THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Prolog Optimizations in GraalVM

by

Thi Thuy Tien Nguyen

Supervisors

Brae J. Webb, Mark Utting, Ian J. Hayes

School of Electrical Engineering and Computer Science,
The University of Queensland.

9 May 2025

Thi Thuy Tien Nguyen
s4833843@uq.edu.au
9 May 2025

Prof Michael Brünig
Head of School
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia QLD 4072

Dear Professor Brünig,

In accordance with the requirements of the Degree of Master of Computer Science (Management) in the School of Electrical Engineering and Computer Science, I submit the following thesis entitled "Prolog Optimizations in GraalVM". The thesis was performed under the supervision of Brae J. Webb, Mark Utting, and Ian J. Hayes.

I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,
Thi Thuy Tien Nguyen

Contents

List of Figures

List of Tables

Abstract

Compiler optimizations are essential for making programs run faster and improving overall system performance. Modern compilers like GraalVM use complex transformation pipelines, typically written in imperative languages for fine-grained control. However, implementation of optimizations in imperative languages is often challenging and error-prone. This project proposes using Prolog, a logic programming language, to express compiler optimizations declaratively for greater clarity and maintainability. This thesis presents the integration of Prolog-based optimization techniques into the GraalVM compiler framework and evaluates their performance compared to native GraalVM optimizations. The project involves: (1) converting GraalVM's intermediate representation (IR) into Prolog terms that describe its structure; (2) expressing optimization strategies declaratively as Prolog rules; (3) querying these rules to identify potential optimization opportunities; and (4) parsing the results to update the IR and apply optimizations. Prolog's declarative syntax facilitates clear and concise expression of optimization logic. Benchmark results indicate that stateless optimizations such as canonicalization perform efficiently within this framework. However, stateful optimizations, including conditional elimination, encounter significant overheads largely attributable to the startup latency of the Prolog engine (Projog), highlighting current performance limitations. The findings suggest that while Prolog integration holds promise for enhancing optimization expressiveness, further improvements in Prolog engine performance are necessary to fully realize its potential in compiler optimization tasks. This work contributes to advancing compiler technology by exploring the feasibility and practical implications of leveraging logic programming in modern compiler frameworks, opening avenues for future research in this domain.

1 Introduction

Compiler optimizations play a key role in improving program execution speed and overall system performance. Most existing optimization techniques are implemented using imperative programming approaches. This project explores an alternative by applying logic programming, specifically Prolog, to express compiler optimizations. The work is carried out within the GraalVM compiler framework. The project assesses the feasibility of using a declarative logic-based approach for compiler optimizations within GraalVM. A comparison is made between this approach and traditional methods in terms of performance and expressiveness. Finally, we evaluate whether optimization rules could be effectively described and applied using Prolog.

Logic programming languages with declarative specifications allow source programs to be analyzed efficiently through queries. By focusing on what needs to be done rather than how to do it, declarative approaches offer greater clarity and intuitiveness. This inherent simplicity and readability facilitates experimentation with new optimization rules, as well as the maintenance of existing ones. Additionally, declarative code can be easier to verify correctness because it is more straightforward and easier to follow. Previous research has extensively utilized Datalog, a prominent logic programming language, for different levels of program analysis [2, 3, 5, 6]. However, there is a notable lack of prior work applying logic programming languages specifically to code optimization and transformation. Spinellis' work in 1999 explored expressing optimizations through declarative specifications rather than traditional imperative code [1]. However, that work was limited to a rudimentary prototype optimizer with a "single optimization specification" and "limited flow-of-control optimizations" [1].

This project develops a Prolog-based optimization framework that integrates with the GraalVM compiler. The approach involved four main steps, developed in parallel to ensure compatibility. First, GraalVM IR nodes are translated into Prolog terms suitable for Prolog queries. Second, optimization rules are written in Prolog using a declarative style to describe transformations. Third, Prolog queries are executed from within GraalVM using Projog, a Java-based Prolog interpreter, to identify potential optimizations. Finally, a recursive descent parser interprets the results returned by Prolog. Together, these steps enables the integration of logic-based optimizations into the GraalVM compilation process. To ensure correctness, the existing GraalVM test suites and new test cases verify the behavior of Prolog rules and validate the integration.

This project implements and evaluates three optimizations: canonicalization, loop-invariant reassociation, and conditional elimination. These are chosen to cover a spectrum of optimization types, from simple, stateless canonicalization rules to more complex transformations involving control flow and data dependencies. Simple optimizations like canonicalization and loop-invariant reassociation are purely declarative and stateless. While conditional elimination might be possible to implement in a stateless way, this project opts for a stateful approach because dynamically updating the knowledge base to track possible knwon variable values makes the analysis for this complex optimization simpler and more manageable.

The framework's performance is evaluated based on optimization throughput, measured as the number of optimization operations performed per second. An optimization operation is defined as the process of building the IR graph for a method and applying the optimization phases. Overall, the study demonstrates the feasibility of integrating Prolog-based optimizations into GraalVM and highlights both the strengths and limitations of this approach. The results show that Prolog's declarative syntax is well-suited for expressing stateless optimizations like canonicalization. However, more complex, state-dependent transformations such as conditional elimination faced performance challenges, mainly due to limitations in the Prolog's engine (Projog) slow startup time.

2 Background

2.1 Graal Intermediate Representation (IR)

The Graal IR [7] models a program's structure and operations using a directed graph that represents both data and control flow between nodes. Each node in this graph is designed to produce at most a single value and follows the Static Single Assignment (SSA) [8] form. 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. In Graal IR, nodes are categorized as fixed or floating. Fixed nodes represent operations with strict execution order, such as control flow instructions, while floating nodes represent computations without fixed order constraints and can be freely reordered within the graph as long as dependencies are maintained. This flexibility allows the compiler during scheduling to reorder floating nodes to optimize execution. This IR framework provides a robust and efficient structure for code analysis and optimization, where optimization processes transform the graph to enhance overall performance. A critical aspect of this project is the conversion of the Graal IR into Prolog-compatible representations, enabling the optimizer to query and reason about the IR for potential optimizations. Consequently, a thorough understanding of the IR's structure and components is essential to accurately translate and utilize it within the Prolog-based optimization framework.

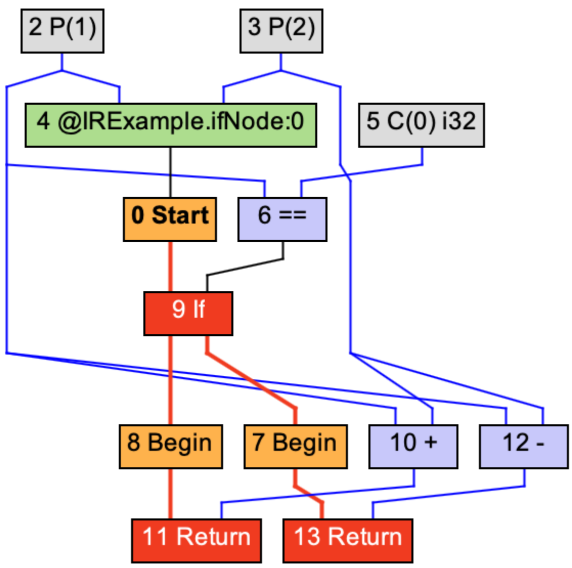

Figure 2.1: Simple IR Graph

Figure 2.1 illustrates a simple IR graph corresponding to the Java code snippet below, with control flow edges highlighted in red. The graph starts at the Start node (node 0), which connects downward to an If node (node 9) that evaluates a condition based on node 6. Following data flow edges upward, this condition node compares parameter 1 (node 2) and constant 0 (node 5). The If node has two successors: the true branch on the left, beginning at node 8, and the false branch on the right, beginning at node 7. Each branch continues downward to Return nodes 11 and 13, respectively. Return node 11 produces its result by following data flow edges upward to node 10, which performs an addition operation, while Return node 13 follows data flow upward to node 12, which performs a subtraction. Both arithmetic nodes take parameter 1 (x) and parameter 2 (y) as the first and second operands.

```
public int ifNode(int x, int y) {
    if (x == 0) {
        return x + y;
    }
    return x - y;
}
```

2.2 GraalVM Compiler Optimizations

In GraalVM, the compilation process is divided into two main phases. The first phase, involving the Graal IR, handles most of the high-level optimizations. This phase is further organized into three tiers: high-tier for high-level optimizations, mid-tier for memory-focused enhancements, and low-tier for low-level IR (LIR) conversion [9]. This project primarily concentrates on high-tier optimizations within the GraalVM, specifically canonicalization rules for the `AddNode` and `AndNode`, as well as loop invariant reassociation, and conditional elimination optimizations.

