MLEgg: Applying Equality Saturation to LLVM

COMP 400 Research Project, advised by Christophe Dubach

KATIE LIN, McGill University, Canada

Compiler optimization is an important job; improvements in compiler efficiency are themselves the most efficient way to optimize code, as they will impact all code generated by it. However, this is not an easy task; generating optimized code is significantly more difficult than just translating one-to-one. Specialized intermediate representations, like allowed by MLIR, can help by allowing different paradigms that programmers use to be preserved and represented. Effectively using these IRs to optimize code, though, is oftentimes a very manual process.

Traditional optimization techniques often rely on hand-written code and deal with only one statement at a time. Dialects of MLIR can help with capturing different paradigms however, oftentimes these paradigms are not transparently represented. Equality saturation, a technique that represents multiple versions of a program simultaneously, solves this issue. By programatically generating and representing equivalent statements, it efficiently optimizes generated code.

This paper describes a proof-of-concept implementation of an MLIR pass that applies equality saturation to a simple program. It is able to parse and interpret instructions, calling an efficient equality saturation runner to generate alternate, more optimized, representations of the program structure. This allows optimizations to be applied automatically and without ordering constraints.

1 INTRODUCTION

Compilers, with a job of translating high-level written code into low-level machine-executable code, have a difficult task. There are lots of languages, and lots of machines, and writing direct adapters would be obscenely time consuming. The LLVM project and its common intermediate representation (or IR) simplifies this issue by creating a common language to translate from and to. This allows for new hardware to be easily adapted and optimized for, and for those benefits to carry across multiple languages.

MLIR is an generalization of the LLVM IR. It extends the instructions in the LLVM IR, allowing for more complex paradigms to be reprsented. For example, loops can be represented directly, instead of as `goto`/`break` combos. In addition, with this extra information, specialized instructions can be genrated for specialized hardware.

Compiler optimization is often a very manual process. Optimizations have to be manually observed by field experts, and contexts wherein they can apply are limited. Verification that they work in all required contexts can range from tedious and error-prone to straight up impossible.

One of the largest challenges when optimizing code is ordering. Because code is rewritten in-place, the prior state of the code is often lost during an optimization. This leads to optimizations that need to be very carefully ordered, as otherwise local optimizations that seem like the best decision can lead to larger optimizations being unable to be applied.

Equality saturation allows for multiple versions of a program to exist simultaneously. The most efficient version of it, determined heuristically, can then be chosen and extracted. Because multiple optimizations of a program can be applied effectively at once, the most optimal ordering of optimizations can be found and applied automatically[1].

This paper shows how this technique can be applied to MLIR. A subset of the Linalg dialect of MLIR, focused on representing linear algebra, is

Author's address: Katie Lin, McGill University, School of Computer Science, Montréal, QC, Canada, katie.lin1@mail.mcgill.ca.

described rule-wise in an equality saturation framework. Programs are assessed and filtered by basic block, and optimizations applied.

This proof-of-concept successfully demonstrates that MLIR instructions can be interpreted and passed to an equality saturation framework. Round-tripping a program in this fashion additionally allows optimizations to be performed on a variable number of statements at once.

2 BACKGROUND AND MOTIVATION

2.1 MLIR and LLVM

LLVM defines an IR that it knows how to translate to machine code. MLIR takes advantage of LLVM's ability to convert its IR to machine code by building *dialects* on top of it that can compile down to LLVM IR. Multiple of these dialects can coexist within a single MLIR program, allowing for rewritings and *lowerings* to affect only parts of a program at once. After MLIR code is generated by a compiler, multiple *passes* are run on it to lower it to LLVM IR.

Dialects often represent higher-level concepts than pure machine code can express. For instance, dialects such as `loop` and `func` represent loops and function calls. Other dialects can represent more abstract concepts, like `tensors`, which are multiple-dimension collections.

Multiple dialects can represent the same thing, at different levels of abstraction. For example, the aforementioned `tensor` can also be represented, more concretely, by instructions in the `affine` dialect. Even more concretely, the multiple-dimension collection can be backed by an actual memory-mapped array.

