accumulate a list of demands until the generator reaches the end of a function definition and then build a dependency graph [33] with all demands and statements. Satisfying all of the accumulated demands can then be formulated as finding an order in which to generate the missing pieces to make the function definition correct. We can find this order by performing a topological sort [33] on the dependency graph.

Our algorithm mostly operates in a top-down manner, except in the generation of new function definitions. We create new functions by first generating a demand for a new function and then satisfying that demand immediately. It might seem odd that we settled for this hybrid approach, but it has at least one benefit. Since functions are only generated on-demand, all generated functions are guaranteed to be called from other parts of the program. Thus, all generated functions will be evaluated, meaning that the algorithm upholds a certain level of efficiency by avoiding the generation of functions that are not used in the program.
4.1 The generator

The implementation of our generator was inspired by the descriptions found in the paper of Pałka et al. [5]. The generator works similarly for statements and expressions, but we will focus on expression in our explanation. For this explanation, we will think of the behavior of the algorithm as a sequence of pairs of phases: the generation phase and the completion phase. The smallest possible sequence is the singleton sequence, consisting of only one pair. We obtain such a sequence from the generation of expressions that are leaves in the AST: integer constants, boolean constants, and variable references. The generation of the expression $1 + a$ will create a sequence of three pairs, first the generation and completion phase for the operator $+$ and then one each for its children: the integer constant 1 and the variable reference $a$.

To generate an arbitrary expression we first choose the type of the expression which we want to generate and enter the generation phase. In the generation phase, we define one pseudo-random generator for every program construct in the language. Apart from these, we also have generators for two types of statements that are often used in imperative programming: statements that mutate a variable by assignment $a = b + c$, as well as statements that consist of a single application of a function, $f()$. Thus, we augment the grammar by duplicating and lifting assignments as well as function applications to the statement-level. It is possible that the generator generates these two types of statements through a combination of the other generators, but since assignments and function applications have to compete with the other expressions, it is not that likely. The augmentation makes it easier to generate programs that have the characteristics of the imperative programming paradigm. As previously mentioned, we also generate demands for new function definitions; they are encoded as applications of unnamed functions.

It is important to note that the pseudo-random generators do not produce the program construct itself, but instead a placeholder. From here on, we will think of these placeholders as skeletons with typed holes in them; each hole has a type, signaling that something of that type should be inserted to generate the complete expression. For example, the skeleton of the plus operator can be visualized as $(\text{int} + \text{int})$, where each $\text{int}$ represents a hole to be filled in by an expression of type integer. The skeleton generators are gathered in a list and filtered according to the type that we want the expression to have.

For example, if we entered the generation phase with a request to generate an expression of type integer, the filtered list would only contain those AST-nodes that return an integer. Among these we can find the following constructs: binary operators $+$, $-$, $\ast$ and $/$, unary operator $-$; applications of functions that return integers; references to any variables declared as integers; as well as integer constants. If the sub-tree being generated has reached the upper size limit, as specified by the user, the filtered list will be filtered again to remove all skeletons which are not leaves. Continuing with our previous example, the list resulting from the second filter operation would only contain references to variables declared as integers as well as integer constants since they are represented as leaves in the AST.

When the list has been filtered, one of the remaining skeletons are chosen uniformly at random, and the completion phase is initiated. Here, the algorithm will complete the expression by filling in all of the holes in the skeleton. To complete a leaf-type expression, we need to insert a random constant, or a random reference picked from a selection of in-scope variables. For completion of non-leaves, the algorithm will call itself recursively for each hole, initiating the generation phase with a demand for an expression of the type of the hole.

As previously mentioned, a demand for a new function is encoded as an application of an unnamed function. To complete this skeleton, the algorithm will first register all parameters in the functions local scope. The algorithm generates the body of a function by calling itself for each statement. When the body has been generated, the algorithm will append a return statement with a local variable reference if the function type is non-void and if the last statement is not a return. Note that while the body of a void-function is not appended with a return, the body itself can still contain early returns, created in the generation process. Finally, the newly generated function is
registered in the global scope. By not registering functions in the global scope until they are fully generated, we
avoid creating functions that are self-recursive or groups of functions that are mutually recursive. However, this
is a double-edged sword. While it does guarantee that all of the generated programs are well-formed in terms of
termination, it also limits the space of possible programs that can be generated.

4.2 Program output

We have chosen to define semantic equivalence in terms of observable output on terminating runs. To make
generated programs produce output, we use natural medium found in most programming languages, the print
function. A print function is flexible since it can be inserted anywhere in a function and can accept any number
of arguments. An example of a print-generation scheme that probably would perform poorly is one where calls to
a print function are generated at random, with arbitrary expressions as input. Such a scheme would not be so
successful at displaying errors that arise from combinations of statements since the category of expressions is
dominated by expressions that are not affected by side-effects on the local environment, everyone except variable
references.

Being able to catch those errors is essential since bugs in heavily tested industry standard compilers are often not
simple enough to be triggered by separate instructions. On the other hand, since any expression can be assigned
to a variable, only printing the values of variables will also include the rest of the expressions. Yang et al. [30] use
a similar print-generation scheme: they store results of computations in global variables and print their values
regularly. Therefore, we choose to generate print statements that print the values of variables. To make sure that
a function’s internal state is always revealed, the generator always generates a call to a print function that prints
all accessible variables before every return statement. If no variables are available, the generator will generate a
print statement that prints the integer constant 1337.
5 Templates

For us to be able to guide the generation, the generator provides a set of configurable parameters. These are communicated to the generator via JSON-files [34] that we call templates. Figure 4 shows an example of a template file. The first property, `paramcount`, defines a closed interval that sets a lower and an upper limit on the number of parameters for function definitions. The second property, `blocklength` defines a closed interval for constraining the number of statements that can be generated in a block. The property `sizelimit` is a constant used to set an upper limit on the size of statements. Similarly, `maxglobals` defines the limit for the maximum number of functions that can be defined. Finally, the last property, `genprobs`, is an object that defines a mapping from each generator category to a constant that represents the probability of it being chosen. Note that it is possible to avoid generating specific program constructs by mapping their generator category to the value zero.

```
{
    "paramcount" : [1, 2],
    "blocklength": [0, 4],
    "sizelimit" : 32,
    "maxglobals" : 2,
    "genprobs": {
        "defstmts" : 16,
        "defs" : 16,
        "locals" : 8,
        "assignstmts" : 8,
        "ifs" : 6,
        "blocks" : 6,
        "whiles" : 6,
        "decls" : 6,
        "globals" : 4,
        "globalstmts" : 4,
        "bops" : 2,
        "prims" : 2,
        "uops" : 2,
        "assignexprs" : 1,
        "rets" : 1,
        "exprstmts" : 1
    }
}
```

Figure 4: The default template which sprung out of the development phase of the generator.

