Thesis for the Degree of Licentiate of Philosophy

Testing an Optimising Compiler by Generating Random Lambda Terms

Michał H. Pałka

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2012

Testing an Optimising Compiler by Generating
Random Lambda Terms
Michał H. Pałka

Technical Report 94L
ISSN 1652-876X
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology and
Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Göteborg, Sweden 2012

Testing an Optimising Compiler by Generating Random Lambda Terms

Michał H. Pałka
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis tries to improve on the relatively uncommon practice of random testing of compilers.

Random testing of compilers is difficult and not widespread for two reasons. First, it is hard to come up with a generator of valid test data for compilers, that is a generator of programs. And secondly, it is difficult to provide a specification, or test oracle, that decides what should be the correct behaviour of a compiler. This work addresses both of these problems.

Existing random compiler test tools do not use a structured way of generating well-typed programs, which is a often a requirement to perform comprehensive testing of a compiler. This thesis proposes such a method based on a formal calculus.

To address the second problem, this thesis proposes using two variants of differential testing, which allows for detecting bugs even when a very limited partial specification of the tested compiler is available. This setup is evaluated practically by performing effective testing of a real compiler.

Contents

Acknowledgements

I would like to thank my supervisors, Koen Claessen and John Hughes, for their guidance and support. Both of them were a continuous source of ideas and contagious enthusiasm. I also wish to thank my examiner, Mary Sheeran for providing valuable feedback throughout my studies.

I would like to thank Robby Findler for accepting the role of discussion leader.

I would like to thank all of my colleagues at the department for making it such an open and friendly environment. I was lucky to have a chance to interact with many extraordinary individuals, many of whom contributed suggestions that helped me in writing this thesis.

I wish to thank Olga for her company and cheering up during this time. I also want to thank all my friends for their encouragement and friendship. Finally, I would like to thank my parents for all the love and support throughout my life.

Chapter 1

Introduction

Correctness of compilers is crucial to most software projects, yet comparatively little stress is put on their quality assurance compared to other software. While there exist examples of formally-verified compilers, such as the CompCert optimising compiler [19] and the Racket compiler and virtual machine [17], such compilers are not common. Instead of using modern testing and verification techniques, writers of production compilers prefer to have them tested 'in the field' by the users. But off-loading the quality assurance to the users has the disadvantage that many serious bugs are discovered only when they cause disruption.

Formal verification is an attractive approach to reliable software, but its cost and complexity are still sky-high, which makes it applicable only to very specialised compilers. For instance, over $3/4$ of the code of CompCert is devoted to verification [19]. Software testing, on the other hand, is the most prevalent and economical way of assuring quality, which, if done right, may result in quicker software development [32].

Relying on the users to find bugs may not sound a bad idea if we ignore the inconvenience caused to them. However, even though the cost of finding bugs is outsourced, the cost of fixing them might be dramatically higher when bugs are found late [32].

Thus, compiler writers resort to test suites[1], which are run continuously during development. But these test suites consist largely of test cases that were taken from bug reports submitted by the users,

[1] For example, the GCC compiler test suite: `http://gcc.gnu.org/onlinedocs/gccint/Testsuites.html`; or the GHC test suite: `http://hackage.haskell.org/trac/ghc/wiki/Building/RunningTests/Adding`.

which means that they tend to find few new bugs.

Having an automatic testing tool for a compiler, on the other hand, would give the advantage of finding bugs early. One possibility is to use random property-based testing. However, this requires having a generator of random programs.

Generating good test programs is not an easy task, since these programs should have a structure that is accepted by the compiler. As compilers often employ multi-stage processing before producing compiled code, in order to test later stages, earlier ones must be completed without error. The requirements for passing a compilation stage can be as basic as a program having the correct syntax, or more complex such as a program being type-correct in a statically-typed programming language.

In this thesis, we investigate generation of random type-correct Haskell programs for the purpose of testing the GHC Haskell compiler, and in particular, its optimising middle-end [29].

We chose the *simply-typed lambda-calculus* [30] as the underlying model of well-typed programs as it is the simplest calculus that has the notion of variable binding and type-correctness, as well as first-class functions, which makes it adequate for representing simple Haskell programs. On top of that, we support polymorphic constants, which makes it possible to generate simple Haskell programs that use polymorphic library functions as well as to encode in them some programming language constructs.

The presented generator of random simply-typed lambda-terms has been successfully applied to testing the GHC Haskell compiler. This compiler contains a particularly sophisticated optimising middle-end, which performs many stages of intermediate-language transformations, such as inlining, let-floating, lambda lifting, specialisation and common subexpression elimination [29]. Such elaborate processing could easily be a source of intricate bugs, making it especially interesting to test.

The GHC compiler has been tested with randomly generated Haskell programs using *differential testing* [23], which involved compiling the same program with different optimisation settings and comparing its observed behaviour. Figure 1.1 shows a diagram of the testing process. Testing uncovered four interesting bugs, three of which were fixed based on our bug reports[2]. We also learned interesting facts about valid optimisations performed by the GHC.

Reported failing test cases were often reasonably small due to the

[2]The remaining one has been fixed as a result of another bug report.

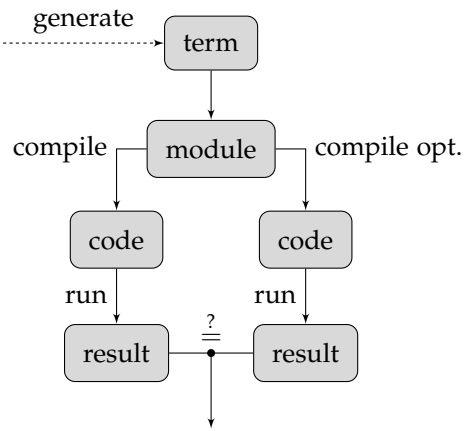

Figure 1.1: Differential testing

process of *shrinking*, which automatically reduces the failing test case as much as possible. Below is one of the found terms that provoked a failure.

```
(\a -> seq a (seq (a []) id)) (\a -> seq undefined (+1))
```

This term can be manually rewritten as follows, to obtain a program that provokes a bug:

```
a = \x -> seq undefined (+1)

main = do
  print $ (a [] `seq` id) [0]
```

This short program turned out to be miscompiled by the tested version of GHC when optimisation was turned off. The failure has been reported as ticket 5625 and the bug causing it was fixed. More information about this failure can be found in Section 1 of Chapter 4.

1 Related work

Even though random testing is not commonplace in compiler development, there are accounts of its successful application. The work of McKeeman [23] and the CSmith tool [35] are both such examples. Both tools employ *Differential testing* where results of programs

compiled with different C compilers are compared to spot bugs. Generating random programs was a central part of both of these efforts, and in both cases the problem was solved using an elaborate, but ad-hoc generator.

McKeeman focuses on testing all stages of a C compiler by generating programs of different level of conformity to the C language: lexically-correct, syntax-correct, etc., and much effort is put into creating program generators for each of these levels. CSmith, on the other hand, focuses on testing optimising middle-ends of the compilers and most effort was put into generating C programs that do not depend on undefined or unspecified behaviour. Assuring this semantic property is, of course, difficult, which is why CSmith uses an array of heuristics and checks for assuring it.

Notwithstanding the effort put into creating these program generators, both test tools found a big collection of previously unreported bugs in the tested compilers. The test cases reported by these tools are usually large and hard to analyse. Only McKeeman discusses automated reduction of size of counterexamples—in case of CSmith they have to be reduced by hand, which can be very labour-intensive.

Lindig [20] created a random testing tool to test whether the C function calling convention is followed by compilers. The generation of programs is much simpler in this case, as only types of functions to be test-called need to be randomly generated. The rest of the program is skeleton code that is generated algorithmically. Despite its simplicity, the tool managed to find a number of discrepancies between compilers that manifested when a function was called from code compiled with another compiler. Automated testing of unexpected cases was important.

Random program generators have also been successfully applied to testing Java libraries [18]. This work defines a formal calculus, whose random terms are then transformed into programs and is able to generate programs containing higher-order features.

Verified compilers The CompCert optimising compiler [19] and the Racket compiler and virtual machine [17] are notable for their quality assurance. The former is a full-fledged optimising compiler that was constructed with formal verification in mind. However, it only supports a subset of C and the catalogue of optimisations performed by it is limited. Furthermore, even though the most important of its parts are formally-verified, which was a costly task, testing was able to find in it previously unknown bugs [35].

The Racket compiler is verified using lightweight formal verification, which is realised by testing the compiler against a formal model. While the compiler supports unrestricted version of the Scheme programming language, it is only capable of performing basic optimisations.

Term generators Generation of random lambda-terms has attracted moderate attention. There were attempts at generating random untyped lambda terms [3, 34], where it is already difficult to obtain a reasonable distribution. Two notable attempts at generating simply-typed lambda terms are based on bit-encoding schemes [33] and enumeration [31], but it is unclear whether it is possible to adapt the latter for random generation.

Unfortunately, none of these works considers any practical applications for randomly generated lambda-terms—the only application considered is examining statistical properties of random lambda terms [3, 34]. Also, none of the typed generators handles parametric polymorphism.

2 Structure

The rest of this chapter introduces *property-based testing*, which is the way of structuring test data generators and test oracles used throughout the paper. In the next chapter we explain our approach to random generation of simply-typed lambda terms. Chapter 3 describes the method of structurally shrinking the generated terms aimed at reducing the sizes of reported counterexamples. We identify two interesting design decisions that we can make in the shrinking method and evaluate their performance based on experimental benchmarks. Chapter 4 presents the failing test cases found for GHC and properties used to find them. We also analyse the consequences of the noticed discrepancies for the programmers using GHC. Chapter 5 describes related work, and Chapters 6 and 7 present future work and conclusions.

Contributions We claim the following contributions:

- We present a generator of random simply-typed lambda terms based on performing random local choice and backtracking. The generator makes use of tailored generation rules and heuristics to avoid excessive backtracking and skewing the distribution

of generated terms too much. The generator also supports generating terms with polymorphic constants. (Chapter 2)

- The generator is applied successfully for finding bugs in GHC optimising compiler using differential testing. (Chapter 4)
- We perform differential testing using two expressions that should give equivalent results and find discrepancies that are independent of optimisation options. (Chapter 4)
- We present a method for shrinking simply-typed lambda terms, which is used for reducing counterexamples found by us during testing. The method proposed by us reduces the terms structurally, but is very effective in reducing the test cases despite the complexity of the processing performed by GHC. Shrinking makes finding the root cause of a bug based on a counterexample dramatically easier. (Chapters 3 and 4)

The presented dissertation is an extended version of existing work [25].

3 Property-based testing

To obtain test oracles for random testing we employ *property-based testing*, which allows us to derive test oracles naturally from logical properties. For example, consider the commutativity law for integers.

$$\forall n\, m.\ n + m = m + n$$

We may turn it into a testable property by writing a Haskell function that checks this equality for two given numbers.

```
prop_comm :: Integer -> Integer -> Bool
prop_comm n m = n + m == m + n
```

This function is an executable version of the above logical property and may be used as an oracle in random testing. We use the word 'property' to denote such executable properties throughout this paper.

A single test on the above property is performed by generating two random numbers `n` and `m`, evaluating the function and checking if its result is true.

Testing this property involves running the function on a finite number of inputs when the number of all inputs is infinite, so testing

can only result in *disproving* the property, by finding a counterexample, or leaving its validity undecided.

The function `prop_comm` implements both the tested code (the + operator) and the test oracle (the == operator), while the random test case generator is separated.

The properties used by us to test the GHC compiler employed differential testing, described in Chapter 4, and an example diagram illustrating the process is shown in Figure 4.1 in that chapter.

QuickCheck [4] is a Haskell library that provides comprehensive support for property-based testing. QuickCheck contains a combinator library for building composable properties as well as random generators for basic Haskell data types. We implemented our random simply-typed term generator in QuickCheck using these basic generators and performed testing of the GHC compiler using properties also written in QuickCheck.

QuickCheck also provides generic support for *shrinking* [4][3] that tries to reduce the size and complexity of the reported counterexamples. We implemented a shrinking method for the simply-typed lambda terms for use in our testing.

The method for generation of random simply-typed lambda terms is independent from QuickCheck, however many other aspects of the testing are heavily influenced by property-based testing, for example the properties and shrinking.

[3] Shrinking is not referred to by name in this paper, but is realised by the function 'smaller'.

Chapter 2

Generation method

To generate random programs for use in testing we consider simply-typed lambda terms [30] that can contain constants in addition to variables. Their syntax is shown in Figure 2.1. Typing rules, shown in Figure 2.2, are standard and constants are typed in the same way as variables. We require that all variables and constants in environments have distinct names.

The aim of the generator is to produce a well-typed term of a certain type, which can contain free variables and constants from a given environment. Of course there are combinations of target type and environment for which no term can be constructed.

Simple generator One possible generation method can be obtained by reading the typing rules shown in Figure 2.2 backwards. To generate a term that is in the consequence of a rule it is firstly necessary to generate terms that are in its premises, if they exist, and then combine them. In other words, the *goal* of generating a term might involve generating the *subgoals* recursively, leading to a procedure that works top-down and produces a term together with its typing derivation. Of course, the derivation must be finite. The rules ensure that the

Variables	$x, y, \ldots$		
Constants	$\mathbf{c}, \mathbf{d}, \ldots$	::=	$\texttt{head}, \texttt{tail}, \texttt{(+)}, \texttt{0}, \texttt{1}, \ldots$
Types	$\sigma, \tau, \ldots$	::=	$\texttt{Int} \mid \texttt{Bool} \mid \texttt{ListInt} \mid \cdots \mid \sigma \to \tau$
Terms	$M, N, \ldots$	::=	$x \mid \mathbf{c} \mid \lambda x{:}\sigma.\ M \mid MN$

Figure 2.1: Syntax for simply-typed λ-calculus

Typing judgements $\Gamma \vdash M : \sigma$
Environment $\Gamma ::= \{x_1 : \sigma_1, x_2 : \sigma_2, \mathbf{c_1} : \sigma_3, \ldots\}$

$$(\textsc{Var})\ \frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma} \qquad (\textsc{Cnst})\ \frac{\mathbf{c}:\sigma \in \Gamma}{\Gamma \vdash \mathbf{c}:\sigma} \qquad (\textsc{Lam})\ \frac{x:\sigma, \Gamma \vdash M:\tau}{\Gamma \vdash \lambda x{:}\sigma.M : \sigma \to \tau}$$

$$(\textsc{App})\ \frac{\Gamma \vdash M:\sigma \to \tau \qquad \Gamma \vdash N:\sigma}{\Gamma \vdash MN:\tau}$$

Figure 2.2: Typing rules

resulting terms are well-typed.

