

# eqsat: An Equality Saturation Dialect for Non-destructive Rewriting

Jules Merckx  
jules.merckx@ugent.be  
Ghent University  
Ghent, Belgium

Jianyi Cheng  
University of Edinburgh  
Edinburgh, UK

Alexandre Lopoukhine  
University of Cambridge  
Cambridge, UK

Bjorn De Sutter  
Ghent University  
Ghent, Belgium

Samuel Coward  
University of Cambridge  
Cambridge, UK

Tobias Grosser  
University of Cambridge  
Cambridge, UK

## Abstract

With recent algorithmic improvements and easy-to-use libraries, equality saturation is being picked up for hardware design, program synthesis, theorem proving, program optimization, and more. Existing work on using equality saturation for program optimization makes use of external equality saturation libraries such as egg, typically generating a single optimized expression. In the context of a compiler, such an approach uses equality saturation to replace a small number of passes. In this work, we propose an alternative approach that represents equality saturation natively in the compiler’s intermediate representation, facilitating the application of constructive compiler passes that maintain the e-graph state throughout the compilation flow. We take LLVM’s MLIR framework and propose a new MLIR dialect named eqsat that represents e-graphs in MLIR code. This not only provides opportunities to rethink e-matching and extraction techniques by orchestrating existing MLIR passes, such as common subexpression elimination, but also avoids translation overhead between the chosen e-graph library and MLIR. Our eqsat intermediate representation (IR) allows programmers to apply equality saturation on arbitrary domain-specific IRs using the same flow as other compiler transformations in MLIR.

## 1 Introduction

To date equality saturation has largely been used outside the compilation flow or has replaced a single compiler pass, with just one work exploring a deep integration [2]. Most works that leverage equality saturation for program optimization develop custom tools [7, 16] built on top of existing equality saturation libraries [20, 22]. More recent work has now started to explore the integration of equality saturation in general-purpose compiler frameworks [5, 21]. These approaches develop an extensible translation layer between the compiler ecosystem [12] and existing equality saturation library implementations [20, 22]. While such an approach offers improved support for non-destructive rewriting of

---

**Listing 1** Python code where subsequent calls to `softmax` and `log` in `model_forward` are hidden behind call barriers, preventing a rewrite from taking place.

---

```
def normalize_probs(logits):
    return softmax(logits, dim=-1)
def compute_log_probs(probs):
    return log(probs)

def model_forward(logits):
    probs = normalize_probs(logits)
    log_probs = compute_log_probs(probs)
    return log_probs
```

---

intermediate representation (IR), it does not fully bridge the gap between equality saturation and compilers. For one, supporting new IR primitives requires additional labor in extending the translation layer between the tools [21]. More importantly, jumping between compiler and external equality saturation library hampers the ability to keep track of equality information as other compiler passes are applied.

Take for example the combination of equality saturation with function call inlining, where the compiler replaces a function call by the code of the function body itself. High-level functions used by programmers abstract a lot of functionality, and can give rise to interesting rewrite opportunities. Some function calls can for example be rewritten to calls to faster, or more precise implementations. An example often encountered in code using deep learning libraries, is that of `logsoftmax`, where a call to a softmax function followed by a call to logarithm can be replaced by a call to the more numerically stable `logsoftmax` operation.

`call(log, call(softmax, x)) → call(logsoftmax, x)`

Existing equality saturation techniques operate on a single function body at a time, potentially with some calls already inlined. In general, however, there is no guarantee that the correct inlining has been applied to expose rewriting opportunities. As an example (Listing 1), a function `model_forward` might subsequently call `normalize_probs` and `compute_log_probs` that in turn call `softmax` and `log`,

respectively. Since those function calls are not inlined, the rewrite opportunity cannot be exploited. Moreover, inlining a function can lead to *less* rewrite opportunities. When `normalize_probs` and `compute_log_probs` are inlined, the rewrite can occur, but if one of both `softmax` or `log` is inlined *as well*, the rewrite opportunity vanishes. In essence, function inlining is subject to a phase-ordering problem similar to the one solved by equality saturation for rewriting. By bringing equality information from term rewriting to other compiler passes such as function call inlining, new rewriting opportunities are revealed.

