Testing an Optimising Compiler by Generating Random Lambda Terms

Michał H. Pałka Koen Claessen
Alejandro Russo
Chalmers University of Technology
{michal.palka,koen,russo}@chalmers.se

John Hughes
Chalmers University of Technology and
Quviq AB
rjmh@chalmers.se

ABSTRACT

This paper considers random testing of a compiler, using randomly generated programs as inputs, and comparing their behaviour with and without optimisation. Since the generated programs must compile, then we need to take into account syntax, scope rules, and type checking during our random generation. Doing so, while attaining a good distribution of test data, proves surprisingly subtle; the main contribution of this paper is a workable solution to this problem. We used it to generate typed functions on lists, which we compiled using the Glasgow Haskell compiler, a mature production quality Haskell compiler. After around 20,000 tests we triggered an optimiser failure, and automatically simplified it to a program with just a few constructs.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification

Keywords

Software Testing, Random Testing

1. INTRODUCTION

Testing a compiler traditionally relies on running it on a suite of hand-written test programs. This approach is problematic for two reasons. Firstly, because collecting a large number of suitable programs is difficult and such programs rarely cover all interesting cases of code [7, 10]. Secondly, because the compiler is tested against the same set of programs over and over again, which means that if a bug is not triggered by any of them, then it will never be found. Using random property-based testing is an alternative that could remedy both of these problems. However, this alternative requires automatic generation of test programs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AST'11, May 23-24, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0592-1/11/05 ...$10.00

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 paper, we study the problem of generating random, type-correct programs. We chose a simple, yet rich, statically typed programming language, namely the *simply-typed lambda calculus* [13]. The lambda calculus (λ-calculus) is very simple — it basically only contains anonymous functions, a feature found in many contemporary programming languages, such as Haskell, Scheme, Python, C♯, Visual Basic, and so on. However, the λ-calculus still captures the two main aspects that makes generating random programs hard: variable binding and type-correctness. While it is quite easy to ensure that generated programs only refer to variables in scope, we found that satisfying the type-checker is much more subtle, and this is the main focus of this paper.

Note that, even when testing compilers for dynamically-typed languages, it makes sense to use statically type-correct programs, since these programs are far more likely to be runnable without crashing immediately.

For our experiments, we have chosen an industrial-strength compiler that accepts expressions from λ-calculus directly, namely the Glasgow Haskell compiler [15] (GHC). This compiler contains a powerful optimiser, which consists of numerous complex transformations operating on the GHC's intermediate Core language, such as inlining, let-floating, lambda lifting, specialisation and common subexpression elimination [12]. Such elaborate processing could easily be a source of intricate bugs, making it interesting to test.

The compiler was tested by compiling randomly generated functions with different optimisation settings and comparing the behaviour of resulting programs under the assertion that optimisation should not change the meaning of the program. These tests revealed actual failures in the compiler.

The remainder of the paper is structured as follows. In the next section we explain the simply-typed λ-calculus, the language we target. In section 3, we explain our basic approach to random generation. Section 4 makes an important extension to *type polymorphism*, essential to generating interesting Haskell programs. Section 5 describes the generation algorithm in detail, including optimisations to reduce the need for search. Section 6 presents the results of our

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

Figure 1: Syntax for simply-typed λ-calculus

compiler testing using the generated λ-terms. Section 7 describes related work, and section 8 concludes.

2. LANGUAGE

The formal language that we choose to develop our ideas is essentially the simply-typed λ-calculus [13] extended with constants and basic types. The calculus allows programs to define and manipulate variables and functions. Specifically, programs are constructed from four different kinds of terms: variables (x), constants ($\mathbf{c}$), function applications, and anonymous functions. Terms of the form $\lambda x : \sigma.\ M$, referred to as *lambda expressions* (λ-expressions), represent functions with argument x of type σ and body M. Types are discussed in the next section; they include, for example, traditional built-in types denoting integers or strings. Terms of the shape $M\ N$ represent application of functions, i.e. the first term M, which is a function, is given the second term N as an argument. Originally a feature of functional programming languages such as LISP, Scheme and Haskell, λ-expressions are now found in many contemporary programming languages, including Python, Ruby, JavaScript, C♯, Visual Basic, and Visual C++.

