CS265 Bril Program Analysis

Ernest Lu

December 2024

A web demo of the project can be accessed here

1 Background

Datalog is a domain-specific declarative language that is used in database applications. A Datalog program consists of a set of initial fact structures along with rules that derive output facts from these input facts. Although this is generally used to derive facts in database applications, the Datalog framework can also be applied to compiler optimizations. A lot of compiler optimizations involve some form of Dataflow analysis, where the framework of getting a set of output facts from a set of input facts may be applied.

2 Datalog Interpreter

First, I implemented a Datalog interpreter to process input facts. In the project, I defined Datalog syntax with the following format:

```
// Datalog file
# edges and reachability queries

.decl edge(x, y) .input;
.decl reachable(x, y) .output;
.rule reachable(x, y) :- 1 edge(x, y);
.rule reachable(x, z) :- 2 reachable(x, y), edge(y, z);
‘
```

Where declarations are specified with the .decl structure, and rules are specified with the .rule structure. In the example above, we have defined an edge declaration and a reachability declaration to be derived from the edge declarations.

We could then associate the above reachability facts with the following input file as such:

```
2
edge(x, y);
edge(y, z);
```

Which should output the following facts:

```
reachable(x, y)
reachable(y, z)
reachable(x, z)
```

This means that y is reachable from z, y is reachable from x, and z is also reachable from x.

We implemented our lexer of these files with Logos, a rust crate for lexers. Datalog evaluation was done under semi-naive evaluation, where we make iterations through rules and generate out output facts until a fixed point. When we consider the declarations that make up a rule, we wish to join the tables for these declaration by their shared keys. The broad goal of doing this is to avoid computing full Cartesian products of tables for declarations.

3 Bril Program Analysis

Dataflow analysis is an intermediate representation analysis that runs until a fixed point. We implemented a liveness analysis that takes in a Bril program, gets a set of liveness facts out of the bril program, then runs a Datalog to determine what variables are live and can be removed. This Datalog framework can also be applied to general dataflow analyses, as Dataflow analysis generates sets of output facts from sets of input facts.

Our liveness analysis is defined with the following facts:

```
# for every node: define:
# edges between nodes
# undefined variable
# variable being used

# edge between two control flow basic blocks
.decl successor(x, y) .input;

# variable is undefined at this basic block
.decl undefined(x, v) .input;

# variable is used at this basic block
.decl var_used(x, v) .input;

# variable is live at this basic block
.decl var_live(x, v) .output;

# we define liveness to occur after the instruction
# rules to define liveness, liveness of successors
.rule var_live(x, v) :- 3 successor(x, y), var_live(y, v), undefined(y, v);
.rule var_live(x, v) :- 2 successor(x, y), var_used(y, v);
```

Where we can infer which variables are live from analyzing a specific point.

For a short example, this Datalog program outputs the following facts:

```
@main() {
  x: int = const 3;
  y: int = const 5;
  y: int = add x y;
  print y;
  x: int = const 4;
}
```

```
Facts:
var_used(default_block_instr_2, x)
var_used(default_block_instr_2, y)
var_used(default_block_instr_3, y)
undefined(default_block_instr_0, y)
undefined(default_block_instr_1, x)
undefined(default_block_instr_2, x)
undefined(default_block_instr_3, x)
undefined(default_block_instr_3, y)
undefined(default_block_instr_4, y)
successor(default_block_instr_3, default_block_instr_4)
successor(default_block_instr_0, default_block_instr_1)
successor(default_block_instr_1, default_block_instr_2)
successor(default_block_instr_2, default_block_instr_3)
```

We can also see that the last line of this statement can be removed because it is unused later. Our liveness derivations should get these variables out as live variables.

```
Live Variables:
var_live(default_block_instr_1, x)
var_live(default_block_instr_2, y)
var_live(default_block_instr_0, x)
var_live(default_block_instr_1, y)
```

This should then output the following program as bril out:

```
@main {
  x: int = const 3;
  y: int = const 5;
  y: int = add x y;
  print y;
}
```

4 Future Work

Although we currently have liveness analysis implemented, the goal is to extend this to support a more dataflow analysis frameworks. Liveness analysis is a specific form of dataflow analysis that focuses on determining which variables are "live" (i.e., potentially used in the future) at various points in a program. This serves as a starting point, but we aim to generalize the approach so that it can handle other types of dataflow analysis.

To achieve this, we would need to define a more flexible framework where different kinds of facts—such as variable reachability, available expressions, or possible values—can be modeled and propagated through the program. Like liveness analysis, these general dataflow analyses would require us to specify a set of initial facts and define rules for how these facts are propagated or derived as we move through the program's control flow.

For instance, in a reachability analysis, we might want to determine which variables can be accessed from a given program point. This analysis would involve defining an initial set of facts (such as the set of variables known to be reachable at the entry points of the program or functions) and then specifying the rules for how reachability is affected by control flow constructs like loops, branches, and function calls. At each program point, we would update the set of reachable variables based on the flow of control and dependencies between different program locations.

To implement this in a more general dataflow framework, we would likely need to adopt a similar structure to the liveness analysis, where facts are propagated forward (or backward, depending on the analysis type) across the program's control flow graph. For each program node (such as a basic block or a statement), we would evaluate how the facts change based on the program's semantics—whether a variable is assigned, used, or passed through a function call, for example.

The flexibility of such a framework could allow us to implement a variety of other dataflow analyses. Some examples include:

Constant Propagation: This analysis seeks to track values that are constant at various points in the program. By propagating constant values through expressions and assignments, we can optimize the program by eliminating unnecessary computations.

Available Expressions: This analysis focuses on identifying expressions that have already been computed and are available for reuse in subsequent program points, thus avoiding redundant calculations.

Reaching Definitions: This tracks which definitions (assignments to variables) reach a particular point in the program, helping to identify potential issues like uninitialized variables or dead code.

May/May-Not Analysis: These analyses track whether a variable might (or might not) be assigned a value at a certain program point, helping to reason about possible execution paths in non-trivial control flow scenarios.

Such analyses could be crucial for static program analysis, where the goal is to reason about the behavior of a program without actually executing it. By leveraging these techniques, we can gain insights into properties like variable usage, memory access patterns, and data dependencies. These insights can in turn drive optimizations like dead code elimination, loop unrolling, constant folding, and more, which can improve the program's performance and reduce its resource consumption.

In summary, while liveness analysis provides one type of insight into program behavior, extending the framework to support general dataflow analyses will enable more sophisticated tools for both static analysis and optimization. The core structure of specifying initial facts and defining propagation rules will remain similar, but the kinds of facts we track and the ways in which they are derived will vary according to the specific goals of each analysis.