In this work, we bring first-class equality saturation to IRs based on static single assignment (SSA). Our native implementation of equality saturation in an existing compiler ecosystem maximizes reuse and facilitates switching between equality saturation and destructive rewriting with just one additional compiler pass. While this flexibility may come at the expense of some performance, specifically for e-matching and congruence closure, in practice, e-graph rewriting often does not dominate the overall runtime. For example, common sub-expression aware extraction methods that utilize integer linear program solving can often dominate [8]. To represent e-graphs in MLIR, we introduce a new MLIR dialect named eqsat, that directly interfaces with existing MLIR dialects. This enables the off-the-shelf reuse of existing compiler passes to implement some of equality saturation’s core algorithms, such as congruence closure.

Our contributions are:

- expressing equality saturation directly in a compiler’s IR via a new compiler IR,
- mapping of equality saturation concepts to existing compiler concepts,
- a framework that maintains the e-graph state across compiler transformations, and
- a prototypical open-source implementation of the proposed approach in xDSL, a Python-Native single static assignment-based compiler closely mirroring MLIR.

## 2 Background

Our work takes inspiration from recent developments in equality saturation and builds on compiler infrastructure used in production systems. We leverage existing rewrite patterns with mutating semantics, and instead apply them non-destructively.

### 2.1 Equality Saturation

Equality saturation is a rewriting technique that applies non-destructive rewrites by keeping track of the original expression alongside transformed ones [18, 20]. At the heart of most equality saturation libraries is the e-graph datastructure that consists of e-classes, each a collection of e-nodes [14]. Each e-node in a particular e-class represents a function (or literal) that is equivalent to the others. During equality saturation,

rewrite patterns are matched against the e-graph. When a match is found, instead of destructively rewriting a term, the new, equivalent term is added as an e-node in the e-class of the original term. By applying rewrites non-destructively, there is no risk of running into the phase-ordering problem, where one rewrite renders more interesting rewrites impossible. In order to efficiently reason over equalities, the e-graph maintains an equality closure under congruence, meaning that different applications of a function on equivalent terms, yield equivalent terms.

Three prior works have combined equality saturation with a mature compiler framework. The SEER project [5], specifically targeted high-level synthesis, optimizing System-C programs using a combination of high-level software rewrites and low-level circuit rewrites. The second, more general work [21], developed a framework for representing any internal MLIR dialect in an e-graph, allowing users to define their own equality saturation rewrites. Both works leveraged existing equality saturation libraries, adding translation layers between the two domains.

In contrast, Cranelift [2], a mature optimizing compiler and code generator, developed an equality saturation optimizer that reuses much of the infrastructure of their IR. This is similar to the approach we present in this paper, so we provide a thorough comparison in Section 7.

### 2.2 Static Single Assignment IR with Regions

SSA is a property of IRs that guarantees that a value is defined exactly once. Modern compilers [1, 11] leverage this property to simplify analysis and transformations during compilation. Values are defined as the results of *operations*, or given as arguments to *blocks*. Operations represent runtime information, such as integer addition, taking a variable number of values as *operands*, and returning a variable number of values as *results*. Blocks group together a number of operations, either encoding a sequence of operations to be run one after the other, or a cyclic graph of relationships between values with no explicit control flow. In contrast to textbook SSA implementations, which represent values passed into a block from divergent control flow using phi nodes, more recent compilers [2, 3, 12] instead use block *arguments*. The flexibility of this data structure has led to its widespread use in modern compilers spanning applications from tensor programs [4, 19] to digital circuits [15].

We use xDSL [10] to implement equality saturation using SSA constructs as defined in MLIR [12]. In MLIR, blocks are nested within *regions*, which are in turn nested within *operations*, allowing users to represent recursive structures of arbitrary depth. Operation definitions are grouped into user-provided *dialects*, which serve as a name space for related definitions. In order to provide reusable infrastructure for user-provided constructs, MLIR leverages *interfaces* and *traits* to encode properties and behaviors of operations, to be

leveraged by transformations such as dead code elimination or common subexpression elimination (CSE).