Suppose that we want to generate a simply-typed λ-term of type `Int` while having access to the following constants: `z` : `Int` and `s` : `Int` $\to$ `Int`.

The simplest term that can be generated is just `z`, as this constant has the right type. This term is generated by applying the Cnst rule with $\mathbf{c}$ *instantiated* with the constant `z`. This rule does not have any recursive premises, only a side condition that `z` must be in the environment, so no subgoals must be generated to finish the generation. Successful generation yields a type derivation tree for the generated term, shown below. We define Γ to be the initial environment $\{\texttt{z:Int}, \texttt{s:Int} \to \texttt{Int}\}$.

$$(\textsc{Cnst})\ \frac{\texttt{z:Int} \in \Gamma}{\Gamma \vdash \texttt{z:Int}}$$

We can generate another term if we apply `s` to `z` and obtain the term `s z`, which also has the right type. To do that by following the typing rules, we must first apply an instance of the App rule where both σ and τ are instantiated with `Int`. Applying this rule requires two subgoals to be generated recursively with their own *generation contexts*, that is their target types together with their environments. Below we show the part of the derivation tree that is determined after selecting the App rule.

$$(\textsc{App})\ \frac{\Gamma \vdash ?_1 : \texttt{Int} \to \texttt{Int} \qquad \Gamma \vdash ?_2 : \texttt{Int}}{\Gamma \vdash ?_1\ ?_2 : \texttt{Int}}$$

The question marks ($?_1$ and $?_2$) represent the subterms that will be generated as subgoals.

The first subgoal has the same environment as the original term, but a different target type, which is `Int → Int`. This subgoal is generated using the Cnst rule instantiated with the constant `s`.

The second subgoal receives the same generation context as the original term and is generated using the Cnst rule.

Solving the subgoals yields the missing parts of the term and its derivation tree.

$$\text{(App)}\ \frac{\text{(Cnst)}\ \dfrac{\texttt{s:Int} \to \texttt{Int} \in \Gamma}{\Gamma \vdash \texttt{s:Int} \to \texttt{Int}} \qquad \text{(Cnst)}\ \dfrac{\texttt{z:Int} \in \Gamma}{\Gamma \vdash \texttt{z:Int}}}{\Gamma \vdash \texttt{s z:Int}}$$

In the same way it is possible to generate more complex terms, for example `s (s z)`. The terms can also contain locally-defined functions in the form of λ-expressions. In this way, we can create more type-correct λ-terms, for example $(\lambda x : \texttt{Int}.x)\ (\texttt{s z})$, or $(\lambda x : \texttt{Int}.\texttt{s}\ (\texttt{s}\ x))\ \texttt{z}$. The Lam typing rule, which is needed for generating a locally-defined function, adds one variable to the environment of its subgoal, which makes it possible to refer to that variable in the body of the function.

Thus, interpreting the typing rules as *generation rules* allows us to generate well-typed terms in a top-down fashion. The four rules are capable of generating every well-typed term—after all a term is well-typed only if there exists a typing derivation for it.

Using a generation rule to generate a part of a term involves choosing its specific instance. Choosing a suitable instance of the Var and Cnst rules is straightforward, as the type of the chosen variable or constant must be the same as the target type. If such variable or constant is not available, then the rules cannot be applied.

Similarly, the Lam rule can only be applied if the target type is functional and type σ from the rule must be equal to the argument of the target type.

This is not the case, however, when the App rule is applied, as the type of the argument is not determined by the generation context and can be chosen freely.

Often a specific instance of the App rule is required to occur in a derivation tree of a term. For example, suppose that a derivation tree might only be constructed by using constant **c** that has type $\sigma_1 \to \sigma_2 \to \sigma_3 \to \tau$ to solve a goal of type τ. The App rule must be then instantiated with σ_3. Furthermore, two more applications of the App rule are required, also with specific types.

In addition, some instances of the App rule cannot occur in a valid derivation tree. Notably, if it is instantiated with an argument type, for which a term cannot be constructed, then no valid derivation containing it exists.

Thus, the derivation is *sensitive* to the way the App rule is instantiated.

Concrete realisation The generation rules give us a *non-deterministic* generation procedure as in a given generation context it might be, and frequently is, possible to use more than one instance of the rules. Unfortunately, we can not have the luxury of allowing the generator to select an arbitrary viable instance, as bad choices may lead the generator to a dead end or non-termination, even if making another choice would result in successful generation.

These problems can be solved by using a procedure that performs backtracking. Whenever a bad choice is made, the procedure will fail and backtrack to another choice. Given that it is possible to run into an infinite loop, a limit on the number of recursive invocations is imposed.

Unfortunately, since the success or failure of the generation depends so much on the way the App rule is instantiated, such generation procedure suffers from excessive backtracking, which is why we propose a different set of generation rules.

1 Alternative rules

We chose to use an alternative set of generation rules instead of adopting the typing rules for that purpose. Consider following method where a term can be generated in two ways:

- We may introduce a lambda expression if the target type is functional. The body of the lambda expression must then be generated to complete the term.

- We may use a symbol from the environment, constant or variable, that possibly needs some arguments to be applied to it to match the target type. The needed arguments become new goals that have to be generated. If the type of the symbol is the same as the target type, no arguments are needed and generation is finished.

The first tactic is captured by the LAM rule in Figure 2.2 and the formal rule for the second one is given below:

$$(\text{INDIR}_\text{V})\ \frac{f:\sigma_1 \to \ldots \to \sigma_n \to \tau \in \Gamma \quad \Gamma \vdash M_1:\sigma_1 \ \cdots\ \Gamma \vdash M_n:\sigma_n}{\Gamma \vdash f M_1 \ldots M_n : \tau}$$

There is also a sister rule for constants, which we omit here. Please note that τ can be any type, also functional, and that the INDIR rules supersede VAR and CNST if $n = 0$.

This choice of generation rules gives us a generation method with interesting properties. First, it is not possible to generate all terms with it, as it never generates any β-redexes. Secondly, it is nevertheless complete, as it will be able to generate a term if a well-typed term exists for a given combination of the target type and environment. And finally, the problem of generation being very sensitive to the way rules are instantiated is reduced. The reason for this is that applying the INDIR rule does not involve choosing any types, and while choosing the wrong variable or constant might result in failed generation there is a finite number of choices to make.

In a way, the INDIR rule is a 'guided' APP rule, which chooses the argument types to suit a specific variable or constant from the environment.

Keeping the APP rule For the purpose of practical generation, we nevertheless keep the APP rule in addition to the INDIR rule to be able to generate β-expanded terms. While the INDIR rule gives higher chances of successfully generating a term without excessive backtracking, the APP rule is capable of generating β-expanded terms.

2 Polymorphic constants

So far we discussed the simply-typed lambda calculus with monomorphic constants, which means that constants have specific types. However, typical Haskell programs make use of *parametric polymorphism*, which requires a richer term representation.

The type system of Haskell is quite complex, but a large part of code in Haskell uses only one aspect of parametric polymorphism, notably *polymorphic constants*[1]. Other variations of polymorphism

[1] In particular, Haskell 98 programs are restricted to Hindley-Milner polymorphism with monomorphic let bindings if no explicit type signatures are provided [26].

Variables	$x, y, \ldots$	
Constants	$\mathbf{c}, \mathbf{d}, \ldots$	$::= \texttt{head}, \texttt{tail}, \texttt{+}, \texttt{0}, \texttt{1}, \ldots$
Type variables	$\alpha, \beta, \ldots$	
Types	$\sigma, \tau, \ldots$	$::= \texttt{Int} \mid \texttt{Bool} \mid \sigma \to \tau \mid \alpha \mid \texttt{List}\ \sigma \mid \cdots$
Polymorphic types	$\Sigma, \Upsilon, \ldots$	$::= \forall\alpha\beta\gamma \cdots .\sigma$
Terms	$M, N, \ldots$	$::= x \mid \mathbf{c} \mid \lambda x{:}\sigma.\ M \mid MN$

Figure 2.3: A simple λ-calculus with polymorphism

in Haskell, such as *higher-rank types* or polymorphic let definitions require a more complicated type system, which is why we decided to support only polymorphic constants.

Polymorphic constants can be used to represent functions operating on polymorphic data structures, such as lists. They are also needed to build expressions that use advanced Haskell libraries, including ones involving monads, applicative functors and arrows.

Polymorphic constants can also be used to encode some programming language constructs, such as if-then-else, which means that support for them does not have to be hard-coded into the generator.

Instead of supporting polymorphic constants, we could replace them with a number of their instances, for example:

$$\texttt{head}_{\texttt{Int}} : \texttt{List Int} \to \texttt{Int}$$
$$\texttt{head}_{\texttt{Int}\to\texttt{Int}} : \texttt{List}\ (\texttt{Int} \to \texttt{Int}) \to \texttt{Int} \to \texttt{Int}$$
$$\ldots$$

However, this would lead to an explosion of constants and, given that there are infinitely many instances of each (non-trivial) polymorphic type, it is not clear which and how many instances should be included.

To accommodate polymorphic constants we must extend our core calculus, as shown in Figure 2.3. Polymorphic types are written as $\forall\alpha\beta\gamma \cdots .\sigma$, where $\alpha, \beta, \ldots$ are type variables, which are allowed to occur in σ and are replaced by types during instantiation. It is illegal for a type variable to occur in a non-polymorphic type even though the syntax allows this.

The typing rules, shown in Figure 2.4, are only slightly different to previous typing rules. All terms have monomorphic types. Lambda-bindings and, in consequence, variables are monomorphic. Also, the occurrences of constants are fully instantiated, and thus all have

$$\begin{array}{lrl} \text{Typing judgements} & & \Gamma \vdash M : \sigma \\ \text{Environment} & \Gamma \quad ::= & \{x_1 : \sigma_1, x_2 : \sigma_2, \mathbf{c_1} : \Sigma_1, \ldots\} \\ \text{Type substitution} & & [\alpha \mapsto \tau_1, \beta \mapsto \tau_2, \ldots] \end{array}$$

$$(\textsc{Var})\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad (\textsc{Cnst})\ \frac{\mathbf{c} : \forall \alpha\beta \cdots . \sigma \in \Gamma}{\Gamma \vdash \mathbf{c} : \forall \alpha\beta \cdots . \sigma[\alpha \mapsto \tau_1, \beta \mapsto \tau_2, \ldots]}$$

$$(\textsc{Lam})\ \frac{x : \sigma, \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau} \qquad (\textsc{App})\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

Figure 2.4: Typing rules for simple λ-calculus with polymorphism

concrete types. The only rule that has changed is Cnst, which now performs instantiation of the constant's polymorphic type.

Generation rules for terms with polymorphic constants also need only small changes compared to previous ones. The rule that changes is $\textsc{Indir}_{\mathrm{C}}$, which introduces constants:

$$\Sigma[\alpha \mapsto \rho_1, \ldots] = \sigma_1 \to \ldots \to \sigma_n \to \tau$$
$$(\textsc{Indir}_{\mathrm{C}})\ \frac{\vdots \quad \mathbf{f} : \Sigma \in \Gamma \qquad \Gamma \vdash M_1 : \sigma_1 \quad \cdots \quad \Gamma \vdash M_n : \sigma_n}{\Gamma \vdash \mathbf{f} M_1 \ldots M_n : \tau}$$

Constant $\mathbf{f}$ can be used if its polymorphic type Σ can be instantiated so that its result is the same as the target type τ, which is realised by the side condition marked with vertical dots.

2.1 Instantiation

Of course, in order to use the $\textsc{Indir}_{\mathrm{C}}$ rule, both $\mathbf{f}$ and an instantiation of its type must be chosen. Depending on the circumstances, this might be easy or difficult. The following examples examine different cases.

Example 1 Consider that the following constant is present in the environment:

$$\texttt{tail} : \forall \alpha . \texttt{List}\ \alpha \to \texttt{List}\ \alpha \in \Gamma$$

and that the target type is `List Int`. If we want to use `tail` to generate this term we have to find an instance of the $\textsc{Indir}_{\mathrm{C}}$ rule. But the only instantiation that can possibly be used here is $[\alpha \mapsto \texttt{Int}]$, and thus the

generation step might look as follows:

$$\text{(INDIR}_{\text{C}}\text{)}\ \frac{\Sigma[\alpha \mapsto \texttt{Int}] = \texttt{List}\ \alpha \to \texttt{List}\ \alpha \quad \vdots \quad \texttt{tail}{:}\Sigma \in \Gamma \qquad \frac{\cdots}{\Gamma \vdash \texttt{?}{:}\texttt{List}\ \alpha}}{\Gamma \vdash \texttt{tail ?}{:}\texttt{List}\ \alpha}$$

Therefore, in this case the instantiation is completely guided by the environment and target type.

Example 2 A slightly more complicated situation occurs when the identity function is to be instantiated. The identity function has following type:

$$\texttt{id}{:}\forall\alpha.\alpha \to \alpha$$

Suppose that, for example, the type of the expression that is to be generated is `Int`. Then, the most obvious choice is to instantiate α with `Int` and generate `id` applied to an `Int` argument. However, it is not the only choice. We may as well instantiate α with `Bool` $\to$ `Int` in which case applying `id` to an argument yields a *function* of type `Bool` $\to$ `Int`. Thus, `id` effectively becomes a *two-argument function* that can be applied to arguments `Bool` $\to$ `Int` and `Bool`. Continuing this way, we may instantiate α with `Bool` $\to$ `Bool` $\to$ `Int` and obtain an instance of `id` that takes *three* arguments. We may carry on adding arguments to `id` like this forever, even if the resulting terms would look uncommon.

This demonstrates that whenever a constant has type that looks like $\forall\alpha \cdots . \cdots \to \alpha$, there will be infinitely many possible ways of instantiating it. Fortunately, it is not likely that constants applied to many 'extra arguments' are relevant for generating interesting terms.

Example 3 Consider the function map and that we want to use it to generate a list of integers. Map is represented with the following constant:

$$\texttt{map}{:}\forall\alpha\beta.(\alpha \to \beta) \to \texttt{List}\ \alpha \to \texttt{List}\ \beta$$

Since the resulting term has the `List` type, the constant must be applied to two arguments. Furthermore, the target type of `List Int` dictates that β in the type of `map` must be instantiated with `Int`. However, nothing constrains how α must be instantiated, making it possible to use any instantiation of it.

