RESEARCH-ARTICLE

Equality Saturation for Optimizing High-Level Julia IR

JULES MERCKX, Ghent University, Ghent, VOV, Belgium

TIM BESARD

BJORN DE SUTTER, Ghent University, Ghent, VOV, Belgium

Open Access Support provided by:

Ghent University

Published: 23 February 2026
Accepted: 21 January 2026
Revised: 14 January 2026
Received: 03 July 2025

Equality Saturation for Optimizing High-Level Julia IR*

JULES MERCKX, Ghent University, Ghent, Belgium
TIM BESARD, JuliaHub, Cambridge, United States
BJORN DE SUTTER, Ghent University, Ghent, Belgium

Compilers are indispensable for transforming code written in high-level languages into performant machine code, but their general-purpose optimizations sometimes fall short. Domain experts might be aware of optimizations that the compiler is unable to apply or that are only valid in a particular domain. We have developed a system that allows domain experts to express rewrite rules to optimize code in the Julia programming language. Our system builds on e-graphs and equality saturation. It can apply optimizations in the presence of control flow and side effects. As Julia uses multiple dispatch, we allow users to constrain rewrite rules by argument types, and propagate type information through the e-graph representation. We propose an ILP formulation for optimal e-graph extraction that exploits opportunities for code reuse and introduce *CFG skeleton relaxation* to rewrite calls to pure functions as well as those with side effects. Use cases demonstrate that our system can perform rewrites on high-level, domain-specific code, as well as on lower-level code such as Julia's broadcasting mechanism. We analyze the required compilation time and the performance impact of these rewrites.

CCS Concepts: • **Software and its engineering → Compilers**; • **Theory of computation →** *Rewrite systems*; *Integer programming*;

Additional Key Words and Phrases: Compiler Optimization, Equality Saturation, High-Level Languages

1 Introduction

Optimizing compilers have become indispensable as programmers rely on them to unravel high-level abstractions into lean and performant code [31, 40, 41]. General-purpose compiler middle-ends typically perform general-purpose optimizations such as common subexpression elimination [49]. Compilers for domain-specific languages perform domain-specific optimizations such as algebraic optimizations [41, 49]. The front-ends of higher-level language compilers can be customized to generate IR that is already optimized for specific targets, such as GPUs, and then fed to the back-ends [15]. Many compilers can also be configured to enable context-specific optimizations, e.g., fusion of operators without maintaining bit-perfect floating point equivalence. However, few optimization pipelines are well suited to support domain-specific and context-specific optimization of code written in a high-level scientific programming language that is used across many domains, such as Julia [16]. In such a language, software libraries build on the generic language infrastructure to provide high-level domain-specific APIs to application developers.

Ideally, those application developers should be able to enjoy domain-specific optimizations for free, i.e., without having to manually tune or rewrite their code, and without facing restrictions on their freedom to exploit the language's rapid prototyping features. Furthermore, it should be easy for application developers to specify or select the context-specific optimization they want enabled. In addition, when multiple libraries are

*New Paper, Not an Extension of a Conference Paper.

Authors' Contact Information: Jules Merckx, Ghent University, Ghent, Belgium; e-mail: jules.merckx@ugent.be; Tim Besard, JuliaHub, Cambridge, Massachusetts, United States; e-mail: tim@juliahub.com; Bjorn De Sutter, Ghent University, Ghent, Belgium; e-mail: bjorn.desutter@ugent.be.

ACM 1544-3973/2026/2-ART
https://doi.org/10.1145/3795883

being reused and composed in an application, be it top-level libraries that provide orthogonal functionality or higher-abstraction-level libraries that provide more abstract APIs on top of lower-level code, the optimization opportunities coming with those libraries should be composable. They should be composable with each other and with the general-purpose and target-specific optimizations that are provided in the main compiler flow and in the target-specific extensions thereof. This inevitably involves solving a phase-ordering problem because some optimizations create opportunities for further optimizations, while others are incompatible and rule each other out. Finally, to obtain a thriving ecosystem of domain libraries, the development of such libraries and the corresponding support for domain-specific optimization should not require the involvement of compiler experts. It then follows that a generic compiler infrastructure is needed that enables developers of domain-specific libraries to specify domain-specific and context-specific optimization opportunities at an abstraction level similar to that of their libraries' APIs, and that ensures sufficient composability.

With our research, we aim to provide that generic compiler infrastructure. In this paper, we present a system that allows programmers, not necessarily compiler developers, to express their domain-specific knowledge as rewrite rules in Julia. Our work is based in part on ideas developed for the Cranelift compiler [25], which in turn uses e-graphs and equality saturation, first introduced by Nelson [51] and Tate et al. [64], respectively. Equality saturation, or EqSat, is a rewrite technique that elegantly deals with the phase-ordering problem by representing the potential results of rewrites while still representing the original program as well.