Tweaking the parameters in the template allows us to guide the generation. Figure 5 shows an excerpt of a program that was generated when the `sizelimit` parameter was set to 4 and the probability constant of binary operators parameter set to 16. The result is a function that mainly consists of nested binary operator expressions. Generating exact values can also be achieved. Figure 6, shows a program where the `paramcount` parameter was set to the closed interval [7, 7], resulting in seven parameters for each function. We can also control the distribution of if-statements and while-statements to generate more interesting programs. Figure 7 shows an excerpt of a program with nested if-statements and while-statements, as well as an early return statement.
bool f() {
    (((5 - -9) - -5) - (2 * 4) * (1 * -3)) * (((1 * 4) * (-4 * -4)) * ((0 + -2) - 2 * 1)) * -3 != g(true || false, 2 * 2, true == false, 2 + -4, -1 == 4) * ((-8 - 6) * (-5 - -3) - ((8 - 4) - (-5 + -9)) - (((-6 + 6) - 0) - (6 * 6 - (3 - 6))) + (g(true, -1, true, -8, true) - (0 + 3) * (-7 - 3)))
    return (((-6 - -9) + (-9 - 6)) - ((8 - 2) - (6 + 3))) * (-7 * (5 + 8) - ((6 + 2) - -5 * 0)) != 2 * ((7 + 1) * (-2 + 3) + (6 * 3 + (7 + 7))) && (-9 * (7 - -1) - (-2 - (2 - 4))) - g(false != true, g(true, -3, true, 2, true), true != true, 1 - -3, false == true) == (((4 - -2) - -8) + ((-8 - 0) - (-1 - 4))) - -8 * ((0 - 4) * (-8 * 7));
}

Figure 5: A generated program mainly consisting of nested binary operations.

void g(int a, int b, int c, bool d, int a1, int b1, int c1) {
    ...
}

Figure 6: A program generated using a template where the paramcount interval was set to [7, 7].

void g(int a, int b, int c, bool d, int a1, int b1, int c1) {
    if (false) {
    } else {
        int d1;
        while (d) {
            if (d1 > -5) {
                int a2;
                while (d) {
                    if (a2 < 5) {
                        print(c1, b1, a1, d, c, b, a);
                        return;
                    } else {
                        a2 = a2 + 1;
                    }
                }
                print(c1, b1, a1, d, c, b, a);
                return;
            } else {
                d1 = d1 - 1;
            }
        }
        int a2;
        while (a2 < 5) {
            int b2;
            while (b2 < 5) {
                b2 = b2 + 1;
            }
            a2 = a2 + 1;
        }
        print(c1, b1, a1, d, c, b, a);
    }
}

Figure 7: A generated program with nested while-loops and if-statements.
6 Evaluation: statement coverage

Statement coverage [7] measures the number of statements in the source code that are executed in proportion to the total number of statements. For the test, we used a reference interpreter and constrained the test to only those sections of the source code that deal with the interpretation of programs. The reference interpreter is implemented in C#, which allowed us to use JetBrains’ tool dotCover [8] to perform the measurements. The goal of the test was to compare the statement coverage of a suite of programs generated with our generator to those of a test suite developed for a compiler course at MDH. This test suite has been developed in an accumulative manner over several course instances: several people have been involved in the process, and new tests have been added periodically as new bugs have been discovered. In total, the reference test suite consists of 27 programs. To make the comparison fair, we also limited the generated test suite to 27 programs. We performed the test by measuring the statement coverage of a batch run consisting of all of the programs in the respective test suite. The results from the tests are shown in figure 8.

<table>
<thead>
<tr>
<th>Test suite</th>
<th>Statement coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>98%</td>
</tr>
<tr>
<td>Generated</td>
<td>99%</td>
</tr>
</tbody>
</table>

Figure 8: Statement coverage of the reference suite and the generated suite.

The generated test suite reached a 99% statement coverage and the reference test suite reached a 98% statement coverage. The additional 1% that the generated test suite did not cover is represented by error-handling code, which was not triggered because all of the programs were type-correct. Therefore, the generated suite reached 100% statement coverage of the sections of the source code that does not deal with error-handling. The 27 generated programs were all generated using the template shown in 4 which sprung out of the development phase of the generator. Thus, we did not need to tweak the template; the first 27 generated test were able to reach higher coverage than the reference test suite. We did not pursue additional tests, but in hindsight, we realize that it could have been interesting to find out how many of the 27 generated programs were needed to match the coverage of the reference test suite.

By looking at the statement coverage report, we were able to conclude that the reason that the reference suite only received 98% coverage is that none of the programs in the suite contain the not equals operator (\(!=\)). We made at least two interesting observations from this result. Firstly, despite being developed by several people over several course instances, no one included a single test for an operator that is frequently used in programming. Although such a trivial mistake would probably not be made in a test suite for an industry standard compiler, the result still shows that writing test suites for compilers, in general, can be challenging; a developer needs to consider many things to reach full coverage. Secondly, while we consider the complete absence of tests for one operator to be a substantial problem, the measurements only reported a difference of one percentage point from the top score, 99%. Because we measured statement coverage, this means that the implementation of the not equals operator represented one percentage point of the total number of statements. Not only does this mean that percentage differences do not correspond to the severity of the missing coverage, but it also means that different implementations can have different coverage percentages, even though they are semantically equivalent. The capacity at which the coverage report can measure the quality of a test suite depends on the structure of the implementation of the compiler or interpreter being tested.
7 Shrinking

Randomized program generation can result in large and complex programs that are difficult to reason about. With shrinking, we can address this problem by repeatedly making a program smaller, until a minimal error-triggering representation is found. To be able to shrink programs, we first need to define what shrinking means. Since programs are modeled using ASTs, we can define shrinking in terms of trees. We shrink a tree by giving it as input to a function, a shrinker, that outputs a smaller version of that tree. There are many ways that a shrinker could operate, but a straightforward approach is by replacing any given child node of an expression with a smaller version of that child. The shrinker can also replace the parent node with any of the children. This means that we can implement a shrinker as a recursive function that operates on trees; the leaves represent the base case as they cannot be made smaller.