Therefore, many instantiations of `map` are possible, and the same problem occurs with several important constants that we might want

to use for generating terms, such as:

$$\begin{aligned}\texttt{ap} &: \forall\alpha\beta.\texttt{A}\ (\alpha \to \beta) \to \texttt{A}\ \alpha \to \texttt{A}\ \beta\\ \texttt{bind} &: \forall\alpha\beta.\texttt{M}\ \alpha \to (\alpha \to \texttt{M}\ \beta) \to \texttt{M}\ \beta\end{aligned}$$

The problem is important as specific instances of `map` and similar constants are likely to be required to generate some interesting terms. For example, `bind` has to be used (and instantiated) in order to build complex monadic expressions.

Also, not surprisingly, if we include a constant that represents function application, exactly the same problem would occur as we would have to select α in the following type:

$$\texttt{(\$)} : \forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \beta$$

Therefore, the problem of instantiating such constants is more general then the problem of instantiating the App rule.

These three examples show that the problem of instantiating polymorphic types in the Indir rule is (a) easily solvable in some cases and (b) choosing arbitrary types might be needed in other cases, which is not crucial for generation, but (c) there are also important cases where instantiation is critical.

We were not able to solve this problem completely, but we present a partial solution of it, which is based on heuristics, in our generation algorithm.

3 Generation algorithm

The algorithm generates terms recursively top-down by applying generation rules. To avoid non-terminating generation each recursive invocation of the algorithm uses a *size parameter*, which is decreased in subsequent invocations.

The first step of each recursive invocation is to create a list of admissible instances of the generation rules, that is the instances that can be used in the current context.

When the size parameter is 0 the list is restricted only to non-recursive rule instances, and thus the generation is forced to succeed or fail without further recursion.

The list is then permuted at random and the first rule instance from it is applied. If the rule instance requires subterms to be generated, these get turned into subgoals and the generation procedure is

invoked recursively with new generation contexts.

When the recursive invocations succeed, their results are combined into the resulting term according to the generation rule and the current invocation of the generator concludes. When they fail, the generator performs backtracking by selecting the next admissible instance of some generation rule from the list. If there are no more instances left, the original recursive invocation fails.

The size parameter used to generate the subgoals of a goal is decreased using the following equation, where p is the number of immediate subterms that need to be generated and s_{g} is the size parameter of the goal.

$$s_{\mathrm{sg}} = \left\lfloor \frac{(s_{\mathrm{g}} - 1)}{p} \right\rfloor \tag{2.1}$$

Example Consider that the environment contains constants $\texttt{tail} : \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha$ and $\texttt{x} : \texttt{List Int}$. Note that we omit writing $\forall$ when a polymorphic type does not have type parameters. If the type of the term to be generated is `List Int`, the admissible rule instances are the following:

- The Indir rule may be instantiated with `tail` and $n = 1$, which requires generating a subterm of type `List Int`. The instantiation involves a substitution $[\alpha \mapsto \texttt{Int}]$ and both τ and σ_1 are instantiated with `List Int`. The instance is presented below in full:

$$(\textsc{Indir}_{\mathrm{C}})\ \frac{\forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha[\alpha \mapsto \texttt{Int}] = \texttt{List Int} \to \texttt{List Int} \qquad \vdots\ \ \texttt{tail} : \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha \in \Gamma \qquad \Gamma \vdash M_1 : \texttt{List Int}}{\Gamma \vdash \texttt{tail}\ M_1 : \texttt{List Int}}$$

- The Indir rule may similarly be instantiated with `x` and $n = 0$.
- The App rule may be instantiated with τ mapped to `List Int` and with σ mapped to any type.
- There is no admissible instantiation of the Lam rule.

If the size parameter is 0, then Indir instantiated with `x` becomes the only admissible instance, as it is the only one that is non-recursive.

The algorithm described above is an ordinary randomised search algorithm with backtracking and size limit. The important part of it is

how the set of admissible rule instances in a given context is selected. Ideally, we should select all the admissible rule instances, but because of the possible many instantiations of some constants, this set might be infinite and therefore we must approximate it by selecting the subset of all possible instantiations. The following procedure is used:

- Admissible instances of $\textsc{Indir}_{\text{V}}$ and $\textsc{Lam}$ are selected completely as there is a finite number of them. Variables are monomorphic and the type of the λ-bound variable is uniquely determined by the generation context.
- Applying the $\textsc{App}$ rule involves choosing the argument type. $N_{\mathsf{app\_tries}}$ types are chosen at random and instances of $\textsc{App}$ based on these types are included in the set.
- There are three cases involving the $\textsc{Indir}_{\text{C}}$ rule, inspired by the examples that we discussed previously:
  1. When the instantiation of a given constant is unique, that instantiation is chosen.
  2. When the constant's type looks like $\forall\alpha \cdots . \cdots \to \alpha$, the constant can be applied to a number of extra arguments. Up to $N_{\mathsf{extra\_arg}}$ extra arguments can be applied, and any types that are not determined are chosen at random.
  3. When the constant's type is not fully instantiated based on the target type, the remaining types are chosen at random.

Limiting the number of instances of $\textsc{App}$ and $\textsc{Indir}_{\text{C}}$ is done by selecting arbitrary types at random in places where any type is allowed. The procedure for random selection of types chooses them based on the set of types available in the environment. First, a set of base types is created by considering different combinations of symbols from the environment and then a 'small' random type is generated from it.

Parameter $N_{\mathsf{app\_tries}}$ is arbitrarily fixed to 5 and this number of instances of the $\textsc{App}$ rule are created, as the rule has a high chance of failing if the wrong type is chosen. Higher values of $N_{\mathsf{app\_tries}}$ did not seem to be more effective in triggering bugs in our testing, but led to excessive backtracking, which slowed down the generator.

Parameter $N_{\mathsf{extra\_arg}}$ is set to 3. Values higher than 1 already did not give any advantage in testing. We decided to conservatively increase $N_{\mathsf{extra\_arg}}$ to 3, as the there is no performance hit associated with that. Choosing an arbitrary value for this parameter prevents

some terms from being present in the distribution of the generator. However, we believe that these terms should occur very rarely in any reasonable distribution and omitting them is acceptable. Note that generation is not sensitive to this parameter as using extra arguments is never required for it to succeed.

Weights The random backtracking algorithm tries the rule instances in a random order in a given recursive invocation. In order to increase the variety of the generated terms, the rules have *weights* assigned to them. Instances of rules with higher weights have higher chances of being tried first. Weights assigned to rules are respectively 2 for rules that use locally-bound variables, 1 for rules that introduce constants and 4 for the rules for introducing an application or a λ-expression. We tried different combinations of weights and found that increasing the weights on the App and Lam rules increases the chances of triggering bugs during our testing. On the other hand, increasing the weight of the App rule beyond 4 (if other weights are low) causes excessive backtracking. We also chose to prefer local variables to constants as using them should lead to expressions with more 'interesting' semantics.

Instance selection Selecting viable instances of rules involving variables Lam and Indir_{V}, is relatively easy and involves checking for equality between monomorphic types. Choosing instances of Indir_{C} is done by performing unification of the target type with possibly modified types of constants.

Generating terms containing `seq` Some of our applications require generating terms that contain occurrences of the following constant.

$$\texttt{seq} : \forall \alpha\beta.\alpha \to \beta \to \beta$$

The purpose of the Haskell function `seq` is to change strictness of an expression without changing its semantics in any other way. The meaning of expression `seq a b` is the same as that of `b`, with the difference that the former is strict in `a` (however, `b` might be also strict in `a` by itself). The function `seq` is often used to eliminate space leaks in Haskell programs by making forcing some expressions to be evaluated. More discussion about function `seq` can be found in Section 2 of Chapter 4.

Using this constant expands the generation space greatly, as matching β against the target type leaves the choice of α completely unconstrained. Given that `seq` is most useful in our applications when its first argument contains locally-bound variables, instead of treating it as ordinary constant, the generator allows it to have only variables as first arguments.

4 Distribution

The generator described above works by performing *random local choice*, influenced by rule weights, and is restricted by the size parameter. The distribution of the generated terms is similar to that for a size-limited generator based on stochastic grammars.

Due to the fact that the size parameter is always distributed evenly between subterms (Equation 2.1), the distribution of the generated terms is biased towards full trees, compared to the uniform distribution of terms with 'n' nodes.

The current design of the generator results in that some terms whose generation involves guessing of types, as described in Section 2.1, might be underrepresented in the distribution if successful generation of subgoals of a rule depends on specific types being chosen in that instantiating that rule.

Chapter 3

Shrinking

Counterexamples found by random generation of lambda terms often look strange and convoluted. However, in order to come up with a useful bug report the counterexample must be convincing to the developers working on the project.

The following term was found to violate a property, described in Section 4, that optimisation should not increase strictness.

$$(\lambda a.\texttt{seq}\ a\ ((\lambda b.\texttt{seq}\ a\ (\lambda c.\texttt{seq}\ b\ \texttt{tail}))\ (a\ (\texttt{head}\ \texttt{undefined}))\ (\texttt{case1}\ (\lambda b.\texttt{length})\ (\texttt{seq}\ a\ \texttt{2})\ (\texttt{seq}\ a\ \texttt{undefined}))))\ (\lambda a.\texttt{seq}\ a\ ((\lambda b.\texttt{seq}\ a\ (\texttt{seq}\ a\ (\texttt{seq}\ a\ (\lambda c.\texttt{seq}\ c\ (\texttt{seq}\ b\ \texttt{undefined}))))))\ (\texttt{seq}\ a\ (\lambda b.\texttt{seq}\ b\ (\lambda c.a)))))$$

The term is a tangible proof that there is *something* wrong with the tested compiler, however to say *what* goes wrong exactly is more difficult. In particular, tracing the execution of the compiler on it is likely to be very labour-intensive because of the size of the term, which would make identifying the root cause of the bug difficult.

Fortunately, it is quite likely that only part of the test case is relevant for triggering the problem and there probably exists a smaller term that triggers the same bug. We use a technique called *shrinking* [4] to search for a simpler counterexample that also triggers a bug. Shrinking was able to reduce the above counterexample to a simpler term that violates the same property:

$$(\lambda a.\texttt{seq}\ a\ (\texttt{seq}\ (a\ \texttt{undefined})\ \texttt{tail}))\ (\lambda a.\texttt{seq}\ \texttt{undefined}\ \texttt{(+1)})$$

The shrunk term is much smaller than the original one, which makes its analysis much easier. It is also *minimal* in the sense that it cannot be further reduced, which suggests that the term contains only parts that are required for triggering the bug. Furthermore, shrinking reduces the variation of terms reported as counterexamples, as different terms originally found during testing often shrink to the same or very similar terms. This makes differentiation between different bugs easier as one bug is typically represented by a set of similar shrunk terms.

Moreover, looking at different shrunk terms that violate one property can give even stronger hints about the problem that is triggered. For example, most terms found by this property contained subterms that look like $\lambda a.$`seq undefined ...`, which define a function that is a defined value itself, but returns $\bot$ given any argument.

Given their size and understandability, shrunk terms are good candidates for including in bug reports as they are more likely to convince the developers that the reported bug is worth fixing.

1 Shrinking simply-typed lambda terms

Shrinking is done by searching for a smaller term similar to the original one that also causes the given property to fail. Smaller *shrinking candidates* are created by reducing the term structurally. When one of the shrinking candidates triggers a failure shrinking will try to reduce it further. The example term shown earlier has been shrunk in 17 steps.

To illustrate a shrinking step, suppose that the term below provokes some failure.

$$(\lambda x.\texttt{id}\ (\texttt{tail}\ x))\ (\texttt{b}\ \texttt{++}\ \texttt{b})$$

Constants that appear in it represent common Haskell functions and have the following types:

$$\begin{array}{lcl} \texttt{id} & : & \forall\alpha.\alpha \to \alpha \\ \texttt{tail} & : & \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha \\ \texttt{(++)} & : & \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha \to \texttt{List}\ \alpha \end{array}$$

Constant `b` is a list of integers and has type `List Int`.

In search of a smaller term that also provokes a failure, we may turn to looking at subterms of the original term. For example, sub-

term `b ++ b`, which has the same type as the original term, could be considered. Term $\lambda x.$`id (tail` x`)` cannot be used as it has a different type. Terms `id (tail` x`)`, and `tail` x, although having the right type, contain an unbound variable[1] so we cannot use them as either.

However, it also makes sense to perform simplifications inside of the term. For example, `b ++ b` may be replaced with `b`, since they have the same type, which gives:

$$(\lambda x.\texttt{id (tail } x\texttt{))\ b}$$

Similarly, we may make a replacement inside of the lambda expression:

$$(\lambda x.\texttt{tail } x\texttt{)\ (b ++ b)}$$

Also, the whole lambda expression, which has type `Int → Int`, may be replaced with its subterm:

$$\texttt{tail (b ++ b)}$$

Another way in which we could simplify a term is by replacing its part with a constant of the right type. For example, if the environment contained constant `[]` (empty list), we could replace one of the list subterms with it:

$$(\lambda x.\texttt{id []})\ \texttt{(b ++ b)}$$

Finally, we can simplify a simply-typed lambda term by performing β-reduction on it, that is inline the function's argument into its body.

The formal properties of the simply-typed lambda calculus ensure that we can do this reliably as two important guarantees are made—that (a) the resulting term is well-typed and that (b) β-reductions always terminate. The second property is non-trivial as the size of a term might increase after a β-reduction.

The considered term contains one function applied to an argument, so it may be β-reduced in one way:

$$\texttt{id (tail (b ++ b))}$$

[1] Terms with unbound variables are not well-typed and thus technically have no type.

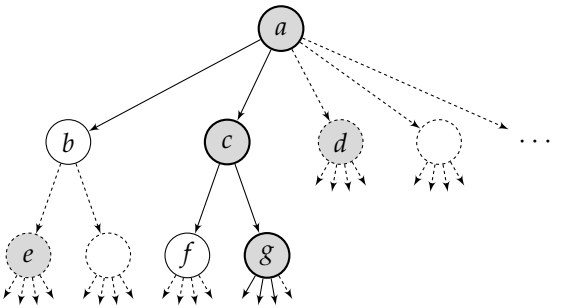

Figure 3.1: Shrinking process. Test cases that fail are marked in grey. Dotted test cases are not considered.

2 Shrinking using QuickCheck

The example above illustrates all ways of shrinking a term. This section presents a semi-formal description of the generic shrinking process performed by QuickCheck.

In each step of the shrinking process a number of shrinking candidates are tested, which are reduced versions of the original test case. When one of them fails, the shrinking step is concluded and the newly found counterexample becomes the starting point for the next shrinking step.