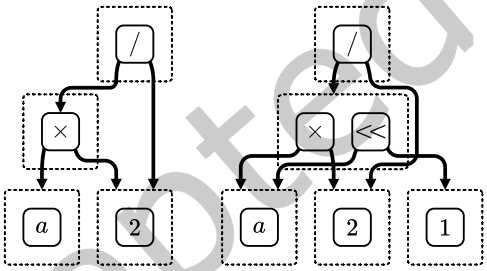

Fig. 1. Left: original e-graph [70] representing the term $(a \times 2)/2$. Right: e-graph after introducing the equivalence $a \times 2 \leftrightarrow a << 1$.

When writing and executing code, users should not take these rewrite rules into account anymore, but instead can focus on writing simple and readable code that maps well onto their mental model of the computations. The compiler then automatically evaluates all relevant rewrites and picks the optimal program. By building on e-graphs and EqSat, one needs not to worry about the order of rewrites. Rewrite rules are applied without removing information from the e-graph, which means that application of a rewrite cannot prevent other rules from firing.

The contributions in this paper are as follows:

- the first use of EqSat in a high-level, dynamically typed programming language optimizer, including expressions spanning control flow transfers, function calls with side effects, value reuse in rewritten expressions, and type-constrained rewrite rules;
- a demonstration of the capabilities of this optimizer on a number of samples;
- an analysis of the required compilation time and performance gains on those samples.

This paper is structured as follows. Section 2 provides background. Section 3 presents our rewriting system. Section 4 presents its capabilities on use cases. Section 5 analyzes its compilation time, and Section 6 demonstrates obtainable performance optimizations. Sections 7 and 8 discuss limitations and related work. Finally, Section 9 draws conclusions.

2 Background

E-Graphs. An e-graph compactly represents a congruence relation on different expression trees [51, 70]. It consists of a collection of *e-classes* that can each contain multiple *e-nodes* representing expression trees that are equivalent to each other according to a user-defined equivalence relation. Each e-node can have multiple children, represented by e-classes. As an example, the left side of Fig. 1 shows the e-graph for the expression $(a \times 2)/2$. In case a represents an integer, a potential equivalent expression for $a \times 2$ is $a << 1$. We can encode this equivalence by adding a new e-node to the appropriate e-class, as shown on the right side of Fig. 1. Because the constant expression 1 is not equivalent to any of the e-classes already present, it is added to a newly created e-class. The strength of e-graphs lies in the fact that both equivalent representations are encoded in the graph at the same time.

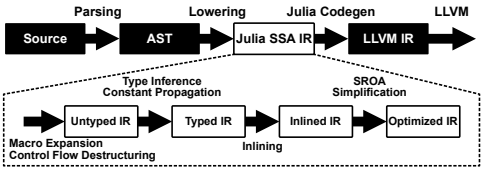

Fig. 2. The native Julia compilation pipeline.

Equality Saturation. Besides in SMT solvers [21, 22, 51], e-graphs are often used for EqSat [18, 64, 70]. EqSat is a term rewriting technique that applies rewrites not by overwriting the original term but by adding the rewritten term to an e-class that represents the original term. By building on e-graphs, it is possible to represent a large number of expressions, potentially exponential in the number of e-nodes, compactly. More importantly, by keeping track of all rewritten expressions, the order in which rewrites are applied no longer matters: One rewrite cannot prevent another one from firing. Central to EqSat is the rebuilding procedure, which is responsible for maintaining the e-graph invariants. When rewriting uncovers two different expressions to be equivalent, all expressions containing these equivalent expressions as subexpressions need to be verified, as these then potentially also need to become equivalent.

Extraction. Once no more new rules can be applied or some timeout is reached during EqSat, one typically wants to determine the optimal expression contained in the e-graph [28, 70]. In its simplest form, extraction starts from a root e-class where one e-node needs to be picked to be extracted. Recursively, all the child e-classes of the picked e-node need to be extracted as well. The end result is a directed, connected graph.

A simple greedy extraction approach can lead to suboptimal results since value reuse is not considered. This can be alleviated by resorting to a more refined extraction technique. Our extraction technique will be discussed in Section 3.5.