**The Pattern Description Language dialect.** The MLIR project contains a number of meta-dialects, such as pdl, which encodes definitions of rewrites on MLIR operations, and pdl\_interp, comprising the actions taken by a state machine when executing the rewrite. Rewrites defined in pdl are composed of two parts, the first a declarative matching pattern, and the second an imperative rewrite procedure. A number of these patterns can be lowered together into a single unified state machine, to be executed by an accompanying interpreter defined in MLIR. The patterns are defined to be applied destructively, meaning that a matched pattern will replace some of the existing IR.

### 3 E-Graphs as a Compiler IR

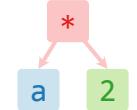
To make equality saturation dynamically available throughout the compilation process, we introduce IR primitives for modeling the data structures at the core of equality saturation. Our primitives are a set of SSA operations implemented in MLIR. Together, they form our new eqsat dialect.

As an example, we consider the IR of the function  $a * 2$  (Listing 2 - top), which takes one argument and multiplies it by two. We can express this function as a simple expression tree consisting of a multiplication node with two children,  $a$  and  $2$ . We can then turn this IR into a trivial e-graph (where each class has just one element) by introducing equality class operations for each result (Listing 2 - bottom). In particular, we introduce three new eqsat.**eclass** operations that use the argument  $\%a$  as well as the results  $\%two$  and  $\%res$  and return corresponding e-classes, each containing exactly one element. We subsequently update the compute operation arith.**muli** to use the newly created equality class result values as its inputs. As the e-graph is being rewritten, circular references may be introduced, which may be disallowed by the parent operation. To preserve correctness, we embed our e-graph in an eqsat.**egraph** operation, which encapsulates the e-graph, preserving the validity of the rest of the IR. The resulting IR models an e-graph (at the right) with three equality classes, each visualized as a dotted frame.

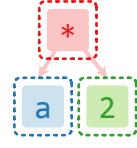
We can now introduce new equalities into our IR. For example, the multiplication in our program is equal to a more efficient left shift. After applying the corresponding rewrite  $x * 2 \rightarrow x << 1$  to our IR (Listing 3), a new bitshift operation as well as the necessary constant operation have been added to the code, and the result of the bitshift has been added as an operand to the same eqsat.**eclass** operation that already referenced the multiplication result. This new IR now corresponds to an e-graph where the results of multiplication and left shift are members of the same e-class (Listing 3 - right). While visualized differently, the reader may observe that each eqsat.**eclass** operation corresponds to exactly one e-class in the egraph, while the edges of the e-graph correspond

**Listing 2** We implemented a compiler pass that inserts operations from our eqsat dialect in order to represent e-graphs within MLIR code directly.

```
func.func @f(%a : i64) -> i64 {
    %two = arith.constant 2 : i64
    %res = arith.muli %a, %two : i64
    func.return %res : i64
}
```

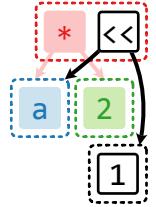


```
func.func @f(%a : i64) -> i64 {
    %two = arith.constant 2 : i64
    %graph_res = eqsat.egraph -> i64 {
        %c_two = eqsat.eclass %two : i64
        %c_a = eqsat.eclass %a : i64
        %res = arith.muli %c_a, %c_two : i64
        %c_res = eqsat.eclass %res : i64
        eqsat.yield %c_res : i64
    }
    func.return %graph_res : i64
}
```



**Listing 3** Contents of the eqsat.**egraph** operation from Listing 2, and corresponding e-graph visualization, after the rewrite  $x * 2 \rightarrow x << 1$  has been applied.

```
%one = arith.constant 1
%c_one = eqsat.eclass %one
%c_two = eqsat.eclass %two
%c_a = eqsat.eclass %a
%res = arith.muli %c_a, %c_two
%res1 = arith.shli %c_a, %c_one
%c_res = eqsat.eclass %res, %res1
eqsat.yield %c_res
```