So far, our definition of a shrinker does not take the type of an expression or the surrounding type environment into account. This could be problematic if we were to use this definition when testing, because the shrinker could introduce type errors in a reduced program. For example, the expression \texttt{false == 1 < 2}, could be reduced to the expression \texttt{false == 1}, which is not type-correct. This is not necessarily a problem since both compilers could agree that this is a type error, and the reduced program would not be considered to have triggered an error in the tested implementation. However, whether or not the two compilers will agree depends on how we define the comparison of the outputs. If we were to define the comparison as a check for string equality, the two compilers would disagree if the error messages were formulated a bit differently. For example, let’s say that the first compiler outputs “Type error: Expected bool.” and the second one outputs “Compilation error: Expected bool.”. Since the comparison would consider the two compiler to disagree, the shrinker will have masked the original error. When the the error-triggering program has been minimized, it will show the type-error instead of the expression or statement that triggered the initial error.

We could avoid error-masking by defining the comparison of the outputs as an approximate match \[35\], perhaps as simple as checking if both outputs contain the substring “error”. However, correcting this problem inside the shrinker will make the shrinker easier to work with from the outside; users will not need to define fuzzy equality checks to use it. Moreover, we also believe that preventing the introduction of errors helps in building a solid foundation for possible future extensions of the language. Therefore, we introduce type-directed shrinking. Just like a type-directed generator, to avoid introducing type-errors, a type-directed shrinker needs to consider the rules of the type system. For expressions, this means that the shrinker has to output an expression that is of the same type as the expression it received as input. There are also errors that can occur when removing certain statements. For example, a non-void function must have a return statement for all paths, and the use of a variable is not valid without a declaration of that variable preceding it. Another possible error would be the removal of the definition of a function that is being called somewhere in the program.

7.1 The shrinker

Our implementation, henceforth referred to as the shrinker, makes use of Haskell’s lazy evaluation. For each call to the shrinker, all of the possible reduced programs, according to our algorithm, will be returned in a list. For large programs, this list could be huge, but because of lazy evaluation, none of the reduced programs will be evaluated until the value is needed. This is important because more often than not, only one or a few reduced programs are actually needed. To signal that there are no more ways to reduce a program, the shrinker will return an empty list. We use lists to model non-determinism, a common Haskell idiom. Conceptually, we can view the shrinker as outputting a set of choices, or paths, that we can take and when we have reached the end of a path, the set is empty. This property allows us to keep the interface of the shrinker simple, while still upholding a certain level of efficiency.

We built the shrinker with several mutually recursive functions, which operate according to the intuition we mentioned earlier: replacing child nodes with a smaller version of themselves and replacing a parent node with
any of its children. The shrinker function for expressions is parameterized by a type, making sure that the output is type-correct. Internally, the shrinker function will filter the list of reduced programs, to make sure that it only returns program of the expected type. When the shrinker encounters a variable declaration, it will first analyze the function in which they occur, checking whether or not the variable is in use. Thus, we only remove unused variable declarations.

For non-void functions, we have to be careful with return statements. If the shrinker were to remove a return statement and cause an execution path to miss a return statement, the shrinker would introduce an error. To avoid this, we prevent the removal of the last return statement in the function. Since the generator always generates a return statement as the last statement in a non-void function, this guarantees that the function will remain type-correct. However, since the shrinker essentially locks access to the last return statement, it will not be able to shrink a non-void function to an empty body. Therefore, we implemented a predicate which decides whether or not a function is minimal. A non-void function is considered to be minimal when it is on the following form: \( t \ f(\ldots)\{ \text{return } e; \} \), where \( e \) is an expression type \( t \). If the return statement refers to a variable declared inside the function body, the function will also be considered minimal when all removable statements have been removed: \( t \ g() \{ t \ a; \text{return } a; \} \). If the predicate should decide that all functions in a program are minimal, the shrinker will output an empty list to signal that the program cannot be further reduced.

7.2 Optimization

The abstract syntax tree of a program is sometimes large, and we want to make sure that the shrinker can minimize them quickly. To do so, we defined the shrinker to be greedy and prioritize larger reductions before smaller ones. This priority manifests itself in the output list by the order in which the reduced programs occur. Programs that are at the beginning of the list have higher priority. We can see this in effect in figure 9, the largest reductions, which output just the leaves of the tree, are placed at the beginning of the list, and the following expressions get successively bigger. Naturally, the shrinker treats all other program constructs similarly. At statement-level this priority is enhanced: the shrinker prioritizes the complete removal of statements before reductions that make statements smaller incrementally.

\[
[ \\
1, \\
2, \\
3, \\
4, \\
1 + 2, \\
3 - 4, \\
1 \ast (3 - 4), \\
2 \ast (3 - 4), \\
(1 + 2) \ast 3, \\
(1 + 2) \ast 4 \\
]\]

**Figure 9:** The output of the shrinker when given the expression \((1 + 2) \ast (3 - 4)\)

At function level, the shrinker will try to find definitions of unused functions and remove them. It finds unused functions with a depth-first search [33] that starts in \texttt{main}, marking every function it visits. The functions are only visited if they are called in another function, making sure that the search only visits functions that are directly or indirectly connected to the entry point of the program. When the search has terminated, the shrinker will remove all unvisited functions. The search is guaranteed to terminate since it only visits previously unvisited functions. Because of these optimizations, the shrinker can shrink programs fast, and for several errors, successive shrinks will lead to a minimized program that only contains one function, not including \texttt{main}. An example of the shrinking process in action is shown in figure 11 and figure 12.
8 Evaluation: differential testing

Testing using the generator and the shrinker requires an entity that drives the test forward, henceforth called the driver. The driver takes as input, a number that represents how many programs that the driver should generate using the generator. For each of the n programs, the driver gives the program to a function, f. The function feeds the program to the tested interpreter or compiler, and to a reference interpreter. When both implementations have executed the program, f compares their outputs and returns the result. If a pair of output strings are equal, the driver continues to the next program; if they are not equal, an error has been found, and the driver initiates the shrinking phase by calling a function, g, with the error-triggering program and a list containing all of the possible reductions as arguments. Recall the interface of the shrinker: for any input program, it outputs a finite list of all possible reductions, and when the program cannot be made smaller, the output list is empty. This allowed us to define g as a simple recursive function. Given a program p and a list of reductions as input, we first check if the list empty. If it is empty is then we return p; if it is not, then we test the head of the list with f. If the outputs are equal, we recurse with the old p and the tail of the list; if the outputs are not equal, we recurse with the head in place of p and all the possible shrinks of the head in place of the old list.