Figure 3.1 contains an illustration the shrinking process. Test case a is the original counterexample found during testing, which is subsequently shrunk. The first shrinking step considers a's shrinking candidates b, c, d, and so on, which are tested in turn. Test case b, which is considered first, succeeds and is discarded. Test case c fails and becomes the 'current' shrunk test case, whose shrinking candidates are tested in the next step. In this step g is found to falsify the property and the process continues with its shrinking candidates. Shrinking terminates when all current shrinking candidates succeed, or when a test case does not have any, and the last failing test case is reported.

Note that the failing test case e from Figure 3.1 is never considered, because shrinking looks only at immediate shrinking candidates of a failing test case. Also, test case d is not considered, because it occurs after the failing c in a sequence of shrinking candidates. Shrinking is a *greedy algorithm* that performs local choice, which is not guaranteed to be optimal. The resulting shrunk test case is only a *local minimum*.

However, good choice of a specific shrinking method tends to give results that are not far away from optimal.

A shrinking method for a particular data type is defined as a function that maps each element of that data type to a list of shrinking candidates defined as the following Haskell function:

```
class Arbitrary a where
  ...
  shrink :: a -> [a]
```

The shrinking candidates should be smaller than the original element, but the measure by which they are smaller can be freely chosen by the developer of the shrinking method. The only formal requirements on the function are that (a) it is total, (b) that lists of candidates are finite and (c) there are no infinite chains of shrinking steps. These requirements ensure that the shrinking process will always terminate, either when all shrinking candidates succeed, or when the current term has an empty list of shrinking candidates.

3 Specific shrinking method

The particular shrinking method for the simply-typed lambda terms that works as described in the beginning of the chapter can be defined by performing 3 kinds of steps:

Rule 1. replace with subterms A subterm may be replaced with its proper subterm, if their types are equal. Care must be taken to avoid referring to variables bound by removed λ-bindings.

Rule 2. β-reduce A term may be β-reduced.

Rule 3. replace with constant Any subterm that is not a constant may be replaced with a constant if the types agree.

There is a possible optimisation that we could introduce in Rule 1—we may restrict it to only consider replacing subterms with their immediate proper subterms of the right type. For example, term `0 + (1 + 2)` may be replaced by `1` using the unrestricted Rule 1, but not using the restricted one, because `1` is a proper subterm of `1 + 2` that can also be used.

The idea behind this restriction is that `1` will be tried anyway in a later shrinking step and that omitting unnecessary shrinking candidates prevents the list from being excessively long. This is

important, for example, when shrinking the data type of binary trees where the total number of all subtrees might be rather large. Since shrinking has to respect types, we expect this optimisation to have a smaller effect in the case of simply-typed lambda terms, however we decided to take it into consideration. The effects of this restriction are investigated later in Section 8.

4 Properties of shrinking

There are two important properties that must be established for our shrinking method: that (a) shrunk terms are well-formed and that (b) there can be no infinite sequence of shrinking steps.

Shrinking produces well-typed terms It is easy to see that rules 1 and 3 turn well-typed terms into well-typed terms as their construction explicitly ensures that. For rule 2, which performs β-reduction, it is not trivial. However, a known result for the simply-typed lambda calculus called *subject reduction*[30] states exactly that β-reduction is type-safe. Even though we allow polymorphic constants, our terms can still be modelled in the simply-typed lambda calculus as the constants are always fully instantiated. Thus, all three kinds of steps preserve well-typedness.

Shrinking terminates Again, it is easy to see that rules 1 and 3 are constructed in such way that they always make terms smaller, so they can never result in non-termination. Rule 2 is more tricky, as it may increase the size of the term due to duplication when the argument term is inlined into the function body. However, another standard result, strong normalisation, states that β-reduction terminates for the simply-typed lambda calculus[30].

Unfortunately, even though no rule can cause non-termination by itself, it is not possible to extend this easily into proof that any combination of steps will always terminate. We expect that the proof can be extended, however the extra steps are non-trivial, and thus we leave it for future work.

Unfortunately, even though no rule can cause non-termination if used in isolation, it is not possible to extend this easily into proof that any combination of steps will always terminate. We expect that the proof can be extended, however the extra steps are non-trivial, and thus we leave it for future work.

5 Design choices of shrinking

The three ways of simplifying a term were chosen because they comprise generic ways of simplifying a term. Parts of a counterexample that are not relevant for triggering a failure might be removed by rules 1 and 3.

It is debatable whether rule 2 that inlines a function argument into its body is a simplification. In some cases the opposite, β-expansion, might lead to a simplified term. However, unrestricted β-expansion is non-terminating, whereas β-reduction terminates even when rules 1 and 3 are present, as we showed previously.

On top of that, the shrinking process turned out to be effective in practice as the shrunk counterexamples were not possible to be reduced further by hand in most cases.

6 Shrinking with batch test cases

In order to speed up testing, a single test case contains a list of terms that are tested together in one run, as described in Section 4. Usually 1000 terms are tested in one batch. Generating test cases that contain a list of terms is straightforward in QuickCheck, where it is enough to compose a generic generator of lists with a generator of terms. Similarly, a shrinking method can be derived from the generic one for lists. However, this solution has some shortcomings.

The generic shrinking method first tries to reduce the length of the list to one element using binary search, and then tries to shrink that element using its own shrinking function.

This method is inefficient in two ways. First, it does not use the information about which term in the list has failed and has to resort to binary search, adding about 10 unnecessary shrinking steps. And secondly, each shrinking candidate is compiled separately without taking advantage of compiling and running them in batches.

An optimised shrinking method would receive the index of the term that has failed the property, and generate a list of shrinking candidates of that term that could be compiled in a batch. Thus, it would avoid the initial search for the failing term and make shrinking that involves many failed attempts much faster.

It turned out that this can be implemented without changing the implementation of QuickCheck, although at a price of making the properties less composable. In order to create a property that uses batch test cases we propose using the following combinator.

```
forAllParShrink
  :: Int -- Number of test cases in one batch
     -> Int -- Number of shrinking candidates in one batch
     -> Gen a -- Test case generator
     -> (a -> [a]) -- Shrinking function
     -> (a -> String) -- Printing
     -> ([a] -> Maybe Int) -- Batch property
     -> Gen Prop
```

The batch property is passed as a function taking a list of test cases and returning a `Maybe Int` where `Nothing` indicates that all test cases succeeded, whereas `Just` n means that test case n was the lowest one that failed.

For each run of the property, the function creates a list of test cases whose length is specified by its first argument. When a failure occurs, the offending test case, which is pointed to by the index returned by the property, is passed to the shrinking function, which generates a list of shrinking candidates. A number of these test cases, limited by the function's second argument, is passed again as a batch to the property.

The fact that the batch property is defined as a function of type `[a] -> Maybe Int` gives it a disadvantage when it comes to composability. Contrast it to the ordinary `forAll` combinator, which takes properties of type `Testable t => a -> t`. The property defined inside of `forAll` may itself use any of the QuickCheck property combinators, such as `collect`, `printTestCase` or `within`, whereas the one defined using `forAllParShrink` cannot, and must be defined monolithically.

On the other hand, the property inside of a `forAll` cannot give any extra information about the failure to the combinator, such as which test case failed in the batch, which is precisely why we have to use the type `[a] -> Maybe Int` for batch properties.

Shrinking using batch test cases implemented like this was sped up by factor of around 10–20 in typical cases, using batches that were limited to having 40 terms.

7 Weaknesses and possible improvements

Reducing the variety of constants The shrinking process is not able to replace a constant in a term with another one because of risk of non-termination. This leads to many similar terms being reported if a particular one is not required to trigger a failure. For example, one

property generated the following terms:

```
(-) a ((-) b 0)
(-) a ((-) b 1)
(-) a ((-) b 2)
(-) a ((+) b 0)
...
```

The counter-examples were shrunk to 12 variations, all using a different combination of constants. Given that shrinking should reduce, if possible, the number of reported counterexamples for a given bug, it would be beneficial if all these terms were *normalised* to a common representative. One way of doing that would be to establish an ordering on all constants and allow a 'larger' constant to be replaced by a 'smaller' one.

A slightly related idea would be to expand constants into their definitions. The expansion, obviously, has to be restricted as recursive definitions could be expanded forever. Such a rule might solve, for instance, the problem of transforming `1` into `0`, as `1` expanded to `(+1) 0` can be reduced to `0`. However, even this simple expansion is problematic because the whole `(+1) 0` can be transformed back to `1` using Rule 3 introducing the risk of non-termination. Also, expanding all non-recursive definitions might make the terms much larger. Thus, it is not clear how to restrict expansion of definitions so that it is useful for shrinking.

Retyping subterms Some cases would benefit from being able to replace a subterm with its proper subterm if it can be *retyped* to match the type of the term that it replaces. For example, if subterm $\lambda x.$`[]` has type `Bool` $\rightarrow$ `List Int` it could also be retyped with `Int` $\rightarrow$ `List Bool` and used to replace a larger term of this type.

Small improvements Two potential improvements can be experimented with that would introduce only small changes to the method. Firstly, subterms could be replaced with variables that are in scope with the right type. And secondly, the shrinking candidates could be ordered in a more complicated way involving, for instance, interleaving of candidates generated by different rules.

8 Shrinking parameters

There are two possible choices in the shrinking method. The first one is the order in which the shrinking candidates generated by different rules should be considered, which in some cases might influence the result of shrinking.

The second choice is whether to optimise shrinking by restricting Rule 1 to only consider replacing subterms with their immediate proper subterms of the right type, as described in Section 3.

To assess the effects that these choices have on shrinking, we parametrised our shrinking method over these choices and conducted an experiment to measure its performance with different parameters. The experiment was performed by collecting pools of randomly generated unshrunk counterexamples for two properties, shrinking them with different combinations of parameters and examining their performance based on the results of shrinking and its speed. The two properties were the ones discussed in Sections 1 and 2 of the next chapter and each of the pools contained 200 terms. While it is certainly worth extending the experiment to cover more properties, or even other applications than just testing GHC, the data that we obtained should give us some idea about the influence of parameters.

The results of the experiment suggest that restricting Rule 1 yields no negative effects on the quality of shrunk terms while improving the speed of shrinking by a small margin. The outcomes are less conclusive about which order of shrinking rules should be used. All orderings deliver comparable quality of shrunk terms, but differ in the numbers of shrinking steps and failed shrink attempts performed during shrinking. Although we can spot orderings that are clearly better than others, it is not possible to nominate the best one, if we assume that batch shrinking (described in Section 6) is used.

In this chapter we present the data that we gathered together with our analysis. We also propose recommendations based on that, but some of our conclusions are not definitive and for that reason we present our analysis in detail to allow drawing alternative conclusions from it.

8.1 Immediate subterms

First, the parameter defining whether to consider only largest applicable subterms was evaluated. Due to the fact that shrinking is a greedy process, it is impossible to predict the outcome of restricting the lists

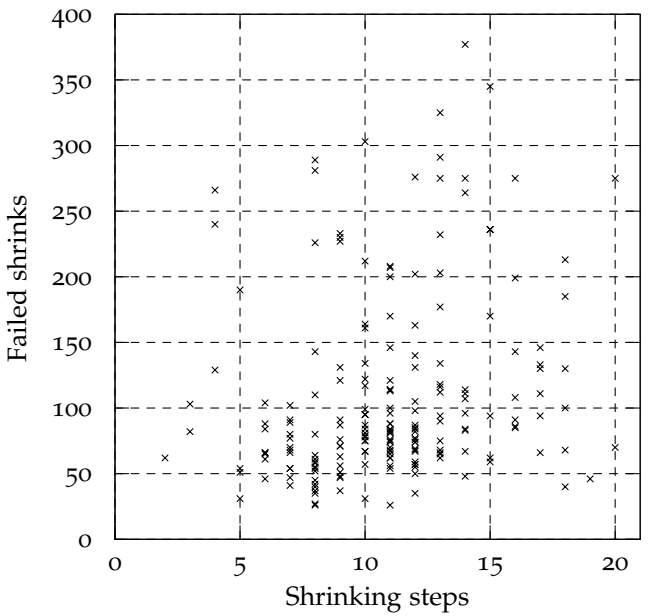

Figure 3.2: Number of shrinking steps and failed shrink attempts for different terms from a pool

of shrinking candidates, but we expected that doing it might reduce the numbers of failed shrinking attempts. At the same time, it might reduce the effectiveness of shrinking.

The experiment showed that using this restriction has no effect on the results of shrinking whatsoever, as all the terms were shrunk in the same way regardless of it. The number of successful shrinking steps was also the same in all cases.

However, the number of unsuccessful shrinking attempts for some terms was higher when all proper subterms were considered during shrinking. The differences were not large on average, with up to 2% more failed attempts in the first property when a specific ordering of shrinking rules was used (the other parameter of shrinking) and up to 7% more in the second one. The biggest discrepancy for any given term was 22% more failed attempts in the first property and 64% in the second one, however these were isolated cases.

There are typically 5–15 more failed shrinking attempts than successful steps for a given term. For example, Figure 3.2 shows a scatter-plot for respective numbers for the first of the considered properties where shrinking was performed with all subterms considered and the with the default ordering of shrinking rules.

It seems to be very beneficial to limit the number of failed shrink attempts since their number is often much larger than the number of successful steps. However, the benefit seems less profound when we consider that using batch shrinking, described in Section 6, makes failed shrink attempts much cheaper, as in our case up to 40 shrinking candidates are tested in one batch.

This optimisation has little effect with testing the properties that we used in our experiment, but we can imagine a situation where terms contain many subterms of the same type (for example involving a binary tree data type), in which case the reduction in failed shrink attempts might be substantial.

8.2 Ordering of shrinking rules

The second parameter of the shrinking method is the order in which the three shrinking rules, mentioned in Section 3, are tried on a term.

We investigated the influence of the ordering of rules on the quality of shrunk terms. For any given two orderings of rules, the shrunk terms were the same in at least 61% of cases for Property 1 and in at least 76% of cases for Property 2. This makes us think that when we change the ordering of rules the results of shrinking remain quite consistent.

To compare different shrunk terms that come from the same original counterexample we decided to use the size of the final shrunk terms as the measure of their quality, and to naturally prefer smaller terms. Figure 3.3 shows a scatter-plot of sizes of shrunk terms that were generated by Property 1 and shrunk using two different rule orderings. The area of each circle in the plot is proportional to the number of data points that occurred in the same place.

