ATLANTIS PRESS

Advances in Computer Science Research, volume 91

International Conference on Modeling, Simulation and Big Data Analysis (MSBDA 2019)

Haskell Compiler Testing Automation Based on Equivalence-Modulo-Inputs Method

Tianchi Li*

School of Software, Tsinghua University, KLISS, Beijing, China

*Corresponding author

Keywords: Software Engineering, Testing Automation, Compiler Testing, Haskell, Type system.

Abstract. Equivalence Modulo Inputs (EMI) has become an effective approach for automatically testing compilers. It has revealed thousands of bugs of C compilers GCC and LLVM. Haskell is a purely functional programming language with static type checking, which satisfies the emerging need of parallel computing for big data on multicore processors. GHC, the mainstream compiler for Haskell, inevitably suffers from bugs. When applying EMI to test GHC, two challenges are coverage definition and typing correctness preservation. This paper introduces an approach to generate Haskell programs as EMI test input of GHC by solving the two Haskell specified challenges.

Introduction

The correctness of compiler is a critical topic in software quality assurance. The bugs hidden in compiler may potentially affect all programs compiled by the compiler. Meanwhile compiler bugs are much harder to recognize. Various approaches have been developed to verify compilers, including formal verification and testing. The representative achievement of formal verification approach is *CompCert*, a compiler with its semantic preservation theorem proved by Coq proof assistant [1]. As for the testing approach, *RDT* (Randomized Differential Testing) together with its variant *DOL* (Different Optimization Levels) and *EMI* (Equivalence Modulo Inputs) have become 3 major classes of compiler testing [2]. EMI project has been continuously revealed thousands of bugs of C compilers [3]. By May 1st, 2019, 1602 bugs of GCC or LLVM have been found by EMI approach and 1007 of them have been confirmed as real bugs and fixed.

The programming language Haskell is widely known for its purely functional programming paradigm support and its flexible type system. The static type checking makes Haskell a safe language in the sense that many bugs can be filtered out as typing error at compile-time. The dominantly mainstream compiler of Haskell is GHC [4]. As a complex software system, GHC inevitably suffer from bugs. By May 1st, 2019, the issue tracker of GHC contains 11984 bug report.

A portion of bugs of Haskell compiler can be triggered by evaluating (executing) an expression of Haskell. An example is the bug reported by GHC issue #5625 [5]. Let expression `a = \x -> seq undefined (+1)`. When printing expression `(a 'seq' a [] 'seq' id) [0]`, the expected behavior is crashing for exception `Prelude.undefined`. When tested in GHC version 7.3.20111022 with compiler optimization flag is set to `-O`, the program behaves as expected. However, if no optimization is set (flag `-O0`), the `[0]` is printed, which violates the expectation. GHC developers confirmed this as a bug (less strict to argument than expected). After fixing the bug, this bug-triggering expression was included in the regression test suite of GHC for the further development of GHC.

The example illustrates a typical class of Haskell compiler bugs, which can be triggered by evaluation an expression. In this paper, we designed an approach to automatically generate such expressions used as black-box test inputs of Haskell compiler, which may reveal potential bugs. EMI provides a methodology of program generation. However, when applying to Haskell compiler, two challenges arise. One is to define expression style coverage for Haskell's functional program, which differs from the statement level coverage of imperative programs in C. Another is to preserve the typing correctness of the whole expression when inserting, deleting and updating sub-expressions the rest of the paper describe the treatment of these two challenges.

Related Work

Differential Testing for Compilers

In software testing, test-oracle problem is the challenge of determining the expected correct behavior or output of the tested software, when the input is given.

RDT (Randomized Differential Testing, or simply Differential Testing) [2] is a widely-used testing technique, which addresses test-oracle problem. When specialized for compiler testing, RDT compares two or more compilers that implement the same specification (e.g. C11 standard for C). The compilers should produce the executables with the same behavior under the same source code input. RDT for compiler testing has a special variant, DOL (Different Optimization Levels) [2]. DOL method compares compilation results under different optimization levels (e.g. `-O0, -O1, -Os, -O2` and `-O3` in GCC) of one compiler. When a test program is respectively compiled under different optimization levels and executed under the same set of test inputs, it may produce different results, thus we know that the compiler under test contains bugs.

EMI

EMI (Equivalence Modulo Inputs) is a methodology for compiler testing, introduced by Vu Le, Mehrdad Afshari, and Zhendong Su in 2014 [3]. The core idea of EMI is to mutate the source program part which is not covered under a given input, then test whether the program behavior changes under the same input.