Canonicalization is an essential early phase in the optimization process, focusing on transforming code into a standardized format and eliminating redundancies. This transformation simplifies and facilitates the application of subsequent optimizations. An example of canonicalization is shown in the code snippet below.

```
// Before
return (x - y) + y;
// After
return x;
```

Loop invariant reassociation identifies expressions inside loops that yield the same result in every iteration and moves them outside the loop. This avoids repeated computation and reduces the loop's execution cost. It is a common technique for improving performance in tight computational loops. By reordering operands to group invariant variables together, the compiler can form a subexpression consisting entirely of invariants, represent it as a floating node, and hoist it outside the loop. An example of loop invariant reassociation is shown in the code snippet below.

```
// Before
for (int i = 0; i < 1000; i++) {
    // inv1 and inv2 are invariant to the loop
    res = ((i + i) + inv1) + inv2;
}

```

```
// After
for (int i = 0; i < 1000; i++) {
    res = (i + i) + (inv1 + inv2);
}
```

Conditional elimination is an optimization that removes conditional branches that will never execute. This helps simplify control flow and reduce unnecessary branching in the program. By eliminating conditions that are always true or false, the resulting code is slightly more efficient and easier to optimize further. An example of conditional elimiation is shown in the code snippet below. The condition $x == 2$ is always false, so the branch can be eliminated.

```
// Before
if (x == 1) {
    if (x == 2) {}
}
// After
if (x == 1) {
}
```

2.3 Prolog Programming Language

In traditional imperative languages, a program consists of a sequence of instructions. This approach emphasizes a step-by-step procedure where each instruction modifies the state of the machine to solve a given problem. In contrast, logic programming languages, such as Prolog, operate on a fundamentally different paradigm. Instead of prescribing a sequence of operations, logic programming focuses on defining a knowledge base composed of facts and rules [10]. After that, users can query the knowledge base to search for objects and relations.

In Prolog, facts represent objects and their relationships, while rules specify the relationship between objects given it satisfies all the conditions. Once the knowledge base is established, users can formulate queries to extract information or solve problems by leveraging the logical relationships defined in the base using the depth-first search algorithm [11]. There may be several ways to achieve a given goal. The system initially selects the first available option. If Prolog fails to resolve a specific subgoal, it will backtrack to explore these previously noted alternatives. This mechanism, referred to as backtracking [11], enables Prolog to systematically search for different solutions by revisiting and trying alternative paths.

Figure 2.2 illustrates a Prolog program. The first three clauses are facts: john is a parent of tom and jessica, and jessica is a parent of both mary and andrew. The final clause is a rule defining the conditions for a grandparent relationship. When a query is made to determine the nodes for which john is a grandparent, Prolog performs a search from the first to the last clause, showcasing its backtracking behavior. Initially, Prolog identifies that john is a parent of tom, but since tom is not a parent of anyone,

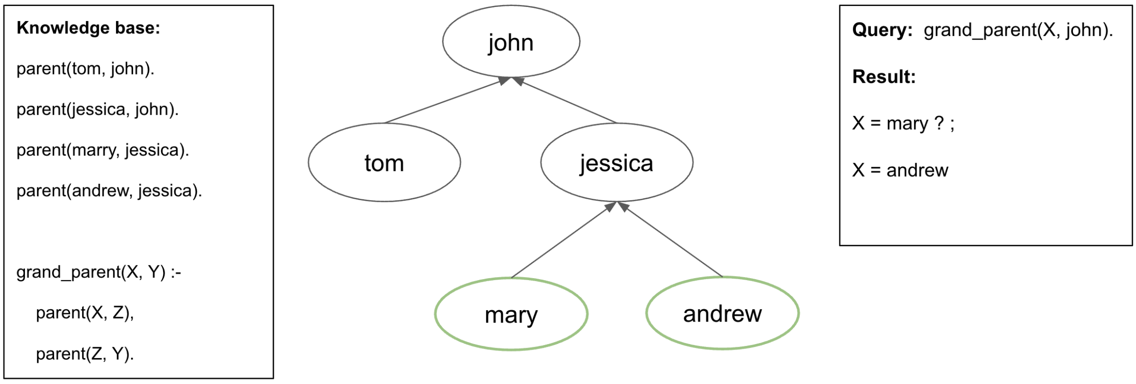

Figure 2.2: Example of Prolog specification

the search backtracks. It then considers the next option: jessica. Prolog then finds that jessica is a parent of mary. After exploring this path, Prolog backtracks again and continues to check the next possibility: jessica is also a parent of andrew, thereby completing the query. In Prolog syntax, any atom or argument starting with a capital letter is treated as a variable, whereas atoms starting with a lowercase letter represent literal values or constants. This distinction is important for pattern matching and variable binding during query evaluation.

3 Literature

3.1 Domain-Specific Languages (DSLs)

Declarative DSLs have made a big impact on compiler optimization by offering targeted solutions for specific optimization tasks. For example, Halide [12] is a domain-specific language and compiler framework designed to optimize image processing and computational photography. Writing efficient code for image processing often requires complex optimizations to exploit parallelism and memory locality. In Halide, the algorithm specifies the computational logic for these tasks, detailing what needs to be done, such as resizing, sharpening, or blurring an image. The schedule defines the execution strategy, including how computations are ordered, parallelized, and managed in memory. This separation of concerns allows Halide to generate highly optimized code by focusing on both the functional description and the efficient execution of tasks. Elevate [13] is another functional language designed to let programmers define custom optimization strategies. It allows for creating composable optimizations rather than sticking to predefined APIs. Empirical results from case studies show that Elevate effectively supports the definition and application of complex optimizations and achieving competitive performance.

Declarative input pattern and output specification

```
# Input pattern         # Output specification
L1:                     N1:
        S1*                     S1
        jeL2                    je N2
        S2*             N4:
        jmpL3                   S2
        S5*                     S4
L2:                             jmp N1
        S3*
L3:                     N3:
        S4*                     S1
        jmpL1                   jne N4
                        N2:
                                S3
                                S4
                                jmp N3
```

Regular Expression

```
(L\d+):(S\d+)*je(L\d+)(S\d+)*jmp(L\d+)
(S\d+)*\3:(S\d+)*\5:(S\d+)*jmp\1
```

Figure 3.1: Example of Optimizer Patterns and Regular Expression [1]

Spinellis created a peephole optimizer that uses a declarative DSL to specify optimizations [1]. Spinellis transformed specifications into string regular expressions, which are then applied to the target code, allowing for adaptive and efficient refinements. Figure 3.1 gives an example of this framework [1]. On the left of the figure is a declarative specification for the one-bit loop optimization [1]. The input pattern is parsed and translated into the regular expression in the right part of the figure. The

[1]je: jump if equal; jne: jump if not equal; jmp: unconditional jump; Sn: code place-holders; Ln: label specification; Nn: new label speciifications

optimizer uses regular expressions to find matches in the program and transform and optimize the code iteratively.

3.2 Logic Programming Languages

DataLog has established itself as a powerful tool in compiler implementation through its applications in complex program analysis and transformation. For instance, the Soufflé Datalog engine [14] introduces a new method for control-flow analysis in Scheme, making it possible to perform scalable and complex analyses of functional programming constructs, and extending the benefits of Datalog-based static analysis to Scheme-like languages. Similarly, DIMPLE [6], a framework for Java bytecode static analyses, is implemented in the Yap Prolog system. DIMPLE leverages logic programming capabilities to provide a declarative language for specifying analyses. The framework facilitates iterative experimentation and delivers efficient implementations that are on par with specialized tools.

Another example is the Doop framework [2] which utilizes Datalog to offer a declarative solution for points-to analysis in Java, providing impressive performance gains and scalability for precise context-sensitive analyses. Figure 3.2 illustrates a Datalog rule that Doop uses to represent a call graph. The query finds and returns a call graph edge if all of the predicates inside `CallGraphEdge` are true. First, it queries the properties of the virtual method call `?call` including its receive object `?base`, `?name`, and `?descriptor`. From this, the analysis retrieves the type of the heap object, represented by `?heaptype`, which refers to the runtime class of the object. This type is inferred from the allocation site where the object was created. Finally, the analysis performs a method lookup based on `?heaptype` to determine the actual target method being invoked.

```
CallGraphEdge(?callerCtx, ?call, ?calleeCtx, ?callee) <-
  VirtualMethodCall:Base[?call] = ?base,
  VirtualMethodCall:SimpleName[?call] = ?name,
  VirtualMethodCall:Descriptor[?call] = ?descriptor,
  VarPointsTo(?callerCtx, ?base, ?heap),
  HeapAllocation:Type[?heap] = ?heaptype,
  MethodLookup[?name, ?descriptor, ?heaptype] = ?callee,
  ?calleeCtx = ?call.
```

Figure 3.2: DOOP's rule for virtual method invocations [2]

Lam et al. developed a framework using Datalog queries and a specialized language called PQL, which employs deductive databases and binary decision diagrams (BDDs) to tackle complex issues like pointer aliasing and heap object management in a context-sensitive manner [3]. PQL helps simplify the process by allowing programmers to write queries in a more intuitive way. Instead of needing to understand the underlying database details, programmers can write code patterns in Java that are automatically converted into Datalog. Figure 3.3 provides an example of a query for simple SQL injection written in PQL and the equivalent Datalog rules. The SQL injection can be caught by checking if the parameter returned from calling `getParameter` on a `HttpServletRequest` is then passed directly as an argument to an `execution` call on the database `Connection`. Similarly, Datalog rules first check if there exists a method call to `getParameter` with a return value stored in a `v1` variable pointing to a heap object `h`. After that, it checks if there exists an invocation of `execute` with an argument `v2` also pointing to the object `h`.