Although minimalistic, simply-typed λ-calculus can represent a wide range of programs. For example, there are no infix operators, but they can be represented as constant functions. λ-expressions define functions taking a single argument, but multi-argument functions can be represented as functions returning functions. Thus 2×3 is represented as `(* 2) 3`, where `(* 2)` is the constant function `*` applied to `2`, returning a function of one remaining argument (`3`) that doubles it, returning `6`. Correspondingly, multi-argument functions are defined using nested λs; the C function `int f (int x, int y) return 2 * x + y;` can be represented by the λ-expression $f = \lambda x : \texttt{Int}.\lambda y : \texttt{Int}.\texttt{+}\ (\texttt{*}\ \texttt{2}\ x)\ y$ Function application brackets to the left, so $(\texttt{*}\ \texttt{2}\ x)$ just means $((\texttt{*}\ \texttt{2})\ x)$, as above. The variables x and y in the function body of f are said to be bound by the λ-expressions; variables which are not bound are said to be *free*. For simplicity, we consider constants such as `+`, `*` and `2` to be free variables defined in a scope enclosing the entire program. Figure 1 shows the formal syntax of the simply-typed λ-calculus.

In simply-typed λ-calculus, variables and expressions have only base or function types. A base type can be, for instance, `Int` or `Bool`. Function types, on the other hand, are of the shape $\sigma \to \tau$, representing a function which takes a value of type σ as argument and returns a value of type τ as result. Observe that σ and τ can be any type—functions can take functions as arguments, and return them as results.

Functions taking many arguments are represented by functions whose result type is a functional type. For example, the function `*`, shown above, has type $\texttt{Int} \to (\texttt{Int} \to \texttt{Int})$, i.e. a function that, after being applied to an argument of type `Int`, returns a function that can be applied again to another `Int` in order to return their product. It is common to treat $\to$ as a right-associative operator and thus write $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$ instead.

In addition to syntactic restrictions, terms must be well-

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

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

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

Figure 2: Typing rules

typed, i.e. functions must only be applied to correctly typed arguments. For instance, the function `*` must be applied to arguments of type `Int`, i.e. numbers, and not to arguments of type `Bool`. Well-typed terms are defined using assertions of the form $\Gamma \vdash M : \sigma$ (*typing judgements*), meaning that term M has type σ provided that the types of free variables in M are as specified in Γ. The *environment* Γ is just a list of elements of the form $x : \sigma$ ($\mathbf{c} : \sigma$), indicating that variable x (constant $\mathbf{c}$) has type σ. Figure 2 formally describes Γ.

The rules defining valid typing judgements are shown in Figure 2. These rules have zero or more premises placed above a horizontal bar, with their consequence below it. The rule (Var) indicates that x has type σ ($\Gamma \vdash x : \sigma$) if x is associated with σ in Γ ($x : \sigma \in \Gamma$). Rule (Lam) is used for typing functions. To determine that a function takes an argument of type σ and returns a result of type τ ($\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$), the rule requires that the body of the function has type τ ($x : \sigma, \Gamma \vdash M : \tau$) when the argument of the function is assumed to have type σ; this assumption is captured by the expression $x : \sigma, \Gamma$. Rule (App) makes sure that the function's actual argument (N) has a type matching the one in the function's type, i.e. σ.

3. RANDOM LAMBDA-TERMS

The typing rules suggest a straightforward generation procedure for well-typed terms. Each typing rule can be interpreted as a generation rule by reading it 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. Our procedure is goal-oriented and has as input the *target type*, which is the type that generated terms will have, and an environment containing all variables and constants that can be used in the terms.

Depending on the target type, a subset of the typing rules may be applicable. For example, the (Lam) rule is only applicable when the target type is a function type. The (Var) rule is only applicable when the environment contains a variable or a constant of the given target type.

