Verifying Compiler Optimisations
(Invited Paper)

Ian J. Hayes(✉), Mark Utting, and Brae J. Webb

The University of Queensland, Brisbane, Australia
{Ian.Hayes,M.Utting,B.Webb}@uq.edu.au

Abstract. Compilers are a vital tool but errors in a compiler can lead to errors in the myriad of programs it compiles. Our research focuses on verifying the optimisation phase because it is a common source of errors within compilers. In programming language semantics, expressions (or terms) are represented by abstract syntax trees, and their semantics is expressed over their (recursive) structure. Optimisations can then be represented by conditional term rewriting rules. The correctness of these rules is verified in Isabelle/HOL. In the GraalVM compiler, the intermediate representation is a sea-of-nodes graph structure that combines data flow and control flow in the one graph. The data flow sub-graphs correspond to term graphs, and the term rewriting rules apply equally to this representation.

1 Introduction

This paper overviews our research on verifying expression optimisations used in the GraalVM compiler developed by Oracle.[1] The compiler supports multiple source languages (Java, Scala, Kotlin, JavaScript, Python, Ruby, ...) and multiple target architectures (AMD64 and ARM) and has variants for both just-in-time and ahead-of-time compilation. It has front ends that generate an intermediate representation (IR) of the program being compiled from the source programming language. The compilation process includes multiple optimisation phases that transform the IR representation of a method/program to a more efficient version, also expressed in the IR. The final phase generates machine code for the target architecture from the optimised IR representation of the program.

Why Verify Compilers? Compilers for programming languages are an indispensable part of the trusted base of a software development platform. Their correctness is essential because an error in a compiler can lead to errors in any of the myriad of programs it compiles.

Why Focus on the Optimiser? For a multi-lingual, multi-target compiler, the machine-independent optimiser is common to all source programming languages and all target machine architectures and hence correctness of the optimiser affects all source languages and all target architectures.

[1] https://github.com/oracle/graal.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023
Y. Li and S. Tahar (Eds.): ICFEM 2023, LNCS 14308, pp. 3–8, 2023.
https://doi.org/10.1007/978-981-99-7584-6_1

The optimiser is a common source of errors within compilers. In a study of C compilers, Yang et al. [7] found that for GCC, with optimisation turned off only 4 bugs were found but with optimisation turned on 79 bugs were found, and for Clang, with optimisation turned off only 19 bugs were found but with optimisation turned on 202 bugs were found.

Errors in an optimiser are often due to subtle edge cases that may not be covered by testing, whereas verification addresses all possible cases. For example, a quirk of two's complement arithmetic is that the most negative 32-bit signed integer $MinInt = -2^{31}$, when negated gives back $MinInt$ (because the largest representable positive integer is $2^{31} - 1$ and hence $-MinInt = 2^{31}$ is not representable as a 32-bit signed integer and the negation of $MinInt$ "overflows" and gives back $MinInt$). One consequence of this is that the absolute value function when applied to $MinInt$ gives $MinInt$, a negative value! Hence a plausible optimisation that replaces $0 \leq abs(x)$ with $true$ is invalid if x is $MinInt$.

Overview. The GraalVM IR for a method consists of a graph structure that combines both control-flow and data-flow nodes [3]. In this paper we overview our approach to verifying the optimisation of data-flow sub-graphs, which represent expressions in the source language. We have developed a model of the IR in Isabelle/HOL [1] and then given the IR a semantics [6] (see Sect. 2). Expression optimisations are given as a set of conditional term rewriting rules (see Sect. 3). Proving the rules correct then corresponds to showing that they preserve the semantics (see Sect. 4). Generating efficient code for an optimiser from a set of rewriting rules is overviewed in Sect. 5.

2 Data-Flow Sub-graphs

GraalVM IR data-flow sub-graphs are,

side-effect free – side effects are factored out into the control-flow part of the graph,

well-defined in context – runtime exceptions such as divide by zero or index out of range are guarded in the control flow graph, so that for example, a divide node cannot be reached if its divisor is zero, and

share common sub-expressions – if the same sub-expression, e, is used in multiple places in an expression f, a single sub-graph representing e is shared by all references to e within f.

Sharing common sub-expressions is essential for generating efficient code but it means that the representation of a term (i.e. a programming language expression) is not a conventional abstract syntax tree but rather a directed acyclic graph structure with a single root node, commonly known as a *term graph* [2]. Figure 1 gives an example of both a conventional tree and (maximal sharing) term-graph representation of the term $x * x + x * x$. Note that in the term-graph representation, the node representing x is shared in the sub-graph representing $x * x$, and the root node of $x * x$ is shared in the whole expression $x * x + x * x$. Also note

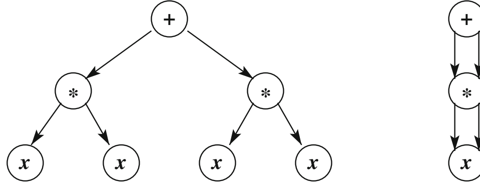

Fig. 1. Abstract syntax tree and term-graph representations of $x * x + x * x$.

that the same sharing occurs if x is a single node or a term graph representing a more complex shared sub-expression.