Simple SQL Injection Query In PQL

```
query simpleSQLInjection()
    uses
        object HttpServletRequest r;
        object Connection c;
        object String p;
matches {
    p = r.getParameter(_);
    c.execute(p);
}
```

Simple SQL Injection In Datalog Rules

$$\begin{aligned} \mathit{SQLInjection}(h) \quad :- \quad & \mathit{calls}(c_1, b_1, \_, \text{“}\texttt{getParameter}\text{”}), \\ & \mathit{ret}(b_1, v_1), vP_c(c_1, v_1, h), \\ & \mathit{calls}(c_2, b_2, \_, \text{“}\texttt{execute}\text{”}), \\ & \mathit{actual}(b_2, 1, v_2), vP_c(c_2, v_2, h). \end{aligned}$$

Figure 3.3: Simple SQL Injection Example in PQL [3]

3.3 Transformation, Rewriting Techniques, and Verification

Transformation and rewriting techniques are essential to modern compiler optimization, enabling sophisticated processes like grammar specification, pattern matching, and program rewriting. Stratego [15] is a language designed for specifying program transformation systems using rewriting strategies, enabling precise control over rule application. The compiler transforms these rules into C code and leverages the ATerm library for effective term representation. It has been used in several applications, including program transformation tools like XT, and domain-specific optimization frameworks like CodeBoost [15]. On the other hand, Veriopt [4] offers a framework for formal verification of Graal compiler optimizations. This framework employs “term rewriting rules on the abstract term representation” [4] and then maps these rules to term graph transformations. In the Veriopt project, a term rewriting system, also known as an optimization phase, comprises a set of rules that can be applied repeatedly to optimize expressions. Each phase is associated with different types of root nodes in patterns, such as AddNode, AbsNode, and MulNode.

phase *Conditional*
 terminating *trm*
begin
optimization *NegateCond*: $((!c)\ ?\ t : f) \longmapsto (c\ ?\ f : t)$
optimization *TrueCond*: $(\mathit{true}\ ?\ t : f) \longmapsto t$
optimization *FalseCond*: $(\mathit{false}\ ?\ t : f) \longmapsto f$
optimization *BranchEqual*: $(c\ ?\ x : x) \longmapsto x$
optimization *LessCond*: $((u < v)\ ?\ t : f) \longmapsto t$
  when $(\mathit{stamp\text{-}under}\ (\mathit{stamp\text{-}expr}\ u)\ (\mathit{stamp\text{-}expr}\ v)$
    $\wedge\ \mathit{wf\text{-}stamp}\ u \wedge \mathit{wf\text{-}stamp}\ v)$
end

Figure 3.4: Conditional canonicalization rules [4]

Figure 3.4 provides an example of such an optimization phase tailored for conditional expressions. *NegateCond* transforms negated conditions using logical equivalences, while *TrueCond* simplifies expressions where the condition is always true by replacing the conditional expression with its true branch. Conversely, *FalseCond* addresses conditions that are always false by eliminating irrelevant branches. *BranchEqual* simplifies expressions where the true and false branches are identical by returning the common branch, thus removing unnecessary conditional checks.

3.4 Identified Research Gaps

Despite the established effectiveness of logic programming languages like Datalog in program analysis, there is limited research on integrating these techniques within modern compiler frameworks such as GraalVM. While significant advancements have been made in transformation and rewriting techniques using domain-specific languages (DSLs) and other paradigms, there is a notable absence of comparative studies on incorporating logic programming-based optimizations into GraalVM. This gap extends to a lack of empirical evidence assessing the impact of such techniques on optimization time and efficiency. Integrating Prolog-based rules with GraalVM's graph-based IR is particularly challenging due to insufficient research on mapping Graal IR to Prolog and ensuring that Prolog-based optimizers work effectively within the Java environment. Addressing these challenges could enhance our understanding of how to apply logic programming in modern compilers and improve compiler optimization practices by providing valuable insights into the performance and effectiveness of these techniques.

4 Implementation

4.1 IR to Prolog Translation

To enable logic-based optimization in GraalVM, the intermediate representation (IR) must first be translated into a form suitable for Prolog. This involves converting each IR node into a Prolog term. These terms can then be used as inputs to Prolog rules for reasoning and optimization, or asserted into the knowledge base when needed. The representation is designed to be simple, uniform, and easy to parse. Each IR node is expressed as a term of the form `node(Id, Type, ...)`, where the first argument is a unique identifier for the node within the method graph, and the second indicates its type (e.g., `add`, `if`, `constant`). The remaining arguments depend on the node type and encode operands, literal values, or successor relationships.

Binay Arithmetic Nodes

Binary arithmetic operations such as addition, subtraction, and similar computations are represented using the following structure:

```
node(Id, Type, FirstOperandId, SecondOperandId)
```

Where:

Id: Unique identifier for this node.

Type: Specifies the arithmetic operation being performed (e.g., `add`, `sub`).

FirstOperandId and `SecondOperandId`: Identifiers of the nodes providing the operands.

Example:

```
node(1, add, 2, 3)
```

This defines an `AddNode` with ID `1`, which computes the sum of the results from nodes `2` and `3`.

If Nodes

An if statement is represented using the following structure:

```
node(Id, Type, ConditionId, TrueSuccessorId, FalseSuccessorId)
```

Where:

Id: Unique identifier for this node.

Type: Specifies the node type (e.g., `if`).

ConditionId: Identifier of the node that evaluates the condition.

TrueSuccessorId: Node to proceed to if the condition is true.

FalseSuccessorId: Node to proceed to if the condition is false.

Example:

```
node(1, if, 2, 3, 4)
```

This defines an `IfNode` with ID `1`, which evaluates the condition from node `2`. If the condition is true, control flows to node `3`; otherwise, it flows to node `4`.

The translation process is implemented by extending the `toString` method of each IR node class to support a new verbosity level: `Verbosity.Prolog`. When this verbosity level is selected, each node formats itself as a Prolog-compatible term that reflects its internal structure and relationships. This approach leverages GraalVM's existing verbosity mechanism, which allows nodes to produce different string representations depending on the context. By introducing the `Prolog` verbosity level, IR nodes can emit a consistent, structured representation that is suitable for use as input to Prolog rules or for assertion into the logic engine. The following code snippet illustrates how the `toString` method of the `IfNode` class was extended to support this additional verbosity level:

```
@Override
public String toString(Verbosity verbosity) {
    if (verbosity == Verbosity.Prolog) {
        return "node(" + String.join(
                ",",
                toString(Verbosity.Id),
                toString(Verbosity.Name).toLowerCase(),
                condition.toString(Verbosity.Id),
                trueSuccessor.toString(Verbosity.Id),
                falseSuccessor.toString(Verbosity.Id)
        ) + ")";
    } else {
        return super.toString(verbosity);
    }
}
```

4.2 Prolog Optimization Rules

Each optimization in this framework is defined in a Prolog file named after the transformation it performs, such as `addNode.pl`. These files contain the logic rules that describe how specific intermediate representation (IR) patterns should be recognized and transformed during compilation.

Stateless optimizations such as canonicalization and loop invariant reassociation are purely pattern based. They operate by matching structural patterns in the IR and returning transformed results without modifying the knowledge base. In contrast, stateful optimizations such as conditional elimination require tracking intermediate state during query execution. These optimizations use Prolog's dynamic capabilities, including fact assertion and retraction, to simulate data flow and propagate information such as possible kwown variable values for conditional elimination optimization.

4.2.1 Add Node Canonicalization

Use cases

This project has implemented eight exploratory canonicalization rules for the `AddNode` as shown below.

$$
\begin{array}{cc}
x+(-y) \rightarrow x-y & -x+y \rightarrow y-x \\
\sim x+x \rightarrow -1 & x+\sim x \rightarrow -1 \\
x+0 \rightarrow x & 0+x \rightarrow x \\
(x-y)+y \rightarrow x & x+(y-x) \rightarrow y
\end{array}
$$

These rules target common arithmetic simplifications that occur in real-world programs and can lead to more efficient generated code by eliminating unnecessary operations. Canonicalization is particularly valuable because it tends to expose further optimization opportunities and simplify control or data flow in the IR. The Prolog rules for these transformations are simple and stateless, making them well-suited for logic-based, declarative expressions. This declarative form makes the rules easy to read and reason about.

Prolog Implementation

```
% (x - y) + y -> x
canonical(node(add, node(Id, sub, IdX, IdY), IdY), Result) :-
    Result = lookup(IdX).
```

This rule matches an `addnode` where the first operand is a `subnode` identified by `Id`, and the second operand is identical to the right operand of that subtraction. The transformation relies solely on node identifiers rather than the internal structure of the nodes. When the same node ID `IdY` appears both as the right operand of the subtraction and as the second operand of the addition, the rule simplifies the expression `(x - y) + y` to just `x` by returning a lookup of `IdX`. Using node identifiers works correctly because the intermediate representation employs common subexpression sharing and global value numbering, ensuring that identical subexpressions share the same node ID. The resulting structure is returned as a new term in the variable `Result`, which is later parsed and reconstructed back into a

GraalVM IR node by the result parser.

Java Implementation Comparision

```
if (forX instanceof NegateNode) {
    // -x + y => y - x
    return BinaryArithmeticNode.sub(forY, ((NegateNode) forX).getValue
       (), view);
} else if (forY instanceof NegateNode) {
    // x + -y => x - y
    return BinaryArithmeticNode.sub(forX, ((NegateNode) forY).getValue
       (), view);
} ...
```

The code snippet above provides example of the Java implementation for canonicalization rules. Both the Java and Prolog versions aim to perform arithmetic simplification, and in this case, both are relatively simple because the underlying transformation rule itself is straightforward. However, the Prolog version stands out in terms of expressiveness and clarity. The Java code, while easy to follow, relies on verbose type checking and manual unwrapping of nodes, using imperative control flow and explicit casting to express the rule. In contrast, the Prolog version concisely encodes the same transformation using a single declarative clause that mirrors the algebraic identity directly. This leads to greater readability and a closer correspondence between the code and the mathematical reasoning behind the optimization, making the intent of the transformation immediately apparent.