We can infer by looking at the plot that no ordering of rules is better than the other when it comes to sizes of shrunk term. Plots for other pairs of orderings and for Property 2 look very similar to the one in Figure 3.3, which makes us conclude that the ordering of rules does not influence the quality of shrunk terms in a measurable way.

The other factor that we considered was the speed of shrinking. Instead of using wall clock time for benchmarking we used two

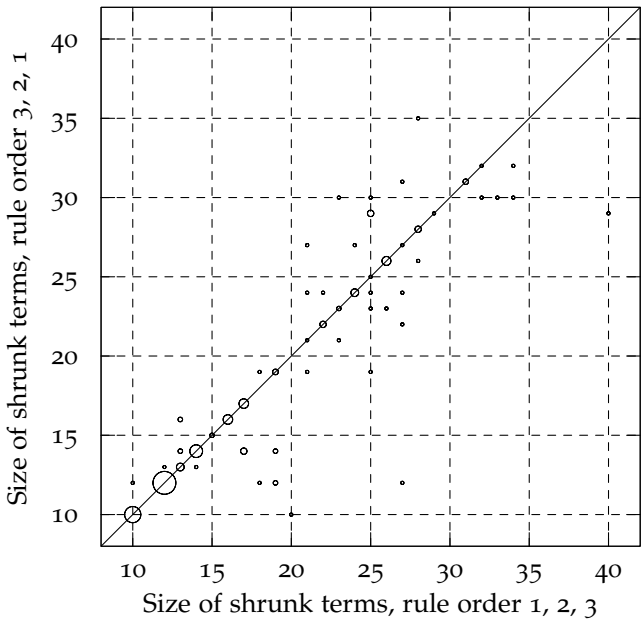

Figure 3.3: Sizes of shrunk terms for different terms from a pool; the diagrams are similar for all other pairs of rule orderings.

numbers: (a) the number of shrinking steps performed during a single shrinking and (b) the number of failed shrink attempts. Table 3.1 shows the mean values of these numbers for different orderings of rules. In all cases we consider shrinking that uses the optimisation presented in Section 8.1.

Looking at Table 3.1 lets us isolate two groups of orderings. The first three generally result in more successful shrinking steps, but fewer failed attempts, whereas for the latter three it is the other way around.

For a possible explanation of this phenomenon notice that in the first group Rule 1 (replacing a subterm with its proper subterm) always precedes Rule 3 (replacing a subterm with a constant), while in the second group the opposite is the case. We suspect that Rule 3 generates very many shrinking candidates and many of them fail

	Property 1		Property 2	
rule order.	shrinks	failed shr.	shrinks	failed shr.
1, 2, 3	10.95	108.43	7.89	61.61
1, 3, 2	10.92	137.92	8.41	119.13
2, 1, 3	12.64	107.93	11.35	52.02
2, 3, 1	7.33	194.85	5.88	155.09
3, 1, 2	6.92	267.45	5.90	310.94
3, 2, 1	6.90	257.39	5.94	290.38

Table 3.1: Shrinking steps and failed shrink attempts

before a successful one is tested. Rule 1, on the other hand, if placed before Rule 3, might successfully reduce a term before shrinking candidates of Rule 3 are considered. Nevertheless, the lower numbers of shrinking steps in the second group might indicate that shrinking candidates generated by Rule 3 are more effective in quickly reducing the term's size.

Looking more closely at the rule orderings in the first group, we can see that ordering 1, 2, 3 seems to be better than 1, 3, 2, because the mean number of shrinks is either very similar (Property 1), or smaller in 1, 2, 3 (Property 2), and the mean number of failed attempts is always lower in 1, 2, 3. Orderings 1, 2, 3 and 2, 1, 3 are closely tied as the mean number of shrinks is lower in 1, 2, 3, but the mean number of failed attempts is better in 2, 1, 3 in Property 2 by a considerable margin.

In the second group 2, 3, 1 seems to be the winner thanks to lower numbers of failed attempts, even though its numbers of shrinking steps are a little higher.

Having a choice between an ordering from the first group and one from the second one, which one should we choose? The numbers of failed shrinking attempts are usually much higher than the numbers of successful shrinks, which hints at the first group. However, given that when testing the GHC compiler we use batch shrinking, failed attempts are much cheaper than shrinking steps as up to 40 shrinking candidates are tested at the same time. With this assumption, all the orderings get much closer to each other in terms of performance. In contrast, when batch shrinking is not used, then the orderings from the first group are clearly better.

Figures 3.4 and 3.5 show scatter-plots of numbers of successful

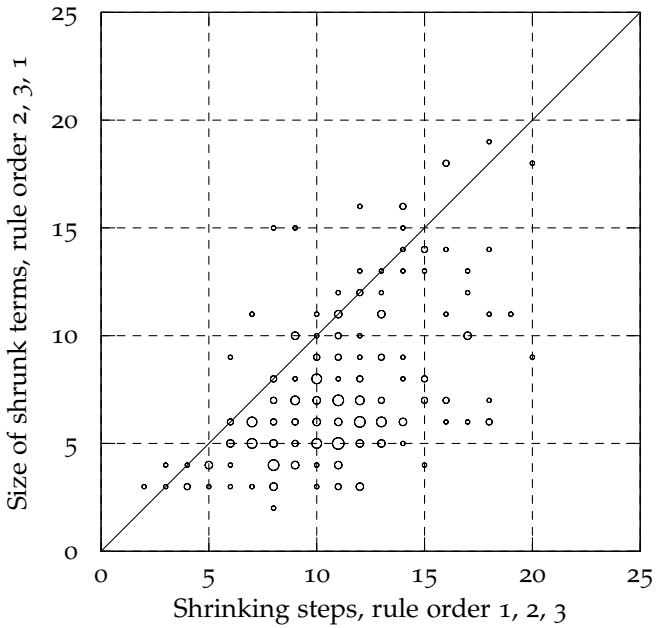

Figure 3.4: Numbers of shrinks for different terms from a pool

shrinks and failed shrink attempts for terms generated by Property 1, when using orderings 1, 2, 3 (first group) and 2, 3, 1 (second group). The scatter-plot of the numbers of failed shrink attempts is plotted using the logarithmic scale.

As we can see in Figure 3.4, ordering 2, 3, 1 generally results in fewer shrinking steps, the difference being more than 10 in some cases. On the other hand, ordering 1, 2, 3 brings about fewer failed shrink attempts, as shown in Figure 3.5. The discrepancies for some terms are as large as over 1000 failed shrink attempts for ordering 2, 3, 1 and less than 200 for the other ordering. Given that ordering 2, 3, 1 generates over 800 more failed shrink attempts for these terms, this corresponds to about 20 more batches used in shrinking.

Comparisons of other orderings from the two groups yield similar scatter-plots to the ones presented here, which prompts us to conclude that the first group results in fewer pathological cases when shrinking

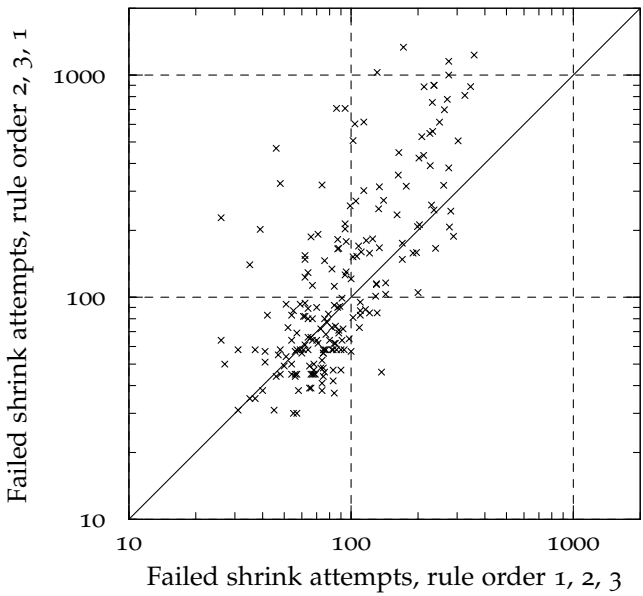

Figure 3.5: Numbers of failed shrink attempts for different terms from a pool

takes a long time.

8.3 Conclusions about shrinking parameters

We conclude that restricting the shrinking candidates to terms where only largest applicable subterm can replace its superterm does not yield a sizeable speed-up. However, we also found that it has no detrimental effect on the quality of shrunk counterexamples, which is why we recommend that it is enabled by default.

The conclusions about which ordering of rules gives the best performance depends on whether we assume that batch shrinking is used. There is no measurable difference in the quality of the shrunk counterexamples when different orderings are used. What differed was the mean numbers of shrinking steps and failed shrink attempts.

The first three orderings shown in Table 3.1 performed fewer failed

shrink attempts during shrinking, which made them appropriate for situations when batch shrinking is not used. Out of this group, orderings 1, 2, 3 and 2, 1, 3 performed particularly well.

When batch shrinking is used, with a maximum of 40 terms in one batch, as in our case, the differences in the average performance between the orderings is much smaller. In this case orderings 1, 2, 3 and 2, 1, 3 from the first group, and 2, 3, 1 from the second one could be considered. Based on the scatter-plot in Figure 3.5 we would still recommend the orderings from the first group to limit the amount of pathological cases. Choosing one of 1, 2, 3 and 2, 1, 3 over the other may impose a hit of 20% in the worst case, which is why we could arbitrarily choose the first one as a safe default.

Clearly, some of these recommendations are conditional and weak, but should serve well as the default values, whereas it should be possible to use alternative parameters whenever the user chooses to do so.

Chapter 4

Applications

We used the generator to test the GHC compiler using *differential testing* [23] directed towards testing the compiler's middle-end. Even though the generator was was capable of generating only a very limited subset of all Haskell programs, it was able to uncover interesting bugs in the compiler.

The testing resulted in finding eight interesting failures and four bugs were reported for GHC and subsequently fixed. The counterexamples obtained were concise enough to be understandable thanks to automatic shrinking.

Apart from bugs detected in the GHC compiler, the testing allowed us to understand better the effects of optimisation performed by it.

1 Strictness changed by optimisation

We investigate whether optimisation performed by GHC influences strictness of the compiled programs. Optimisation is performed on a program in order to turn it into a more 'optimal' one without changing its observable behaviour, where more 'optimal' might mean, for example, one that runs quicker. However, erroneous transformations might change the semantics of a program. In particular, changes in strictness, while subtle, can influence the program's observable behaviour—for example, the program may crash or consume more memory than it should.

To detect whether the GHC's optimiser modifies strictness of some expressions we used differential testing where observed output of an optimised program is compared with observed output of the same

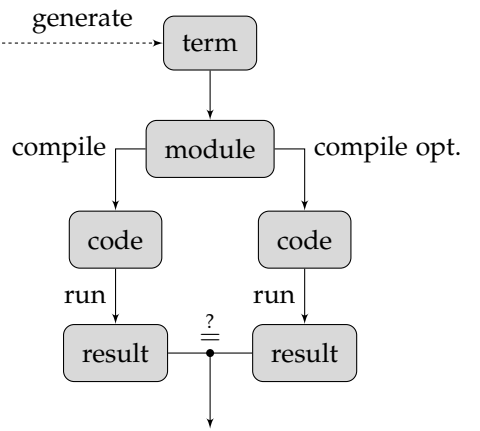

Figure 4.1: Differential testing

program compiled with no optimisation, as shown in Figure 4.1. Changes in strictness of expressions might result in less or more output to be printed by tested functions before crashing.

A generated term whose strictness is to be analysed is compiled as part of a module whose skeleton is shown in Figure 4.2. The generated term is bound to the `code` variable and is a function of type `[Int] -> [Int]`.

The `main` function of the module is devised to print the results of the `code` function applied to a small number of partially-defined lists of integers defined by the constant `inputs`, which we elide here. Expression `print $ code x` prints the result of the function, and thus forces its evaluation.

When an error is encountered during the evaluation, it is caught as an exception by `E.catch`. The exception handler prints a message indicating an exception, but disregards its exact kind as we do not consider changing the exception kind as a change in program's semantics following the Haskell Report [21]. The program is then allowed to continue to evaluate the function for all remaining inputs.

To provide accurate information on partial values potentially returned by the observed function, it is necessary to turn off output buffering, which is done by calling `hSetBuffering` in the `main` function. Had buffering been active, the printing code would try to print a whole line of characters at once and if an exception were triggered before the buffer is flushed, the output that had already been generated

```
module Main where
import Control.Monad (forM_)
import qualified Control.Exception as E
import System.IO (hSetBuffering, stdout,
  BufferMode (NoBuffering))

handler (E.ErrorCall s) = putStrLn $ "*** Exception: "

inputs :: [[Int]]
inputs = ...

code :: [Int] -> [Int]
code = ...

main = do
  hSetBuffering stdout NoBuffering
  forM_ inputs $ \x -> do
    E.catch (print $ code x) handler
```

Figure 4.2: Module skeleton

would be discarded. Therefore, we would not be able to distinguish between different partially-defined results. On the other hand, when buffering is turned off, characters are printed one by one and the exception handler will trigger only after the last defined character is printed. Even though a more accurate technique is available for discovering the semantics of partial values [7], we chose this method as fast and accurate enough.

Each module created from the skeleton is compiled in two variants. The unoptimised variant is compiled without any optimisation options, which results in the default `-O0` 'zero' optimisation level. The optimised variant is compiled with `-O -fno-full-laziness`, which turns on typical optimisation options, but turns off the *full laziness* optimisation, which is also known as *let-floating*. The reason for leaving out full laziness is that it is known to change the strictness of compiled code and that examples involving it are less interesting.

The initial environment for terms contains simple integer constants and functions, like `0` and `+`, as well as common functions operating on lists from the Haskell prelude. The initial environment that is used in this section is presented in Figure 4.3.