EMI method can be formalized as follows: Let $Comp$ be the compiler under test, P and Q be two source programs while P and Q accept the same type of input, denoted as T. Let i be an input of type T. And compiling and executing process can be formalized as function application: Let $Comp(P)$ be the executable program compiled by Comp from source program P, and Comp(P)(i) be the output of Comp(P) given the input i. If $\mathrm{Comp(P)(i)} = \mathrm{Comp(Q)(i)}$, define P and Q are *equivalent modulo* i. P and Q are called *EMI variant* of each other. The insight of EMI method is to mutate the program part of P which is not covered when Comp(P) is executed given input i. The mutant, denoted as P′, should be equivalent to P modulo i. If P and P′ is not equivalent modulo i, then Comp is judged to contain bug.

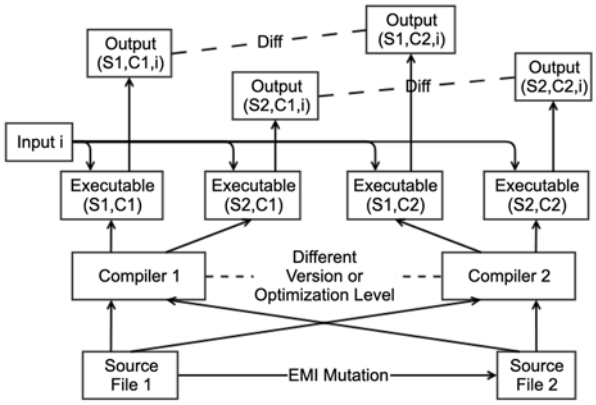

Figure 1. The framework of DT-EMI composed approach for Haskell compiler testing

Type Syntax:

$$T ::= B \quad \text{(base type)}$$
$$\mid (T_1 \to T_2) \quad \text{(function type)}$$

Term Syntax:

$$t ::= x \quad \text{(variable)}$$
$$\mid (t_1\ t_2) \quad (\lambda\text{-application})$$
$$\mid (\lambda x : T.t) \quad (\lambda\text{-abstraction})$$

Figure 2. The syntax of types and terms of simply-typed lambda calculus

Haskell Specified EMI Method

Differential testing and EMI method can be combined to detect compiler bugs. Figure 1 shows the framework of our composed approach to test Haskell compiler. In the framework, EMI mutation is the key and challenging step and the solutions are described in this section.

The existing implementations based on EMI method focused on imperative language, especially C. When implementing EMI method for Haskell, one challenge is to define expression style coverage for Haskell's functional program, which differs from the statement level coverage of imperative programs in C. Another challenge is to preserve the typing correctness of the whole expression when inserting, deleting and updating sub-expressions.

Language Definition

Haskell has rich language features, including those specified by language standard (Haskell 2010 report) and those implemented exclusively by GHC extensions. This makes Haskell hard to be handled entirely. In this paper, we select a subset of Haskell language as the mutation source and target. The selected language is the Haskell subset which can be desugared to simply-typed lambda calculus [6]. The BNF-style grammar in figure 2 shows the formation rules of types and terms (expressions) of simply-typed lambda calculus. Unlike type theory, in Haskell, the base types are from a given finite set, e.g. {`Bool, Int`}.

Coverage Definition

EMI mutation begins at measuring the coverage of program given a input. In this paper, *coverage* denotes the information of which part of the program is covered (executed) and which part is not, and *coverage rate* denotes the percentage of covered part among the program.

A Haskell program can be treated as a lambda expression of type `IO ()`, with an environment of bindings of lambda expressions to named variables. Therefore, unlike C programs, Haskell programs has no statement concept, and there are only (usually nested) lambda application of expressions. While the coverage of C programs is defined on statement level, we here define the coverage of Haskell programs on expression level.

Typing Correctness

Typing judgements and typing rules are common techniques in the study of lambda calculus and type theory [6]. In the Haskell specified EMI, typing rules are utilized to generate well-typed terms of given type.

A *typing context* Γ is a set of mapping of the form $x: T$, expanded as $\{x_1: T_1, \ldots, x_n: T_n\}$, indicating the variable x_i is assigned to a type T_i. Without loss of generality, in this paper, if a typing context Γ occurs, it is assumed to be *consistent*. Γ is consistent if there is no x, T_1, T_2 such that $x: T_1 \in, x: T_2 \in$, and $T_1 \neq T_2$. A *typing judgement* is a three-place relation $\Gamma \vdash t: T$ read as "under the assumptions in Γ, the term t has the type T". Γ is called *antecedent* of the judgement, and $t: T$ is called *succedent*. Typing judgement $\Gamma \vdash t: T$ is valid if there is a proof for $\Gamma \vdash t: T$ by applying *typing rules*.