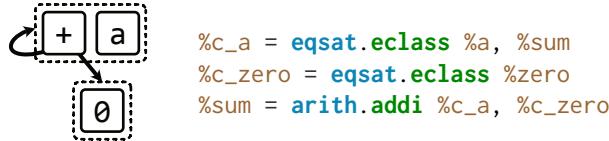


to use-def edges that connect the results of eqsat.**eclass** operations with the arithmetic operations arith.**muli** and arith.**shli**. Hence, an e-graph can be trivially embedded into an SSA-based compiler IR.

Concretely, our eqsat dialect consists of three operations which in combination with the use-def information offered by an SSA-based compiler IR are sufficient to represent e-graphs and carry out equality saturation:

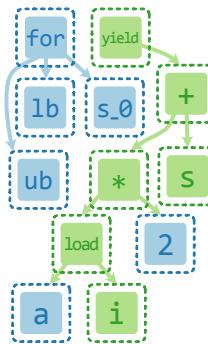
- The eqsat.**eclass** operation takes one or more values (analogous to e-nodes), and produces a single result.
- The eqsat.**egraph** operation encompasses a piece of code on which equality saturation can be executed. All eqsat.**eclass** operations must be contained within an eqsat.**egraph** operation. Operations within this operation's region can access values defined outside of it, but not the other way around.
- The eqsat.**yield** operation is a terminator that closes off an e-graph. It takes as operands the eqsat.**eclass** results that are exposed by the eqsat.**egraph** operation to the rest of the program.

**Listing 4** Cycles in e-graphs can be represented compactly in MLIR IR by using graph-regions.



**Listing 5** Example of an `scf.for` operation, and the corresponding e-graph. `eqsat.eclass` operations have been left out in this example for clarity. Operations *inside* the loop body can access values from *outside*.

```
%s = scf.for %i = %lb to %ub
  iter_args(%s = %s_0) {
    %x = memref.load %a[%i]
    %term = arith.mulf %x, %two
    %s_new = arith.addf %s, %term
    scf.yield %s_new
}
```



**Cycles.** Depending on the equality rules used during equality saturation, cycles can appear in the e-graph. For example, applying the rule  $a + 0 \rightarrow a$  on an e-graph containing the expression  $a + 0$  introduces a cycle as illustrated in Listing 4. In typical SSA-based IRs, value uses can only occur after their definition. MLIR, however, supports the concept of graph regions, where this restriction is lifted. The region encompassed by an `eqsat.egraph` operation is such a graph region, allowing cycles in the use-def chain to occur.

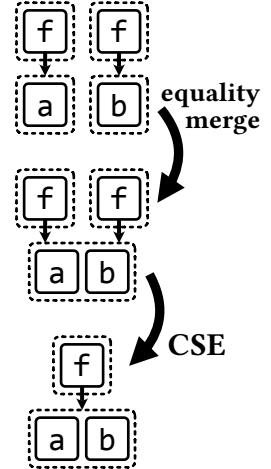
**Control Flow.** By virtue of MLIR’s region-based IR, there are dialects that can be used to represent control flow such as if-else-statements or for loops in a structured manner. For example through the use of the `scf` dialect, which offers `scf.for`, `scf.if`, and other operations. In contrast to simple arithmetic operations, these control flow operations carry a region containing the control flow body, allowing the IRs to express programs with control flow without the use of basic blocks and phi-nodes. The presence of these nested control flow regions does not hinder equality saturation but rather allows rewrites to naturally occur across control flow. For example, values defined outside of a for loop are still accessible within the loop while the inverse is not true (Listing 5). Simply inspecting the graphical e-graph structure does not reveal these scope constraints, they are encoded only in the IR.

