Prolog Optimizations in GraalVM

Thi Thuy Tien Nguyen
School of Electrical Engineering and Computer Science
The University of Queensland, Qld., 4072, Australia

Abstract

This paper investigates the integration of a Prolog-based optimization framework into the GraalVM compiler to explore the use of logic programming in compiler optimizations. GraalVM's intermediate representation (IR) is converted into Prolog facts, with optimization logic expressed through Prolog predicate rules. Experiments on canonicalization and conditional elimination show that Prolog enables clear and declarative rule definitions. Benchmark results show that stateless optimizations like canonicalization perform reasonably well, whereas stateful tasks like conditional elimination face notable overhead due to Projog's limitations, suggesting the need for further performance enhancements.

1. Introduction

Compiler optimizations are essential for improving program performance. Much of the focus has been on improving the compilation process through traditional imperative approaches. Logic programming languages, such as Prolog, offer unique advantages due to their declarative nature, making them well-suited for tasks like program analysis. However, their application in the optimization phase of compilers remains largely unexplored. This project investigates the feasibility of integrating a Prolog-based optimization framework into the GraalVM compiler. It aims to determine whether Prolog's declarative approach can improve the expressiveness and effectiveness of compiler optimizations compared to existing GraalVM techniques. The primary objectives are to assess the technical feasibility of implementing this framework, develop Prolog-based rules for various optimization techniques, and evaluate their performance and integration within GraalVM's existing infrastructure.

2. Background

In GraalVM, the Graal IR models a program's structure and operations using a directed graph that illustrates both data and control flow between nodes [1]. Data flow is represented by input edges pointing upward to the operand nodes, control flow is depicted by successor edges pointing downwards to the successor nodes. This IR framework provides a robust and efficient structure for code analysis and optimization transformation. A critical aspect of this project involves converting the Graal IR into Prolog facts to enable the optimizer to query these facts for potential optimizations.

In traditional imperative languages, a program consists of a sequence of instructions. This approach emphasizes a step-by-step procedure to solve a given problem. In contrast, logic programming languages, such as Prolog, focus on defining a knowledge base composed of facts and rules [2]. Facts represent objects and their relationships, while rules imply the relationships. Logic programming languages with declarative specifications allow programs to be analyzed efficiently through queries. By focusing on what needs to be achieved rather than how to achieve it, declarative approaches offer greater clarity and intuitiveness. Previous research has extensively utilized Datalog, a prominent logic programming language, for different levels of program analysis. Tools like Soufflé [3] apply Datalog to perform advanced control-flow analysis, while DIMPLE [4] uses Prolog to analyze Java bytecode, supporting flexible experimentation and high-performance results. However, there is a notable lack of prior work regarding the application of logic programming languages for the optimization and transformation of code.

3. Methodology

This project approach to integrating Prolog into GraalVM consists of three phases: writing Prolog rules, generating Prolog facts, and querying Prolog for optimization. In the first phase, optimization rules are translated into Prolog predicates, initially focusing on canonicalization and conditional elimination. The second phase involves generating Prolog facts representing the IR nodes and their relationships. The final phase queries the Prolog rules for optimization and transforms the IR accordingly. These phases will be developed in parallel to ensure compatibility between Prolog facts and rules. Existing test suites in GraalVM will be leveraged to ensure the correctness of the integration. For evaluation, optimization throughput will be benchmarked and compared with GraalVM's existing techniques to assess performance improvements.

4. Results

A. Canonicalization: Add Node

```
// 1 of 9 rules
canonical(add(X,Y,Op), '(x - y) + y -> x'):-
    X = node(IdSub,-,XSub,Y),
    member(associative,Op).
```

$Uncanonical$	$Canonical$
$x + \sim x$	-1
$\sim x + x$	-1
$-x + y$	$y - x$
$x + -y$	$x - y$
$x + 0$	x
$0 + x$	x
$(x - y) + y$	x
$x + (y - x)$	y

TABLE I: Canonicalization Rules for Add Node

This project has implemented 9 canonicalization rules for Add node which are shown in TABLE I. An example of a Prolog rule for canonicalization is shown in the code snippet above.