Given that we want to test the strictness of compiled functions, we

```
seq    :: a -> b -> b
id     :: a -> a
[]     :: [a]
0      :: Int
1      :: Int
2      :: Int
(+)    :: Int -> Int -> Int
(+1)   :: Int -> Int
(-)    :: Int -> Int -> Int
(:)    :: a -> [a] -> [a]
enumFromTo :: Int -> Int -> [Int]
enumFromTo' :: Int -> Int -> [Int]
head   :: [a] -> a
tail   :: [a] -> [a]
take   :: Int -> [a] -> [a]
(!!)   :: [a] -> Int -> a
length :: [a] -> Int
filter :: (a -> Bool) -> [a] -> [a]
map    :: (a -> b) -> [a] -> [b]
null   :: [a] -> Bool
(++)   :: [a] -> [a] -> [a]
odd    :: Int -> Bool
even   :: Int -> Bool
(&&)   :: Bool -> Bool -> Bool
(||)   :: Bool -> Bool -> Bool
not    :: Bool -> Bool
True   :: Bool
False  :: Bool
foldr  :: (a -> b -> b) -> b -> [a] -> b
(==)   :: Int -> Int -> Bool
(==)   :: Bool -> Bool -> Bool
(==)   :: [Int] -> [Int] -> Bool
case1  :: (a -> [a] -> b) -> b -> [a] -> b
undefined :: a
```

Figure 4.3: Initial environment. Many instances of `(==)` are needed because our generator does not support Haskell type classes. The constant `enumFromTo'` is our own definition where the second argument is the length of the enumeration. The constant `case1` is another definition that performs case analysis on lists.

must make sure that their arguments are not inlined, otherwise each function would be compiled and optimised many times together with each of its arguments. Fortunately, iterating through inputs using the `forM_` function is enough to stop GHC from inlining the arguments.

Due to high fixed cost associated with compilation and linking a module, 1000 terms are included in a single 'batch' module during testing, which requires a slightly different module skeleton and a suitable output comparison procedure.

Testing the property for a given generated term is done by (1) creating a module using the skeleton from Figure 4.2, (2) compiling it twice with different optimisation settings, (3) running the compiled modules and (4) comparing their output for equality. If the outputs are different then we have found a counterexample that has a different observable behaviour depending on the optimisation options. We can define the property in a more formal way as follows, where t is the term, module is a function that creates a module out of a term, and c_* and run are functions that, respectively, compile and run the module yielding its output:

$$\forall t.\mathsf{run}(\mathsf{c}_{\mathsf{opt}}(\mathsf{module}(t))) = \mathsf{run}(\mathsf{c}_{\mathsf{noopt}}(\mathsf{module}(t)))$$

1.1 Results

The property described above led to observing a discrepancy between optimised and unoptimised code for about one in 10000 terms, which takes about 3 minutes of CPU time when terms are tested in batches.

One failure for each 10000 tests is an unusually low number for a QuickCheck property, and we might speculate why this number is so low. One likely reason is the very high number of all expressions even among terms of small size. It is possible that failing test cases comprise a small number of all expressions. Another possible reason is an imperfect distribution of our generator, which may increase the probability of generating the same terms many times.

All counterexamples that we present here have been shrunk by the shrinking process, which means that any further shrinking results in a test case that does not trigger a failure.

Failure 1 The following snippet is an example expression that violated the property.

```
foldr (\a -> seq) id ((:) 0 (undefined::[Int]))
```

For clarity, it can be rewritten as follows by η-expanding one of the subexpressions, while keeping the same behaviour:

```
foldr (\a b c -> seq b c) id (0 : undefined :: [Int])
```

The expression exhibits different observable behaviour depending on compilation options, which indicates that at least one of the versions is compiled wrongly. However, to find out which one we must determine by hand what is its correct semantics.

The intended semantics of the higher-order function `foldr` is to return its first argument applied to the first element of the list (variable `a` gets bound to `0`) and to the result of `foldr`'s recursive call (`b` gets bound to the result of the recursive call). Function `foldr` is defined as a two-argument function, but is applied to three arguments in its context.

The whole expression should reduce to `\c -> seq (foldr ...) c`, which acts as the identity function except when expression `foldr ...` crashes, in which case the whole expression should also crash when applied to an argument. The expression should in fact crash as the recursive `foldr` is applied to the `undefined` list.

We were able to construct a simple program that demonstrates the incorrect compilation, shown below, which is simpler than the original module and more suitable for submitting a bug report.

```
main = print $
  wrap
   (foldr (\a b c -> seq b c) id (0 : undefined::[Int]))
   [0]
```

The tested expression is passed as argument to `wrap`, which acts as the identity function and an example list `[0]` is applied to the resulting expression. The purpose of `wrap` is to prevent the expression from being simplified together with its argument, which is achieved by implementing `wrap` in such way that GHC cannot reduce it[1].

```
wrap :: a -> a
wrap x = [x]!!0
```

When the program is compiled using GHC without optimisation, it prints

```
program: Prelude.undefined
```

[1] Using the builtin function of GHC named `GHC.Exts.lazy` has the same effect.

which indicates that the expected exception has been raised. However, with optimisation turned on, the expression gets incorrectly compiled into the identity function and `[0]` is printed instead. A likely explanation for this is that `seq` is somehow omitted from the generated code.

The demonstrated change in observable behaviour caused by optimisation violates the semantics defined by the Haskell Report. To assess the seriousness of this bug we will analyse its possible consequences.

The bug might cause a crashing function to successfully return a result. This does not seem dangerous, but can have two implications. First, it is a surprise factor for the programmer, which may hinder the understanding of the code. And furthermore it may invalidate a partial correctness arguments about a program, which state that *Program has property P if it terminates*. If the program's overall correctness relies on such partial correctness argument, the programmer might be mistakenly convinced about its correctness.

A more common manifestation of the bug might be, however, a space leak caused by an omitted `seq` application. Common wisdom about Haskell programs compiled using GHC is that optimisation occasionally reduces their performance, which often happens by introducing space leaks. This and similar bugs might have a role in causing the performance regressions.

Failure 2 Another counterexample yielded by the same property is this expression.

```
seq (seq (head []) (\a -> undefined))
```

Identifying incorrect compilation again requires analysing the expression's semantics. The nested expression containing `seq` should evaluate to an exception, since its first argument is `head` of an empty list. Therefore the whole expression should also be a function that crashes. To demonstrate the error we can use the same skeleton program as with the previous term.

```
main = print $
  wrap (seq (seq (head []) (\a -> undefined))) [0]
```

Compiling the program with no optimisation yields the correct result.

```
program: Prelude.head: empty list
```

However, when compiled with optimisation, the expression behaves as the identity function and yields `[0]`. Thus, the semantics in the optimised version is changed to more lazy, which is a similar to the problem triggered by the expression in Failure 1.

The fact that the counterexample that we are considering has been shrunk allows us to draw some conclusions about the bug that has been triggered. From the way the shrinking is performed, we know that neither of the shrinking candidates generated from the reported term failed the property. This means that if we simplify the counterexample in a structural way it will no longer be a counterexample.

For example, if we replace the expression `head []` with `undefined` the term will be compiled correctly, or at least testing will not detect any error. Note that these two terms ought to have the same semantics, nevertheless using `head []` is required to trigger a bug. We might speculate that `undefined` is correctly identified by the compiler as a crashing expression, while `head []` is not, which makes the compiler perform different transformations each time.

Another term that would look like a good candidate for replacement is `\a -> undefined` and again replacing it with `undefined` makes the failure go away. If treated as functions, both terms behave as undefined functions, but it is possible to differentiate between them by executing `seq` on them. One hypothesis might be that the compiler at some point assumes that `\a -> undefined` is equal to `undefined` 'for all practical purposes', but the distinction between them turns out to be relevant in this context.

This speculation might be further reinforced by looking at some other counterexamples for the same property, presented below.

```
seq (seq ((!!) ([]::[] Bool) 0) (\a -> (undefined::Int)))
seq (seq ((!!) ([]::[] Bool) 0) (\a -> (undefined::Int)))
seq (seq ((+1) (undefined::Int)) (\a -> (undefined::Int)))
seq (seq (even (undefined::Int)) (\a -> (undefined::Int)))
seq (seq ((+) (undefined::Int) 0) (\a -> (undefined::Bool)))
```

All of them are structured similarly to the original one, having a crashing expression (but not `error ...` or `undefined`) as the argument of the nested `seq`, and `\a -> undefined` as the second argument of the other `seq`. This may suggest which combination of features is needed to trigger a failure and which details are not relevant in an expression.

Apart from the expressions shown above, several other groups of expressions could be distinguished among the ones reported by the property. It is impossible to say, based on testing, if each of the

groups is caused by a distinct bug, or whether there is one bug that causes all the failures. By looking at bug fixes that were added to the GHC we deduced that, for example, this failure was probably caused by a different bug than Failure 1.

Failure 3 Counterexamples presented in previous examples were terms whose behaviour was always more lazy in the optimised version of the program, and indeed all found terms that were scrutinised by us during testing had this characteristic. We decided to check if it is possible to find a term that is more strict when it is compiled with optimisation.

For this we modified the function that compares outputs of two variants of a program compiled with different optimisation levels to signal failure only when the optimised version prints *less* output, indicating that it is more strict. As expected, terms like this were much harder to come across, but still possible to find at a rate about 100 times lower than the previous kind.

The following term was found by the modified property.

```
(\a -> seq a (seq (a []) id)) (\a -> seq undefined (+1))
```

For clarity we might rewrite it as follows:

```
let a = \x -> seq undefined (+1) in a `seq` a [] `seq` id
```

If we consider its semantics: variable `a` is bound to a function whose semantics is equivalent to that of `\x -> undefined`, which means that `seq a` x is defined, but `seq (a` y`)` x is undefined for any defined x and y.

Given that the considered expression performs `seq` both on `a` and on `a []` its result should be undefined and the correct outcome of a program that evaluates this expression should be a raised exception. Strangely enough, compiling the expression with no optimisation gives a program that prints `[0]` (when the expression is applied to `[0]`) instead of crashing and with optimisation turned on the result is correct.

It seems, thus, that our working hypothesis that unoptimised programs are correct and should be treated as reference for testing is not always true as we have just found a program that gets fixed by compiling with optimisation.

This very unexpected bug was reported as ticket 5625[2] in the GHC bug tracker and was subsequently fixed.

[2] Available at http://hackage.haskell.org/trac/ghc/ticket/5625. All GHC tickets can be accessed in this manner by altering the ticket number.

As previously, we may look at other reported counterexamples to determine what features of the expression are relevant to causing failures. The counterexamples are shown below, each spanning two lines.

```
(\a -> seq (a (seq a (undefined::[] Int))) id)
    (\a -> seq (undefined::[] Int) (+1))
(\a -> seq a (seq (a (undefined::[] Int)) tail))
    (\a -> seq (undefined::[] Int) (+1))
(\a -> seq a (seq (a ([]::[] (Int -> Int))) (:)))
    (\a -> seq (undefined::[] Bool) (+1)) 0
(\a -> seq (a (seq a (undefined::[] Int))) id)
    (\a -> seq (undefined::Bool) (\b -> head))
(\a -> seq (a ([]::[] Int) (undefined::Int)) (seq a))
    (\a -> \b -> seq (undefined::Int) (+1)
```

The common features visible in these expressions are (a) subexpression `\x -> seq undefined` *e* (slightly modified in the last example) that is bound to the variable `a` of the first function, and (b) that variable `a` is used twice in the function. Given the expected semantics of the subexpression, it may also be important that `a` is applied to an argument at least once in each counterexample.

It is worth noting that it is actually possible to reduce this counterexample further by hand. It appears that the bug is triggered only if `a` is not inlined and the reported term contains two occurrences of `a`, which prevent it from being inlined. But if we move out `a` to become a top-level definition and export it from the module then only one occurrence of `a` is needed. The module demonstrating the bug becomes then:

```
module Main (a, main) where

a = \x -> seq undefined (+1)

main = do
  print $ (a [] `seq` id) [0]
```

1.2 Were the bugs fixed?

In the three examples presented above, we demonstrated that GHC may subtly change the semantics of expressions by changing their strictness. The compiled expressions were too lazy, which in two

cases was caused by optimisation, while in one case optimisation unexpectedly fixed the error.

The counterexamples reduced by the shrinking process were concise enough for us to initially analyse and understand the failures. Even though we have no expertise about the internals of the GHC compiler, we were able to make educated guesses about the failures based on the counterexamples. Furthermore, the found test cases were of high enough quality that they could be used in bug reports. Out of the three presented expressions, one was used to submit a bug report for GHC.

The bugs causing all three failures have been fixed. The term in Failure 1 is no longer miscompiled by GHC 7.3.20111127 when option `-fpedantic-bottoms` is used. The option was introduced as a fix for another bug that we reported (ticket 5587). Therefore, we conclude that the fix affects also this failure.

Failure 2 has been fixed before GHC version 7.3.20111022. However, we cannot identify a specific bug that is relevant. The bug concerning Failure 3 has been fixed before GHC version 7.3.20111127 through ticket 5625.

Failure 4 Out of the hundreds of reported counterexamples, those that were scrutinised by us always contained `seq`s. This is not surprising as `seq` is a construction that is a difficult case for the compiler. However, `grep`ping through the counterexamples revealed that there are several that do not involve `seq`, all rather similar.

Here is the simplest of such terms that has been found:

```
\a -> foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0
```

The correct semantics of this function applied to a defined list is to concatenate the list with itself and then execute a fold on it. The fold ignores the elements of the list, but threads through the initial value of the fold. The initial value, which is `undefined ()`, is returned as the result of the fold and is applied to `0`, so the whole expression should crash with an exception.

Compiling the code with no optimisation yields correct results, but the following program was found to demonstrate incorrect compilation:

```
f :: [Int] -> [Int]
f a = foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0

main = print $ f []
```

When compiled with no optimisation it correctly prints

```
program: Prelude.undefined
```

However, when optimisation is turned on, the program does not crash, and does not print any output. This is surprising, as we would expect that the program would either crash or print a list of integers, which is the type of values returned by `f`.

To investigate how this happens, we looked at the intermediate Core representation of the program, which revealed that the code that is supposed to print the resulting list is missing. The only sensible reason why a compiler would omit this code is that it expects the function to crash before returning the value. This seems a likely guess, as GHC strictness analysis marks `f` as a function returning bottom. To confirm our findings, we decided to fool the GHC's strictness analyser using the `wrap` function.

```
f :: [Int] -> [Int]
f a = foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0

main = print $ wrap $ f []
```

When the result of `f` is passed through `wrap`, the printing code is included and a (somewhat less) incorrect result gets printed.

```
[]
```

But unlike previous examples, where expressions were lazier as a result of some `seq`s not being executed, here they do not occur and the returned value seems arbitrary. Indeed, it turned out that the body of `f` is polymorphic in the result type and we can make it return an integer, as follows:

```
f :: [Int] -> Int
f a = foldr (\b -> \c -> c) (undefined ()) (a ++ a) 0

main = print $ wrap $ f []
```

This program, when optimised, prints:

```
1099511628032
```

What is more, when we choose `(Int, Int)` as the result type, the program dies with a segmentation fault.