The following example illustrates how generation works. Suppose that we have to generate a term of type `Int` and that two constants are available in the initial environment Γ, representing the number zero and the successor function.

$$\Gamma = \{\texttt{zero} : \texttt{Int}, \texttt{succ} : \texttt{Int} \to \texttt{Int}\}$$

We start off with the initial environment Γ and the target type, whereas the term that we want to generate, which is as yet unknown, is denoted by a placeholder **?**.

$$\Gamma \vdash \mathbf{?} : \texttt{Int} \qquad (1)$$

Both the (Var) rule, using constant `zero`, and the (App) rule can be applied at this point to obtain a term of type `Int`. Choosing the former immediately terminates the generation, since a variable has no subterms, and the generated term would be complete. Choosing the latter, on the other hand, requires generating two subterms, i.e. a function returning an `Int` and its argument. Type σ in the (App) rule is neither determined by the environment nor by the target type and, as a consequence, can be freely chosen. We arbitrarily decide to select the (App) rule and to fix the σ type to `Int`. Since both σ and τ from the (App) rule are determined to be `Int`, to finish the generation it is necessary to generate two subterms, denoted by $?_1$ and $?_2$, with the following types.

$$\Gamma \vdash ?_1 : \texttt{Int} \to \texttt{Int} \qquad\qquad \Gamma \vdash ?_2 : \texttt{Int}$$

To generate the first subterm, any of the three rules might be used. In particular, the (Lam) rule is permitted by the fact that `Int` $\to$ `Int` is a functional type. Nevertheless, we opt for the simple alternative of using the (Var) rule and therefore $?_1$ is replaced by `succ`.

$$\Gamma \vdash \texttt{succ} : \texttt{Int} \to \texttt{Int}$$

To generate $?_2$, we choose simply to use the constant `zero` from the environment exercising the (Var) rule.

$$\Gamma \vdash \texttt{zero} : \texttt{Int}$$

Observe that the term $?_2$ is generated under the same environment and the target type as the main term $?$ from equation 1, which means that the generator could proceed in exactly the same ways here as at that point.

To finish the generation, our procedure constructs the main term from equation $? = ?_1\ ?_2$, since the (App) rule was invoked in the first step, which gives us the final term.

$$\Gamma \vdash \texttt{succ zero} : \texttt{Int}$$

3.1 Naïve Approach

The generation procedure derived from the typing rules is *non-deterministic* in that more than one rule may be applicable at any given point, and a single rule may be applied in many different ways. Thus, in order to implement the procedure, we must supply a way of choosing the rule to apply whenever this ambiguity occurs. Unfortunately, not all choices are equally good and some of them might lead to a dead end or to non-terminating generation. A simple remedy is to impose a size limit on generated terms and allow the procedure to *backtrack* and choose another option whenever it goes astray.

Once the size limit and backtracking are in place, we can choose any strategy for selecting the rule to apply without compromising the strength of the generator, since it will be able to backtrack from any bad choice. Therefore, it is reasonable to adopt a very simple strategy of choosing the rules at random.

Although this simple approach is capable of generating every well-typed term smaller than a given size limit, it has a serious shortcoming. Whenever the (App) rule is used, the type of the argument is neither determined by the target type nor the environment, and thus any type can appear there. This yields too many possibilities even if the size of types could be restricted.

A bad choice at this point can be a serious problem, as often *only a very specific choice of types will allow the search to progress.* For example, suppose that the target type is `Int` and the constant $\texttt{f} : \texttt{String} \to \texttt{Bool} \to \texttt{Int}$ is available. Constant `f` might be used to construct the required term by applying it to two arguments, of types `String` and `Bool` respectively. However, for that to happen, the (App) rule must be invoked twice, and the argument types guessed to be `Bool` and `String` respectively at the two invocations. Unfortunately, guessing these types correctly is very unlikely to occur, and on average, a large number of backtracking steps will be needed. This makes the generation of terms very inefficient, and moreover, hard-to-guess terms (like this one) will occur very rarely in the results of the generator.