Julia. Julia is a general-purpose programming language that is primarily used for scientific computing. Being a dynamically typed language, functions can be defined for unconstrained argument types. A function can also be redefined for specific argument types (and counts) to express that it should behave differently for that combination of types. The different behaviors, i.e., the definitions of a function for different types and counts of arguments, are called *methods* in Julia. In essence, a function is an identifier, such as `f` or `print`, that it shares with all its methods. When a function call is to be executed, the Julia run-time system will look up and dispatch the function's method that best matches the types and count of all the call's run-time arguments. This is called *multiple dispatch*.

In Julia, operators are syntactic sugar for functions. Even basic functionality like array indexing can be customized with methods for different types of arrays and indices. A method can, e.g., redefine how elements in a `Symmetric` matrix are accessed, avoiding the need to store symmetric elements twice. Methods that implement more abstract computations on matrices by accessing those matrices' elements through index operations will automatically benefit from that custom access method when they are invoked on a `Symmetric` matrix. Because all operators are functions, Julia expressions correspond by and large to trees of function calls.

Fig. 2 shows Julia's just-ahead-of-time compiler flow. At run time, when a method needs to be dispatched and executed, this flow is invoked on it. We will call this the current method. The compiler first lowers the method's AST into an SSA-based IR. Because all operators are functions, that IR code consists mainly of control flow, phi-nodes, literals, assignments to SSA values, and function calls. It hence directly reflects all expressions from the source code, including all called functions and operators, the main difference being that each sub-expression result is assigned to its own SSA value in the IR. On this IR, the compiler applies type inference based on the run-time argument types, constant propagation, method inlining, and some other basic optimizations such as scalar replacement of aggregates (SROA). From this Julia IR, the compiler then generates LLVM IR, to which many more optimizations are applied by the LLVM compiler before it generates assembly code for the target architecture.

Cranelift and Acyclic E-Graphs. Cranelift [25], a compiler back-end mostly used to generate code from Rust and WebAssembly, is the first production compiler to use e-graphs for code optimization. As Cranelift aims at JIT compilation, where compilation time is critical, they have introduced *acyclic e-graphs* (ægraphs). Like regular e-graphs, these represent multiple equivalent expressions in e-classes. But, while regular EqSat fully propagates the effects of introduced equalities, this is not the case with ægraphs. Equalities are greedily applied, and e-graph invariants are not fully restored. This means that some rewrites are potentially not discovered, but the compilation time is shorter and more predictable.

More relevant to our work is the introduction of the *CFG skeleton*, a data structure kept alongside the e-graph to keep track of control flow instructions and instructions with side effects, two constructs that do not fit the EqSat paradigm for optimizing expressions. Its workings and how we leverage this technique in our work will be explained in Section 3.1.

Because e-graphs in Cranelift are used in the compiler back-end, they model expressions extracted from the IR after inlining, which can be relatively large, but mainly contain simple, low-level instructions, many of which are pure. Cranelifts' e-graphs and its use of the CFG skeleton are hence designed to conserve all control flow and side effects, instead focusing on the rewriting of pure expressions in between control flow and operations with side effects.

3 A Rewrite-Based Compiler Middle-End

Our system allows programmers to specify custom rewrite rules in Julia syntax. These rules are then applied on an e-graph representing the code of a method. Once the rules are applied and the e-graph is saturated or a timeout has been reached, the compiler will try to extract the optimal code embedded within that e-graph.

Rules have a left-hand side and a right-hand side. Each side consists of a regular Julia expression, i.e., of a tree of function calls, plus variables and/or literals. Two example rules are `sin(~x::Number)^2 + cos(~x::Number)^2` ⟶ `1` and `translate(~p::Vec2D, 0, 0)` ⟶ `~p`. In these rules `sin`, `cos`, `^`, `+`, and `translate` are the names of the called functions. The `~` prefix identifies pattern variables in a rule. These can match any input values of the expression's function calls, be it the values of other expressions, variables, or literals. The `::` notation is used to specify the types of values for which the rule can be applied. If no type is specified, values of any type can be matched. A rewrite rule can be applied whenever the pattern of function calls, literals, and variables on its left-hand side matches an IR code fragment.