4.2.2 And Node Canonicalization

Use cases

This project has implemented two canonicalization rules for the `AndNode` as shown below.

$$x \,\&\, y \rightarrow y \quad \text{if } \sim \texttt{mustSetX} \,\&\, \texttt{maySetY} = 0$$

$$x \,\&\, y \rightarrow x \quad \text{if } \sim \texttt{mustSetY} \,\&\, \texttt{maySetX} = 0$$

These rules, while still stateless and pattern-based, are slightly more involved than typical structural rules because they depend on type analysis of the operands. Specifically, they require inspecting the `mustSet` and `maySet` bitmasks associated with the operands. In the context of compiler optimizations, `mustSet` and `maySet` are abstract representations of known and possible values at the bit level. The `mustSet` bitmask identifies bits that are guaranteed to be 1 in the operand, while the `maySet` bitmask identifies bits that could potentially be 1. The operand `x` satisfies the following properties: `mustSetX & x = mustSetX` and `~maySetX & x = 0`. These bitmasks enable reasoning about the outcome of bitwise operations without knowing exact operand values at compile time. These rules help reduce redundant operations and enable further simplifications by eliminating unnecessary bitwise computations when the operand properties make them semantically redundant.

Prolog Implementation

```
% Complement of a bitmask
bitwise_not(X, Result) :-
    Mask is (1 << 64) - 1,
    Result is X xor Mask.

% x & y -> y if ~mustBeSetX & mayBeSetY == 0
canonical(node(and, X, Y, StampX, StampY), Result) :-
    StampX = stamp(_, _, MustBeSetX, _),
    StampY = stamp(_, _, _, MayBeSetY),
    bitwise_not(MustBeSetX, ComplementMustBeSetX),
    (ComplementMustBeSetX /\ MayBeSetY) = 0,
    find_id(Y, IdY),
    Result = lookup(IdY).
```

This rule matches an andnode where the bitwise AND operation can potentially be simplified based on the bitmask metadata of its operands. It takes as input an andnode term consisting of two operand nodes along with their associated stamp information. Each stamp contains four pieces of data: the minimum value that the node could have, the maximum value that the node could have, the mustBeSet bits, and the mayBeSet bits. For this rule, only the mustBeSet and mayBeSet fields are relevant. The rule computes the bitwise complement of the first operand's mustBeSet bits and then performs a bitwise AND with the second operand's mayBeSet bits. If this result is zero, it indicates that the original AND operation can be simplified to just the second operand. The rule then extracts the unique identifier of this operand and returns it as the simplified result. The bitwise complement operation is implemented using a helper predicate bitwise_not because the Projog engine does not provide a built-in predicate for this operation. This method leverages precise bit-level information to safely optimize the code. A similar predicate is used to handle the complementary case where x is returned instead of y.

```
% These predicates are used to find the ID of a node.
find_id(node(Id, _), Id).
find_id(node(Id, _, _), Id).
find_id(node(Id, _, _, _), Id).
find_id(node(Id, _, _, _, _), Id).
find_id(node(Id, _, _, _, _, _), Id).
```

The code snippet above defines a set of helper predicates that enable the Prolog rules to extract the unique identifier associated with a node from its term representation. In this framework, each IR node is represented as a compound Prolog term where the first argument is the node's ID. Since nodes may have varying arities depending on their type and number of operands, multiple clauses of the `find_id/2` predicate are defined to handle nodes with different numbers of arguments. This abstraction allows Prolog rules to uniformly access node identifiers without needing to know the exact structure of the node, improving both readability and generality of the optimization rules.

Java Implementation Comparision

```
IntegerStamp xStamp = (IntegerStamp) rawXStamp;
IntegerStamp yStamp = (IntegerStamp) rawYStamp;
if (((~xStamp.mustBeSet()) & yStamp.mayBeSet()) == 0) {
    return forY;
} else if (((~yStamp.mustBeSet()) & xStamp.mayBeSet()) == 0) {
    return forX;
}
```

The code snippet above provides the Java implementation for canonicalization rules. Both the Java and Prolog versions implement the same underlying logic for the canonicalization of the bitwise AND operation, but they reflect their respective language paradigms in different ways. The Java code uses a single if-else statement to handle complementary cases within one procedural block, making the control flow explicit and linear. Meanwhile, the Prolog version expresses these cases as separate predicates, each capturing a specific rule declaratively. This separation aligns with Prolog's logical programming style, allowing each case to be reasoned about independently. While the Prolog code is somewhat longer, it offers clarity through modular rules. Both approaches have similar control flow, but the expression differs to suit the strengths of each language.

4.2.3 Loop Invariant Reassociation

Use cases

This project has implemented rules to match a variety of loop invariant reassociation patterns, as shown below.

$$
\begin{array}{ll}
inv1 - (i + inv2) \rightarrow (inv1 - inv2) - i & (i + inv2) - inv1 \rightarrow i + (inv2 - inv1) \\
(inv2 - i) + inv1 \rightarrow (inv1 + inv2) - i & (i - inv2) + inv1 \rightarrow i + (inv1 - inv2) \\
inv1 - (inv2 - i) \rightarrow i + (inv1 - inv2) & inv1 - (i - inv2) \rightarrow (inv1 + inv2) - i \\
(inv2 - i) - inv1 \rightarrow (inv2 - inv1) - i & (i - inv2) - inv1 \rightarrow i - (inv1 + inv2)
\end{array}
$$

$$
(i \ \mathtt{Op} \ inv2) \ \mathtt{Op} \ inv1 \quad \rightarrow i \ \mathtt{Op} \ (inv1 \ \mathtt{Op} \ inv2)
$$

In these patterns, `inv1` and `inv2` represent invariant expressions that do not change within the loop, while `i` denotes a loop-variant variable. The goal of these rules is to isolate the loop-variant component in order to enable hoisting of loop-invariant computations outside the loop. The last rule in the table represents a generalized reassociation pattern where both occurrences of the operator `Op` must be the same and drawn from a specific set of associative operations: addition, multiplication, bitwise AND, OR, XOR, and arithmetic max or min. These operations are associative and commutative, meaning the order of operands does not affect the result.

Prolog Implementation

```
% Base case: NodeId should not be in LoopNodes.
is_invariant(Node, NodeId, LoopNodes) :-
    \+ member(NodeId, LoopNodes).

% Recursive case: Check invariants for the children nodes X and Y.
is_invariant(Node, NodeId, LoopNodes) :-
    node_type(Node, NodeType, X, Y, IdX, IdY),
    is_invariant(X, IdX, LoopNodes),
    is_invariant(Y, IdY, LoopNodes).

% Case: Left is invariant, Right is variant
find_reassociate(X, Y, IdX, IdY, LoopNodes, IdX, Y, left) :-
    is_invariant(X, IdX, LoopNodes),
    \+ is_invariant(Y, IdY, LoopNodes).

% Case: Right is invariant, Left is variant
find_reassociate(X, Y, IdX, IdY, LoopNodes, IdY, X, right) :-
    \+ is_invariant(X, IdX, LoopNodes),
    is_invariant(Y, IdY, LoopNodes).
```

The above Prolog code defines logic for identifying loop-invariant computations and potential reassociation opportunities. The predicate `is_invariant/3` determines whether a node is invariant with respect to a list of loop-related node identifiers given by `LoopNodes`. In the base case, a node is considered invariant if its identifier, `NodeId`, is not a member of `LoopNodes`. In the recursive case, the predicate assumes the node has two child nodes, `X` and `Y`, with corresponding identifiers `IdX` and `IdY`. It recursively checks that both children are invariant. The predicate `find_reassociate/7` defines cases for recognizing expressions where one side is invariant and the other is not. In the first case, if the left child `X` is invariant and the right child `Y` is not, then it identifies `X` as the match and sets the side to `left`. In the second case, if the right child `Y` is invariant and the left child `X` is not, then it identifies `Y` as the match and sets the side to `right`. This logic enables detection of partial invariant expressions that could be reassociated to improve performance.

```
% Entry point for invariant reassociation
find_reassociate_inv(Node, LoopNodes, R) :-
```

```
    node_type(Node, NodeType, X, Y, IdX, IdY),
    find_reassociate(
        X, Y, IdX, IdY,
        LoopNodes, Match1Id, Other1, MatchSide1
    ),
    node_type(
        Other1, Other1Type, Other1X, Other1Y,
        Other1XId, Other1YId
    ),
    find_reassociate(
        Other1X, Other1Y, Other1XId, Other1YId,
        LoopNodes, Match2Id, Other2, MatchSide2
    ),
    find_id(Other1, Other1Id),
    find_id(Other2, Other2Id),
    reassociate_rule(
        NodeType, Other1Type, Match1Id, Match2Id,
        MatchSide1, MatchSide2, Other2Id, R
    ).
```

The predicate `find_reassociate_inv/3` serves as the entry point for identifying potential opportunities to reassociate invariant computations. The predicate takes a node (`Node`), a list of nodes inside the loop body (`LoopNodes`) that are not loop-invariant, and returns a reassociation result (`R`). It begins by extracting the type and subcomponents of the input node via `node_type/6`. It then attempts to find a matching reassociation pattern for the first level of operands using `find_reassociate/7`, which yields a matched node identifier (`Match1Id`), the non-matching “other” node (`Other1`), and which side the match occurred on (`MatchSide1`). The process is repeated for `Other1`, attempting to trace a second level of potential reassociation. The identifiers for these “other” nodes are resolved using `find_id/2`, and finally, all gathered information is passed into `reassociate_rule/8`, which checks whether a valid transformation rule applies and produces the reassociation result (`R`). Unlike the canonicalization rule, which typically operates on a single node in isolation, this rule explores a chain of connected nodes.

```
% Rule for addnode with subnode
reassociate_rule(
    addnode, subnode, Match1Id, Match2Id,
    MatchSide1, MatchSide2, Other2Id, R
) :-
    reassociate_add_sub(Match1Id, Match2Id, Other2Id, MatchSide2, R).
```