3.2 Refined Approach

In order to address the problem of guessing argument types, we introduce another typing rule (inspired by the proof synthesis method from [18]).

$$(\textsc{Indir})\ \frac{f : \ldots, \Gamma \vdash M_1 : \sigma_1 \ \cdots\ f : \ldots, \Gamma \vdash M_n : \sigma_n}{f : \sigma_1 \to \ldots \to \sigma_n \to \tau, \Gamma \vdash f\ M_1 \ldots M_n : \tau}$$

This rule is logically unnecessary, as it follows from the other typing rules, but as a *generation* rule it is far superior to the (App) rule. It creates a term that is a variable (or constant) from the environment applied to a number of recursively generated argument terms. What is important here is that even though the resulting term contains a number of term applications, no types have to be guessed as they are determined by the type of the function. Since no guessing is involved, the search is no longer so erratic and generation is much more efficient. However, if we were to *replace* the (App) rule by this one, then we would no longer be able to generate all λ-terms—in particular, terms such as $(\lambda x : \texttt{Int}.x)\ \texttt{1}$, in which a λ-expression is applied directly, could no longer be generated. Therefore, we choose to keep both rules just to make it possible to generate λ-expressions that are directly applied to arguments.

4. POLYMORPHISM

Parametric polymorphism refers to the use of the same code at several different types. First introduced by Damas and Milner in ML [4], it has become a standard feature of functional languages such as Miranda and Haskell, and is now also found in mainstream languages via, for example, Java generics [19]. Opportunities for polymorphism arise when the same underlying term can be typed in several different ways. For instance, the identity function $\lambda x : ?.x$, which takes an argument and returns it, can be assigned either of the types `Int` $\to$ `Int` or `Bool` $\to$ `Bool` choice of $?$. Rather than use question marks as types, we introduce *type variables* $\alpha, \beta, \ldots$, and say that $\lambda x : \alpha.x$ has the type $\alpha \to \alpha$ for any type α, where α might take the value `Int`, `Bool`, or any other type. To express the fact that α can be arbitrarily chosen, we assign to the identity function the type $\forall\alpha.\ \alpha \to \alpha$, a *polymorphic type* where $\forall\alpha.$ indicates the changing type variable.

Polymorphism is heavily used in conjunction with *parametrised datatypes.* For example, let `List` α be the type of lists with elements of type α. Two useful functions on lists are `head`, which returns the first element of the list, and `tail`, which returns all the elements except for the first one. The types of these functions are naturally polymorphic:

$$\texttt{head} : \forall\alpha.\texttt{List}\ \alpha \to \alpha \qquad\qquad \texttt{tail} : \forall\alpha.\texttt{List}\ \alpha \to \texttt{List}\ \alpha$$

When these functions are used, then α may be instantiated to any type, allowing the same code to manipulate lists with any type of element.

Variables	$x, y, \ldots$	
Constants	$\mathbf{c}, \mathbf{d}, \ldots$	$::=$ `head`, `tail`, `+`, `0`, `1`, ...
Types	$\sigma, \tau, \ldots$	$::=$ `Int` $\mid$ `Bool` $\mid \sigma \to \tau$ $\mid \alpha \mid$ `List` α
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 3: A simple λ-calculus with polymorphism

Figure 3 shows the formal syntax for a simple polymorphic λ-calculus. The syntax for terms is exactly the same as for simply-typed λ-calculus (see Figure 1). The only changes are to the syntax for types, where we introduce type variables (α) and polymorphic lists (`List` α).

4.1 Generation of Terms

The approach described in Section 3 generates terms with *monomorphic* types such as `Int`, `Bool`, and `Int` $\to$ `Int`, i.e. types that do not involve type variables. In this paper, we only consider generating terms with monomorphic types. However, we allow the use of polymorphic *constants*, such as `head` and `tail`, in the terms we generate. This allows us to generate programs that make use of Haskell's list-processing library functions.

Generation of random terms using polymorphic constants introduces the problem of *instantiating* a polymorphic type to a monomorphic one when the constant is used. In some cases, instantiating a polymorphic type is straightforward. For example, suppose that we want to generate a term of type `List Int`, and we have available an environment Γ containing the constants `lst` : `List Int`, and `tail` : $\forall\alpha.$`List` $\alpha \to$ `List` α. Clearly, if we choose to generate a call of `tail` with type `List Int`, then we must instantiate α to `Int`, and generate an argument which also has the type `List Int`. We could now choose `lst` : `List Int` to be the argument, and generate the term `tail lst`.