As expressions in Julia source code and the SSA IR produced for them contain the same function calls, it is straightforward to search for such matches in a method's SSA IR code. In other words, there is no need to have rule authors express equivalences in terms of IR constructs. They can instead express them in regular source code expressions similarly to how those would occur in source code to be optimized. This syntax using regular Julia code expressions, with the addition of unbound, optionally typed variables, covers all rewrites where a single expression tree is replaced with a new expression tree. This syntax does not allow rules matching

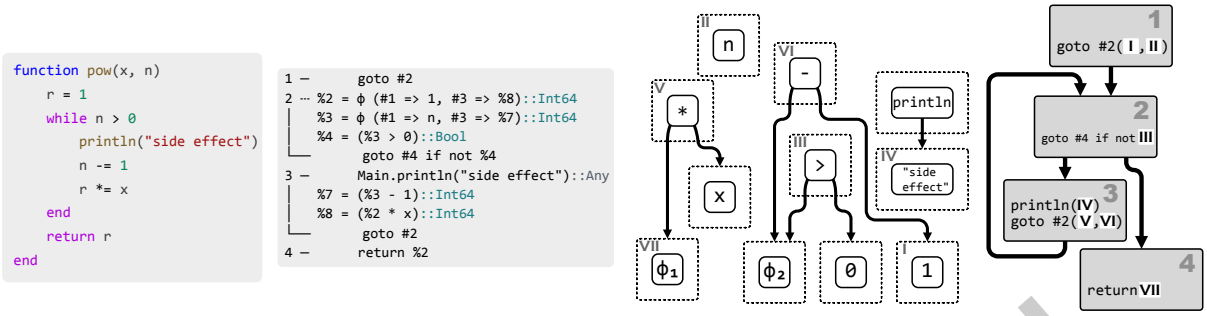

Fig. 3. From left to right: a Julia method defined for the function pow that computes a number's power, the corresponding Julia IR that is generated for two integer arguments, and the e-graph and CFG skeleton for this method.

multi-patterns [35, 72]. Such patterns can match multiple expression trees and rewrite them each, potentially reusing variables in multiple output expressions.

As in many existing compiler middle-ends, the optimization based on these rewrite rules/patterns will operate on an intermediate representation (IR) that supports optimizations while still preserving high-level language-specific information [41, 44, 63]. In our case, this is the Julia compiler's SSA-based IR [16].

A consideration to be made for pattern rewriting is at what point in the compilation to apply patterns. Concretely, patterns can be matched on IR before or after function calls have been inlined. Before inlining, the current method's IR still contains function calls as they were written by the programmer. At that stage in the compilation, high-level rewrite rules can typically be matched and applied, i.e., rules involving functions of high-level, abstract, and domain-specific APIs. However, some rewrite opportunities could then remain hidden behind call barriers. This can happen, e.g., when APIs with domain-specific function names wrap more generic computations such as underlying linear algebra computations. If the programmer used the domain-specific wrapper APIs while the rewrite patterns are expressed in terms of the underlying linear algebra operations, those patterns will not be found in the IR code before inlining.

Ideally, when exploring rewrite opportunities, the compiler should hence consider all possible combinations of inlined and not-yet-inlined functions. Support for such an exploration is for future work. Instead, our current prototype implementation performs rewriting optimization only before inlining. We note that many interesting rewrites that are not covered by classical compiler optimization occur before inlining, when high-level API usage has not yet vanished from the code. For this reason also, the example snippets of IR in this paper contain code that has not been inlined yet.

3.1 From IR to E-Graph and Back Again

To apply rewrites on the Julia IR of a method, we first convert it to an e-graph and a CFG skeleton. For this, we leverage methods first introduced in the Cranelift compiler [25]. Fig. 3 shows an example Julia function pow together with its SSA IR, e-graph, and CFG skeleton. The structured control flow present in the original source code has been lowered into four basic blocks in the Julia IR, each terminated by an instruction that jumps to another block or returns.

Each e-node in the e-graph corresponds to an SSA value in the IR. For example, the value %7 in the IR corresponds to the e-node labeled -. The dotted boxes around the e-nodes represent the different e-classes. Since no equalities have been registered yet in this e-graph, each e-class contains exactly one e-node. In this representation, the phi-nodes are leaf nodes, i.e., their dependencies are not included in the e-graph. This is because an SSA phi function introduces a branch in an expression that cannot be represented in an e-graph directly. The lack of

phi-node information implies (i) that rewrites including `phi`-node dependencies are not supported, and (ii) that the e-graph by itself does not contain sufficient information to reconstruct all control flow of the original code. For (i), an example of a useful rewrite that is not supported is to reduce a phi-node to a value if all possible values in the phi-node are equivalent.

For (ii), it is the CFG skeleton that provides the extra information needed to reconstruct valid IR. To that extent, the CFG skeleton contains all operations responsible for control flow and all operations with side effects, including function calls that have side effects. All operations in the skeleton reference the e-classes they depend on. To visualize these references, we use the roman numerals in Fig. 3. Each roman numeral labels an e-class of which the corresponding SSA value is used in an operation with side effects, or in a control flow statement. For example, since a call to `println` has side effects, it is stored in the third block of the CFG skeleton, referencing e-class **IV**.