Earlier in this chapter, we discussed the situation where a partial correctness argument might be invalidated if some expression in a

program is evaluated successfully by mistake instead of crashing. This example suggests that GHC itself can fall into this trap. If GHC's analysis concludes that `f []` will crash, then the compiler is 'careless' about handling its result, because it believes that the result will never be returned. Due to an error in optimisation, the result is nevertheless returned, which may lead to the effects that we demonstrated. Thus, failure to throw an exception may lead to worse consequences than space leaks or too lazy expressions.

The bug was reported as ticket 5626 and fixed as a result of another bug report.

2 Evaluation order

We decided to investigate whether optimisation changes the order of evaluation of expressions, which is interesting for two reasons. First, it is interesting to see which changes actually take place to get an understanding of optimisations performed by GHC; and secondly, to relate that to the changes that GHC is allowed to make, as it is possible that the evaluation order is changed in an invalid way.

The Haskell programming language is designed with the intention that the evaluation order should not matter for programs. Specifically, the evaluation order is not observable, unless the program uses unsafe programming constructs, and the compiler is free to use any evaluation order as long as the non-strict semantics is preserved [21].

Still, the evaluation order is important for reasons of efficiency, which is why GHC gives certain guarantees about it to make it possible to write efficient programs. Essentially, the evaluation order used by GHC is determined by the call-by-need evaluation strategy, with the exception that in some cases the order might be chosen freely by the compiler [27].

A Haskell program may exhibit different levels of space usage depending on what is the exact order of evaluation [12], but efficient programs rely on the additional guarantees provided by GHC for predictable space behaviour.

Determining the evaluation order is impossible in general, since it is not observable as far as pure computations are concerned, but we can use a trick to observe it. To detect which of two subexpressions of an expression is evaluated first, we can make use of catching exceptions. Not unexpectedly, catching exceptions is an impure operation, which is why it lets us observe the evaluation order.

Consider expression e that contains two subexpressions a and b.

$$e = \ldots a \ldots b \ldots$$

We can replace the two subexpressions with error terms as follows:

$$e' = \ldots \texttt{error "aaa"} \ldots \texttt{error "bbb"} \ldots$$

If evaluation of the term requires both subterms to be evaluated, one of the exceptions will be thrown and precisely which one gets thrown depends on their relative evaluation order. It is reasonable to expect that the error term that is evaluated first will yield an exception.

Replacing a subexpression with the `error` x expressions carries the risk that the presence of undefined subterms will affect the optimisations performed by the compiler on the expression. Thus, instead of placing the `error` x terms inside of the expression it is better to create a function that is later applied to suitable error terms and make sure its arguments are not inlined.

$$e'' = \lambda ab. \ldots a \ldots b \ldots$$

$$e'' \ (\texttt{error "aa"}) \ (\texttt{error "bb"})$$

Testing was performed using the same approach as with the previous property, that is by comparing outputs of the same module compiled with different optimisation options. The module skeleton was slightly different as this time we were interested in discriminating between different exception kinds. Therefore, the exception handler also prints the exception string:

```
handler (E.ErrorCall s) = putStrLn $ "*** Exception: " ++ s
```

The modules were again compiled with no optimisation and with `-O -fno-full-laziness` options. The criterion for selecting offending terms was that a different exception from the two passed as arguments is thrown by each variant of the module.

2.1 Results

Failure 5 Terms that exhibit the above behaviour turned out to be quite common. One of the simplest terms found is the following:

```
\aa bb -> seq aa (seq bb aa)
```

We can rewrite it using the infix notation for `seq` for clarity.

```
\aa bb -> aa `seq` bb `seq` aa
```

The term represents a function that forces the evaluation of both of its arguments and returns the first one. When applied to two error terms the unoptimised and optimised versions produced different results. The unoptimised one printed the exception thrown by the first error term:

```
*** Exception: aa
```

The optimised threw the other exception, indicating that `bb` is evaluated first.

```
*** Exception: bb
```

Inspecting the GHC's internal Core representation of the optimised program reveals that the code of the function was transformed into `\aa bb -> bb `seq` aa`, omitting the `seq` operation on `aa` altogether.

This result surprised us at first, as `seq` is often used to eliminate space leaks by making sure that its first argument is evaluated before the second one. For example, the definition of the standard library function `foldl'` relies on performing `seq` to avoid a space leak.

To answer whether omitting this `seq` is acceptable, we consulted the Haskell Report [21], which provides the following definition of `seq`:

```
seq bot b = bot
seq a   b = b if a /= bot
```

This definition is a semantic one and ensures that `seq` is strict in its first argument, however it says nothing about the evaluation order. A naïve implementation of `seq` would simply force the first argument before returning the second one, and this would result in the behaviour which programmers seem to rely on. However, if evaluating the second argument of `seq` forces the evaluation of the first one by itself, the compiler might omit the `seq` operation without violating the semantics. This is indeed the case in the discussed example, as returning `aa` means that it will be forced by the caller of the function, so forcing it beforehand is not necessary.

The lesson from this example is that `seq`, which is implemented correctly by the compiler, *might still not guarantee that its first argument will be evaluated before the second!* Unfortunately, many of the Haskell programs and libraries rely on this guarantee to avoid space leaks, such as the above mentioned `foldl'` function.

This fact, while neglected by many, has been known before. For example, as outlined in [22], it is apparent that an operation that forces a specific evaluation order is needed for implementing efficient parallel computations in Haskell. However, restricting `seq` to allow only a specific evaluation order might eliminate some optimisation opportunities, which is why another variant of it called `pseq` was introduced. The new construction has the same semantics as `seq`, but is guaranteed to evaluate its first argument before its second.

Thus, consistently with [22] and the Haskell Report, we would make the following recommendation.

- Whenever strict ordering of evaluation of expressions is needed, `pseq` should be used.
- The `seq` operator should be used to change the strictness of expressions, but not to enforce the order of evaluation.

Failure 6 Here is another term reported using the property.

```
\aa bb -> (+) (length (take bb ([]::[] Bool))) aa
```

The term performs integer addition of an expression that depends on arguments `bb` and `aa`. At first it seems that `bb` should always be evaluated first, since + should evaluate its first argument before the second one. However, the + operation is one of the cases where GHC is allowed to alter the evaluation order for reasons of efficiency [27]. Thus, the change of evaluation order in this case is also legitimate.

Many more terms involving `seq`, + and functions involving + were reported by the property. Unfortunately, only manual inspection of them was able to establish whether the changes in the order of evaluation were allowed, which means that we checked only a small number.

To perform more effective testing of this issue, a more accurate property would have to be constructed. However, it would require precise modelling of the Haskell semantics, which is an ambitious task by itself. It is not clear if a truly 'differential' technique that does not rely on a complex oracle could be applied here.

3 Equivalence of inlined expressions

Another property that we constructed compares the observable behaviour of a `let` expression and its reduced form, which we can

portray with the following equation.

$$\texttt{let x } = e \texttt{ in } C[\texttt{x}] \;\approx_{\mathsf{obs}}\; C[e]$$

Notation $C[t]$ denotes an expression with zero or more gaps that are filled with occurrences of expression t. The two expressions should behave in the same way if no impure language constructs are used by them.

The initial goal of this effort was to reveal impure behaviour of some library functions. We failed to reach it. However, an interesting compiler bug was found in the process.

The property is implemented in a similar way as previous ones. However, only one module is created that contains both variants of expressions that are compared. For each test case two terms are generated, one representing the expression e and one representing the context $C[\bullet]$. The second term is then used to create expressions $C[\texttt{x}]$ and $C[e]$.

To disregard the effects of possibly changed evaluation order the property treats all thrown exception types as equal, as was the case in our first property.

Failure 7 The following expression was found to behave differently than its reduced form when compiled with GHC:

```
let x = error "aaa" in seq (seq (tail ([]::[Int])) (\a -> x))
```

When we run this expression as part of the program below, it yields the correct result, that is it crashes and prints the following exception: `Prelude.tail: empty list`.

```
print $
  wrap
    (let x = error "aaa" in
       seq (seq (tail ([]::[Int])) (\a -> x)))
    [0]
```

However, if we replace the generated expression in this program with its second variant, shown below, the program prints `0` instead.

```
seq (seq (tail ([]::[Int])) (\a -> error "aaa"))
```

It is interesting that the program gets miscompiled regardless of the optimisation level used and even unoptimised compilation yields the same erroneous results. Thus, this failure could not be detected using

our previous approach of using differential testing with different optimisation levels.

The approach of differential testing using pairs of expressions like the one in this example is inspired by traditional property-based testing where simple logical properties, such as equality laws, are tested. This gives us additional bug-finding power, which is not possible with traditional differential testing when only one compiler, the one under test, is available.

And as in any programming language, in Haskell there are many possible schemes of generating equivalent expressions. The failure that we found suggests one more such scheme, apart from a `let` expression and its reduced form. As it turns out, if we replace `error "aaa"` with `undefined` in the offending expression, as shown below, it is again compiled correctly.

```
seq (seq (tail ([]::[Int])) (\a -> undefined))
```

Thus, one possible scheme is pairs of expressions where occurrences of `error "aaa"` are replaced with `undefined`.

The failure has been reported as GHC ticket 5557 and fixed before version 7.3.20111022.

4 Equivalence of different error expressions

The bug that we found using the property described above led us to investigate the following property.

$$C[\texttt{error "aaa"}] \approx_{\text{obs}} C[\texttt{undefined}]$$

This property is implemented similarly to the previous one, but requires generating only one term that represents $C[\bullet]$.

As expected, testing this property yielded the same, and similar, failures as the previous one, and unfortunately no fresh failures.

Thus, we decided to modify the property and tested the following one instead.

$$C[\texttt{hiddenError}] \approx_{\text{obs}} C[\texttt{undefined}]$$

Here `hiddenError` is an expression that crashes, but is defined in such a way that makes it impossible for GHC to determine its semantics at compile time. We hoped that using an expression with a concealed

crash might make GHC assume that it will not crash and use a transformation that is only valid for non-crashing expressions on it.

Failure 8 Indeed, testing found interesting terms that were miscompiled, one of which is presented below.

```
hiddenError = error "hidden error"
main = do
  print $ seq
            (head (map (\a -> \b -> hiddenError)
                       (hiddenError::[] Bool)))
            id
            [1]
```

When this program is compiled with optimisation using the `-O -fno-full-laziness` options, it prints `[1]` instead of crashing. As visible in the program, it is actually enough that `hiddenError` is just a plain definition using `error ...`, without further obfuscation. However, using `error ...` directly does not trigger the bug.

The failure has been reported as GHC ticket 5587, which now contains a comprehensive explanation of its causes. The offending expression, which contains a `head` function applied to `map`, undergoes complex transformations using rewrite rules [28] thanks to list fusion [10]. This process leads to a subexpression that is a case expression whose one branch crashes, while the other one is a function that expects one more argument. For reasons of performance, this subexpression gets η-expanded, so that it crashes only when the extra argument is applied, changing its behaviour in some contexts.

This particular counterexample has an interesting feature, in that the code that triggered the bug was created indirectly by GHC by transforming the original expression using inlining and rewrite rules. The resulting expression had elements that were not possible to generate using our generator, but were needed to cause the failure, such as case-expressions. Thus, given the multi-pass operation of the GHC optimiser, it is possible to subject the compiler to a wider variety of expressions than the ones that are possible to generate directly.

It is also worth noting that even though the shrinking process only tries to reduce terms structurally, it shrank this counterexample very well. The sequence of rewrite steps performed during the optimisation process reveals that the counterexample is in fact minimal.

The bug has been fixed by introducing a new compiler option `-fpedantic-bottoms`. Using this option causes the compiler to omit the

erroneous transformation. However, the default is still to perform it. The motivation for this is that increased performance might be attained at a price of changes in the semantics. However, it is not known at this point what is the performance penalty of `-fpedantic-bottoms`.

5 Summary

Testing the GHC compiler using randomly generated simply-typed lambda terms and differential testing proved to be effective, even when only a small fragment of the Haskell language is covered. We found many interesting failures, eight of which we presented here. Four of these counterexamples resulted in high-quality bug reports, which were acknowledged by the GHC developers as relevant.

We managed to obtain succinct counterexamples without understanding the inner workings of GHC, only by using shrinking, which automatically reduces a failing test case to a smaller one. Shrinking usually resulted in test cases that were not possible to be shrunk further by hand. Shrunk counterexamples for a single property are often similar and looking at their similarities allows for making educated guesses about the cause of the failures.

The fact that the test cases have been reduced by shrinking proved to be of enormous help. In one case it was possible to find a well-hidden error in one of the GHC's phases of compilation in a matter of minutes, something that would not be possible if the reported test case was large.

All the reported bugs concerned strictness of expressions being incorrectly changed in the process of optimisation. We also investigated whether optimisation might result in changes in the evaluation order of expressions. However, we were not able to detect any bugs there. What we discovered, instead, was that in many cases the order of evaluation is unspecified, such as when the operator `seq` is used, which is permitted by the Haskell Report. On the other hand, programmers often rely on `seq` having a determined evaluation order, which may cause their programs to exhibit space leaks if an alternative order is used. Thus, in addition to finding bugs in the compiler, our testing method can be used to support or disprove hypotheses about the compiled programs, which may help in understanding of the compiler.

We used differential testing using one compiler implementation, but comparing behaviour of programs compiled using different opti-

misation levels. In addition to that, we used another form of differential testing where two programs are compared, which are different but have equivalent semantics. The second approach led to discovering failures that were not possible to discover using the first one.

Chapter 5

Related Work

1 Compiler test tools

CSmith [35] is a random C program generator aimed at testing compilers. It attempts to generate C programs that avoid undefined or unspecified behaviour [15] without compromising the expressiveness of the generated programs. To achieve this goal CSmith employs a relatively complex program generator that uses different techniques for producing safe programs. Firstly, it avoids some unsafe behaviours simply by introducing structural constraints. And secondly, for cases where this would be too restrictive, the generator resorts to performing static analysis on already generated code fragments to determine whether a given operation is safe, or by inserting runtime safety checks in the generated code.

Evaluation of test results is done using differential testing with different compilers, or different options to the same compiler. The comparison of effects of two executions is performed by comparing checksums of non-pointer global variables sampled at the end of each execution. A variety of compilers were tested, including GCC, LLVM, CompCert and commercial C compilers. CSmith was able to uncover as many as 325 previously unreported bugs in all compilers altogether, most of them in GCC and LLVM. Even CompCert, which has a formally verified core, exhibited a number of bugs.