**Listing 6** By moving equality saturation primitives into the compiler, CSE subsumes egraph rebuilding.

```
%c_a = eqsat.eclass %a, %sum
%c_b = eqsat.eclass %b
%res0 = test.f(%c_a)
%res1 = test.f(%c_b)
```

```
%c_ab = eqsat.eclass %a, %b
%res0 = test.f(%c_ab)
%res1 = test.f(%c_ab)
```

```
%c_ab = eqsat.eclass %a, %b
%res = test.f(%c_ab)
```



## 4 Rebuilding

At the core of equality saturation is the congruence invariant that ensures that all equalities in an e-graph are closed under congruence. To conserve this invariant more efficiently, Willsey et al. [20] proposed an explicit e-graph rebuilding step, where the merging of parent e-classes is carried out for all the new equalities that have been inserted after matching all patterns, instead of immediately after each single insertion.

We make the observation that, by embedding e-classes as operations in an SSA-based IR, applying CSE maintains this invariant as well. As an example, take a function `f` being applied twice with different e-class operands containing `a` and `b`, respectively (Listing 6). When an equality  $a \leftrightarrow b$  is introduced, the equality saturation pass merges the equivalent e-classes. All uses of the original e-classes now use this merged e-class instead. At this point, the congruence invariant is not satisfied, as the two e-classes containing `f` are equivalent because they refer to the same e-class operand. Applying CSE on the MLIR code removes the duplicated application of `f`, thus restoring the invariant.

By utilizing a standard CSE pass as provided by MLIR, the implementation of equality saturation is made considerably simpler. Importantly, the CSE pass itself was implemented without equality saturation in mind. Compared to the full e-graph rebuilding algorithm, however, CSE can be less efficient as it is applied on the whole e-graph instead of incrementally, on newly introduced equalities. In the future, it might be possible to leverage incremental CSE<sup>1</sup> for more efficient e-graph rebuilding.

<sup>1</sup>[https://mlir.llvm.org/doxygen/CSE\\_8cpp\\_source.html](https://mlir.llvm.org/doxygen/CSE_8cpp_source.html)

**Listing 7** A declarative rewrite pattern written in MLIR’s pdl dialect to rewrite  $a + 0$  into  $a$ .

```
pdl.pattern : benefit(1) {
  %0 = pdl.type
  %a = pdl.operand
  %2 = pdl.attribute = 0: i32
  %3 = pdl.operation "arith.constant" {"value"=%2}
    -> (%0: !pdl.type)
  %zero = pdl.result 0 of %3
  %5 = pdl.operation "arith.addi"(%
    %a, %zero: !pdl.value, !pdl.value
  ) -> (%0: !pdl.type)
  pdl.rewrite %5 {
    pdl.replace %5 with (%a: !pdl.value)
  }
}
```

**Listing 8** Part of the result after lowering the pdl rewrite pattern from Listing 7 to the pdl\_interp dialect. This imperative code contains simple instructions and primitive control flow to match one or more patterns.

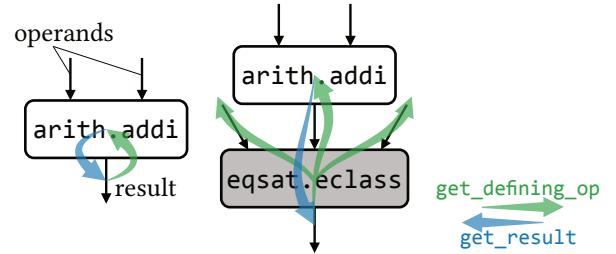
```
pdl_interp.func @matcher(%arg0: !pdl.operation) {
  %0 = pdl_interp.get_operand 1 of %arg0
  %1 = pdl_interp.get_defining_op of %0 : !pdl.value
  pdl_interp.is_not_null %1 : !pdl.operation -> ^bb2, ^bb1
^bb1:
  pdl_interp.finalize
^bb2:
  pdl_interp.check_operation_name of %arg0 is "arith.addi"
    -> ^bb3, ^bb1
^bb3:
  pdl_interp.check_operand_count of %arg0 is 2 -> ^bb4, ^bb1
  :
}
```

## 5 Grafting E-Matching

In order to find and apply rewrites on an e-graph, e-matching needs to be performed. Here, the e-graph is searched, verifying if any of the provided patterns are matched in the equivalence structure. With the pdl dialect it is possible to describe complex patterns. Static types of operation results and operands can be matched in order to describe complex type constraints (Listing 7). Additionally, *multi-patterns*, where multiple expressions are matched and rewritten together, are trivially expressed in pdl’s declarative format.