Cranelift's SSA IR format, like MLIR [1] and Swift [63], uses basic block arguments. These offer an alternative to phi-nodes for representing control-dependent data flow in SSA IRs. Whereas phi-nodes represent such data flow within a basic block, basic block arguments represent such data flow at the basic block level. Even though Julia IR contains phi-nodes, we opt to use block arguments in the CFG skeleton because all information regarding out-of-block uses of expressions that need to be materialized within a block is then explicitly represented within that block itself.

To illustrate, the references to e-classes **I**, **II**, **V**, and **VI** in the two `goto #2` statements in the skeleton are basic block arguments. They are passed as arguments to control flow statements and can be references in the destination block, similarly to how functions have arguments. The relation between basic block arguments and phi-nodes in the original Julia IR is simple. For example, the two phi-nodes in block `#2` in the Julia IR correspond to the two basic block arguments passed in the CFG skeleton by blocks `#1` and `#3`. Note that while the e-graph in Fig. 3 contains nodes labeled phi, these are purely labels and do not carry the information contained in SSA phi-functions. The phi-nodes in the e-graph can be interpreted as being the corresponding basic block arguments.

```
a = f(…)
if condition
   b = g(…, a, …)
else
   …
end
```

Fig. 4. Prototypical example of an expression `b = g(..., f(...), ...)` being split by a condition.

It is the CFG skeleton that allows the conversion back to valid Julia IR. As long as each statement in the CFG skeleton is *elaborated*, meaning that that statement and all of its dependencies are materialized in the generated IR, the resulting program has the same control flow and side effects as the original program.

The need to support control flow stems from expressions that are split over multiple basic blocks for the sake of code conciseness, readability, and maintainability. The most important case concerns if-then-else patterns in which different expressions are computed under different conditions, and of which a common subexpression is computed beforehand as shown in the pseudo-code in Figure 4. Five idioms for which such patterns commonly occur (as witnessed by the referenced examples from Julia libraries) are the following:

Keyword Arguments (kwargs) Besides arguments that provide data to a function, it is quite common to have keyword arguments that parametrize the semantics of the function by controlling which parts of the function get executed. The if-condition then is a direct check of a kwarg's value such as `!flipkernel` [11].

LAPACK-style wrappers with `uplo`/`fmt`/`trans`/... chars BLAS/LAPACK-like libraries often use characters to indicate the format of an array (transposed or not, upper/lower triangular, etc.). If-conditions then check the format (e.g., with `fmt == :coo` [3] or `M.uplo == 'U'` [10]) to decide which exact computations to execute.

API Version Checks It is not uncommon to invoke API functions conditioned on the version of the used libraries. The if-condition then is a version check such as `CUSPARSE.version()` ≥ `v"11.7.2"` [4].

Low-level Math In scalar 'mathy' code, it is common to find numeric checks that decide on the best option to implement certain computations. Examples of such if-conditions are `n <= 24` [5] and `ax > sqrt(floatmax(ax)/2)` [6].

Imperative Type Checks For readability or to avoid too many methods, imperative type checks are sometimes used instead of multiple dispatch to determine which code to execute. Standard libraries provide examples of type checks serving as if-conditions, such as `B isa CuVectorT` [7], `a isa AbstractArray` [2], and `a isa Integer` [8].

Whenever conditional control flow splits a larger expression as in Figure 4, our work can handle them.

To convert an e-graph back to IR, the statements in the CFG skeleton are then used as starting points. This ensures that side effects and control flow are kept intact. IR is created by materializing each statement in the CFG skeleton as well as all its dependencies in the e-graph. Using the *scoped elaboration* algorithm [25], materialization occurs only for values that have not yet been materialized in statements that dominate the current statement in the CFG. Otherwise, the previously materialized SSA value is reused.

Our implementation uses Metatheory.jl [18], a Julia EqSat package based on egg [70]. At its core, it provides an efficient implementation of the e-graph datastructure and EqSat algorithms, and a system that allows to register new types of e-nodes. In our work, we use this system to implement e-nodes that represent Julia IR statements. Metatheory.jl also offers macros to easily specify rewrite rules with a high-level, declarative syntax. We extended this rule syntax to support type annotations, as will be discussed in Section 3.2. Lastly, Metatheory.jl contains a generic extraction algorithm. However, this algorithm is unsuited for our problem, as extraction needs to take into account the information from the CFG skeleton as well. In Section 3.5 we will discuss our extraction approach.