8.1 The tests

To evaluate the generator and the shrinker we tested their bug-finding capabilities with a set of seven anonymized interpreters from a compiler course at MDH. Before conducting the tests, we were told that some of the interpreters potentially had some bugs in them, but we were not told which ones and of what sort the bugs would be. In our tests, we ran the driver with n = 100 meaning that each interpreter would have to output the correct result for 100 programs to pass the test, and in the one case where the tested interpreter would pass the test, we ran the test once more. The generator generated programs according to the template in figure 4, the same one that we used for the statement coverage test. It results in programs consisting of up to two relatively small functions that can contain all possible program constructs.

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Test result</th>
<th>Bug(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Failed</td>
<td>Return statements only propagate one level.</td>
</tr>
<tr>
<td>2</td>
<td>Failed</td>
<td>Uninitialized variables are not created.</td>
</tr>
<tr>
<td>3</td>
<td>Failed</td>
<td>Early returns are ignored.</td>
</tr>
<tr>
<td>4</td>
<td>Successful</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>Failed</td>
<td>Not equals operator ((!=)) not implemented.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uninitialized variables are not created.</td>
</tr>
<tr>
<td>6</td>
<td>Failed</td>
<td>Void functions without returns are not allowed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Wrong operator precedence.</td>
</tr>
<tr>
<td>7</td>
<td>Failed</td>
<td>Unknown.</td>
</tr>
</tbody>
</table>

Figure 10: Results of testing the interpreters with the generator and the shrinker.

The table in figure 10 shows the results of the conducted tests: we tested seven interpreters and only one was found to be correct. Several of the interpreters failed on the first program and all failed within the first ten. To be confident of the outcomes of the test of the fourth interpreter, we reran the test, exposing it to a total of 200 programs. Note that the fifth interpreter has not implemented the not equals operator, the very same operator that was not present in the reference test suite. When we tested the seventh interpreter, it did not even run the programs but instead output an error message about a null reference.

The bugs were identified by analyzing the minimized program together with the expected and actual output. For some reduced programs, it was not obvious what the triggered bug could be, mostly because some of the tested interpreters had substandard and vague error messages without positional information. However, this was not a problem, we just reran the test multiple times and accumulated the output from each test. By looking at all the minimal programs and corresponding expected and actual outputs together, we were able to identify the cause.
of the error. Since the bugs were not only triggered for each run of 100 programs, but also within the first few programs, executing additional test runs was not time-consuming. Running the test a few extra times also allowed us to identify another bug for two of the interpreters. We do not know whether or not we found all bugs since we did not inspect the source code. Potentially, some of the bugs we found could be masking other bugs. In the test, we noticed that the generator was able to produce programs faster than the interpreter could evaluate them. Figure 11 and figure 12 show an example of the command-line output from the testing of the first interpreter.

```c
bool g(int a, bool b) {
    print(b, a);
    return b;
{
    int c;
    while (1 > a) {
        b = false;
        b = b;
        if (c < 5) {
            b = b;
            print(b, a, c);
            return b;
        } else {
            c = c + 1;
        }
    }
    b = b;
    b = b;
}

int f() {
    g(-7 * 8, true);
{
    print(1337);
    return 5;
    g(-2, true);
    true;
    g(3, true);
}

    if (g(--8, g(7, true))) {}
    7;
    print(1337);
    return 3;
}

int main() {
    f();
    return 0;
}
```

**Figure 11:** Testing the first interpreter: command-line output showing the bug-triggering program.
Here is the minimized program:

```c
int f() {
    return 5;
}
print(1337);
return 3;
}

int main() {
    f();
    return 0;
}
```

Number of shrinks: 11
Expected output : ""
Actual output : "1337"

**Figure 12:** Testing the first interpreter: command-line output showing the shrinking phase.

To be able to compare our results to something, we also tested the interpreters using the same test suite that we employed in our statement coverage evaluation. The test suite consists of 27 programs which together reached 98% statement coverage of the reference interpreter, the same number as the generator. The results are shown in the table in 13.

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Test result</th>
<th>Bug(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Successful</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>Successful</td>
<td>N/A</td>
</tr>
<tr>
<td>3</td>
<td>Successful</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>Successful</td>
<td>N/A</td>
</tr>
<tr>
<td>5</td>
<td>Successful</td>
<td>N/A</td>
</tr>
<tr>
<td>6</td>
<td>Failed</td>
<td>Void functions without returns are not allowed. Wrong operator precedence.</td>
</tr>
<tr>
<td>7</td>
<td>Failed</td>
<td>Assignment operator does not return value.</td>
</tr>
</tbody>
</table>

**Figure 13:** Results of testing the interpreters with the reference test suite.

As shown in figure 13, we managed to run the test suite on the seventh interpreter. The interpreter would only fail with the null reference error for two of the programs, so we were able to select a bug-triggering program and analyze it. The cause of the error was found by repeatedly removing statements, then running the program, and then adding statements back in until it was clear what line caused the error. As a process for identifying the cause of errors, this was more cumbersome than the experience we had with the generator and the shrinker, where we could just re-run the test. However, as we noted previously, we could not interact with the seventh interpreter from the driver since all programs gave a null reference error, and therefore we were not able to shrink the programs using the shrinker. However, we are sure that the shrinker could have minimized the program because the assignment operator error is essentially equivalent to the error that the shrinker identified in the fifth
interpreter. From a shrinking perspective, the AST-nodes that trigger both errors are identical: expression nodes with two children.

