Integrating Prolog into GraalVM Optimization Phases

Tien Nguyen – Supervisors Brae J. Webb, Ian J. Hayes and Mark Utting

Project Objectives

- Integrate a Prolog-based framework into GraalVM.
- Implement optimization rules as Prolog predicates.
- Benchmark Prolog optimizations vs. native GraalVM performance.

Motivation

- Proven effectiveness of logic programming languages in program analysis.
- Limited research on using logic programming language for code optimization and transformation.

GraalVM IR

- GraalVM performs optimizations using its intermediate representation (IR), a directed graph that models data and control flow between program nodes.

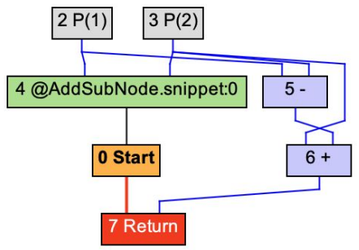

Figure 1: Simple GraalVM IR for (x - y) + y

Implemented Optimizations

- Add node canonicalization.
- And node canonicalization.
- Loop invariant reassociation.
- Conditional elimination.

Outcomes

- Feasibility proven: successfully integrated Prolog into GraalVM using the Projog library.
- Performance trade-offs: slower execution due to Projog's speed and dynamic fact handling.

Method

1. Generating Prolog facts from IR nodes

Extend IR nodes' toString with Verbosity.Prolog to output Prolog facts.

```
@Override
public String toString(Verbosity verbosity) {
   if (verbosity == Verbosity.Prolog) {...}
      else {...}
}
```

Snippet 1: toString method of an IR node

2. Writing Prolog rules

- <optimization>.pl files
- Stateless: canonicalization, loop invariant reassociation
  - purely pattern based, do not modify facts
- Stateful: conditional elimination
  - dynamic creation/retraction of facts (stamps for variable values)

```
% (x - y) + y -> x
canonical(node(addnode, node(Id, subnode, X, Y), Y, Op), Result) :-
   member(associative, Op),
   find_id(X, IdX),
   Result = lookup(IdX).
```

Snippet 2: One canonicalization rule for add node

3. Querying and parsing results

- Use Projog library to integrate Prolog into GraalVM.
- Recursive descent parser processes Prolog query results.
- Grammar of Prolog results:

```
term        ::= lookupTerm | nodeTerm | constExp
constExp    ::= booleanTerm | number
lookupTerm  ::= "lookup" "(" number ")"
nodeTerm    ::= "node" "(" type "," args ")" | "node" "(" type "," number ")"
args        ::= term ("," term)*
type        ::= identifier
```

```
// Lookup existing node in the IR graph
public ValueNode visitLookup(ExpNode.LookupNode lookupNode) {
   return (ValueNode) graph.getNode(lookupNode.id);
}
// Create a new node
public ValueNode visitAddNode(ExpNode.AddNode lookupNode) {
   ValueNode left = node.left.evaluate(this);
   ValueNode right = node.right.evaluate(this);
   return BinaryArithmeticNode.add(left, right);
}
```

Snippet 3: Interpreter for lookup node and add node

Benchmark Results

- Stateless Optimizations: Minimal runtime overhead.
- Stateful Optimizations: Notably slower due to dynamic fact management.
- Metric: operations/sec. One operation includes IR graph construction and applying all optimization phases.

Optimization	Test	Without Prolog	With Prolog
Add node canonicalization	addSub (x - y) + y -> x	3,050	2,938
Loop invariant reassociation	mulMul (i1 * ni) * i2 -> (i1 + i2) * ni	370	334
Conditional elimination	2NestedIfs	1,661	394
	8NestedIfs	517	243

Table 1: Benchmark Results for Optimization Tests (Operations/Sec)

Example

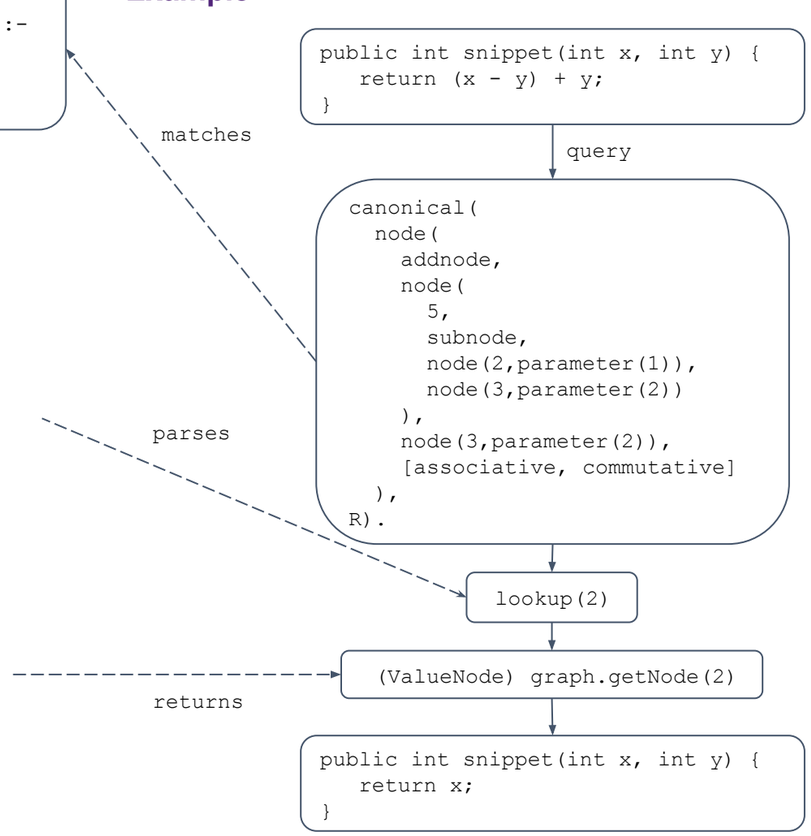

Diagram 1: Prolog-Based Optimization Workflow