Typically, pdl rewrite patterns are first lowered into operations from the pdl\_interp dialect. This dialect consists of lower level, imperative matching operations, making it simpler for an interpreter to match and rewrite. Listing 8 shows the pdl\_interp code that is obtained by lowering the pdl code from Listing 7.

The existing lowering pass from pdl to pdl\_interp also combines multiple patterns into one search routine, reusing information used for different patterns and bailing out early as soon as none of the patterns can be matched anymore.



**Figure 1.** (left) In MLIR’s default, destructive rewriting, `pdl_interp.get_result` and `pdl_interp.get_defining_op` are each other’s inverse. (right) In equality saturation, this is not the case because each value comes from one of multiple equivalent operations.

In order to reuse this existing rewrite infrastructure for matching patterns in the IR that has been extended with eqsat operations, we have implemented an alternative interpreter over pdl\_interp operations that takes into account the extra indirections caused by eqsat.eclass operations. Most importantly, this interpreter is responsible for backtracking and trying out all possible values in an e-class, as is done by most equality saturation frameworks, and described by De Moura et al. [9].

As it turns out, for equality saturation, most pdl\_interp operations can be interpreted exactly the same as for classical rewriting, with the exception of `pdl_interp.get_result` and `pdl_interp.get_defining_op`, as these operations go back and forth between an operation and its result. Similarly, the behavior of `pdl_interp.create_operation`, and `pdl_interp.replace` needs to be adapted to take into account the e-class operations. The `pdl_interp.get_result` operation takes an operation and returns its result value (Figure 1). Using our eqsat dialect, the result of all operations in the e-graph are eqsat.eclass operations, which is not what the existing pattern matching code expects. Instead, our interpreter digs one level deeper, returning the result of the eqsat.eclass operation.

Similarly, for `pdl_interp.get_defining_op`, an operation that returns the operation that defines a particular value, the link from result to defining operation is interrupted by an eqsat.eclass operation (Figure 1). To handle this, the interpreter keeps track of each `pdl_interp.get_defining_op` operation. And when a pattern match fails, the interpreter goes back to the latest instance and retries matching with the next operand of the eqsat.eclass operation.

For `pdl_interp.create_operation`, instead of blindly creating and inserting a new operation, the interpreter will now first verify if an identical operation already exists in the program and use that one. This serves the same purpose as the use of hashconsing in typical e-graph libraries. Lastly, `pdl_interp.replace` now does not remove the operation being replaced, but rather inserts the replacement values in the correct e-class.

**Listing 9** Equivalent representations of sign-extension from a two-bit value to a four-bit value.

---

```
%2 = moore.sextr %a : i2 -> i4

%0 = comb.extract %a from 1 : (i2) -> i1
%1 = comb.replicate %0 : (i1) -> i2
%2 = comb.concat %1, %a : i2, i2
```

---

By again depending on existing compiler infrastructure to implement part of equality saturation, development burden is greatly reduced. Additionally, since MLIR combines all patterns in a single matching routine, large rulesets with overlapping subpatterns can more efficiently be pruned compared to most existing equality saturation frameworks. For example, in egg patterns are matched one by one, in a top-down manner. Another approach looks at pattern matching on e-graphs from a database perspective [23], converting e-graphs into a relational database and viewing pattern matching as a relational join. Similar to this approach, the generated MLIR pattern matcher is able to search for patterns top-down, bottom-up, or a combination thereof using existing pattern matching infrastructure.