However, suppose the environment contains the constants

$$\begin{array}{ll} \texttt{map}: \ \forall\alpha\beta.(\alpha \to \beta) \to \texttt{List}\ \alpha \to \texttt{List}\ \beta & \\ \texttt{lst}: \texttt{List Int} & \texttt{lst2}: \texttt{List Bool} \end{array}$$

where `map` $f\ l$ returns the list of type `List` β obtained by applying the function f, of type $\alpha \to \beta$, to every element of the list l, of type `List` α. If we choose to generate a term of type `List Int` which is a call of `map`, then as subgoals we must generate an f of type $\alpha \to$ `Int` and an l of type `List` α—*but α is not determined, and can be chosen freely.* This leads to similar problems to those described in Section 3.1 with the rule (App). In this particular example, α can be chosen to be at least `Int` or `Bool`, since we have a list of integers (`lst` : `Int`) and booleans (`lst2` : `Bool`) in the context, but in general the choice of α is difficult.

In fact, the problem of instantiating polymorphic functions is a generalisation of the problem of instantiating the (App) rule. To see this, it is enough to make rule (App) redundant by introducing a polymorphic constant

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

that, when used in the (Indir) rule, generates the same subterms as rule (App).

We adopted a simple solution to this problem, which is crude, but reasonably effective. Instead of instantiating undetermined type variables with any possible type, we use a method for randomly generating types that avoids those for which it is impossible to construct a term. Firstly, a set of types is constructed, which initially consists of the types from the environment. Then, further types are added to the set using the rule that if both a functional type and its argument type are present, then we can also add the type of the function's result. This set is then used to generate random types either by selecting them directly or by creating a functional type based on them when instantiating. Since polymorphic types can also be present in the environment, this procedure sometimes involves specialising them.

5. GENERATION ALGORITHM

We describe the generation algorithm informally to avoid overwhelming formal notation. It works by applying generation rules, which build successive parts of the generated terms, and by backtracking when generation reaches a dead end. The basic structure of the algorithm is as follows:

- A list of generation rules that are applicable is constructed based on the target type and the symbols contained in the environment. If one rule can be applied in several different ways, for instance, with different symbols from the environment, then it is represented several times in the list.
- The list of rule applications is then randomly shuffled and the first one from the randomised list is applied.
- The generation procedure is invoked recursively for the premises of the selected rule.
- If all the recursive calls finish with success, all required subterms are available and the term is constructed according to the rule.
- If generating any of the premises fails, then the generator backtracks and the next rule from the list is selected instead.
- If all rules have been tried without success, then the generation fails.

Thus, the generator first tries to exhaust all possibilities for constructing subterms required by a generation rule, before deciding to select another one—the search is depth-first.

An awkward special case arises when applying the (Indir) rule to a function whose result type is just a type variable—for example, the identity function `id`$:\forall\alpha.\alpha \to \alpha$. By instantiating α to a function type with n arguments, we can apply (Indir) to generate a call of `id` with $n+1$ arguments, for any n, and whatever the target type! For instance, if the target type is `Int`, then we can instantiate α to `String` $\to$ `Bool` $\to$ `Int` and generate a call `id` $F\ S\ B$, where subgoal F must have type `String` $\to$ `Bool` $\to$ `Int`, S must have type `String`, and B must have type `Bool`. There are thus infinitely many ways to apply the (Indir) rule to such a function. To remove the possibility of infinite backtracking we only allow the (Indir) rule to consider at most three extra parameters. This trade-off prevents the (Indir) rule from generating some terms, however they can still be generated using the (App) rule.

5.1 Optimisations

Sometimes we reduce backtracking by omitting rule applications which we know will fail. For example, when the (Indir) rule using one symbol fails, we can conclude that it will also fail with any other symbol of the same type, as the premises for the rule application are the same. Therefore, the generation algorithm considers one version of the (Indir) rule for each unique type present in the environment, and the exact symbol is chosen at random afterwards.

We also prioritise rule applications to speed up generation and reduce backtracking. Rules with higher weights have a higher chance of being selected before others. We use weights to first try rules that have a high chance of success, using other rules only occasionally, or when the preferred ones fail. In particular, applying rules which involve guessing types usually has a low probability of success—but is necessary sometimes to produce certain terms.