3.2 Rules and Types

With multiple dispatch, the semantics of a function call are decided not only by syntax (i.e., the function name), but also by the types and counts of their arguments. This necessitates a way to represent this information in rewrite rules. For example, a user might want to rewrite matrix multiplications into a function call to some external BLAS library. This means that a call to `Base.:*`, a base library function, should be rewritten, but only if its arguments are matrices. Users can specify type constraints in the rewrite rules by associating a type with an unbound variable. For example, a rule for rewriting a matrix multiplication can be written as `~A::Matrix * ~B::Matrix` → `blas_call(~A, ~B)`.

E-class analyses [70] offer a framework to associate and propagate lattice information in an e-graph. We designed a novel e-class analysis for tracking types in the e-graph. For each e-class, this analysis maintains the most specific type that is compatible with all the e-nodes in that class. During e-graph construction, this type is simply the same as the types of statements in the Julia IR. When a rewrite rule determines that two e-classes are equal, the e-classes are merged into a new e-class containing e-nodes from both classes. When this happens, the resulting e-class type is determined by the `typejoin` function, a standard Julia function that returns the most specific type that is a supertype of all its arguments. For rewrite rules that introduce a new function call, such as the rewrite of a matrix multiplication into `blas_call`, we run Julia's type inference on the new function using the argument types stored in the e-classes as the new e-node for the function call is added to the e-graph. Similarly, if the type associated with an e-class changes, we rerun type inference for all functions in e-classes containing e-nodes that depend on the changed e-class.

Rewrite rules that change the type of a value produced by an expression can cause problems. If the value is used by a function that has no method defined for the new type, no valid code can be produced anymore. Our current implementation detects this when the type information of parent nodes is recomputed. When type inference finds no method for arguments with the new type, an error is thrown. It is the responsibility of the rule authors and users to either ensure that rules only introduce new types that are compatible with all the functions

that depend on the value, or that new methods are defined to make it so. In future work, we will implement the necessary support to skip the application of rewrite rules that can result in the production of invalid code.

During EqSat, for a small amount of different types that are joined, we construct a `Union` type that keeps track of all individual possible types. When the union grows too large, we fall back on a single type that is their most specific joint supertype. For example, if a value of type `Integer` is introduced in an e-class with associated type `Float64`, the new associated type becomes `Union{Integer, Float64}`. If, throughout EqSat, this union grows further, the associated type could become `Number`. This is in line with how the Julia compiler generates specialized code for values with a small union type. Here, tracking small unions of specific types prevents the saturation process from erroring when an existing function in the e-graph is not defined for the more general supertype but is defined for the specific types in the union.

To be sound, the type information associated to e-classes by this analysis should always be conservative, i.e., sound overapproximation (super types) of all possible run-time types that could flow to the e-class once a subgraph is extracted for it. The key insight is that by using Julia's existing type system operations (`typejoin` and type inference), the analysis inherits much of the soundness reasoning from Julia itself. Our argument for soundness then is as follows:

- The initial types of the e-classes are obtained from Julia's type inference, which we assume to be sound.
- The `typejoin` operation is a standard Julia operation, also used in its type inference, among others. Joining multiple types with it produces a common supertype, so this preserves soundness.
- Whenever a new function call is introduced by rewrite rules, corresponding to a new e-node in some e-class, we rely again on Julia's type inference, which returns a (conservative) type for the call's return value when the function would be executed on arguments that have the types associated to the new e-node's child e-classes. Julia's type inference supports concrete types and union types for those arguments. So we can, in all cases, rely on the soundness of Julia's type inference to obtain conservative type approximations for newly added nodes.
- The iterative type propagation as new nodes are being added and types associated to e-classes are being updated by invoking Julia's type inference converges to a sound fixpoint. When an associated type changes, be it because of (i) a typejoin, (ii) adding a type to a union of types following the application of a rewrite rule, or (iii) after (iteratively) re-running a type inference on some dependent e-classes, the change will be monotonic in the sense that (i) the new type will a supertype of all the joint old types, or (ii) a grown union of types, or (iii) a type equal or higher in the hierarchy than the old inferred type. Because Julia's type hierarchy has a finite height, the re-inference process is guaranteed to terminate, even in the case of cycles in the e-graph.
- Upon termination, the types associated to all classes are conservative type approximations, because Julia's sound type inference has been executed for all e-nodes (either before the e-graph saturation started, or during it) in the final e-graph to determine the nodes' types based on all their child classes' finally associated types, and because the e-classes' types are overapproximations (supertypes or union types) of the types of their e-nodes.
- The analysis correctly detects type errors: when at any point type inference finds no applicable method for newly introduced types, an error is thrown. This is sound because it prevents generating invalid code.