While this time we did manage to run the seventh interpreter, improving on the previous test in that respect, the test suite did not perform as well as the generator and the shrinker overall. While the shrinker and the generator found bugs in all but one of the six first interpreters, the test suite only found bugs in one of them. Thankfully, the sixth interpreter had good enough error messages for us to quickly identify that the errors were the same as those we found with the generator and the shrinker.
9 Discussion

In this section, we discuss the results of our tests and implications that they carry, both on their own and together. Furthermore, we discuss type-directed generation and shrinking as compared to traditional testing using test suites: not only their differences but also how we could use the methods together. Finally, we deal with some of the limitations of the current implementation of the generator and the shrinker, and present potential improvements, including controlled error generation.

9.1 Statement coverage

The generator achieved higher statement coverage [7] than the reference test suite. The fact that we did not need to tweak templates to reach full statement coverage indicates that the language is perhaps too small for the results to have any severe implications. We believe that the results do show a clear advantage of the generator: the process of generating 27 programs took just a couple of seconds. Naturally, first, the generator has to be implemented, requiring an initial investment of time. Once it is built, compiler developers can use the generator to produce tests and devote more of their time to working on actually fixing bugs. Furthermore, the fact that the reference test suite was missing tests for the not equals operator shows that even though a developer might have a long writing a test suite, there is no guarantee that the test suite has enough coverage.

The reference test suite contains 27 tests, a number which is quite substantial given the size of the language. A strong reason for having a large number of tests is that we can divide separate tests into separate files so that if a test fails, we can get an understanding of what the error could be with just a quick look at the program. When testing, we do not want to have to rely only on the error messages of the compiler, but also structure our test programs to be effective in communicating the source of an error. This has a disadvantage: compiler developers have to write many tests. Our language currently consists of 14 operators, the majority being arithmetic operators and comparison operators. If we were to extend our language to include all of the 49 operators in C [6], we would need at least 35 additional tests to cover their basic functionalities. Furthermore, if we want to test additional properties, such as whether or not an operator correctly shortcuts evaluation or not, we have to write even more tests. For a larger language, manual test writing can become cumbersome. With the generator, on the other hand, we just have to add all the operators to the AST datatype, and the generator will be able to output programs that make use of them.

9.2 Differential testing

The results show a clear advantage to the generator in terms of triggering bugs. While the reference test suite only triggered three bugs in two of the interpreters, the generator was able to trigger seven bugs in five of the interpreters. The fact that the reference suite did not trigger bugs in the first four implementations is alarming. It may not necessarily carry any severe implications since the reference test suite is substantially smaller than an official test suite for an industry standard compiler, such as the GCC test suite [36]. However, the results do highlight the fact that writing effective test suites by hand is not easy, and that random program generation can help in the testing of compilers. Thus, we believe that our results support those of Pałka et al. [5], Midtgaard et al. [13], Yang et al. [30], Zhang et al. [31], Le et al. [37] and Lindig [38].

We also believe that the tests demonstrated the capabilities of the shrinker as a means of identifying the cause of errors. When we tested the capabilities of the reference test suite, we had to identify one of the bugs through cumbersome fiddling with different combinations of statements. Since the error message simply stated that a reference was null, we received no guidance on our hunt. Instead, we were left to analyze the program, which to our relief was relatively small. How the author of the program had structured the code was crucial in reducing the time it took for us to find the cause of the error - another testimony showcasing the challenges of writing tests for compilers. The shrinker, on the other hand, does not rely on the compiler for support. Instead, it helped us in finding bugs on its own. The process was straightforward: first, we analyzed the minimized program together
with the expected and the actual output, and if we were unsure about the cause of the error, we just reran the test.

9.3 Statement coverage for compilers

The results of the statement coverage test showed that the generated test suite had a slightly higher code coverage than the reference test suite, 99% versus 98%. The generated suite had full coverage of the reference interpreter, except for the sections devoted to error-handling, while the reference suite was missing programs that make use of the not equals operator (`!=`). In our second test, the reference test suite only triggered three bugs, while the generator triggered seven bugs. It turned out that the reference suite was missing several types of tests: programs with the not equals operator, programs with uninitialized variables, programs with returns from blocks, and programs with early returns in general. The statement coverage did flag the fact that the test suite was missing a test for the not equals operator, but it could not provide any warnings to indicate any of the other missing tests.

This result is alarming since statement coverage as a measure for quality of a compiler test suite could falsely indicate to developers that they are testing the compiler thoroughly. There are several other ways to measure code coverage, and statement coverage is perhaps one of the weakest [7]. However, the results suggest that there is a ceiling in terms of code coverage, regardless of what measurement we use to calculate it. If coverage does reach that ceiling, we still cannot consider a test suite sufficient. Because of this, we believe that code coverage is not an appropriate measure of quality for compiler test suites. Note that we exclude exhaustive path coverage testing [7] from this analysis since it is not a feasible approach for any large and complex system.

9.4 Type-directed generation and shrinking vs. test suites

Writing a program that triggers a bug in an industry standard optimizing compiler is not an easy thing to do. Once we get past writing the first test cases, it is hard to devise programs that will trigger the not so obvious errors, because the system is too large and complex for humans to reason about as a whole. Therefore, we believe that the traditional approach of writing test suites will never be able to trigger all the bugs in optimizing compilers; the quality of the test data, the programs, is ultimately limited by a group of humans’ understanding of a system that is too large and complex for them to understand fully. There exist several substantial test suites, like the GCC test suite [36], which have large coverage. However, to reach that kind of coverage, a large number of test programs have to be gathered, and that takes time; the development of GCC test suite spans several decades. Despite this, the work of Yang et al. [30], Zhang et al. [31], Le et al. [37] and Lindig [38] have shown that a large number of bugs have still managed to escape the test suite. Essentially, the test suite is a collection of real programs taken from different fields of the industry, as well as programs written explicitly with the aim of exercising the compiler. The programs are either programs that may be challenging to compile but ultimately written to be read by humans, or programs that are written to be challenging according to a human’s belief of what kind of programs are likely to trigger bugs. In any case, the test data will be constrained to a specific area of the space of all possible programs, leaving large areas untested.