B. Conditional Elimination

```
// Case 1: X equals a constant
if (x == 1) {
    // this block is simplified to false
    if (x == 2) {}
}
// Case: 2: X is larger than a constant
if (x > 1) {
    // this block is simplified to false
    if (x == 0) {}
}
```

The above code snippet demonstrates conditional elimination of nested if conditions that are handled in this project. In both cases, the inner if condition becomes unreachable and can be safely removed.

```
// Entry  predicate
process_if_nodes(NodeId, Result) :-
    node(NodeId, if, _, _, _),
    update_state(NodeId, DominatorId),
    is_true_successor(NodeId, DominatorId,
    IsTrueSucc),
    check_guard(NodeId, DominatorId,
    IsTrueSucc),
    try_fold(NodeId, DominatorId, IsTrueSucc,
    Result).
```

The entry predicate process_if_nodes/2 is responsible for processing if nodes, checking guard conditions, and determining if branches are unreachable.

C. Benchmarking

Integrating Prolog rules into the GraalVM optimizer leads to slower performance overall compared to the current Graal optimizer as shown in TABLE II. While the slowdown in canonicalization tests is relatively minor, with only slight performance differences, the conditional elimination tests show a significant decrease in throughput when using Prolog.

$Test$	$With\ Prolog$	$Without\ Prolog$
$canonical1$	1052348 ± 9217	1050055 ± 11484
$canonical2$	997088 ± 9253	1009246 ± 10954
$canonical3$	767676 ± 10135	772046 ± 15732
$condElim1$	560 ± 9	79171 ± 583
$condElim2$	339 ± 11	37074 ± 229

TABLE II: Benchmark Results for GraalVM Optimizations With and Without Prolog (Ops/Sec)

5. Discussion

This project successfully demonstrates the feasibility of integrating Prolog into GraalVM using the Projog library. Prolog facts are generated to represent multiple IR nodes in the compiler, and predicate rules are implemented to express the optimization logic. Canonicalization rules, such as those for AddNode, are simple, stateless, and use a singleton Projog class, resulting in consistent performance with minimal overhead. The first run is slower due to the initial class setup and rule loading, but later runs are faster. Conditional elimination is much slower because it must reinitialize the Projog engine and reload rules for every run. This is due to the need to manage dynamic facts and frequent creation and removal of Stamp objects, which represent known values of variables and are used to evaluate the correctness of nested conditions. This suggests that while Prolog integration may offer benefits in rule specification simplicity and clarity, it comes at a notable performance cost, especially for certain optimization phases. In addition, Projog itself is relatively slow, and performance could be improved by using a faster Prolog library.

6. Conclusion

This project demonstrated the feasibility of integrating Prolog into the GraalVM compiler for expressing optimizations. The results show that while Prolog provides a clear and expressive way to define optimization rules, performance remains a concern, particularly in stateful and dynamic scenarios such as conditional elimination. Canonicalization, being stateless, benefited from a more efficient integration pattern using a singleton Projog class. Future work will focus on optimizing performance further, potentially by adopting a faster Prolog engine and extending the framework to support more complex optimization patterns.

References

[1] M. Sipek, B. Mihaljevic, and A. Radovan, “Exploring aspects of polyglot high-performance virtual machine GraalVM,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, May 2019, pp. 1671–1676. [Online]. Available: https://ieeexplore.ieee.org/document/8756917/

[2] M. Bramer, “Clauses and predicates,” in *Logic Programming with Prolog*. London: Springer London, 2013, pp. 13–27. [Online]. Available: https://doi.org/10.1007/978-1-4471-5487-7_2

[3] D. R. Silverman, Y. Sun, K. K. Micinski, and T. Gilray, “So you want to analyze scheme programs with datalog?” *CoRR*, vol. abs/2107.12909, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2107.12909

[4] W. C. Benton and C. N. Fischer, “Interactive, scalable, declarative program analysis,” in *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM, Jul 2007, pp. 13–24. [Online]. Available: https://dl.acm.org/doi/10.1145/1273920.1273923