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 different optimizations 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, particularly its slow startup time, 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 specifies 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.

Datalog has established itself as a powerful tool in compiler implementation through its applications in complex program analysis and transformation. Tools like Soufflé [2] apply Datalog to perform advanced control-flow analysis, while DIMPLE [3] uses Yap 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. Spinellis' work in 1999 explored an alternative method for expressing optimizations through declarative specifications. Specifically, he transformed specifications into string regular expressions, which are then applied to the target code, allowing for adaptive and efficient refinements [4]. However, Spinellis's work was limited to a rudimentary prototype of an optimizer with a "single optimization specification" and "limited flow-of-control optimizations" [4].

3. Methodology

This project's approach to integrating Prolog into GraalVM consists of three phases: writing Prolog rules, generating Prolog facts, and querying and parsing Prolog results for optimization. In the first phase, optimization rules are translated into Prolog predicates. The second phase involves generating Prolog facts representing the IR nodes and their relationships. The final phase queries the Prolog rules for potential optimization and transforms the IR accordingly. A custom Prolog result parser is implemented using recursive descent parsing to parse and convert Prolog output back into IR nodes. 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.

4. Approach

A. Prolog Rules

1. Add Node Canonicalization

$$x + (-y) \rightarrow x - y \qquad -x + y \rightarrow y - x$$
$$\sim x + x \rightarrow -1 \qquad x + \sim x \rightarrow -1$$
$$x + 0 \rightarrow x \qquad 0 + x \rightarrow x$$
$$(x - y) + y \rightarrow x \qquad x + (y - x) \rightarrow y$$

This project has implemented 8 canonicalization rules for the add node which are shown above. The Prolog rules for these canonicalization are simple and stateless, using a singleton Projog class. The rules are loaded once and reused for each run. An example of a Prolog rule for canonicalization is shown in the following code snippet.

```
% x + -y -> x - y
canonical(node(addnode, X, node(IdNegate,
    negate, Y)), Result) :-
    find_id(X, IdX),
    find_id(Y, IdY),
    Result = node(subnode, lookup(IdX), lookup
    (IdY)).
```

This rule takes an IR node representing an addition operation and checks whether one of the operands is a negate node. If this condition is met, it rewrites the addition as a subtraction operation and returns a new IR node term as the result which will be parsed and created by the result parser.

2. And Node Canonicalization

$x \ \& \ y \rightarrow y$ if $\sim mustSetX \ \& \ maySetY == 0$

$x \ \& \ y \rightarrow x$ if $\sim mustSetY \ \& \ maySetX == 0$

This project has implemented 2 canonicalization rules for the and node which are shown above. These rules are also simple and stateless. However, they are more complex than the and node rules as they require checking the mustSet and maySet values of the operands. In compiler optimizations, mustSet and maySet are bitmasks used to describe known properties of an operand's bit values. mustSet indicates the bits that are definitely set (i.e., must be 1), while maySet indicates the bits that may be set (i.e., could be 1).

```
% A stamp: stamp(lowerBound, upperBound,
    mustSet, maySet, bits)

% x & y -> y if ~mustSetX & maySetY == 0
canonical(node(andnode, X, Y, StampX, StampY),
     Result) :-
    StampX = stamp(_, _, MustSetX, _, _),
    StampY = stamp(_, _, _, MaySetY, _),
    bitwise_not(MustSetX, ComplementMustSetX),
    (ComplementMustBeSetX /\ MayBeSetY) = 0,
    find_id(Y, IdY),
    Result = lookup(IdY).
```

This rule takes in an *and node* with operands X and Y, along with their associated stamps. It checks if the bitwise AND of the complement of X's must-set bits and Y's may-set bits is zero. If so, it outputs a lookup to Y. lookup(IdY) will be parsed by the result parser to retrieve the corresponding node from the IR graph.