Our generator, on the other hand, is not limited by human imagination and not necessarily guided by a predefined notion of how to write bug-triggering programs. The configurational aspect of the generator allows for a user to resize and move the space of possible programs that the generator can output, but not to control how the programs themselves are composed. Moreover, by setting the probabilities of the different program constructs to equal values, the space will be widened, and the generator will produce a large variety of different programs, several of which a software developer would probably never think of writing. This entails that by exposing an optimizing compiler to a large enough set of randomly generated programs, it is likely that some programs will eventually trigger a bug that would never have been triggered by traditional testing using test suites. Critics could argue that programs that are too different from those that are written in production code to execute a meaningful task will only trigger bugs that are obscure and ultimately not severe enough to be worth the time. Nonetheless, Yang et al. [30], Zhang et al. [31] and Le et al. [37] were able to find bugs in GCC that were marked by the developers as release-blocking, the highest priority level.
Moreover, while writing a test suite is a time-consuming process, testing a compiler using our generator is not.

In the tests, the bottleneck in terms of execution speed was the interpreters’ evaluation of the programs; the generator produced programs faster than the interpreters could consume them. This observation implies that there is probably a sweet spot in terms of the size of programs and their bug-triggering ability. Too large programs will lower the rate at which the test can trigger bugs because programs take too long to evaluate. Programs that are too small will instead lower the rate because of their lack of complexity. With the generator’s configurational interface, it is possible for users to address this by allowing them to optimize the generated programs for fast evaluation. A compiler developer can, for example, customize the programs for a particular optimization pass by only generating some constructs, or by only generating function definitions of specific lengths. Finally, the fact that the generator produced programs faster than the interpreter could evaluate them implies that there is room for more computations in the generation process, for example, static analysis.

There is a difference between triggering bugs and finding bugs. Programs that trigger bugs and produce the wrong output at compile time or at run time signal that there is an error in the implementation, or perhaps even the specification. A program that triggers a bug will not necessarily help in finding the error. When a compiler developer is analyzing a program to identify an error, the program’s ability to communicate the error is a critical part of the process, and the size of the program dramatically affects this property. For each statement in the program, the developer has to analyze the code generator [39] translates the AST representing that statement to the target hardware language. Several types of optimization passes transform sets of statements, potentially resulting in an exponential growth of options to consider. The smaller the program is and the less noise it contains, the better it will be at assisting developers in finding the cause of the errors.

We believe that the shrinker is a tool that can successfully help compiler developers in finding the cause of errors. The command-line output shown in figure 11 and figure 12 is an example of the shrinkers ability to transform a program that triggers a bug without communicating what the cause of the error is, into one that not only triggers the bug but also clearly points to the location of the error. The shrinker can also be used separately from the generator, for example with a test suite. Implementing this change is as simple as taking the implementation of the test driver and replacing the use of the generator with an iterator that goes through each test in a test suite. This method would be complementary to the one employing the generator to produce programs, and compiler developers can use them both together. The combination would offer the generator’s non-determinism and ability to find previously unknown bugs and the accuracy of hand-written test suites, allowing developers to also specifically target specific code patterns that are known to cause problems.

The approach developed by Pałka et al. [5] is fundamentally different to traditional compiler testing using test suites: instead of relying on the subjectivity of the human mind we let the non-determinism of pseudo-random generators create the test data. With a mechanism for shrinking, the method becomes a powerful tool for not only triggering bugs but also identifying them. All of these ideas spring from property-based and QuickCheck. One of QuickCheck’s co-author, John Hughes, has shown that property-based testing can be used to test several types of large and complex systems found in software industry [17], such as distributive systems. For each of these systems, he has first created a model of the system and then applied property-based testing to find bugs that the developers either did not even know they had or could not locate. The work of Hughes indicates that property-based testing can be superior to human intuition when it comes to finding errors in large and complex systems. We believe that type-directed generation and shrinking can be used successfully in the testing of industry standard compilers for imperative programming languages.

9.5 Limitations: the generator

In theory, the generator should only be limited by the rules of the type system and what is considered to be well-defined behavior in terms of runtime semantics. In practice, however, it is not possible to create a generator that is capable of generating all well-formed programs without also sometimes violating the rules of the type system or runtime semantics. Consider non-termination: to avoid it we need to be able to tell whether or not an
arbitrary program will terminate, which essentially is the halting problem, proven unsolvable by Alan Turing [40]. Thus, it is not possible to create the perfect generator because of the limitations of computation. Nonetheless, our generator can still be improved. In the following sections, we discuss some of the limitations of the current implementation, and ways to move past them.

To avoid errors in generated programs, as well as reduced programs, the generation of functions and while-statements is limited. As previously mentioned, the generator will insert a return statement as the last statement of any non-void function. However, it is possible to write programs that do not have a return as the last statement, but still, have one return for every execution path. We could lift this constraint by adding control-flow analysis to both the generator and the shrinker. The generator constrains the generation of while-statements in the sense that it will not mutate the loop variable from the inside of the loop body, except for the increment or decrement it inserts to make the loop terminate. We can solve this by analyzing the expressions that mutate the loop variable to make sure that they do not stop the loop from terminating. In short, the analysis could check for increments and decrements, and the generator could alter the mutation if it is deemed to be erroneous. Even if a total analysis is possible, the computational cost would probably be impractical. At some point, we would probably want to terminate the analysis and alter the mutation in such a way that termination is guaranteed.

The most significant limitation of them all stems from the prevention of the generation of self-recursive functions and groups of mutually recursive functions. With this decision, we are effectively ruling out a vast number of type-correct programs that are well-formed in terms of termination. To be able to tap into this area of programs, we could apply a similar technique as with while-statements and analyze the program to find out whether it is likely to terminate or if there is a risk that it does not. However, there is a simpler solution available, which is effectively employed by Yang et al. [30]: allow the generator to generate programs that are not guaranteed to terminate and use a breaking mechanism to cancel the evaluation of the program after a certain length of time. This option does not agree with the overall approach we have had so far: to build a generator that can generate programs without errors. That is to say, we are not against generating programs that contain errors, but we want to be able to do so in a controlled manner. Maybe our approach is just not feasible in the long run; recursive functions could be the point at which the compromise of the number of programs the generator can produce is not worth the control that the limitations provide. After all, the primary purpose of the generator is to generate programs that trigger bugs in compilers. Naturally, the best way to increase its capacity to do so is to extend the space of possible programs it can generate.

9.6 Limitations: the shrinker

Sometimes, a function will not be removed by the shrinker, even though the function itself is not causing the error. One example where this can occur is in a program like that in figure 14.

```c
int f() {
    return 3;
}

int main() {
    print(1 + f());
    return 0;
}
```

Figure 14: A bug-triggering program.