Even rule prioritisation is not always enough to prevent massive backtracking, we also limit the number of 'dangerous' rules—the ones involving guessing types—that can be applied recursively. Undoubtedly, this rules out the generation of some complex terms, but we still generate enough interesting terms to find compiler failures.

5.2 Distribution

The distribution of generated terms is ad-hoc, but produces an acceptable rate of terms that trigger failures—we have tweaked it to achieve good results in our own testing. Weights assigned to rules are respectively 4 for rules that use locally-bound variables, 2 for rules that introduce constants, 8 for the rules for introducing an application or a λ-expression, and 6 for the `seq` rule.

It is unclear what the "best" distribution would be. There is no reason to believe, for example, that a uniform distribution over terms of a specific size would be *more* effective at revealing bugs, and it might well be less so. In any case, we do not have the goal to approximate "programs that could be written by real programmers", because how well the generated terms correspond to this notion is hard to determine. So, we are pragmatic, and consider that success in finding bugs is the most important measure of a good distribution.

6. TESTING GHC

We use the generation of random λ-terms described in Section 3 and 4 to test the Glasgow Haskell Compiler [15] (GHC). Haskell is a purely functional programming language with *lazy evaluation*—expressions are by default compiled as *closures*, and their evaluation delayed until the value is actually required. GHC is the most popular and complex Haskell compiler—its main part consists of approximately 120,000 lines of code. Part of its complexity comes from an elaborate code optimiser. The optimiser transforms code in many stages, one of which is the *strictness analyser* [12], which identifies expressions whose value is *always* required eventually, and which can therefore be compiled for immediate evaluation (avoiding the costly closure mechanism) without changing the semantics of the program. Since Haskell is purely functional, changing the *order* of evaluation in this way does not change the program semantics.

Determining whether code has been compiled by the compiler correctly is, of course, difficult since the semantics of Haskell is complex. However, one way to check the correctness of optimised code is to compare it to the unoptimised version. Optimisations should only make programs more efficient without changing their semantics. Establishing program equivalence cannot be done automatically in general, since the input domain of many programs is infinite, but we can demonstrate that programs are *not* equivalent just by finding one random input on which their output differs. More formally, it should be the case that $\forall x \in \mathbf{dom}\ f : [\![f]\!]_{\mathbf{ghc\ \text{-}O1}}(x) = [\![f]\!]_{\mathbf{ghc}}(x)$ where f is a Haskell program, $[\![f]\!]_{\mathbf{ghc\ \text{-}O1}}$ and $[\![f]\!]_{\mathbf{ghc}}$ represents the optimised and unoptimised compiled version of f, respectively. If this equation does not hold for a program f and an input x, then optimisations are changing the semantics of the program. The techniques described in Section 3 and 4 can provide us with several f and x where the equation above does not hold, thus showing that GHC is buggy. Although Sections 3 and 4 present techniques to generate random programs in a very simple setting, i.e. simple-typed λ-calculus, our approach detects failures in such a complex piece of code as GHC in just a few minutes.

6.1 Correctness of the Strictness Analyser

To evaluate the correctness of the strictness analyser, we decided to test compilation of functions that operate on lazy data structures, i.e. data structures with elements or components which are computed on demand. In particular, we focus on testing Haskell programs manipulating lazy lists. Such lists can contain "undefined" components, represented as closures of expressions, that raise an exception when evaluated. A function operating on lists can receive such a partially-defined list as an argument, and still yield a result, as long as it does not inspect the parts of the list that are undefined. For example, in Haskell, a function that returns the second element of a list can be successfully applied to lists of length two even if the first element is undefined.

We automatically generate random lambda terms of type `List Int` $\to$ `List Int` to test GHC's strictness analyser. Programs are compiled with GHC's optimisations turned on and off, respectively. Both compiled versions are then run with a number of simple partially-defined lists as input in order to compare their outputs. Since the results of the compiled functions are also lazy lists, instead of just yielding a result or failing completely, most commonly the functions yield a partially-defined result. Clearly, if two functions are equivalent, they should yield the same partially-defined result when applied to the same partially-defined argument.