Each leaf node of a term represents either a constant, a parameter to the method in which the expression occurs, or a control-flow node, such as a method call node, where at runtime the value associated with that node will have already been calculated by the control-flow execution, e.g. the result of a method call.

The semantics of expressions is defined over their abstract syntax tree (or tree for short) form. Term-graphs are a (more efficient) representation of a tree (i.e. a data refinement). For any term graph there is a unique corresponding tree and hence the semantics of a term graph is defined as the semantics of the corresponding tree. Our Isabelle/HOL semantics for an expression, e, is parametrised with respect to a context consisting of a list, p, of *parameters* of the method in which e occurs and a *method state*, m, consisting of a mapping from control-flow node identifiers (e.g. for a method call node) to their (pre-computed) values. The following relation represents evaluating term e in a context consisting of method state m and parameters p to value v.

$$[m, p] \vdash e \mapsto v \tag{1}$$

In Isabelle/HOL this is a (deterministic) relation, rather than a function; if the value of e in context $[m, p]$ is not well defined, the relation does not hold.

Values may be integers, object/array references or the special undefined value.[2] The semantics needs to take into account the bit width of integer values (e.g. 32 or 64) because unbounded integers do not have the same semantics. For integers, their values are represented by a 64-bit value (using the HOL Word library) plus a bit-width b, where $0 < b \leq 64$; only the low-order b bits of the 64-bit word are significant. In two's complement arithmetic, for an expression such as $(x + y) - y$, the calculation of $x + y$ may overflow but subtracting y then "underflows" the value back to x, allowing $(x + y) - y$ to be replaced by x, even with the presence of overflow.

To validate our semantics we have developed a tool that translates GraalVM test cases written in Java to their Isabelle/HOL representation. Each test case is run in Java and its result compared with the value determined by our executable Isabelle/HOL semantics (see [4] for more details).

[2] GraalVM IR handles floating point numbers but we have not addressed those as yet.

3 Term Rewriting Rules

Expression optimisations are based on the algebraic properties of the expressions, e.g. $x * 0 = 0$, and can be expressed as conditional term rewriting rules [5], for example, in the following rewriting rules x, y, t and f represent arbitrary expressions, c represents an integer constant, and $\ll$ is the left shift operator. The compiler performs static analysis that tracks the lower and upper bounds of a node, which are stored in the node's *stamp* so that for Rule 6 if the upper bound for x is less than the lower bound for y, $x < y$ must be true in that context. A division node within the graph can only be reached after a (control-flow) check that the divisor is non-zero, otherwise an exception is raised.[3] That allows an optimisation like Rule 3 to be valid because the case when x is zero cannot occur.

$$x * 0 \longmapsto 0 \tag{2}$$
$$x/x \longmapsto 1 \tag{3}$$
$$(x + y) - y \longmapsto x \tag{4}$$
$$x * c \longmapsto x \ll \log_2 c \quad \textsf{when} \ \ isPower2\ c \tag{5}$$
$$x < y \longmapsto true \quad \textsf{when} \ \ upper(stamp(x)) < lower(stamp(y)) \tag{6}$$
$$\neg false \longmapsto true \tag{7}$$
$$(true\ ?\ t : f) \longmapsto t \tag{8}$$

The rewriting rules can be applied to any sub-term and in any order. In practice, it is better to optimise all sub-terms of a term e before applying rules to optimise e itself. An exception is when optimising a conditional $(b\ ?\ t : f)$, in which case it is better to first optimise b (e.g. using Rule 6 or Rule 7) because if Rule 8 can then be applied then f is eliminated from the expression and then only t needs to be optimised.

4 Verifying Term Rewriting Rules

This section briefly overviews the verification of rewriting rules (for more details see [5]). We say term e_1 is *refined by* term e_2 if and only if for all contexts $[m, p]$, if e_1 evaluates to a well-defined value v, so does e_2.

$$e_1 \sqsupseteq e_2 = (\forall m\ p\ v\ .\ [m, p] \vdash e_1 \mapsto v \implies [m, p] \vdash e_2 \mapsto v) \tag{9}$$

To show a rewriting rule, $e_1 \longmapsto e_2$ when *cond*, is correct, we show that if *cond* holds e_1 is refined by e_2.

$$cond \implies (e_1 \sqsupseteq e_2) \tag{10}$$

For Rule 2, the right side (i.e. 0) is valid in all contexts but the left side (i.e. $x * 0$) is only well defined in contexts where x is well defined. For the division

[3] Our treatment of the semantics assumes that all division nodes are so guarded.

node, the semantics defines $0/0$ to be a special undefined value, and hence Rule 3 is valid because the values of the two sides of a rewriting rule only need to be equal if the left side is well defined.