CSmith has no means of reducing the size of a failing test case, as it would be difficult to ensure that a shrunk test case is also free from undefined or unspecified behaviour. Programs containing 8k–16k tokens gave the highest rate of triggering bugs, and reducing them by

hand was employed to obtain understandable test cases.

Lindig [20] created a simple tool called Quest for testing the C function calling convention of C compilers. This tool randomly generates programs containing C functions that execute consistency checks to verify that their arguments have been passed correctly. Program generation is *type-driven*, that is the type of a function is first picked at random and a suitable body is generated algorithmically. Although Lindig claims that his method does not require a language specification, he relies on a partial specification stipulating that the consistency checks should succeed. The scheme was able to detect bugs related to passing function arguments in 5 different compilers. Bugs found by Quest were triggered by surprisingly simple code, which is explained by the fact that the static test suites used to by compiler writers contain very few kinds of argument and result types of functions.

McKeeman [23] presents a case of differential testing of C compilers using inputs of various *quality levels*. Starting with sequences of any ASCII characters, which have the lowest quality level, the inputs range through valid sequences of tokens and syntactically correct programs to reach programs with well-defined semantics. This led to successful finding of errors in different stages of the compilers tested. Additionally, starting with a test case from any level, 'nearby' test cases are created by introducing small changes to the original test case, which often causes a tested compiler to crash, uncovering a bug.

The test case generator was implemented as a Tcl script, which is based on a context-free-grammar-based generator enhanced to support context-sensitive features, like tracking defined variables. Grammar rules are weighted and termination is ensured by assigning small enough weights to recursive rules.

If a failing test case is found, a shrinking process is applied to reduce its size. Failing test cases can be as big as 600 lines of code and can often be shrunk to just several lines of code. However, this might require about 10000 compilations.

Instead of avoiding illegal operations at higher quality levels, whenever there is a discrepancy in the behaviour of two compiled versions, the program is rerun with all potentially problematic operations replaced by their error-checking variants. If an error is detected, the test case is discarded.

The highest level of *quality* that can be generated comprises of programs with meaningful semantics. Programs of this level are generated from specific templates that define their high-level structure,

which guarantees certain semantic properties. Of course the diversity of the generated programs is traded here for semantic correctness, as the programs are much more specific than those from lower quality levels.

Our work does not have the breadth of CSmith or the McKeeman's tool, as we cover a much smaller part of the language that we generate. However, in that part we are able to generate very interesting programs, thanks to using a formal calculus that ensures well-typedness. We also had to solve problems that are absent while generating C programs, such as generating higher-order and curried functions and parametric polymorphism. Like McKeeman's work, our testing tool shrinks counterexamples, but does it in a type-safe way that guarantees to preserve typing and makes it much more efficient by using batch testing.

Hanford [13] presented an early example of a recursive, grammar-based random program generator used for testing compilers. The generator is based on context-free grammars, which are dynamically modified during generation to accommodate some context-sensitive behaviour, for example when a new variable is introduced. The generator has a limited support for backtracking, which occurs when it is not possible to rewrite some non-terminal. The tool has been used to test compilers for simple properties, such as using programs that are syntactically-correct, or containing syntax errors, for example integer expressions in place of boolean ones.

2 Shrinking

Shrinking proved to be a very effective technique in property-based testing and is now standard in Haskell QuickCheck [4] and Erlang QuickCheck [14]. Shrinking allows for defining generic shrinking methods for polymorphic data types, which can be composed with shrinking methods for their element types. For instance, the default shrinking method for lists of integers uses the shrinking method for lists and also that for integers to reduce individual elements.

A similar technique has been invented concurrently, called *delta debugging* [36], which is broader, but when applied to test input it bears resemblance to shrinking. For example, the standard method for reducing strings using delta debugging is very similar to the default shrinking method for lists. Delta debugging has been applied successfully to obtain small failing test cases for large and complex

software.

Like shrinking, delta debugging finds a failing test case that is locally minimal. However, delta debugging assumes a different model for reducing test cases. A test case is first decomposed into a number of independent changes that represent transformation of an empty test case into the original test case. Then, a locally-minimal set of changes is determined, that results in a failing test case. Shrinking in QuickCheck, on the other hand, places very loose requirements on each specific shrinking method.

3 Library test tools

Klein et al. [18] created a testing tool that generates random programs to test an object-oriented library. Their generator is capable of producing higher-order object-oriented programs (which override methods) and supports monitoring of pre- and post-conditions, which are used to establish the validity and result of the test. Their generation method uses generation rules similar to ours, with random rule selection, size bound, and backtracking. Rather than our INDIR rule, which generates calls of functions in the environment only when their result type matches the target type, they use a rule that can generate a call of *any* function in the environment at any time, binding its result to a fresh local variable, which can then in turn be used in another attempt to generate a term of the target type. The advantage of their approach is that it is easier to generate calls of functions in the environment; the disadvantage is that many of the local variables they create are never used, because their types do not match the target type. Klein et al. do not consider polymorphic types, nor do they shrink failing test cases to minimal examples as we do.

Wrangler, a refactoring tool for Erlang has also been tested using random program generation [8]. A rich program generator has been created, which is capable of generating full modules. Even though Erlang is an untyped language, the generator takes types into consideration in order to avoid argument mismatches when calling functions. Similarly, Daniel et al. [6] exhaustively generate Java programs (up a to certain size) in order to test the refactoring engines in Eclipse and NetBeans. Different from our approach, some of the generated programs are not valid inputs for the Java compiler.

Generating random sequential programs is practised in testing monadic code with QuickCheck [5] and in testing stateful programs

with Erlang QuickCheck [14]. Such a program is usually a sequence of actions, which might contain variables, but all variable handling is outsourced to the programmer using QuickCheck. Often such generated programs are not parametrised, which makes it possible to ensure that preconditions of all actions are satisfied.

4 Testing of formal models

Redex [16] is a tool for lightweight verification of programming language formal models. Formal models are randomly tested whether they satisfy the stated properties using randomly generated expressions of the 'object' language. Generation of size-bounded terms is based on grammars (syntax of the object language is defined using a context-free grammar), and malformed expressions, for example containing free variables, are filtered out. Naïve generation and filtering is, of course, not enough to test more complex models, as many reduction rules are never exercised. To raise the likelihood of generating expressions that could be reduced with these reduction rules, their left-hand sides are used to guide the generation. Redex has been successfully used to formalise and test a nine existing formal models and find mistakes in all of them.

5 Typed term generators

Djinn [2] solves the *type inhabitation* problem for simply-typed λ-calculus, that is, it returns *any* term instead of a random one for a given type. It is based on a terminating proof procedure for intuitionistic propositional logic [9], which makes it find a term of a given type reliably when one exists. It is limited, however, in that it does not perform *polymorphic instantiation*, which means that it cannot generate some terms involving polymorphic constants.

Vytiniotis and Kennedy [33] present encoding of data types into streams of bits, which can be used for their random generation. In their approach to generating simply-typed λ-terms, the target type is *never* fixed, and thus the generation never fails, eliminating the need for backtracking. This way of generating well-typed terms can also be extended to simply-typed lambda calculus with polymorphic constants. Reducing the problem of random data generation to encoding in bit-streams has the consequence that improving the distribution

of generated data corresponds to inventing efficient compression schemes, such as Huffman coding.

The λ-term *enumerator* developed by Yakushev and Jeuring [31] creates function applications in the same way as our method, by generating a candidate type for the argument, and trying to generate the argument afterwards.

6 Untyped term generators

Statistical properties of random untyped λ-terms have been explored in [3], which also explores a method of generating them using Boltzmann sampling. Generation of random untyped λ-terms is tackled in [34], which employs counting of possible subterms to achieve uniform generation distribution. Correspondingly, the work in [24] examines the proportion of simple types that are inhabited, that is, for which it is possible to create a term of that type.

TGGS [11] is a random test data generation system based on context-free grammars enhanced with context-sensitive constructs, like imperative actions conditional clauses and stacks, which serve a similar rôle to attributes in attribute grammars. The system generates data by expanding non-terminal symbols by choosing grammar rules at random and backtracks whenever that is not possible, rolling back all relevant imperative actions. It is possible to affect the distribution of the generated data using weights, which influence how often different grammar rules are chosen.

Chapter 6

Future work

The potential for finding bugs in GHC using the presented method has not at all been exhausted during the testing that we performed. Although most of the presented bugs have been fixed, and our properties find counterexamples at a much lower rate, we are still able to find new interesting error cases using them. Many more new properties and variations of existing ones are likely to yield even more new counterexamples.

The same method can be applied to other Haskell compilers. However, it might be less effective for compilers that do not perform as sophisticated optimisation as GHC. When more Haskell compilers are available, it would be interesting to perform the standard variant of differential testing i.e. cross-testing of two different implementations.

Given that the subset of Haskell that is randomly generated is very limited, there is room for improvement by adding support for more language constructions that can occur in the generated terms. For example, `let` and `where` clauses could be generated in a similar way to function applications and `case` expressions could be generated by having polymorphic constants that are required to be fully-applied. Polymorphic `let` bindings could also be supported by means of introducing polymorphic constants in their bodies and dummy type constants (simulating 'rigid' type variables) in the type of bound expressions.

The biggest technical challenge to solve in the current generator is to improve on the generation of terms involving polymorphic constants like `map` or monadic `bind`, which suffers from problems described in Section 2.1. The problem can be alleviated, for example, by allowing the generator to generate terms with partially-specified

types. One argument of `map` would be generated with a partially-specified type while the other argument would be generated with a type that agrees with that of the already generated subterm.

More drastic redesign is also possible in order to solve this problem. Theorem proving techniques might be used to track unresolved type variables and propagate the changes whenever they are refined in one of the subterms. However, we have been trying to avoid using theorem proving techniques as this approach would change the scope of this project too radically. We nevertheless think that using such an approach would be legitimate in well-typed term generation.

Even more radical would be creating a generator that approximates the uniform distribution of terms (of a given size) by counting all possible terms that can be generated, as it is done in AGATA [1]. However, it is not clear whether this approach is feasible computationally as the data type of well-typed terms is very complex, and quite large terms are needed to perform useful testing.

Another approach that could possibly be used is to adapt the technique proposed by Vytiniotis and Kennedy [33]. The advantage of this approach is its simplicity and elegance, but a naïve generator seems to be very skewed towards creating terms with partially-applied constants, which suggests that much effort might be required to correct the distribution.

Chapter 7

Conclusions

We applied property-based testing and random program generation for testing a sophisticated optimising Haskell compiler. Even though we generated a limited subset of Haskell, we were able to find interesting bugs in the compiler. Found counterexamples were reduced structurally using shrinking, which made them understandable and well-suited for bug reports.

The properties used for finding the bugs employed differential testing by comparing the behaviour of the same program compiled with different optimisation levels. Also, we used an alternative form of differential testing where the behaviour of two equivalent programs is compared, which was used to find more bugs. In addition to bug reports, we learned about valid interesting behaviour of GHC, and in particular about changes to the default order of evaluation that it performs.

We have two positive observations about structural shrinking of counterexamples. First, even though the process is oblivious to the complex internal workings of the tested compiler the shrunk counterexamples could not be reduced by us further by hand in most cases, which suggests that the results were close to optimal. And secondly, looking at shrunk counterexamples allowed us to make educated guesses about the cause of the failures, even without referring to or understanding the compiler's code.

Unfortunately, testing a compiler using a random program generator is hardly a *fully automatic* technique, which we hoped it to be in the beginning. In contrast, we found that effective testing requires spending effort on creatively devising properties. In particular, some unexpected bugs were found by properties that were created for an-

other purpose. However, the technique brings some automation to finding compiler bugs.

We were satisfied with the relevance and quality of counterexamples that we found for GHC with reasonable effort. Based on this experience we think that random compiler testing is an attractive technique for finding compiler bugs, which could be scaled up to perform much more comprehensive testing than we performed.

References

[1] J. Almström Duregård. AGATA: Random generation of test data. Master's thesis, Chalmers University of Technology, Dec. 2009.

[2] L. Augustsson. Announcing Djinn, version 2004-12-11, a coding wizard. http://permalink.gmane.org/gmane.comp.lang.haskell.general/12747, 2005.

[3] O. Bodini, D. Gardy, and B. Gittenberger. Lambda terms of bounded unary height. In *Proceedings of the 8th Workshop on Analytic Algorithmics and Combinatorics*, 2011.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2000.

[5] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 65–77, New York, NY, USA, 2002. ACM.

[6] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the 6th meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2007.

[7] N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In *Mathematics of Program Construction*, pages 85–109. Springer, 2004.

[8] D. Drienyovszky, D. Horpácsi, and S. Thompson. QuickChecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*. ACM, 2010.

[9] R. Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57(3), 1992.

[10] A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[11] R. F. Guilmette. TGGS: a flexible system for generating efficient test case generators. Technical report, RG Consulting, 1995.

[12] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 265–276, New York, NY, USA, 2001. ACM.

[13] K. V. Hanford. Automatic generation of test cases. *IBM Syst. J.*, 9(4):242–257, Dec. 1970.

[14] J. Hughes. QuickCheck testing for fun and profit. In M. Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-69611-7_1.

[15] ISO. ISO C Standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[16] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM.

[17] C. Klein, M. Flatt, and R. B. Findler. The racket virtual machine and randomized testing. Available from `http://plt.eecs.northwestern.edu/racket-machine/`, 2010.

[18] C. Klein, M. Flatt, and R. B. Findler. Random testing for higher-order, stateful programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2010.

[19] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[20] C. Lindig. Random testing of C calling conventions. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging*. ACM, 2005.

[21] S. Marlow. Haskell 2010 language report. `http://www.haskell.org/definition/haskell2010.pdf`.

[22] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[23] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[24] M. Moczurad, J. Tyszkiewicz, and M. Zaionc. Statistical properties of simple types. *Mathematical. Structures in Computer Science*, 10, Oct. 2000.

[25] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[26] S. Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[27] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 25–36, New York, NY, USA, 1999. ACM.

[28] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233, Sept. 2001.

[29] S. L. Peyton Jones. Compiling Haskell by program transformation: a report from the trenches. In *Proceedings of European Symposium on Programming*. Springer-Verlag, 1996.

[30] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[31] A. Rodriguez Yakushev and J. Jeuring. Enumerating well-typed terms generically. In *Approaches and Applications of Inductive Programming*, volume 5812 of *LNCS*. Springer Berlin / Heidelberg, 2010.

[32] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, 2002.

[33] D. Vytiniotis and A. J. Kennedy. Functional pearl: every bit counts. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

[34] J. Wang. Generating random lambda calculus terms. Technical report, Boston University, 2005.

[35] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[36] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.