## 6 Future Work

Our first step will be to explore mechanisms for combining multiple cost models, useful in cases where performance must be traded off with floating-point accuracy [16, 17] in linear algebra micro-kernel compilation [13]. Combining cost models is also inevitable when the program being rewritten is expressed in terms of operations in multiple dialects, with associated cost being computed by separate cost models. Furthermore, having equivalent program representations at different levels of abstraction provides the opportunity to combine analyses across abstraction levels [6]. In circuit design, for example (Listing 9), the moore dialect uses just one operation, while the comb dialect utilizes three operations to achieve the same result. Naturally, the single operator is simpler to analyze, say via an interval analysis, while the three-operator implementation is easier to lower into real hardware.

As discussed (Section 3), region-based control flow operations do not inhibit equality saturation. Currently, however, the pdl dialect cannot be used to match regions of operations. This means that, while it is possible to match code in regions, it is not yet possible to match complete control flow operations and rewrite those. In the future, allowing this could open up doors to not only rewrite code in the presence of control flow, but also rewrite control flow operations themselves.

## 7 Related Work

The idea of embedding equality saturation in a compiler has been explored before with Craelift's acyclic e-graphs (ægraphs) [2]. Although similar to our work, we have made a number of different design decisions that lead to distinct capabilities.

Firstly, functions in Craelift's IR contain control flow graphs (CFGs), possibly consisting of multiple basic blocks and unstructured control flow. The presence of a CFG complicates building an e-graph representation, as phi-nodes (or block arguments) complicate def-use dependencies by making them conditional on control flow. To resolve this, Craelift introduces the concept of a CFG skeleton, a data structure storing the fixed CFG such that the function can be reconstructed from the e-graph representation. The CFG skeleton has the downside of prohibiting control flow rewrites, meaning that rewrites that span multiple basic blocks can occur, but that the structure of the basic blocks and control flow is fixed.

By building on MLIR, we instead target code using structured control flow constructs. Here, every function consists of a single basic block, and control flow is captured by operations containing nested code regions. Conceptually, this can allow control flow to be rewritten just the same as other operations.

Secondly, Craelift's IR is strictly SSA, and does not have the concept of graph regions making it less straightforward to represent cyclic e-graphs in IR directly. Instead, Craelift rewriting runs in a single pass, applying multiple rewrites eagerly the moment each instruction is added to the e-graph. By leveraging graph regions, we are able to represent cyclic e-graphs, allowing us to execute full equality saturation in our framework.

Lastly, by targeting MLIR, instead of a lower level IR such as the one found in Craelift, we open up the possibility of rewriting in many different domains, and across different abstraction levels.

## 8 Conclusion

Existing work using equality saturation for code optimization mostly does so by using external libraries. This prevents combining the advantages of equality saturation with existing compiler analyses and transformations, and requires additional work in order to support new program constructs. By bringing equality saturation to the compiler in the form of IR primitives, the fundamental barrier between equality saturation and compiler passes is lifted. We have shown that existing compiler passes such as CSE, and compiler infrastructure such as MLIR's pdl dialect can be harnessed to more easily implement equality saturation. Furthermore, this approach opens up many new opportunities for using equality saturation to further enhance existing compiler passes.

## Acknowledgments

This work has received funding from the European Union's Horizon EUROPE research and innovation program under grant agreement no. 101070375 (CONVOLVE).