Simply-typed lambda calculus contains three *typing rules*:

$$\frac{x: T \in \Gamma}{\Gamma \vdash x: T}(\mathrm{T-Var}) \quad \frac{\Gamma, x: T \vdash t: T'}{\Gamma \vdash (\lambda x: T.\, t): T \to T'}(\mathrm{T-Abs}) \quad \frac{\Gamma \vdash t_1: T \to T' \quad \Gamma \vdash t_2: T}{\Gamma \vdash (t_1 t_2): T'}(\mathrm{T-App})$$

Each typing rule consists of two parts. One is a type judgement below a horizontal line, called the *conclusion* of the rule. Another is some (maybe zero) type judgements above the horizontal line, called the *premises* of the rule.

The rule (T-Abs) formalizes typing of lambda abstraction: If $t: T'$ is provable under assumption $\Gamma, x: T$, then $(\lambda x: T.\, t): T \to T'$ is provable under a weaker assumption, Γ. Rule (T-App) formalizes typing of lambda application: If $t_1: T \to T'$ and $t_2: T$ are both provable under assumption Γ, then $(t_1 t_2): T'$ is provable under the same assumption Γ. Rule (T-Var) formalizes the base case: If $x: T$ is in the assumption Γ, then $x: T$ is provable under assumption Γ.

Algorithm 1 describes the generation procedure of a lambda term. In the input arguments, Γ stands for the available primitive terms of basic types; Tar stands for the target type, of which the generated term should be; Rules stand for the typing rules (in this specific calculus, Rules={(T-App), (T-Abs), (T-Var)}); $M \in N$ is the threshold for typing rules instantiation, since a typing rule may have infinite instances.

Line 1 of *Algorithm 1* get the instances of all available typing rules. Instantiation means replacing the metalanguage variables (e.g. T) denoting types in the typing rule with concrete type, e.g. Int. In line 11, subterm-needed retrieves from rule instance RI all the term occurring in the succedent of premises of RI. For example, if RI is an instance of (T-App) and the conclusion is $(t_1 t_2)$, subterm-needed will retrieve t_1 and t_2 because they occur in the two premises of (T-App) respectively.

Algorithm 1 Gen: Generating lambda term of given type

Input: Γ, Tar, Rules, Size, M
Output: out

```
 1: RIs = inst(Rules, Γ, Tar) // get rule instances
 2: result = ∅
 3: if (length(RIs) = 0) then
 4:   return result
 5: end if
 6: if (length(RIs) = ∞) then
 7:   RIs = select-subset(M, RIs)
 8: end if
 9: RIs = shuffle(RIs)
10: for (RI in RIs) do
11:   p = length(subterm-needed(RI))
12:   if (p=0) then
13:     return term-in-conclusion(RI)
14:   else
15:     sub-result = ∅
16:     for t in subterm-needed(RIs) do
17:       sub-result += Gen(Γ, type-of(t), Rules, M) // recursively generate subterms
18:     end for
19:     result += cartesian-product(sub-result)
20:   end if
21: end for
22: return result
```

Implementation

As described in the algorithm design section, Haskell specified EMI consist of two major components. One is measuring the coverage of Haskell program under a set of input, and another is manipulating the Haskell program according to coverage information. Figure 3 shows the implementation of the two major components and the supporting components. The coverage measurement is implemented by HPC, the program coverage utility released with GHC. The Haskell program manipulation is implemented by customizing refactoring in the framework of Haskell-tools.

Coverage Measurement Based on HPC Library

Andy Gill from Galois Inc. and Colin Runciman from University of York introduced HPC (Haskell Program Coverage) in 2007 [7]. HPC can instrument Haskell programs, thus measuring the coverage information of the program under a set of input and visualize the coverage in various ways like program highlighting and statistical chart. Figure 4 is an interface of the hpc-markup in HPC. It visualizes the program coverage. Yellow highlight for expressions never evaluated; green for expressions always evaluated `True`; red for expressions always evaluated `False`.

We utilized HPC to extract coverage of Haskell programs under a set of inputs. Coverage information acts as input of the mutation step. Our method is to analyze the HPC generated *tix* and *mix* file. For example, after the instrument step of HPC, Haskell expression `addInt x y` is turned into *tix* file with content `tick 0 (tick 1 ((tick 2 addInt) (tick 3 x)) (tick 4 y))`. After the executing step, a *mix* file `Tix [ TixModule "TestModule" 2454134535 5 [1,0,1,1,1]]` is generated. Analyzing the .mix file.