To compare partially-defined lists, we traverse each list from left to right, printing its value on the output. We then compare lists by comparing the generated output. For example, a list containing the numbers from 0 to 4 is printed as `[0,1,2,3,4]`. Square brackets and commas are the Haskell notation for lists. In contrast, if we encounter an undefined value, then the list is printed as `[0,1,2,3,*** Exception:`, where `*** Exception:` is appended by an exception handler. Note that this approach does not guarantee to distinguish different partially-defined lists, but it works well enough for our purposes, even though more accurate methods are available [6].

6.2 Generating Random Haskell Functions

To generate random functions on lists, we gave our generation algorithm an initial environment containing list operations, functions from the standard `Prelude` module (e.g. `(+)`, `(-)`, `(&&)`, `(||)`, `map`, `length`, and `filter`) and constant values (e.g. `0`, `1`, `True`, and `False`). Having such a rich initial environment increases the possibility of generating interesting terms.

Haskell provides programmers with a way to control lazy evaluation, via the built-in operation `seq` which *forces* the evaluation of terms. This operation takes two arguments and forces the evaluation of the first one before returning the second. We wanted to test whether GHC's strictness analyser accounts for this behaviour correctly.

The type of `seq` is $\forall \alpha \beta . \alpha \rightarrow \beta \rightarrow \beta$. Observe that the type of the first argument is completely unrelated to the second one. We could just include function `seq` in the initial environment like any other function, but because its type is so general, doing so leads to overuse of `seq` in the generated terms—a call of `seq` can be inserted *anywhere*, with any value as first argument. We therefore used a custom generation rule that restricts the first argument of `seq` to be a local variable.

6.3 Testing Environment

Testing was performed on GHC version 6.12.1 configured for x86-64 systems, running on a modern laptop[1]. Because starting the compiler is quite costly, at around 0.5 s, we placed 1,000 generated functions in each module, thus amortising this cost across a large number of tests. The choice of the number 1,000 roughly balances compilation time with the time spent on test case generation, yielding a total time of around 20 s for generation, compilation and testing of 1,000 terms. Using a larger number of terms in a module would not improve performance considerably.

The generated modules invoke the generated functions on about 20 simple partially-defined lists, and print the results. The small set of test data is sufficient, as even two partially-defined lists are enough to uncover most compiler failures that were found. Each generated module was compiled with the default optimisation level (compiler flag -O1) and with no optimisation (compiler flag -O0), the compiled code was executed, and the outputs were compared.

6.4 Results

We limited the size of generated functions to a maximum of 70 function applications or λ-expressions per generated term. After generating and testing around 20,000 random terms of type `List Int` $\rightarrow$ `List Int`, which took around 15 minutes, a failure in GHC's strictness analyser was found. The random functions exposing failures in GHC consisted of terms of size between 30 and 50. These functions were then automatically simplified by a procedure known as *shrinking* (see next section), roughly halving their sizes. After this simplification phase, the functions that triggered the compiler failure are the following ones[2] (in Haskell syntax):

```
seq (id (\a -> seq a id) (undefined::Int))

seq (seq (\a -> length) (\a -> seq a seq)
                         (head ([]::[] Bool)))

seq (seq (\a -> null) (\a -> seq a (\b -> length))
                       (head ([]::[] Bool)))
```

(Note the first parameter of `seq` is not always a variable in these examples—this is a result of the shrinking process).

We illustrate the failure using the first test case above. The expected behaviour of that function is to raise an exception immediately, on application to any argument. The term should be equivalent to `seq (undefined :: Int)`, which forces the evaluation of an undefined value before proceeding with any other computation. However, this behaviour is not reflected by the code compiled with optimisations.

Argument	-O0	-O1
`[undefined]`	`*** Exception:`	`[*** Exception:`
`[1,undefined]`	`*** Exception:`	`[1,*** Exception:`
`[2,1,undefined]`	`*** Exception:`	`[2,1,*** Exception:`

Table 1: Outputs of a bug-triggering function

We can see in Table 1 that the function behaves as expected when compiled without optimisations (column `-O0`). However, if the same function is compiled with optimisations (column `-O1`), the function returns a partially-defined list instead of directly raising an exception. In fact, the function returns a list which is equal to its argument (column *Argument*).

