Project Summary for CMPSC263 22Fall

Hongbo Wen

December 5, 2022

1 Introduction

Our goal in this project is to build an LLVM IR interpreter and a micro optimizer for the interpretation, based on Racket, Rosette, and E-graph. The meaning of this project is to show the potential ability of the Racket language on symbolic execution/optimization and how to build a prototype workflow fast.

1.1 Terminology

- LLVM IR: A low-level intermediate representation used by the LLVM compiler framework, and become very popular in recent years.
- Racket: Racket is a functional language, the child of Scheme and grandchild of LISP. Different from other languages in the LISP family, it provides features that support building new domain-specific and general-purpose languages.
- Rosette: Rosette is a dialect of Racket, which introduces the theory of vector and solver to achieve program optimization, solving, and synthesis.
- E-graph: Equivalence graphs are a data structure for compactly storing many equivalent expressions, and could be used to find the minimum-cost equivalence representation of an expression.
- Keywords: LLVM;Interpreter;Optimization.

1.2 Prototype

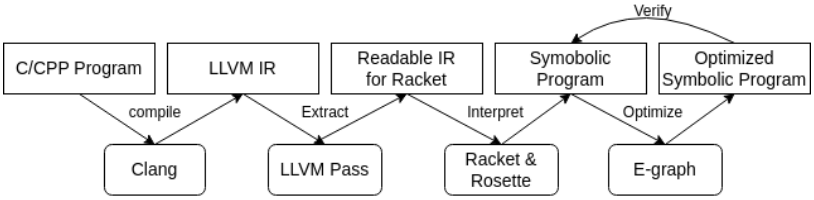

Figure 1: Workflow

This workflow aims to show how our system works:

1. The c/cpp program is compiled to LLVM IR by Clang.
2. The LLVM IR is passed by a pass we developed to generate another JSON file, which is readable for Racket.
3. The IR for the whole program is interpreted as the symbolic program, which concludes paths and terminal values.
4. The symbolic program is sent to the E-graph to find the best optimization of it.

2 Problems and Solutions

2.1 How to make LLVM IR readable for Racket

The first question is how to make LLVM IR readable for Racket and there are three ways to implement it:

1. Write a parser for LLVM IR in Racket.
2. Use LLVM C API by Racket C ffi interface.
3. Use LLVM Pass to extract information and then load it in Racket.

2.1.1 Write a parser

This way is the worst to do in this project. Because LLVM IR has complex grammar, and if we want to parse the IR file, we need to implement the whole parser. Its workload is worthless because our project isn't major aiming at this. Further, it will spend much time debugging the parser.

2.1.2 LLVM C API

This way is useful because we could only implement what we want. However, the LLVM C API is limited. For example, it is hard to extract the type as a string of a value or manipulate a value in LLVM C API. Further, wrapping the interface also requires workload.

2.1.3 LLVM Pass

This way is the most direct way to implement and the JSON format could be a bridge, further, we could do whatever we want because the LLVM pass using LLVM CPP API, LLVM's core system, so there is no worry about any limitation.

2.2 The hash map should support symbolic keys and values

2.2.1 An example to expose the problem

In order to interpret a program, at least we need a variable table, otherwise, we need to implement a register allocation. To simplify the problem, we use a hash map to save the binding from the LLVM value name to its value, represented by a rosette bit vector. However, things will be difficult if we talk about symbolic execution. Let's see an example:

```
int symbolicI32() { return 0; }
int main() {
  int x = symbolicI32();
  if (x >= 128) {
    int z = x + 256;
    return z;
  } else {
    int y = x << 2;
    return y;
  }
}
```

Notice that *symbolicI32* will be translated into a symbolic 32-bit vector. So in this example, when the symbolic value is larger than or equal to 128, we will write z to the variable table. Otherwise, we write y to the variable table. To summarize, we have to write a symbolic key: $if(x \geq 128)\ z\ else\ y$ into the hash map, which isn't supported by the original hash map, because the hash map isn't lifted in Rosette. In Rosette, lifting means the behavior is supported to be used for symbolic values. For example, normal *add* is lifted to support add-on symbolic values and returns another symbolic value. However, the hash map isn't lifted so we don't have the free lunch.

2.2.2 Solution

To lift the hash map simply, we have to support two cases of reading and writing, one is when the key is a symbolic value, and another is when the value is a symbolic value. To save time, I used an implementation from another researcher who also works on Racket. But the key point is how he implemented this feature.

1. Lift the key: So if the key is a symbolic value, such as $if(x \geq 128)\ z\ else\ y$, we will first secure the key, which means if the key doesn't exist, we will insert a void into the key's slot. Then we transfer the condition from the key to the value. Finally, we write the key-value pair to the hash map. Notice the value here is a symbolic value.

2. Lift the value: Now we only have the symbolic value, we just merge them with the current value, and the automatic path merge was happening here. We could also develop dead code elimination in this stage: we could test all of the conditions of values and remove those unsatisfied branches.