3. Loop Invariant Reassociation

```
// Case 1 - Before
for (int i = 0; i < 1000; i++) {
    res = ((i + i) + rnd1) + rnd2;
}
// Case 1 - After
for (int i = 0; i < 1000; i++) {
    res = (i + i) + (rnd1 + rnd2);
}

// Case 2 - Before
for (int i = 0; i < 1000; i++) {
    res = ((i * rnd1) * i) * rnd2;
}
// Case 2 - After
for (int i = 0; i < 1000; i++) {
    res = i * i * (rnd1 * rnd2);
}
```

The above code snippet demonstrates two cases of loop invariant reassociation handled in this project. In both examples, invariant expressions are reassociated and grouped together to enable more efficient computation. Although not explicitly shown in the snippet, the invariant computation are hoisted outside the loop during IR graph construction, reducing redundant computation and improving performance.

```
find_reassociate_inv(Node, LoopNodes, R) :-
    node_type(Node, NodeType, X, Y),
    find_reassociate(X, Y, Match1Id, Other1),
    node_type(Other1, Other1Type, Other1X,
    Other1Y),
    find_reassociate(Other1X, Other1Y,
    Match2Id, Other2),
    find_id(Other1, Other1Id),
    find_id(Other2, Other2Id),
    (
        % (other2 + match2) + match1
        % -> other2 + (match2 + match1)
        NodeType == Other1Type == addnode
            -> R = node(addnode,
                        lookup(Other2Id),
                        node(addnode,
                            lookup(Match1Id),
                            lookup(Match2Id)
                        )
                    );
        ...
    ).
```

The `find_reassociate_inv` rule takes as input a binary arithmetic node and the set of nodes that belong to a loop. Among its two operands, X and Y, the rule searches for a reassociation opportunity by traversing the dataflow graph upward from each operand. Unlike the canonicalization rule, which typically operates on a single node in isolation, this rule explores a chain of connected nodes. A candidate node for reassociation is expected to have two operands: one loop-invariant and one non-invariant. The rule then follows the non-invariant operand further to potentially identify a second non-invariant operand. It checks whether their combinations involve associative operators (such as addition or subtraction), analyzing the operators and their order to determine if the expression can be restructured. If so, it returns a transformed expression that groups the invariant computations together, enabling hoisting outside the loop. The rules for loop invariant reassociation are more complex than the canonicalization rules, however they are still stateless.

4. 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 two cases of conditional elimination of nested if statements handled in this project. In both cases, the inner if condition becomes unreachable and can be safely removed. The Prolog rules for conditional elimination are stateful and the most complex in this project. As the analysis traverses downward through the dominator tree, it creates stamps to track the known possible values of variables. When the traversal moves back up the tree, these variable states are retracted to maintain correctness. This stateful tracking enables accurate elimination of unreachable branches based on previously established conditions.

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

The Prolog rule `process_if_nodes` follows a series of steps to handle conditional elimination. First, it identifies an if node. Second, the `update_state` predicate is invoked to find the dominator of the if node. This predicate also retracts any stale variable stamps associated with nodes lower in the dominator tree than the current node. Third, the rule checks if the node is a true or false successor of its dominator node using the `check_successor_type` predicate. Fourth, the `check_guard` predicate checks the condition of the dominator node and stores the variable stamps accordingly. Finally, the `try_fold` predicate evaluates the condition of the node and eliminates the condition if it is redundant based on existing variable stamps. This sequence of operations ensures that unreachable branches are identified and removed by maintaining a stateful representation of variable values across the dominator tree.

B. Generating Prolog Facts

To generate Prolog facts from the IR, each node in the IR is visited and formatted according to a new verbosity level: Verbosity.Prolog. Under this verbosity, each node is represented as a Prolog fact, following a standardized structure. For general nodes, the format is node(id, nodeType, ...args), while for control-flow nodes, it follows node(id, nodeType, ...args, nextNode), where nextNode represents the successor in the control flow. This uniform representation allows the IR to be emitted as structured Prolog facts for rule-based optimization and analysis.