The above Prolog code defines a specific pattern-matching rule for reassociating arithmetic expressions in the case where the first operation is addition and the second operation is subtraction. The predicate `reassociate_rule/8` captures this transformation by invoking `reassociate_add_sub/5`, which encodes two concrete transformation patterns. The `reassociate_rule/8` predicate takes as input the types of the first and second operations (e.g., `addnode` and `subnode`), the identifiers of two matched subexpressions (`Match1Id` and `Match2Id`), information about which sides of the expressions matched (`MatchSide1` and `MatchSide2`), the identifier of the other node involved in the reassociation (`Other2Id`), and finally returns the reassociated expression `R`. This input allows `reassociate_rule/8` to select and apply the appropriate transformation pattern based on the structure and position of the invariant subexpressions.

```
% (inv2 - i) + inv1 -> (inv1 + inv2) - i
reassociate_add_sub(
    Match1Id, Match2Id, Other2Id, left,
    node(
        subnode,
        node(addnode, lookup(Match1Id), lookup(Match2Id)
    ),
    lookup(Other2Id))
).

% (i - inv2) + inv1 -> i + (inv1 - inv2)
reassociate_add_sub(
    Match1Id, Match2Id, Other2Id, right,
    node(
        addnode,
        node(subnode, lookup(Match1Id), lookup(Match2Id)
    ),
    lookup(Other2Id))
).
```

The two clauses of `reassociate_add_sub/5` define transformation rules that restructure arithmetic expressions to group loop-invariant computations. The first clause handles the case where the second invariant appears on the left side of a subtraction, while the second clause deals with the scenario where the invariant is on the right. In both cases, the transformation rearranges the expression so that the invariant parts are grouped together. This is just one example of a family of rules used to optimize expressions through reassociation. Similar predicates exist for other operator combinations such as `reassociate_sub_sub`, `reassociate_sub_add`, and so on, each encoding the specific transformation rules for their respective operator pairings.

Java Implementation Comparision

```
ValueNode m1 = match1.getValue(node);
ValueNode m2 = match2.getValue(other);
ValueNode a = match2.getOtherValue(other);
if (isNonExactAddOrSub(node)) {
    ValueNode associated;
    if (invertM1) {
        associated = BinaryArithmeticNode.sub(m2, m1, view);
    } else if (invertM2) {
        associated = BinaryArithmeticNode.sub(m1, m2, view);
    } else {
        associated = BinaryArithmeticNode.add(m1, m2, view);
    }
    if (invertA) {
        return BinaryArithmeticNode.sub(associated, a, view);
    }
    if (aSub) {
        return BinaryArithmeticNode.sub(a, associated, view);
    }
    return BinaryArithmeticNode.add(a, associated, view);
}
...
```

The code snippet above provides the Java implementation for reassociating invariant in loop. Both the Java and Prolog implementations perform loop invariant reassociation but differ significantly in style and clarity. The Java version uses several intermediary boolean variables like `invertM1`, `invertM2`, and `aSub` to control the flow and decide which arithmetic operation to apply. This approach, while compact, can be harder to follow because the logic is spread across multiple conditions and temporary variables, making the reasoning less direct. In contrast, the Prolog version is generally longer since it encodes each specific reassociation case as a separate predicate clause. This explicit case-by-case structure makes the Prolog code easier to understand and reason about, as each rule clearly corresponds to a particular transformation without relying on mutable state or complex branching.

4.2.4 Conditional Eliminationtion

Use cases

This project has implemented rules to match a variety of condtional elimination cases, examples of which are shown below.

```
// 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 Prolog rules for conditional elimination represent the most stateful and structurally complex analysis in this project. Unlike previous optimizations, which are generally stateless and operate locally on individual nodes, conditional elimination must track and manage global control flow context throughout the analysis. There are three related but distinct representations involved: the intermediate representation (IR) graph, the control flow graph (CFG), and the dominator tree. The IR graph models the program's operations, while the CFG captures possible execution paths between blocks. The dominator tree is computed by Graal through a postorder traversal of the CFG. Although dominators can be computed on the fly, where each node's dominator is either itself (if it is the start node) or the intersection of its parents' dominators, this project uses Graal's precomputed dominator tree and asserts blocks as facts into the Prolog knowledge base before querying. In this context, a block is represented as `block(BlockId, BeginNodeId, EndNodeId, DominatorBlockId)`, encoding a high-level basic block (`HIRBlock`) with its start and end nodes and a reference to its immediate dominator block in the control flow graph. For an if statement, the end node of the block is the actual if node that performs the branching. This approach simplifies queries by providing direct access to dominator relationships without recomputing them during analysis.

On the other hand, because neither the IR graph nor the dominator tree encode explicit parent and child links between outer and inner if nodes, the analysis must dynamically assert and retract facts describing current conditions and variable states as it traverses the dominator tree. A central concept in this analysis is the stamp. A stamp represents the known value range of a variable at a certain point in the control flow. A stamp is presented as `stamp(NodeId, UperBound, LowerBound, Level)`. Each stamp records the level of the if node where the value range was derived. This level indicates the depth of the condition in the nested control flow hierarchy, which is critical for defining the scope of the information. By knowing the level, the analysis can determine when a constraint no longer applies, such as when exiting a nested if block, and retract the stamp accordingly. Stamps are inferred from control flow conditions. For example, if the analysis encounters a branch that only executes when the variable x is less than five, it can assert a stamp recording that the value of x must be less than five on that path, and that this constraint originates at the current conditional node level. These known ranges are later used to simplify or eliminate subsequent conditional expressions, allowing the system to reason more effectively about the program's behavior.

Because the analysis moves down through the dominator tree and later back up, the state must be carefully managed. When descending into a node, new stamps may be asserted to reflect updated knowledge about variable values. However, when the traversal moves upward again, those stamps must be retracted to avoid applying invalid assumptions to unrelated parts of the graph. This stateful approach ensures that the analysis remains sound and accurate, while still enabling powerful optimizations such as the elimination of unreachable branches and the simplification of guarded conditions.

Prolog Implementation

```
// Entry predicate for processing if nodes
process_if_nodes(Node, Result) :-
    node(NodeId, if, _, _, _),
    update_dominance_info(NodeId, DominatorId),
    check_successor_type(NodeId, DominatorId, SuccType),
    check_guard(NodeId, DominatorId, SuccType),
    try_fold(NodeId, DominatorId, SuccType, Result),
    Node = lookup(NodeId).
```

The predicate `process_if_nodes/2` identifies and processes all nodes in the knowledge base that represent conditional branches, i.e., nodes of type `if`. It does not take a specific input node but instead searches through the facts to find nodes matching the pattern `node(NodeId, if, _, _, _)`. For each such node, it updates the analysis state by determining the dominator node via `update_dominance_info/2`, which finds the controlling predecessor in the control flow graph. It then examines the type of successor nodes related to this dominator using `check_successor_type/3`, and inspects the guarding conditions with `check_guard/3`. Based on these checks, `try_fold/4` attempts to simplify or fold the conditional node. When `process_if_nodes/2` is queried, it systematically returns pairs of `NodeId` and `Result`, such as `(NodeId = 1, Result = node(boolean, false))`, reflecting the outcome of attempting to optimize each conditional branch.

```
% Predicate to find dominator of a block
find_dominator(NodeId, DominatorId) :-
    block(_, _, NodeId, DominatorBlockId),
    block(DominatorBlockId, _, DominatorId, _).

% Update child node level
update_child_level(ParentNodeId, ChildNodeId) :-
    level(ParentNodeId, Level),
    NewLevel is Level + 1,
    asserta_unique(level(ChildNodeId, NewLevel)).

% Update global state
update_dominance_info(NodeId, DominatorId) :-
    find_dominator(NodeId, DominatorId) ->
    (
```

```
            % Update node level and retract stale stamp
            update_child_level(DominatorId, NodeId),
            retract_stale_stamps(NodeId)
        );
        % First if node doesn't have a dominator
        asserta_unique(level(NodeId, 0)),
        fail.
```

When visiting a new node, the `update_dominance_info/2` coordinates process to find the node dominator, its level in the tree and retract stale stamps. First, it attempts to find its dominator. The predicate `find_dominator/2` finds the immediate dominator of a node by looking up the `block` information. After finding the `block` containing the `NodeId`, the predicate retrieves its dominator block identifier `DominatorBlockId`. It then locates the `block` with that ID and returns the if node's identifier corresponding to that dominator block as `DominatorId`. Once the dominator is found, `update_child_level/2` updates the child node's level relative to its parent by fetching the parent's current level, incrementing it by one, and recording this new level for the child node using `asserta_unique/1`. If no dominator is found (such as for the root or first `if` node), it assigns level zero to the node and returns failure. This mechanism tracks the depth of nodes in the dominator tree or control flow hierarchy, which is critical for later analysis or optimizations.

```
% Predicate to find the maximum levels of the tree
find_max_level(MaxLevel) :-
    findall(Level, level(_, Level), Levels),
    max_list(Levels, MaxLevel).