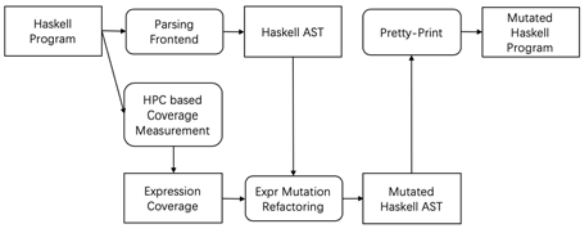

Figure 3. The implementation framework of Haskell specified EMI method.

```
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
             | otherwise = error
                 "attempting to compute reciprocal of number <= 1"
  where
  (digits, recur) = divide n 1 []
divide :: Int -> Int -> [Int] -> (String, Int)
divide n c cs | c `elem` cs = ([], position c cs)
              | r == 0      = (show q, 0)
              | r /= 0      = (show q ++ digits, recur)
  where
  (q, r) = (c*10) `quotRem` n
  (digits, recur) = divide n r (c:cs)

position :: Int -> [Int] -> Int
position n (x:xs) | n==x      = 1
                  | otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n =
  "1/" ++ show n ++ " = " ++
  if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
  where
  p = length d - r
  (d, r) = reciprocal n

main = do
  number <- readLn
  putStrLn (showRecip number)
  main
```

Figure 4. Interface of hpc-markup in HPC. Coverage is marked with highlight. [7]

Program Manipulation Based on Haskell-tools Library

Programs are highly structural data. Program manipulating needs infrastructures like parsers and AST transformers. In the Haskell specified EMI method, the haskell-tools library is utilized to manipulate Haskell program. Haskell-tools is a toolkit bringing convenience for developing Haskell programs, developed by Boldizsár Németh [8]. It can *refactor* Haskell programs. Haskell-tools has several official built-in *refactorings*, including renaming definition, extracting binding etc. Haskell-tools is scalable by providing users a set of APIs to customize refactorings. We customize several refactoring configurations to implement the sub-expression replacement algorithm of Haskell expressions with new expression generated by *Algorithm 1*, thus completing EMI mutation specified for Haskell program.

Result

Our approach got some preliminary results. One successfully generated Haskell expression is `\x -> zipWith (==) x undefined`. This is a function of type `Eq a => [a] -> [Bool]`. When this function is applied to empty string argument, it triggers the bug of GHC version 7.10.0.20141227 as reported by GHC issue #9949 [9]. Our EMI implementation generated this expression from `\x -> zipWith (==) x []`. When argument `x` is an empty list, the third argument of `zipWith`, i.e. `[]`, is not covered because the result can be determined without evaluating the third argument. Then `[]` is replaced by many type-compatible expressions generated by algorithm 1. The `undefined` belongs to Γ. In this way, once `[]` was replaced by `undefined` and the result is a bug-triggering expression.

Conclusion and Future Work

In this paper, we proposed an approach to adapt EMI compiler testing technique to Haskell compiler GHC, discussed the challenges of coverage definition and typing correctness preserving, gave solutions to the two challenges. A preliminary goal have been reached that some bug-triggering Haskell expressions for old version of GHC have been generated by our tool.

Many successive works can be done based on this paper. One may support richer type system than simply-typed lambda calculus, covering more functionality of the Haskell compiler. Continuous generation of expressions can be used as stress testing for GHC. The GHC independent part of our work may be used to test other Haskell compilers like LHC (Haskell compiler with LLVM backend).

References

[1] X. Leroy et al., "The compcert verified compiler," Documentation and users manual. INRIA Paris-Rocquencourt, vol. 53, 2012.

[2] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016, pp. 180–190.

[3] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in ACM SIGPLAN Notices, vol. 49, no. 6. ACM, 2014, pp. 216–226.

[4] S. P. Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler, "The glasgow haskell compiler: a technical overview," in Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference, vol. 93, 1993.

[5] https://gitlab.haskell.org/ghc/ghc/issues/5625, accessed: 2019-05-01.

[6] S. Thompson, Type theory and functional programming. Addison Wesley, 1991.

[7] A. Gill and C. Runciman, "Haskell program coverage," in Proceedings of the ACM SIGPLAN workshop on Haskell workshop. ACM, 2007, pp. 1–12.

[8] https://github.com/haskell-tools/haskell-tools, accessed: 2019-05-01.

[9] https://gitlab.haskell.org/ghc/ghc/issues/9949, accessed: 2019-05-01.