Let’s say that the tested compiler for some reason has a bug in the implementation of code generation for binary operators: instead of outputting an instruction addition for plus, the compiler will output an instruction for subtraction. Thus, the correct output string will be "4" but the tested compiler will output "-2". Even though
the call to \( f \) could theoretically be replaced with any integer value that is not zero, meaning that \( f \) is not essential for the triggering of the bug, the shrinker will not be able to shrink the program further. When the shrinker for binary operator expressions is given the input \( 1 + f() \), it will output the list \([1, f()]\), a list of the operands, neither of them triggering the bug in the tested compiler. For such a case we could instead implement a function inlining feature in the shrinker. In its simplest form, the inlining feature would be able to inline calls to functions with zero parameters that return an expression at the call site. Inlining the function \( f \) at the call site would result in the program shown in figure 15.

```c
int f() {
    return 3;
}

int main() {
    print(1 + 3);
    return 0;
}

Figure 15: \( f \) inlined at the call site.
```

Now, the shrinker can remove \( f \) since no other statement refers to it. The reduced program, shown in figure 16, is the smallest program that will trigger the error.

```c
int main() {
    print(1 + 3);
    return 0;
}

Figure 16: A minimal bug-triggering program.
```

Now, consider the program in figure 17.

```c
int g() {
    return 1 + 2;
}

int f1() {
    return g();
}

int f2() {
    return f1();
}

int f3() {
    return f2();
}

int main() {
    print(f3());
    return 0;
}

Figure 17: A chain of function calls.
```
Given that the shrinker will perform at most one inline for any reduced program, the shrinker will be able to shrink the program to its minimal form in four steps. However, the shrinker could also perform inlining greedily, applying several or all possible inlinings for one reduction. Intermediate steps could be placed further behind in the output list, to guarantee that if the largest inline reduction fails, the shrinker will still be able to inline as far as possible. Also, the inlining feature could be extended to inline functions that take several parameters and consist of several statements, such as \( f \) in figure 18.

```c
int f(int a, int b) {
    int c;
    c = a + b;
    c = a + b + c;
    return c;
}
```

**Figure 18**: Function that takes several parameters and consists of several statements.

To successfully inline \( f \) we would first have to transform the function in several steps, as shown in figure 19.

```c
int f(int a, int b) {
    int c; => int f(int a, int b) {
    c = a + b; => int c; =>
    c = a + b + c; return a + b + a + b;
    return c; }
}
```

**Figure 19**: Once \( f \) has been shrunken to the rightmost form, it can be inlined at the call site.

These kinds of transformations resemble transformations that optimizing compilers perform. In contrast to optimizing compilers, we can allow the shrinker to be wrong sometimes. Optimizing compilers have to be careful not to change the semantics of the program when performing reordering or substitution. As an example, changing \( f() + g() \) to \( g() + f() \) can change the meaning of the program if \( f() \) and \( g() \) mutate some global state. With our shrinker, on the other hand, we can allow this since if we were to choose a reduction that caused the error to disappear, we could just try another one. There are several more possible transformations that we will not go into, one example being to remove all unused parameters in a function.

### 9.7 Controlled error generation

As of yet, the generator does not support controlled generation of programs that trigger errors. One of the goals for the implementation of the generator and the shrinker have been to guarantee that the generated programs are type-correct. In the implementation of the generator, we perform a set of actions to prevent the generated programs from containing type-errors. If we were to refrain from these actions, we would effectively make the generator generate programs with type errors. However, we believe that merely omitting or relaxing the strictness of some actions in the implementation does not provide enough control. We can instead implement features that allow a user to specify what kind of errors the generator should produce, and how likely they are to occur in a program. For example, a user might want only to generate programs that with some probability contain function calls where one of the arguments is of the wrong type. Another useful error could be missing variable declarations, where expressions refer to undeclared variables. These kinds of errors are useful in testing the type-checker and the implementation of scoping.
Let’s say that the user specifies that the generator should omit variable declarations with some probability $k \in [0, 1]$ by providing a flag, $\text{declErr} = \text{true}$, and the probability $p = k$. We can satisfy this request by some additional logic in the generation of functions, as shown in the algorithm in figure 20. Other errors, such as the previously mentioned error where one of the arguments in a function call is of the wrong type, can be created similarly.

1. Generate a function $f$
2. If $\text{declErr} = \text{true}$ then continue, else go to step 4
3. for each declaration $d \in \text{body}(f)$
   (a) generate a decimal number $x \in [0, 1]$
   (b) if $x < p$ then remove $d$ from the body of $f$
4. Register $f$ in the global scope

**Figure 20:** Algorithm for removing variable declarations.
10 Related work

The majority of the research we have seen is focused on testing compilers for the C programming language [6]. This is hardly surprising, since historically speaking, C is arguably the most established programming language in the software industry. Furthermore, implementations for testing C compilers can be adapted for testing compilers for most popular programming languages in the software industry. Besides research concerning the testing of C compilers, we also found several research projects from the functional programming community.

We chose to divide this section into two main categories: methods for which there exists an implementation for testing C compilers, and methods that focus on functional programming languages. Even though we present the methods in two separate categories for this survey, our division does not imply any excluding definition for any of the methods. For example, the methods that have been implemented for the testing of C compilers are based on ideas that are general enough to be adapted to programming languages that are unrelated to C.

10.1 Testing C compilers

Lindig [38] proposes a testing method that looks for errors in the implementation of function calls. While calling conventions for a platform can be different in several ways, all calling conventions have in common that any argument that is passed to a function has to be received unaltered. Lindig’s testing scheme first generates a random program consisting of calls to functions that asserts that the arguments are passed in unaltered. The program is then compiled and run; if any of the assertions should fail, a bug has been found. Lindig found 13 previously undetected bugs in mature C compilers, including GCC and Intel’s commercial C compiler ICC [41].

Lindig also compares the random programs generated by his tool to those found in a compiler test suite. The comparison shows that the functions calls and function declarations generated by his tool are more likely to contain an even distribution among the types that are used for arguments and returns. Separately, functions in the test suite were dominated by void, pointers, and int types. Lindig believes that the underrepresentation of types in the test suite is a reason why his tool was able to find so many bugs in otherwise mature compilers.