Verifying optimisations as term rewriting rules is much simpler on the tree representation than on the term-graph representation because, in a term graph, a replaced node may be referenced in multiple places in the graph. To show that term-graph rewriting is correct, we make use of a theorem that shows that if $e_1 \sqsupseteq e_2$ and a term graph matching e_1 is replaced by the corresponding term graph for e_2, then the semantics of the overall graph is preserved.

5 Generating Code for Optimisations

The approach described above represents optimisations as a set of rewriting rules. One could naively translate each rule to code and apply them repeatedly until no rule was applicable.[4] We are currently developing an approach to generate an efficient optimiser from sets of rewriting rules. In practice, there is often overlap between rules in the matching process, for example, all rules with the same node at the top-level of their pattern will perform the same initial match. In practice there are many rewriting rules for each kind of top-level node and hence in generating code we would like to factor out such matching so it is only done once. The factoring can also be applied to sub-expressions of the pattern.

To handle code generation we need to introduce more basic matching primitive, $\mathsf{match}\, e\, p$, that matches term e with pattern p; it takes an initial substitution s and if the match succeeds, returns s updated with instantiations for the free variables within p. Matches can be composed using, $m_1 \fatsemi m_2$, to form a combined match that takes a substitution, s, and returns s updated for both matches, if they both succeed, but if either fails their combination fails. For a condition C, $\mathsf{test}\, C$, fails is C does not hold for its input substitution s, otherwise it passes through s. Alternative rules can be combined using, $r_1\, \mathsf{else}\, r_2$, meaning take the result of r_1 if it succeeds, otherwise try r_2. For example, (Rule 2 else Rule 5) expands to,

$$(\mathsf{match}\, e\, (x * y) \fatsemi \mathsf{match}\, y\, (\mathsf{con}\, c) \fatsemi \mathsf{test}(c = 0) \fatsemi \mathsf{apply}\, 0)\ \mathsf{else}$$
$$(\mathsf{match}\, e\, (x * y) \fatsemi \mathsf{match}\, y\, (\mathsf{con}\, c) \fatsemi \mathsf{test}(isPower2\, c) \fatsemi \mathsf{apply}(x \ll \mathsf{eval}(\log_2 c)))$$

which after factoring out the initial matches becomes the following.

$$\mathsf{match}\, e\, (x * y) \fatsemi \mathsf{match}\, y\, (\mathsf{con}\, c) \fatsemi ((\mathsf{test}(c = 0) \fatsemi \mathsf{apply}\, 0)\ \mathsf{else}$$
$$(\mathsf{test}(isPower2\, c) \fatsemi \mathsf{apply}(x \ll \mathsf{eval}(\log_2 c))))$$

Generating efficient code in a programming language, such as Java, from rewriting rules expressed using these primitives is relatively straightforward.

For many rewriting rules (e.g. Rule 4), if all sub-expressions of the left side (e.g. x and y) have already been optimised, then a successful application of the rewriting rule results in a term that cannot be further optimised and hence the optimisation of that term is complete.

[4] Each set of rewriting rules is given a measure function to ensure rewriting terminates.

6 Conclusions

Conditional term rewriting rules allow one to succinctly formalise expression optimisations. Each rule can be separately verified to show that it preserves the semantics of an expression, whenever the rule is applicable. In the context of the GraalVM compiler the rewriting rules can also be applied to its term-graph representation of expressions. Given a set of valid conditional rewriting rules, each representing individual optimisations, the challenge is then to combine them to form an efficient optimiser by factoring out common matches and avoiding applying rules in situations where they cannot possibly succeed.

Acknowledgements. Mark Utting's position and Brae Webb's PhD scholarship are both funded in part by Oracle Labs. Our thanks go to Paddy Krishnan, Andrew Craik and Gergö Barany from Oracle Labs Brisbane for their helpful feedback, and to the Oracle GraalVM compiler team for answering questions. Thanks also to honours students that have contributed to advancing the project.

References

1. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
2. Plump, D.: Essentials of term graph rewriting. Electron. Notes Theor. Comput. Sci. **51**, 277–289 (2002). https://doi.org/10.1016/S1571-0661(04)80210-X
3. Stadler, L., Würthinger, T., Simon, D., Wimmer, C., Mössenböck, H.: Graal IR: an extensible declarative intermediate representation. In: Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop. APPLC '13, pp. 1–9, February 2013
4. Utting, M., Webb, B.J., Hayes, I.J.: Differential testing of a verification framework for compiler optimizations (case study). In: FormaliSE 2023. IEEE (2023)
5. Webb, B.J., Hayes, I.J., Utting, M.: Verifying term graph optimizations using Isabelle/HOL. In: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. CPP 2023, pp. 320–333. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3573105.3575673
6. Webb, B.J., Utting, M., Hayes, I.J.: A formal semantics of the GraalVM intermediate representation. In: Hou, Z., Ganesh, V. (eds.) ATVA 2021. LNCS, vol. 12971, pp. 111–126. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88885-5_8
7. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, pp. 283–294. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993532