3.3 Constants in the E-Graph

Rewrite rules are usually symbolic in nature. They match an expression and replace that expression with a new one, plugging in any matched variable. Modern EqSat frameworks such as egg [70] and Metatheory.jl [18], which is used in this work, also support *dynamic rewrites*. When dynamic rewrite rules are applied, the expression in

the right-hand side is first evaluated (potentially using additional e-class analysis data) and the original code is rewritten into the result of that execution, rather than into the literal right-hand side expression itself.

When literals are part of the e-graph, dynamic rewrites can be used to perform simple constant folding. In normal Julia code, literals of primitive types and composite types with known size can appear directly in the IR. The values of these literals are embedded in the e-graph. The e-classes containing the corresponding e-nodes of the literals have an inferred type associated with them, but compared to e-classes containing nonliteral e-nodes, the actual value of that type is available as well. To support constant folding of literals, we extend rewrite rule type annotations with a special `Comptime{T}` type that only allows to match against literals of type `T`. Using $\Rightarrow$ instead of $\rightarrow$ to signify dynamic rewrite rules, a rule such as `~a::Comptime{Integer} + ~b::Comptime{Integer}` $\Rightarrow$ `~a + ~b` will match any addition between two integer literals and insert the constant-folded result of this addition in the respective e-class.

Dynamic rewrite rules also open opportunities for partial evaluation-like optimization [26, 49] beyond the constant propagation capabilities of the Julia compiler. Whereas the compiler does not propagate dynamic array arguments because their values could be mutated at any time, a programmer can override this default behavior with dynamic rewrite rules. To achieve this, we extended the rewrite rule format with rules of the form `function.arg[2]` $\rightarrow$ `W`. With this rule, the programmer can introduce an equivalence between the function's second argument in its e-graph and a constant array `W` that the programmer has defined in the context where the optimizer is invoked. Note that this "rule" is unsound and should not be applied to arbitrary programs, but serves more as a user directive to the optimizer during optimization of a specific instantiation of a method. A constant array holding `W`'s contents is then added as an equivalent e-node to that argument's e-class in the e-graph, to which dynamic rewrite rules can then be applied. The optimized function that is returned after EqSat is specialized for that constant argument and might not even depend on the argument value passed to it at the call site anymore. An interesting case is, e.g., when `W` is a matrix holding the constant weights of a neural network layer. Computations that only depend on `W` or other compile-time-known variables can then be partially evaluated.

3.4 Rewrites in the Presence of Side Effects

As discussed in Section 3.1, the CFG skeleton is used to ensure that all statements with side effects, such as the `println` in the example of Fig. 3, are materialized when the final optimized program is extracted.

We leverage Julia's built-in static effect analysis to determine whether a function call has side effects. This analysis determines several properties for each IR statement. The program property `effect_free` signals whether a statement is "free from externally semantically visible side effects" [9]. For our purpose, however, this analysis is often too strict and too imprecise, resulting in some functions not being labeled `effect_free`, even though they do not have side effects that require them to remain in a program. For example, many functions that allocate memory are not marked effect-free, even if the allocated memory never gets accessed again. As a result, even if rewrite rules subsequently introduce equalities that would make calls to these functions superfluous, they will still be included in the optimized code. In short, there is a mismatch between Julia's effect analysis and the desired freedom to operate of our rewrite system.

In Cranelift, side effects are less of a concern because the code it rewrites is lower level and typically contains a large amount of simple, low-level instructions, many of which are pure. In high-level Julia code, by contrast, the IR we work on generally is much smaller and consists only of a few calls to high-level functions/methods. Because these functions/methods have more complex behavior than simple, low-level instructions, they also often include behavior with potential side effects according to Julia's built-in analysis.

To overcome this mismatch when rewriting Julia code, we allow users to prefix rewrite rules with the `relaxed` keyword. Whenever a relaxed rule matches code, all the function calls that are part of the CFG skeleton and that

```
function f(A::AbstractMatrix)
    eig = eigen(A)
    if f_condition(A)
        x = tr(A)
    else
        x = det(A)
    end
    return eig, x
end
```

```
1 — %1 = Main.eigen(_2)::Union{…}
│   %2 = Main.f_condition(_2)::Bool
└──      goto #3 if not %2
2 — %4 = Main.tr(_2)::Float64
└──      goto #4
3 — %6 = Main.det(_2)::Float64
4 ┄ %7 = ϕ (#2 => %4, #3 => %6)::Float64
│   %8 = Core.tuple(%1, %7)::Tuple{…}
└──      return %8
```

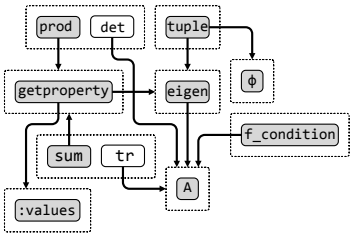

Fig. 5. Example method for function `f` for which EqSat reveals opportunities for reuse. `tr` and `det` can be rewritten as a sum and a product over the eigenvalues stored in `eig`. Optimal e-nodes for extraction are shaded gray.

occur in the left-hand side of the rule are "detached" from the CFG skeleton. When the final optimized code is extracted from the e-graph, detached statements are not forced to be materialized. We call this technique *CFG skeleton relaxation*.