Yang et al. [30] have developed a method which they have used extensively for three years to identify more than 325 bugs residing in several industry-standard C compilers, among them GCC and LLVM [42]. Their testing scheme is based on differential testing with a set of compilers and random C programs generated by their tool CSmith. The generator in CSmith is unique in the sense that it is capable of generating programs containing a large subset of the C language constructs, while still being free from the 191 undefined behaviors listed in the C standard [43]. CSmith achieves this by running static analysis and dynamic checks during the generation process. Out of all the bugs that CSmith identified, 25 of the GCC bugs were classified as release-blocking, and the 202 bugs found in LLVM represents 2% of all LLVM bug reports.

Le et al. [37] present a method for testing compilers that they call Equivalence Modulo Inputs, or EMI. Their approach is based on modifying a given source program to create programs that are structurally different but still give the same output as the original program. These programs are then compiled by the same compiler. If any of the compiled programs should differ in output, a bug has been found. EMI has two significant benefits:

1. Tests can be constructed for a single compiler, without the need for a reference.
2. EMI is flexible in the sense that it can easily be applied to new domains - a change in the language of generated programs does not require an entire rewrite of an implementation.

Le et al. have successfully applied EMI on several industry-standard C compilers and reported on 147 unique bugs in GCC and LLVM.
Zhang et al. [31] have developed Skeletal Program Enumeration, or SPE, a method based on the observation that most bug-triggering programs are small, or can be reduced immensely. SPE generates random programs from a small input program by first analyzing the syntactic structure of the program, and then generating all unique permutations of variable arrangements. The naive approach to program enumeration is inefficient because identifiers in a program can be separate in name but equal in value. Therefore, many of the generated programs will be considered equal based on their output. The algorithm behind SPE can avoid duplication and still generate all unique permutations. Together with the fact that SPE does not favor large programs, this property makes the method exhaustive and fast. With their method, Zhang et al. have found 217 bugs in GCC and LLVM [42] in less than six months. Moreover, they have also implemented their method for the functional programming language Scala [19].

10.2 Testing functional compilers

Pałka et al. [5] have developed a method for generating type correct randomized programs based on what essentially amounts to running the type system backward. The generation algorithm operates by successively applying rules from the type system for simply-typed lambda calculus in a bottom-up, goal-oriented fashion. They use their algorithm to generate random programs to test an industry-standard compiler for the purely functional programming language Haskell [12], and manage to identify bugs in the optimization stage. Their method also incorporates a shrinking scheme to minimize the erroneous source code - simplifying the program, while still making sure that the same error is triggered. By automating this process, the method of Pałka et al. can expose the compiler to hundreds of tests in a few minutes, and when a bug is found, the erroneous program is minimized to just a few constructs.

The method of Pałka et al. [5] does not take semantic underspecification into account. Semantic underspecification allows implementations to differ while remaining correct according to the language specification. A common example of semantic underspecification is the evaluation order of parameters to functions. Midtgaard et al. [13] have extended the method of Pałka et al. by introducing effects into the type system in order to avoid generating programs that depend on evaluation order. More precisely, they model the possibility that a chosen type rule will generate a term that in combination with certain other terms are subject to evaluation order dependence.

With their generator, Midtgaard et al. successfully test an industry-standard compiler for the functional programming language OCaml [44]. Midtgaard et al. find bugs in the compiler by generating programs and performing differential testing with a reference compiler. Since the OCaml language standard does not specify evaluation order, successful use of differential testing would not have been possible without their extension.

Ofenbeck [45] et al. have developed a Scala-based framework for random program generation based on a provided specification of the grammar of a programming language. To generate code for a programming language, the user of the framework has to supply the grammar of the language encoded with functions and data types. Since the generator operates completely based on the provided specification of the grammar, the framework can be used to generate random programs for any programming language that can be encoded using Scala’s type system. The generator is also able to provide a Scala version of the generated program that can be used as a reference to compare against. Furthermore, the generator offers test case shrinking, as seen in the work of Pałka et al. Thanks to these capabilities, their framework can be used to construct effective automated random testing schemes for programming languages for which there is only one compiler implementation.

However, because of the generality of their framework, their generator is unable to avoid generating programs that exhibit language specific undefined behaviors or trigger exceptions. Such issues are left to the user to encode in their provided specification. Moreover, if there is no possible one-to-one translation between Scala and the user-specified grammar, a comparison based test using the generated Scala reference program is likely to give false negatives. For this reason, the user-specified grammar will most likely be limited to a subset of their selected programming language.
11 Conclusions

With this thesis, we have attempted to answer the following question - whether we can successfully use type-directed generation and shrinking for the testing of compilers for imperative programming languages. We have implemented a generator that is capable of generating type-correct programs that contain all of the program constructs of the language we defined, a subset of C. In a test that measured statement coverage, the generated programs achieved the same level of coverage as a reference test suite. We also implemented a type-directed shrinker that can take any program that the generator produces and shrink it to a smaller program, without introducing errors. With the generator and the shrinker, we constructed a test that exposed an interpreter to a set of programs and compared the output of the interpreter to that of a reference interpreter. If the tested interpreter provided the wrong output for some program, the test would enter a shrinking phase were the error-triggering program was minimized. By running the test on seven anonymized interpreters from a compiler course at MDH, we showed that the generator is a powerful tool for triggering errors: it found bugs in five of the interpreters, a total of seven bugs. Moreover, the shrinker turned out to be a powerful tool in determining the cause of these errors.

To evaluate our results, we ran the previously mentioned reference test suite on the same seven interpreters and found that it would only trigger errors in two of the interpreters, three errors in total. Furthermore, identifying the cause of one of the errors proved cumbersome without the help of a tool like the shrinker. Together, the results showed that even though the generator and the test suite achieved the same statement coverage, the reference test suite triggered less than half of the bugs that the generator triggered. These results suggested that statement coverage and other types of code coverage are perhaps not reliable measures of quality for compiler test suites. To answer our problem formulation, we compared type-directed generation and shrinking to testing using traditional test suites and discussed how we would use the methods together. Finally, we reached the conclusion that type-directed generation and shrinking can be used successfully for the testing of compilers for imperative programming languages.

For future work, we are interested in pursuing some of the ideas for improvements of the current implementation, which we discuss in the previous section. These improvements include but are not limited to: enhancing the generator with program analysis, making it possible to generate self-recursive functions and mutually recursive groups of functions, implementing controlled error generation, as well as enhancing the capabilities of the shrinker, for example with function inlining.
References