So in this case, the final value of z should be: $if(x \geq 128)\ x + 256\ else\ void$.

2.3 SSA format or Load-Store format

Essentially, those two representations have no difference in meaning. But, it is hard to mock the behavior of memory and register directly, so our interpreter only concludes two tables to record the value of each variable. The SSA format is more likely an interpreted one because every instruction could be regarded as a value and we just save it into the local variable table. Also, we could never interpret allocate, load, and store instructions if we use this format. Thus I select the SSA format to be the format we interpret.

2.4 The usage of E-graph implementation on Racket, Regraph

E-graph is a technique that could be used to find the lowest cost expression of the given expression. But the biggest challenge here is not about the algorithm itself, instead, it is how to use its implementation on Racket. So I used a 3rd party package, which is also developed by another group of researchers. However, there isn't an example code in README or documents. Thus, the only way to understand how to use it is to read its source code. I found that, compared with the standard e-graph implementation on Rust, we need to loop the rewrite phase in a loop and decide the bound condition by ourselves. So that's why my program doesn't work. After solving this, I manually loop the rewrite phase 10 times and now the optimizer works.

3 Findings

3.1 Problem Scope

From the start of this project, I'd like to build the workflow on the whole program level. However, as I went deeper and deeper, I found out that actually, this workflow is more suitable for the optimization of a hot basic block or function run-time. In the other words, its scope should be limited to peephole optimization or operator rewriting. The reasons are:

1. We need to mock the behavior of the memory if we want to optimize the whole program, however, symbolic execution on the memory takes much time and is too complex.
2. The overhead of verifying is high, which means we can't afford the time cost of a huge program snippet run-time.
3. The e-graph works well if the given expression is small, however, its efficiency becomes unstable as the expression goes bigger.
4. The major advantage of Racket is to express the instructions as an expression, it is very like the behavior of a peephole-level code. However, for the whole program, analysis of the control flow will be really hard for Racket's representation, which means it is hard to do optimization concerned about the control flow.

To summarize, this workflow could be used for the whole program scope when offline. On the contrary, when online it should be only used for peephole optimization of the program snippet which is really important, such as critical operators, content of large loop, or float point computing which requires really precise.

3.2 Applied Optimizations

We have the implementation three kinds of classical optimization. And the mechanism of how they work is a key gain from my aspect.

3.2.1 Constant Folding

Assume we have an expression as: $(+\ (+\ a\ 1)\ 2)$, essentially, it has a better representation as $(+\ a\ 3)$. The merge of 1 *and* 2 is what we called *constant folding*. Let's provide three rewrite rules first:

1. $(+\ a\ b) \rightarrow (+\ b\ a)$
2. $(+\ a\ (+\ b\ c)) => (+\ c\ (+\ a\ b))$;
3. $(+\ 1\ 2) => 3$

So the problem is, the constant could be infinite, we can't list all of the rewritten rules of constants. But, the stuff we need here, is to compute instead of rewrite because we could easily compute $(+\ 1\ 2)$! We will use the *precompute* phase provided by the Regraph. In this phase, all of the functions whose arguments are constants will be matched, and those functions will be computed immediately. Thus, in this case, the $(+\ 1\ 2)$ will be computed to 3 immediately, this phase also works for other operators and other constants. That's how constant folding works in e-graph.

3.2.2 Strengthen Reducing

In e-graph, the optimization could be formalized as: find the lowest cost of expression under given rewrite rules. The cost of an expression is defined as: An atomic expression is counted as 1, such as a constant or symbolic value, and a function is counted as 1 + the sum of the cost of its arguments. The e-graph tries to find all of the possible rewrites of the given expression and finally extracts the lowest-cost expression. Let's see an example to know the mechanism better:

```
int symbolicI32() { return 0; }
int main() {
  int x = symbolicI32();
  int y = symbolicI32();
  int i = x * y;
  int j = y * x;
  int k = 2 * y * x;
  int z = i + k + j;
  return z;
}
```

In this example, finally, we will get $(* \ 4 \ (* \ x \ y))$. But during the iteration, we will get:

1. Loop 1 $(+ \ (* \ x \ (* \ 2 \ y)) \ (<< \ (* \ x \ y) \ 1)$
2. Loop 2 $(+ \ (* \ x \ y) \ (* \ 3 \ (* \ x \ y)))$
3. Loop 3 $(* \ 4 \ (* \ x \ y))$

The main idea of E-graph is to avoid local optimum, such as the result of loop 1. Because rewriting rules depend on the format of an expression. And the Regraph provides a compact representation of those rewriting to avoid an exponential explosion of the search space.

3.2.3 Path Pruning

Section 2.2 has described how the path pruning works, thus, we will not repeat it here.

4 Summarize

In this project, I faced many problems and solved them, getting familiar with the tools and techniques of the workflow, the limitation of this project, and its suitable scope. It is a little pity that it is hard to be applied on a runtime system directly.