6.5 Shrinking

The randomly generated functions that provoke failures are typically too big to be comfortably read and understood. Clearly, it is desirable to find smaller functions that show the presence of bugs. Fortunately, it is likely that not all parts of a randomly generated function are needed to reveal a bug, and that smaller, similar terms provoke the same bugs. With this in mind, we establish a phase of *shrinking* [3] for failing test cases. This phase consists of creating a number of smaller variants of bug-triggering functions. These variants are then tested to determine if they trigger the same failure. If that is the case, we repeat the shrinking process to search for an even smaller term that results in a failure. Shrinking finishes when a minimal test case is found that provokes a failure, which might (or not) be caused by the original bug.

In this case, shrinking is done using three simple rules. Firstly, a sub-term can be replaced by any of its sub-terms as long as variable bindings and its type are preserved. Secondly, a subterm, that is not a constant, can be replaced by any constant of the same type. And thirdly, an application of a λ-expression $(\lambda x.M)\ N$ can be replaced by the body M, with x replaced by the actual parameter N—a β-reduction. In simply-typed λ-calculus, β-reductions cannot continue indefinitely, so the shrinking process always terminates.

7. RELATED WORK

Random testing used for finding bugs in compilers and programming language tools has received some attention in recent years. Lindig [10] created a tool for testing the C function calling convention of the GCC compiler. This tool randomly generates only the *types* of functions; their bodies just checked that the parameters were received correctly.

Wrangler, a refactoring tool for Erlang has also been tested using random program generation [7]. 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. [5] exhaustively generate Java programs (up to certain size) in order to test the refactoring engines in Eclipse and NetBeans. Different from our approach, some of the generates programs are not valid inputs for the Java compiler.

Probably the work related most closely to ours is Klein et al. [9], who generated random programs to test an object-oriented library, finding a large number of bugs. Their generator is capable of producing higher-order object-oriented programs (which override methods) and supports monitoring of pre- and post-conditions. Their generation method

[1]The machine had a 2.4 GHz Intel Core 2 CPU and 4 GB of RAM.

[2]Readers can refer to `http://www.cse.chalmers.se/~palka/testingcompiler/` for further details

uses generation rules similar to ours, and backtracks occasionally just as ours does. 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 which 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. In a sense, we generate terms top down, while they generate them bottom up. 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.

Vytiniotis and Kennedy [16] present encoding of datatypes into streams of bits, which can be used for their random generation. In their approach to generate simply-typed λ-terms, the target type is *never* fixed, and thus the generation never fails, eliminating the need for backtracking.

The λ-term *enumerator* developed by Yakushev and Jeuring [14] 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.

Djinn [1] 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 [8].

Statistical properties of random untyped λ-terms have been explored in [2], which also explores a method of generating them using Boltzmann sampling. Generation of random untyped λ-terms is tackled in [17], which employs counting of possible subterms to achieve uniform generation distribution. Correspondingly, the work in [11] examines the proportion of simple types that are inhabited, that is, for which it is possible to create a term of that type.

8. CONCLUSIONS

Generating random and type correct programs for compiler testing is quite a difficult problem, because type correctness is a *global* property which must be achieved by a sequence of *local* choices. It is easy for a random generator to make a bad choice early on, painting itself into a corner in which generation cannot be completed at all, or can be completed only by generating very trivial programs (in which, for example, variables are defined but almost never used). We have presented a workable approach, in a simple setting—the simply-typed λ-calculus. In contrast to earlier work, we have considered type polymorphism, and shown that it introduces further complications for the generator.

We show the *value* of our approach by applying testing the optimiser of GHC finding surprising optimiser failures in a few minutes on an ordinary laptop. Moreover, the generator can be easily adapted to test other compilers by adding a term-printing function producing the syntax of the programming language and providing a suitable initial environment.

9. REFERENCES

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

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

[3] 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.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1982.

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

[6] N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In *MPC*, 2004.

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

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

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

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

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

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

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

[14] 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.

[15] The GHC Team. The Glasgow Haskell Compiler. Software release. `http://haskell.org/ghc/`.

[16] 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.

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

[18] M. Zaionc. Mechanical procedure for proof construction via closed terms in typed λ calculus. *J. Autom. Reason.*, 4:173–190, June 1988.

[19] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial: A Short Course on the Basics, 4th Edition*. Prentice Hall PTR, 2006.