2.2 Equality Saturation and egg

Equivalence graphs (or *e-graphs*) are the main driving force behind equality saturation. E-graphs are a cheap way of representing equivalence relations across many expressions [3]. By storing classes of operations, and building relations between them, equivalences can be stored much more efficiently than relations on individual expressions. The recursive nature of e-graphs also allows a potentially infinite amount of equivalences to be stored in a finite manner.

However, especially at a larger scale, equality saturation can be expensive. Even with the space-saving nature of e-graphs, saturating one is still an operation that grows exponentially. Oftentimes in practice on larger datasets, equality saturation is only run for a certain (single-digit) number of steps, as doing more would be cost-prohibitive.

The egg library is an implementation of equality saturation in Rust. Given a defined language and a defined set of rules that operate on the language, it is able to take an arbitrary expression and cost function and optimize it. One of `egg`'s largest improvements on traditional e-graphs is changing the invariant to only hold after a *rebuilding* operation. Relaxing the invariant allows insertion operations to be more efficient.

2.3 Motivation

Equality saturation was built to drive compiler optimization [3]. Much research has been done into applying equality saturation techniques to modern programming languages, especially in the field of idiom recognition [2]. However, these examples have been focused on optimizing high-level languages, or languages with easily-defined semantics.

Optimizing a lower-level representation, like LLVM IR, would allow for these benefits to apply in a language-agnostic manner. This optimization would be much harder, as idioms wouldn't be nearly as easily recognizable, and information is lost in the lowering process. MLIR splits the difference by allowing for specificity and retained information while still maintaining the interoperability benefits.

3 OVERVIEW

The project pairs MLIR and egg to apply equality saturation to an IR directly—in this case, the Linalg dialect. It acts as an MLIR pass, adapting

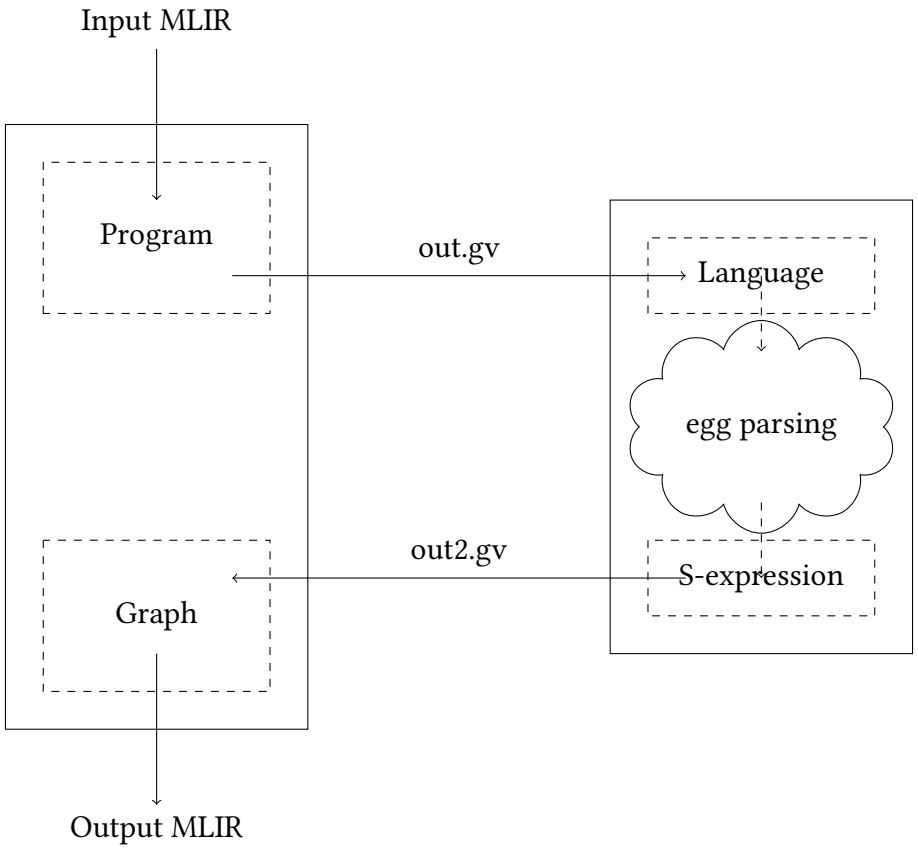

Fig. 1. An overview of the structure of MLEgg.

the instructions found within to an egg expression and using the tools that library provides to perform equality saturation. It then adapts in reverse, bringing that result back to the MLIR pass and applying it.