% Retract all stamps that are guarded by the lower node level
retract_stale_stamps(NodeId) :-
    level(NodeId, MinLevel),
    find_max_level(MaxLevel),
    findall(Num, between(MinLevel, MaxLevel, Num), StaleLevels),
    retract_stamps(StaleLevels).
retract_stamps([]).
retract_stamps([Level|Rest]) :-
    retractall(stamp(_, _, _, Level)),
    retract_stamps(Rest).
```

To ensure consistency in the analysis, stale or outdated information about nodes lower in the dominator tree must be removed when visiting a new node. The predicate `retract_stale_stamps/1` achieves this by first determining the level of the current node, then finding the maximum node level recorded so far. It collects all levels between the current node's level and the maximum and retracts all corresponding `stamp/4` facts for those levels, effectively clearing cached data that may no longer be valid due to new information higher in the tree.

```
% Case: NodeId is the true successor
check_successor_type(NodeId, DominatorId, true) :-
    node(DominatorId, if, TrueSucc, _, _),
    node(TrueSucc, begin, NodeId).

% Case: NodeId is the false successor
check_successor_type(NodeId, DominatorId, false) :-
    node(DominatorId, if, _, FalseSucc, _),
    node(FalseSucc, begin, NodeId).

% Case: NodeId is neither (default case)
check_successor_type(_, _, unknown).
```

The above Prolog predicate check_successor_type/3 determines whether a given NodeId is a true successor, a false successor, or neither (unknown) with respect to an if node identified by DominatorId. It starts by retrieving the if node corresponding to DominatorId, which has two successor nodes: TrueSucc and FalseSucc. Then it checks if the given NodeId matches the beginning node of the true branch (TrueSucc) in the first predicate; if so, it binds Result to true. Otherwise, it checks if NodeId matches the beginning node of the false branch (FalseSucc) in the second predicate; if so, it binds Result to false. If neither condition holds, it assigns Result to unknown in the final predicate.

```
check_guard(DomOp, IdX, DomValueY, SuccType, NodeId) :-
    % Do nothing if dominator value is null
    DomValueY == null -> true;
    (
        level(NodeId, GuardId),
        member(DomOp,['==']), SuccType == true
            -> asserta_unique(stamp(IdX, DomValueY, DomValueY, GuardId
               ));
        member(DomOp,['<']), SuccType == false
            -> asserta_unique(stamp(IdX, DomValueY, max_int, GuardId))
               ;
        true
    ).

% check case both conditions are binary conditions,
% have the same node x, and y of the dominator condition is a constant
check_guard(NodeId, DominatorId, SuccType) :-
    node(NodeId, if, _, _, ConditionId),
    node(ConditionId, _, IdX, _),
    node(IdX, parameter(_)),
```

```
    node(DominatorId, if, _, _, DominatorConditionId),
    node(DominatorConditionId, DominatorOp, IdX, DominatorY),
    node(DominatorY, constant(DominatorYValue, _)),
    check_guard(DominatorOp, IdX, DominatorYValue, SuccType, NodeId).
```

The above Prolog code is designed to recognize and remember useful information from conditional checks in earlier parts of a program's control flow, which can later help optimize the program. The first predicate, `check_guard/3`, operates at a higher level by examining the current node's condition and its dominator's condition. It ensures both conditions relate to the same variable with identifier `IdX` and that the dominator's condition compares that variable to a constant. If these checks pass, it calls `check_guard/5` to attempt recording a `stamp` representing the possible known variable values. The second predicate, `check_guard/5`, takes the comparison operator from the dominator node, the variable's identifier `IdX`, the constant value `DomValueY` from the dominator node, the type of successor branch (true or false), and the current node `NodeId`. It first skips processing if the dominator value is missing. Otherwise, it retrieves the node's level in the control flow and, based on the operator and successor type, records a `stamp` fact. For example, if the operator is equality and the branch is true, it records that the variable equals the constant. If the operator is less-than and the branch is false, it records that the variable is at least the constant value. This `stamp` acts as a fact that the system can later use to simplify expressions or optimize code.

```
% Constant condition: value stamp available and deterministic
try_fold('>=', ValueY, MinX, MaxX, true) :-
    ValueY \= null, MinX == MaxX, MinX >= ValueY.
try_fold('>=', ValueY, MinX, MaxX, false) :-
    ValueY \= null, MinX == MaxX, MinX < ValueY.
...
try_fold(_, ValueY, _, _, unknown) :- ValueY \= null.
try_fold(_, ValueY, _, _, unknown) :- ValueY == null.
try_fold(_, _, _, _, unknown).  % General fallback

% try fold the condition
try_fold(NodeId, DominatorId, SuccType, Result) :-
    node(NodeId, if, _, _, ConditionId),
    node(DominatorId, if, _, _, DominatorConditionId),
    node(ConditionId, Op, IdX, IdY),
    node(DominatorConditionId, DominatorOp, IdX, DominatorY),
    node(IdX, parameter(_)),
    node(IdY, constant(ValueY, _)),
    node(DominatorY, constant(DominatorYValue, _)),
    (
        stamp(IdX, MinX, MaxX, _) ->
            try_fold(Op, ValueY, MinX, MaxX, Result);
```

```
    ).
```

The above Prolog code attempts to simplify or "fold" conditional expressions by using known constant value ranges to decide the truth of conditions. The predicate `try_fold/4` serves as the entry point: it extracts the operator and operands from the current conditional node and its dominator node, ensuring both involve the same variable `IdX` and that the compared values are constants. It then looks up any known value range for `IdX` stored as a `stamp`. If such a range exists, `try_fold/4` calls the more specific `try_fold/5` predicate to attempt to evaluate the condition conclusively as `true`, `false`, or `unknown`. `try_fold/5`, defines specific cases for different operators such as `'>='`. For example, if the operator is `'>='`, the constant `ValueY` is not null, and the known value range for the variable is a single value (`MinX` equals `MaxX`), then it returns `true` if that value satisfies the condition or `false` otherwise. Other operators have similar specialized rules. If none of these match, the predicate defaults to `unknown`, indicating the condition cannot be conclusively evaluated. A key limitation of the current code is that it only handles comparisons where a parameter is compared to a constant value; it does not support folding conditions involving two variables or more complex expressions.

Java Implementation Comparision

Both the Prolog and Java implementations of conditional elimination are inherently complex due to the nature of the analysis, involving intricate state tracking and control flow reasoning. The Java version typically spans multiple classes and methods, spreading the logic across different parts of the codebase, which can make following the overall flow challenging without jumping between files. In contrast, the Prolog implementation, while lengthy, expresses the logic declaratively within a set of predicates, making the reasoning about pattern matching and rule application more direct. However, Prolog's declarative style may require understanding predicate dependencies and backtracking, which can be less intuitive for those unfamiliar with logic programming. Overall, both versions balance complexity differently: Java through imperative, object-oriented constructs with explicit state management, and Prolog through concise logical rules with implicit control flow.

4.3 Query Result Parser

A critical part of integrating Prolog-based optimization into the GraalVM compilation pipeline is interpreting the results of Prolog queries in a way that can be applied to the compiler's intermediate representation (IR). This task is handled by a custom Prolog result parser, which translates the Prolog's output terms into corresponding GraalVM IR node constructions. The parser is implemented using a recursive descent approach and is designed to be modular, extensible, and robust to varying term structures. After optimization opportunities are identified through Prolog queries, the results returned must be interpreted as transformations to be applied to the IR graph. These transformations can include the reuse of existing nodes or the creation of new computation nodes. To support this, the parser must handle a structured grammar of Prolog terms that describe IR transformations. The design emphasizes clarity and extensibility, allowing new node types or term patterns to be added as the optimization logic evolves.

4.3.1 Supported Term Types

The parser currently supports two main categories of Prolog terms: lookup terms and node terms. Each of these is parsed into a distinct internal representation that maps to an IR node construction or modification. A lookup term refers to an existing node in the IR graph by its identifier. This supports reusing nodes in the optimization result rather than duplicating them. For example, `lookup(42)` instructs the parser to retrieve the IR node with ID `42` from the existing graph and use it as part of the resulting computation. A node term is used to represent the creation of a new IR node, such as arithmetic or constant node. Each term specifies a node type followed by one or more arguments, which can themselves be terms (allowing for recursive nesting). For instance, an addition node that sums two constants might be represented as: `node(addnode, node(constantnode, 1), node(constantnode, 2))`. This term defines an `AddNode` with two operand nodes created from constant values `1` and `2`.

4.3.2 Grammar Specification

The Prolog terms parsed by this component follow a well-defined recursive grammar. The grammar is shown below using a variant of Extended Backus-Naur Form (EBNF):

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

This grammar supports recursive composition, allowing complex IR structures to be described using nested terms. For example, a `nodeTerm` can take other `nodeTerms` or `lookupTerms` as arguments, enabling the construction of complete subgraphs within a single expression. The top-level rule `term` covers all supported input formats: `lookupTerm` refers to an existing IR node by ID and `nodeTerm` defines a new node with a specific type and arguments. The `args` rule enables a comma-separated list of nested terms, making the grammar flexible enough to capture deep IR trees. This grammar ensures a clean separation between symbolic IR node construction and raw value references, which is critical for correctly interpreting the semantics of Prolog query results.

4.3.3 Recursive Descent Parsing

The parser is implemented as a recursive descent parser, where each non-terminal rule in the grammar corresponds to a function in the code. For example:

- `parseTerm()` dispatches to `parseLookup()` or `parseNode()` based on the parsed identifier (`lookup` or `node`).
- `parseLookup()` parses a `lookupTerm` of the form `lookup(number)`.
- `parseNode()` parses a `nodeTerm`, handling both cases where the second argument is either a `constExp` or a list of nested terms.
- `parseArgs()` handles an arbitrary-length comma-separated list of `terms`, recursively parsing each argument.
- `parseConstExp()` parses constant expressions, which are either boolean literals (`true`, `false`) or numeric literals.

This structure makes the parser naturally align with the grammar, making it easier to maintain and extend.

4.3.4 IR Reconstruction

Once the Prolog result parser has constructed an internal representation of the parsed terms, these must be converted into actual GraalVM IR nodes. This is accomplished using the *visitor pattern*, a design approach that allows each parsed node type to define how it should be transformed into the corresponding IR node. Each parsed node class implements a `visit` method tailored to its specific node type. The visitor traverses the parsed term tree recursively, invoking the appropriate visit methods to construct the IR nodes. This pattern cleanly separates the parsing logic from IR construction, supporting easy extensibility when adding new node types. For example, the visitor method for an addition node might look like this:

```
@Override
public ValueNode visitAddNode(ExpNode.AddNode node) {
    // Recursively evaluate the left and right operands
    ValueNode left = node.left.evaluate(this);
    ValueNode right = node.right.evaluate(this);
    // Create a GraalVM AddNode from the evaluated operands
    return BinaryArithmeticNode.add(left, right);
}
```

Here, `visitAddNode` recursively evaluates its child nodes by calling `evaluate` on each, which triggers the visitor on those subnodes. After obtaining the operand IR nodes, it constructs a new GraalVM `AddNode` representing the addition operation. This recursive descent through the parsed term tree ensures that the full IR subtree corresponding to the Prolog query result is reconstructed accurately. The visitor pattern also facilitates modularity by encapsulating node-specific IR creation logic within each visit method, making the parser both robust and extensible.

4.3.5 Example Use Case

The following node representation describes an `AddNode` where the first operand is the existing IR node with ID 3, and the second operand is a multiplication of a constant 2 with another existing node with ID 5.

```
node(
    addnode,
    lookup(3),
    node(mulnode,node(constantnode,2),lookup(5))
)
```

The parsing and interpretation process proceeds as follows:

1. The parser identifies the outer `node(addnode, ...)` term, representing an addition node.
2. It encounters `lookup(3)`, which instructs the parser to retrieve the existing IR node with ID 3 from the current graph.
3. The parser then processes the inner `node(mulnode, ...)` term recursively. This involves:
   - Creating a `constantnode` with the numeric value 2.
   - Retrieving the existing IR node referenced by `lookup(5)`.
   - Constructing a `MulNode` from the constant node and the looked-up node.
4. After fully parsing the term tree, the corresponding GraalVM IR nodes are constructed using the visitor pattern:
   - The visitor's `visitAddNode` method is called with the parsed `AddNode` term.
   - It recursively evaluates its operands by invoking `evaluate` on the left child (the looked-up node with ID 3) and the right child (the constructed `MulNode` subtree).
   - The `MulNode` subtree is itself created by the visitor through its `visitMulNode` method, which similarly evaluates its operands.
   - Finally, the visitor assembles these evaluated child nodes into a new `AddNode` instance that reflects the optimized IR subtree.

4.4 Prolog Optimization Engine

To integrate all stages of the optimization pipeline, a coordinating class is used to drive the complete transformation process. This class coordinates the end-to-end interaction with Prolog, from loading logic rules to returning reconstructed intermediate representation (IR) nodes. Each Prolog-based optimization is encapsulated within its own Java class. For example, optimizations targeting canonicalization rules for the `AddNode` are implemented in the `AddNodeProlog` class, while conditional elimination optimizations are handled by the `ConditionalEliminationProlog` class. This modular design promotes separation of concerns, allowing each optimization to define the specific IR node types it operates on, the logic for translating IR nodes into Prolog terms, and the structure of the Prolog queries it executes. Stateless optimizations, such as canonicalization and loop-invariant reassociation, use a singleton class instance to avoid reloading rules and reduce startup overhead. In contrast, stateful optimizations require a fresh instance for each execution, as they may modify or depend on internal state during processing.

The coordinator leverages the `Projog` library, a Java-based Prolog interpreter, to allow query execution and rule evaluation within the Java environment. Unlike traditional native Prolog runtimes (such as SWI-Prolog) that run as separate processes, Projog runs entirely within the Java Virtual Machine as a library (delivered via a JAR file). This means it does not require starting an external Prolog process or communicating between separate processes using interprocess communication (IPC) mechanisms like sockets or standard input/output streams. Additionally, there is no need to install or manage an external Prolog runtime on the system. This tight integration enables Prolog queries to be invoked directly within the GraalVM infrastructure as part of the compilation pipeline, simplifying deployment and improving performance by avoiding overhead associated with external processes.

Each optimization class follows a common execution pipeline:

1. **Rule Loading:** At initialization, the engine loads a `.pl` file containing Prolog rules into the instance's knowledge base. These rules define the conditions and transformations associated with a particular IR pattern (section 4.2).

2. **IR-to-Prolog Translation:** The IR node to be optimized is recursively translated into a corresponding Prolog term (section 4.1).

3. **Fact Generation (Optional):** Before constructing the query, additional facts representing relevant structural or contextual information are dynamically asserted into the knowledge base.

   This step is necessary because the original nested IR graph structure is difficult to analyze directly for conditional elimination. For example, in conditional elimination, each `if` node has a fact asserted to record its position in the dominator tree, which helps the query reason about control flow order.

4. **Query Construction:** Construct a query with the translated Prolog term.

5. **Query Execution:** The query is submitted to the Projog engine. If a matching rule is found, the engine binds `ResultTerm` to a Prolog term that encodes the optimized form.

6. **Result Parsing:** The Prolog term returned as the result is parsed using the `PrologResultParser`. This converts it back into a Java representation suitable for IR construction (section 4.3).

7. **IR Reconstruction:** Finally, the parsed representation is transformed into a new GraalVM IR node using the visitor-based reconstruction mechanism described earlier (section 4.3).

End-to-End Optimization Example

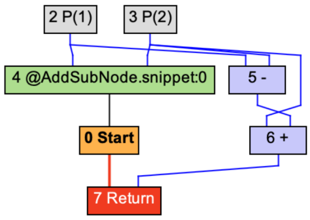

Figure 4.1: GraalVM IR Representation of (x - y) + y

The example illustrated in Figure 4.2 demonstrates the full optimization pipeline for a canonicalization rule applied to an addition operation. Starting from the IR graph representation of the original code (Figure 4.1), the process begins by translating the relevant IR subtrees into Prolog terms and constructing a query (step 1). The constructed query expresses a search for a canonicalized form of an addition operation. It takes two arguments: the first argument describes the addition operation and its operands in detail, while the second argument is a variable that will hold the result returned by the Prolog engine. The first operand of the `addnode` is a `subnode` represented by node 5, and the second operand is a parameter node 3 representing the variable `y`. The `subnode` itself has two operands: node 2, representing the variable `x`, and node 3, representing `y`. The Prolog query is then executed, and the engine returns a result, `lookup(2)`, indicating that the expression simplifies to an existing node, node `x` (step 2). This result is parsed into an internal representation by the Prolog result parser (step 3), which is subsequently used to reconstruct the corresponding GraalVM IR node by retrieving node ID 2 from the graph (step 4). The final output reflects the optimized method, where the original expression `(x - y) + y` is simplified to just `x`. This sequence demonstrates how the system translates, queries, parses, and reconstructs optimized IR seamlessly within the compilation pipeline.

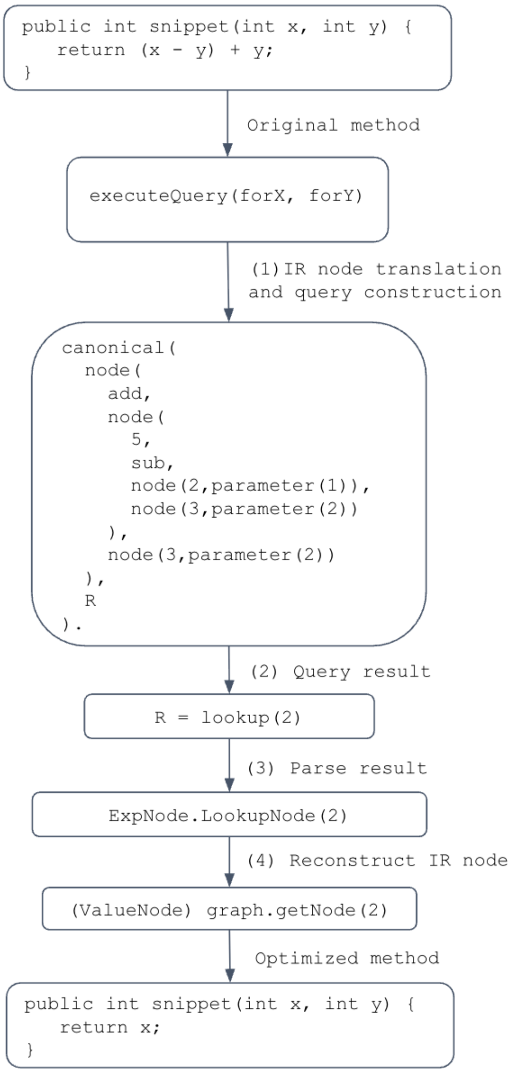

Figure 4.2: Prolog-Based Optimization Workflow

5 Performance Evaluation

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 gives a consistent and stable throughput metric measured in operations per second, where a higher value indicates faster execution time. Benchmark is conducted twice: once using the standard GraalVM (without Prolog), and once after incorporating Prolog into GraalVM.

Canonicalization Tests

Test Descriptions:

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

Test Results:

Test	*Without Prolog*	*With Prolog*	*Difference*
addNot	2,964	2,830	−134
addSubNode	3,050	2,938	−112
negateAdd	2,242	2,133	−109
addNeutral	3,005	2,920	−85

Table 5.1: Benchmark Results for Add Node Canonicalization Tests (Operations/Sec)

The benchmark results for the canonicalization tests (Table 5.1) show minor performance reductions when using Prolog. 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. This slowdown is primarily attributed to the overhead introduced by the initial startup of the Prolog engine, which occurs for every optimization and is more pronounced in the case of stateful optimizations (conditional elimination).

Loop Invariant Reassociation Tests

Test Descriptions:

- *subSub*: $(inv1 - i) - inv2 \rightarrow inv1 - inv2 - i$
- *subAdd*: $(inv1 + i) - inv2 \rightarrow i + inv1 - inv2$
- *addAdd*: $(inv1 + i) + inv2 \rightarrow i + inv1 + inv2$
- *mulMul*: $(inv1 * i) * inv2 \rightarrow inv1 * inv2 * n$

Test Results:

$Test$	$Without\ Prolog$	$With\ Prolog$	$Difference$
$subAdd$	367	330	−37
$subSub$	304	268	−36
$addAdd$	375	339	−36
$mulMul$	370	334	−36

Table 5.2: Benchmark Results for Reassociation Tests (Operations/Sec)

Similarly, the reassociation tests (Table 5.2) reveal a modest slowdown when Prolog is used. Although there is some performance degradation, the effect of Prolog on the reassociation phase remains limited and does not severely impact throughput.

Conditional Elimination Tests

Test Descriptions:

- *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.