C. Querying Prolog Rules and Parsing Results

The Prolog result parser is implemented using a recursive descent approach and currently supports parsing three types of terms: lookup terms, node terms, and constant expressions (boolean or numeric). A lookup term, written as lookup(number), instructs the interpreter to retrieve an existing node from the IR graph using its id. A node term, such as node(nodeType, args...), represents a new IR node to be created by the interpreter, with nodeType indicating the kind of operation (e.g., addnode, subnode, constantnode, etc.) and args as its operands. The parser also handles constant expressions, which can be either boolean values (true, false) or numeric literals. The grammar is defined as follows:

```
term ::= lookupTerm | nodeTerm | constExp
constExp ::= booleanTerm | number
lookupTerm ::= "lookup" "(" number ")"
nodeTerm ::=
    "node" "(" nodeType "," args ")" |
    "node" "(" nodeType "," number ")"
args ::= term ("," term)*
nodeType ::= identifier
identifier ::= [a-zA-Z_][a-zA-Z0-9_]*
booleanTerm ::= "true" | "false"
number ::= '-'? [0-9]+
```

5. Results

This project defines an operation throughput metric to evaluate the impact of integrating Prolog into GraalVM. An operation is defined as the process of building the IR graph for a method and applying all optimization phases, including canonicalization, loop invariant reassociation and conditional elimination. The operation is repeatedly performed for five seconds. The total number of completed operations is divided by five to calculate the average number of operations per second (operation throughput). This benchmark is conducted twice: once using the standard GraalVM (without Prolog), and once after incorporating Prolog into GraalVM. Comparing both results indicates whether Prolog improves or degrades the performance of the optimization process.

$Test$	$Without\ Prolog$	$Without\ Prolog$
$negateAdd$	$2,242$	$2,133$
$addNot$	$2,964$	$2,830$
$addNeutral$	$3,005$	$2,920$
$addSubNode$	$3,050$	$2,938$

TABLE I: Benchmark Results for Add Node Canonicalization Tests (Operations/Sec)

negateAdd: $-y + x \rightarrow x - y$
addNot: $\sim x + x \rightarrow -1$
addNeutral: $0 + x \rightarrow x$
addSubNode: $(x - y) + y \rightarrow x$

$Test$	$Without\ Prolog$	$With\ Prolog$
$subSub$	304	268
$subAdd$	367	330
$addAdd$	375	339
$mulMul$	370	334

TABLE II: Benchmark Results for Reassociation Tests (Operations/Sec)

subSub: $(i1 - ni) - i2 \rightarrow i1 - i2 - ni$
subAdd: $(i1 + ni) - i2 \rightarrow ni + i1 - i2$
addAdd: $(i1 + ni) + i2 \rightarrow ni + i1 + i2$
mulMul: $(i1 * ni) * i2 \rightarrow i1 * i2 * n$

The benchmark results for both the canonicalization and reassociation tests show only minor performance reductions when using Prolog. In the canonicalization tests (Table I), the performance drop is minimal, with only slight decreases in operations per second. For example, in the negateAdd test, throughput drops from 2,242 ops/sec to 2,133 ops/sec, and in the addNot test, it decreases from 2,964 ops/sec to 2,830 ops/sec. Similarly, in the reassociation tests (Table II), the slowdown is relatively small, with a drop from 304 ops/sec to 268 ops/sec in the subSub test and from 375 ops/sec to 339 ops/sec in the addAdd test. Overall, while there is some performance degradation, the impact of Prolog on both canonicalization and reassociation phases is not significant.

$Test$	$Without\ Prolog$	$With\ Prolog$
$condElim1$	$1,661$	394
$condElim2$	$1,386$	355
$condElim3$	951	310
$condElim4$	517	243

TABLE III: Benchmark Results for Condtional Elimination Tests (Operations/Sec)

condElim1: 2 nested ifs.
condElim2: An outer if containing 2 child if statements, one of which contains a nested if.
condElim3: An outer if containing 3 child if statements, one of which contains a nested if.
condElim4: 8 nested ifs.

The results in Table III show a noticeable drop in optimization throughput when Prolog is used for conditional elimination. The number of operations per second is significantly lower with Prolog across all tests. For example, condElim4 drops from 517 to 243 ops/sec, and condElim1 from 1,661 to 394 ops/sec. Moreover, the data indicates that performance degrades further as the nesting depth of if statements increases. This suggests that while the Prolog-based approach is functionally correct, it currently introduces performance costs. This is likely due to the complexity of interfacing with the Prolog engine and the overhead associated with managing deeply nested, stateful logic.

6. 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.

7. 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] 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

[3] 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

[4] D. Spinellis, "Declarative peephole optimization using string pattern matching," *ACM SIGPLAN Notices*, vol. 34, pp. 47–50, Feb 1999. [Online]. Available: https://dl.acm.org/doi/10.1145/307903.307921