4 IMPLEMENTATION

The program is effectively implemented as a series of translation layers between MLIR code and the egg equality saturation library. This allows for maximum extensibility while not duplicating prior work done on other libraries. Due to language differences and to aid flexibility in development, the project is implemented in two parts: a program which runs the MLIR pass and a sub-program which takes a simple graph and runs equality saturation on it, outputting a new graph with references to the original. Lastly, because expressions are most easily represented as discrete acyclic graphs, the Dot language is used as an interchange format to pass between the two parts.

4.1 MLIR Pass

Like any other MLIR pass, MLEgg takes in a programmatic representation of the MLIR code as it currently exists in the lowering process and must return the same. Filtering for only operations that the program has rules for is simple, using the MLIR API. In order to guarantee that branching and loop instructions won't affect the equality saturation process, this filtering process is done per basic block.

With a filtered list of operations to consider, the program generates a directed acyclic graph showing parent-child relationships between every operation and every value each operation takes as input. This heredity graph is equivalent to the input expression to the equality saturation runner, but it's trapped on the wrong side of the program-subprogram wall.

4.2 Translating with Dot

In order to pass the graph to egg, the MLIR pass writes it to a file and calls the equality saturation subprocess, telling it to look for the file as input. The subprogram then parses the input file and recreates the graph. Using the information given by the graph, the subprogram builds an egg `RecExpr`.

Instead of using inter-process communication, or writing the entire program as one overarching unit, the program is developed in two components in two languages. This simplifies the development process by removing a layer of complexity. Rust and C++ are interoperable, but using standard Unix file manipulation tools allows for transparency during the interprocess portion.

Dot is a simple and functional file format for expressing graphs. Because of its heavy usage, it is well-documented and can serve as an easy visual debug tool.

The graph interpreter used to read the Dot file is nontrivial, as it needs to account for the possibility that there is more than one root node. Because the built graph data structure does not have references to parent nodes, it creates a unique set of all nodes that are children of other nodes and then finds the difference between that and the entire set of nodes. This creates the set of nodes that are not the children of any other nodes; or, in other words, the set of root nodes.

4.3 Using egg

Most of the program is aimed around wrapping MLIR statements in such a way that they can be interpreted by egg. The most crucial step is building the input expression. Because egg needs a set of known operators to operate on, the input expression must be defined in terms of a `Language` – a user-defined enum that contains a set of operations that rules can refer to. Thus, in order to properly convert the input DAG to an input expression, the DAG must be parsed and re-expressed within the `Language`.

While parsing, in order to simplify the reconciliation process later on, the original node ID is preserved within the data string.

Once the expression is built, the runner can be called with the expression and predefined rules as inputs. The outputted fully-saturated graph can have a cost function applied – in this case, one minimizing node count – to extract the most efficient expression from it. This expression serves as the output which should be represented by the program.

4.4 Translating back with Dot

Due to limitations in the egg library, this expression is only representable as a stringified s-expression. This s-expression must then be manually parsed, generating another DAG which can then be output to the main program.

Because of the ubiquity of the Dot format, the egg library actually includes a Dot printer for an arbitrary expression. However, the printer is aimed at pretty-printing an output for explanatory reasons, and thus has many extraneous nodes. Since this project only uses a subset of the Dot language for simplicity, it is easier to write a bespoke graph traversal and printing method.

4.5 Transforming MLIR

With the transformed graph read back into the MLIR pass, the actual instructions need to be modified to match the new state. The new graph contains information about what operations should exist and which values they should operate on. These values either exist within the current set of instructions

already, or are the result of another new operation generated by the equality saturation process.

The new instructions are generated at the end of the original basic block. This maintains the guarantee that the original nodes that the new instructions reference will be instantiated when the new instructions need them.

Finally, as all the original instructions have been replaced, they can be deleted.