Test Results:

$Test$	$Without\ Prolog$	$With\ Prolog$	$Difference$
$condElim1$	1,661	394	−1267
$condElim2$	1,386	355	−1031
$condElim3$	951	310	−641
$condElim4$	517	243	−274

Table 5.3: Benchmark Results for Condtional Eliminiation Tests (Operations/Sec)

The results in Table 5.3 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. *condElim1* is the simplest and fastest test as it only has 2 nested ifs, which is reflected in the much higher throughput without Prolog. However, when applying Prolog, its throughput drops the most, and all test results become more uniform. This is largely explained by Prolog's significant startup overhead, which imposes a fixed cost that dominates the runtime for smaller tests. As a result, this overhead "levels" the performance across different test complexities, causing a uniform but overall slower throughput.

6 Discussion

This project successfully demonstrates the feasibility of integrating a Prolog engine into the GraalVM optimization pipeline using the Projog library. By translating GraalVM's IR nodes into Prolog terms and implementing optimization logic as Prolog predicate rules, the system is able to perform a variety of compiler optimizations in a declarative and modular fashion. The results confirm that Prolog can be used as a backend for expressing compiler optimizations, making this integration both technically viable and functionally complete.

One key strength of the Prolog-based approach is its expressiveness. The declarative nature of Prolog allows optimization rules such as those for AddNode canonicalization to be written in a concise and readable manner that closely mirrors their mathematical definitions. This makes the logic easier to reason about and verify. However, during development, writing and debugging Prolog rules proved more challenging compared to Java. Prolog lacks the rich tooling and step-by-step debugging support available in Java environments, and its inference-based execution model can make it harder to trace errors or unexpected outcomes. Additionally, some optimization tasks, particularly conditional elimination, were difficult to express in a purely declarative form. They require dynamically asserting dominator tree blocks as facts to capture control flow relationships, and continuously updating knowledge about variable states as the analysis traverses different branches and nested levels of the dominator tree.

Developing the Prolog representation format presented several challenges that required iterative refinement. A key difficulty was managing the initial incompatibility between the node representation and the evolving design of the Prolog rules used for analysis and optimization. The representation needed to balance simplicity with expressiveness; for example, deciding whether a control-flow node like `if` should include only the identifiers of its condition and successors, or also embed more detailed structural information, such as the full definitions of those subnodes. These decisions were guided primarily by experimentation. Similarly, designing the Prolog result parser involved several challenges. First, deciding on a clear and expressive grammar that could accurately capture the variety of IR node structures was critical. The grammar needed to be both precise and flexible to handle recursive nesting and different term types. Ensuring extensibility was another key challenge; as new IR node types and optimization patterns evolved, the parser had to accommodate these changes without significant rewrites. The use of the visitor pattern was instrumental in overcoming this, as it cleanly separates parsing from IR reconstruction logic and allows node-specific behavior to be encapsulated and extended independently. Throughout development, an iterative improvement approach was adopted. The Prolog representation format, parser grammar, logic, and visitor methods were refined based as more rules were introduced. This approach enabled incremental validation across components, ensuring the overall robustness and maintainability of the parser.

In terms of performance, the impact of Prolog integration varies depending on the nature of the optimization. Stateless rules like those used for canonicalization perform well, with only minimal overhead. This is achieved by using a singleton Projog instance that loads rules once and reuses them across invocations. On the other hand, stateful optimizations such as conditional elimination show significant slowdowns. These optimizations require reinitializing the Prolog engine for each run because of the need to assert and retract dynamic facts such as stamps to track known variable values. As a result, rule loading and engine setup must be repeated frequently, causing noticeable degradation. Furthermore, the Projog engine is relatively slow and could potentially be faster; however, due to time constraints, exploring other Prolog engine options was not possible.

Looking forward, several improvements could be explored. Investigating alternative Prolog engines or approaches might help reduce overhead. Finding ways to express conditional elimination rules in a purely declarative manner without requiring dynamic assertion of facts would simplify the integration and improve performance. Moreover, it would be worthwhile to explore other optimization domains that are particularly well suited to declarative logic, where Prolog's strengths in pattern matching and symbolic reasoning can be leveraged more effectively. Finally, the infrastructure could potentially be reused for other purposes, such as program analysis for security or evaluating the effectiveness of newly proposed optimizations.

7 Conclusions

This project successfully demonstrated the feasibility of integrating Prolog-based optimization techniques into the GraalVM compiler framework. By converting GraalVM's intermediate representation into Prolog terms and expressing optimizations as declarative rules, it was possible to implement a functional optimization pipeline within GraalVM using Projog. The results show that Prolog's declarative syntax offers clear and expressive rule definitions, particularly well suited for stateless optimizations like canonicalization.

However, performance limitations were observed, especially for stateful optimizations such as conditional elimination. These phases suffered from significant overhead due to repeated engine initialization and dynamic fact management in Projog. While stateless optimizations incurred only minimal overhead thanks to reuse of a singleton Prolog instance, stateful optimizations were substantially slower. Additionally, the Projog engine itself proved relatively slow, and time constraints prevented exploring alternative Prolog implementations that might offer better performance.

Despite these challenges, the project highlights the potential of leveraging logic programming for compiler optimizations. The declarative nature of Prolog facilitates reasoning about optimization rules and modularizes the optimization logic. Future work could focus on identifying more performant Prolog engines, devising purely declarative methods for complex transformations like conditional elimination, and exploring other compiler optimization domains that can benefit from Prolog's strengths in symbolic reasoning and pattern matching.

References

[1] 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

[2] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," *ACM SIGPLAN Notices*, vol. 44, pp. 243–262, Oct 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1639949.1640108

[3] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, Jun 2005, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/1065167.1065169

[4] B. J. Webb, I. J. Hayes, and M. Utting, "Verifying term graph optimizations using Isabelle/HOL," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Jan 2023, pp. 320–333. [Online]. Available: https://dl.acm.org/doi/10.1145/3573105.3575673

[5] R. van Tonder, "Towards fully declarative program analysis via source code transformation," *CoRR*, vol. abs/2112.12398, 2021. [Online]. Available: https://doi.org/10.48550/arXiv.2112.12398

[6] 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

[7] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck, "Graal IR: An extensible declarative intermediate representation," in *2nd Asia-Pacific Programming Languages and Compilers*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:52231504

[8] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88*. ACM Press, 1988, pp. 12–27. [Online]. Available: https://dl.acm.org/doi/10.1145/73560.73562

[9] 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/

[10] 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

[11] K. R. Chowdhary, “Logic programming and prolog,” in *Fundamentals of Artificial Intelligence*. New Delhi: Springer India, 2020, ch. 5, pp. 111–141. [Online]. Available: https://doi.org/10.1007/978-81-322-3972-7_5

[12] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Halide: decoupling algorithms from schedules for high-performance image processing,” *Communications of the ACM*, vol. 61, pp. 106–115, Dec 2017. [Online]. Available: https://dl.acm.org/doi/10.1145/3150211

[13] B. Hagedorn, J. Lenfers, T. Koehler, S. Gorlatch, and M. Steuwer, “A language for describing optimization strategies,” *CoRR*, vol. abs/2002.02268, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2002.02268

[14] 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

[15] E. Visser, “Stratego: A language for program transformation based on rewriting strategies system description of Stratego 0.5,” in *Rewriting Techniques and Applications*, A. Middeldorp, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 357–361. [Online]. Available: https://doi.org/10.1007/3-540-45127-7_27