## References

- [1] 2024. GCC, the GNU Compiler Collection - GNU Project. <https://gcc.gnu.org/>
- [2] Bytecode Alliance. 2024. Craelift. <https://craelift.dev>.
- [3] Apple Inc. 2024. *The Swift Programming Language*. <https://docs.swift.org/swift-book/documentation/the-swift-programming-language/> Version 6.1.
- [4] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler support for sparse tensor computations in MLIR. *ACM Transactions on Architecture and Code Optimization (TACO)* 19, 4 (2022), 1–25.
- [5] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for HLS using E-graph Rewriting with MLIR. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, La Jolla, CA, 1029–1044. doi:[10.1145/3620665.3640392](https://doi.org/10.1145/3620665.3640392)
- [6] Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Combining E-Graphs with Abstract Interpretation. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. Association for Computing Machinery, Orlando, FL, 1–7. doi:[10.1145/3589250.3596144](https://doi.org/10.1145/3589250.3596144)
- [7] Samuel Coward, Theo Drane, and George A Constantinides. 2024. ROVER: RTL Optimization via Verified E-Graph Rewriting. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43 (2024), 4687–4700. doi:[10.1109/TCAD.2024.3410154](https://doi.org/10.1109/TCAD.2024.3410154)
- [8] S. Coward, L. Paulson, T. Drane, and E. Morini. 2022. Formal Verification of Transcendental Fixed- and Floating-point Algorithms using an Automatic Theorem Prover. *Formal Aspects of Computing* 34, 2 (2022). doi:[10.1145/3543670](https://doi.org/10.1145/3543670)
- [9] Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, Vol. 4603 LNAI. Springer-Verlag, Bremen, 183–198. doi:[10.1007/978-3-540-73595-3\\_13](https://doi.org/10.1007/978-3-540-73595-3_13)
- [10] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladotis, Michel Steuwer, and Tobias Grosser. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) (CGO '25). Association for Computing Machinery, New York, NY, USA, 179–192. doi:[10.1145/3696443.3708945](https://doi.org/10.1145/3696443.3708945)
- [11] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, CGO. IEEE, 75–86. doi:[10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665)
- [12] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization* (CGO). 2–14. doi:[10.1109/CGO51591.2021.9370308](https://doi.org/10.1109/CGO51591.2021.9370308)
- [13] Alexandre Lopoukhine, Federico Ficarelli, Christos Vasiladotis, Anton Lydike, Josse Van Delm, Alban Dutilleul, Luca Benini, Marian Verhelst, and Tobias Grosser. 2025. A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) (CGO '25). Association for Computing Machinery, New York, NY, USA, 163–178.
- [14] Charles Gregory Nelson. 1980. *Techniques for program verification*. Ph. D. Dissertation. Stanford University.
- [15] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) (ASPLOS '21). Association for Computing Machinery, New York, NY, USA, 804–817. doi:[10.1145/3445814.3446712](https://doi.org/10.1145/3445814.3446712)
- [16] Pavel Panchevka, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 50. Association for Computing Machinery, 1–11.
- [17] Brett Saiki, Jackson Brough, Jonas Regehr, Jesus Ponce, Varun Pradeep, Aditya Akhileshwaran, Zachary Tatlock, and Pavel Panchevka. 2025. Target-Aware Implementation of Real Expressions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) (ASPLOS '25). Association for Computing Machinery, New York, NY, USA, 1069–1083. doi:[10.1145/3669940.3707277](https://doi.org/10.1145/3669940.3707277)
- [18] 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*, Vol. 44. Association for Computing Machinery, 264–276. doi:[10.1145/1480881.1480915](https://doi.org/10.1145/1480881.1480915)
- [19] Nicolas Vasilache, Oleksandr Zinenko, Aart JC Bik, Mahesh Ravishankar, Thomas Raoux, Alexander Belyaev, Matthias Springer, Tobias Gysi, Diego Caballero, Stephan Herhut, et al. 2022. Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction. *arXiv preprint arXiv:2202.03293* (2022).
- [20] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchevka. 2021. Egg: Fast and extensible equality saturation. In *Proceedings of the ACM on Principles of Programming Languages*, Vol. 5. Association for Computing Machinery. doi:[10.1145/3434304](https://doi.org/10.1145/3434304)
- [21] Abd-El-Aziz Zayed and Christophe Dubach. 2025. DialEgg: Dialect-Agnostic MLIR Optimizer using Equality Saturation with Egglog. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) (CGO '25). Association for Computing Machinery, New York, NY, USA, 271–283. doi:[10.1145/3696443.3708957](https://doi.org/10.1145/3696443.3708957)
- [22] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 468–492. doi:[10.1145/3591239](https://doi.org/10.1145/3591239)
- [23] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. 2022. Relational e-matching. *Proc. ACM Program. Lang.* 6, POPL, Article 35 (Jan. 2022), 22 pages. doi:[10.1145/3498696](https://doi.org/10.1145/3498696)