5 EVALUATION

A qualitative evaluation of MLEgg follows, demonstrated using the following basic MLIR program.

```
module {
    %lhs = tensor.empty() : tensor<3x2xf32>
    %rhs = tensor.empty() : tensor<2x4xf32>
    %init = tensor.empty() : tensor<3x4xf32>
    %matmul = linalg.matmul ins(%lhs, %rhs:
        tensor<3x2xf32>, tensor<2x4xf32>)
        outs(%init: tensor<3x4xf32>) ->
        tensor<3x4xf32>
    %new = tensor.empty() : tensor<4x3xf32>
    %init2 = tensor.empty() : tensor<3x3xf32>

    %next = linalg.matmul ins(%matmul, %new:
         tensor<3x4xf32>, tensor<4x3xf32>)
        outs(%init2: tensor<3x3xf32>) ->
        tensor<3x3xf32>
}
```

Across each program, the program guarantees that it can run, then recurses and processes each basic block. Within the example program, there is only one basic block – it considers the entire program at once.

The program then filters out all the instructions in the basic block that it doesn't have rules for. With the remaining instructions, every node is given an ID, and a graph is generated showing heredity relationships between instructions and arguments. For instance, in the example program, only the two `linalg.matmul` instructions are considered. The heredity graph shows that the instruction labeled `%next` depends on the instruction labeled `%matmul`, as follows.

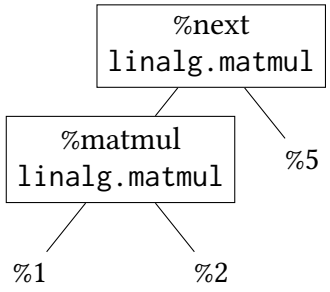

Fig. 2. Heredity graph

This graph is then written to a file and passed to the equality saturation sub-program. The sub-program reads the graph and converts it to the s-expression in the requisite language that the egg library expects as input. In this example, with the language represented using an identical string as MLIR, the s-expression looks as follows.

```
(linalg.matmul (linalg.matmul 1 2) 5)
```

Fig. 3. Input s-expression

The program then saturates the graph using the provided rules. From the fully saturated graph, a new expression is extracted, using a cost function optimizing for node count. In this example, the extracted s-expression is slightly different from the input expression.

```
(linalg.matmul 1 (linalg.matmul 2 5))
```

Fig. 4. Extracted s-expression

This s-expression is then parsed, written out, and returned to the MLIR pass. The MLIR pass uses the new heredity graph to create new instructions for the program, based on the child instructions referencing the previous graph. The new instructions are created at the end of the basic block, so as to

not interfere with preexisting instructions. Finally, the old instructions are deleted, leaving just the new instructions.

```
module {
    %1 = tensor.empty() : tensor<3x2xf32>
    %2 = tensor.empty() : tensor<2x4xf32>
    %init = tensor.empty() : tensor<2x3xf32>
    %4 = tensor.empty() : tensor<4x3xf32>
    %init2 = tensor.empty() : tensor<3x3xf32>

    %5 = linalg.matmul ins(%2, %4: tensor<2
        x4xf32>, tensor<4x3xf32>) outs(%init:
         tensor<2x3xf32>) -> tensor<2x3xf32>
    %3 = linalg.matmul ins(%1, %5: tensor<3
        x2xf32>, tensor<2x3xf32>) outs(%init2
        : tensor<3x3xf32>) -> tensor<3x3xf32>
}
```

Note that in the worked example above, the two instructions were indeed successfully swapped.

6 CONCLUSION

This paper has shown a working example of a program applying equality saturation to an MLIR dialect. The proof of concept can be extended, given a larger set of rules, to other operations and other dialects.

REFERENCES

[1] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '09). Association for Computing Machinery, Savannah, GA, USA, 264–276. ISBN: 9781605583792. DOI: 10.1145/1480881.1480915.

[2] Jonathan Van Der Cruysse and Christophe Dubach. 2024. Latent idiom recognition for a minimalist functional array language using equality saturation. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 270–282. DOI: 10.1109/CGO57630.2024.10444879.

[3] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5, POPL, Article 23, (Jan. 2021), 29 pages. DOI: 10.1145/3434304.