Technically, it is possible to write relaxed rules that get rid of side effects that are depended upon by other parts of the code, thereby breaking its intended semantics. We note, however, that all rewrite rules discussed in this paper and that needed relaxation as we will discuss in Section 4, did not introduce problems in our use cases.

Finally, we note that semantic rule verification is outside the scope of this work. As a rewrite rule might be correct in one domain but incorrect in another, such as reduced precision in deep learning, correctness is up to the rule authors and users. We found that, in practice, for high-level functions that have no obvious program-related side effects, relaxing the CFG skeleton yields the desired results. Problems might arise when the code being optimized makes use of low-level internal functions that could depend on side effects, but we did not encounter such cases in our experiments.

3.5 Extraction

When Julia IR is generated from an optimized e-graph and its corresponding CFG skeleton, each statement in the CFG skeleton is considered a root node for e-graph extraction. This means that, in contrast to vanilla egraph extraction, we essentially need to run multiple extractions on the single e-graph to recover our program [19]. These extractions are not independent of each other. Take, for example, the code and corresponding IR and e-graph in Fig. 5. The method `f` computes and returns two different values `eig` and `x` derived from matrix `A`. `eig` stores the eigen vectors and values, and `x` stores the trace or the determinant of the matrix, depending on control flow. In this example, two rules from a collection of linear algebra identities have been applied: `tr(~A::AbstractMatrix)` ⟶ `sum(eigen(~A).values)` and `det(~A::AbstractMatrix)` ⟶ `prod(eigen(~A).values)`. These give rise to the extra e-nodes in the e-classes containing `det` and `tr`. Intuitively, we can see that an optimal extraction might be one in which the result of the call to `eigen` is reused to compute the determinant and/or trace instead of explicitly calling those functions. Indeed, that extraction is valid since the call to `eigen` dominates both the call to `det` and to `tr`, so its result is available at these call sites.

Greedy extraction techniques, albeit fast for extracting single expressions [70], are not guaranteed to extract the graphs with the minimum cost, even when using very simple cost models. For example, if two children of an e-node share a subgraph (when extracting one or more expressions) or if the children of two root nodes share a subgraph (when extracting multiple expressions), greedy extraction would overestimate the cost because it would ignore the sharing.

Alternatively, the problem of finding an optimal extraction can be formulated as an ILP problem. The state of the art in ILP extraction formulations applicable here ensures an acyclic extracted graph [30], which is important for further elaboration to produce linearized IR. That ILP formulation extracts only one expression, however, not

multiple expressions for the multiple root nodes of multiple statements in a CFG skeleton. By introducing, for example, a virtual parent node to the ILP formulation that depends on all root nodes, the optimal solution will take into account e-node reuse. However, this reuse does not take into account dominance relations between statements in the CFG skeleton, which in turn can lead to redundant code. We hence instead adapt and extend the original ILP formulation to enable extraction of multiple expressions taking into account e-node reuse from dominating expressions.

Informally, the ILP formulation ensures that:

- Each e-class corresponding to a statement in the CFG skeleton is picked. This ensures all the necessary expressions will be computed in the rewritten code.
- If an e-class is picked, then at least one of its e-nodes is picked. This ensures that each necessary (sub)expression is computed in at least one way.
- If an e-node is picked, then at least all of its children e-classes are picked. This ensures that for each necessary (sub)expression, its operands will be computed.
- Picked e-nodes do not form a cycle. This prevents that cyclic graphs are extracted because such graphs would represent infinite, non-terminating expressions that cannot be properly evaluated or represented.
- Only e-nodes that were picked in dominating extractions can be reused. This prevents incorrect reuse, i.e., reuse of some expression result that is not guaranteed to be available at the point of reuse in the CFG skeleton.
- E-classes are picked or reused only where needed for statements in the CFG skeleton. This is to prevent (sub)expressions from getting hoisted in the CFG skeleton. Hoisting expressions can be benificial for code size, but doing so aggressively can also introduce partially dead computations or alter program semantics when (sub)expressions with side effects are hoisted. Preventing hoisting completely like the current formulation does can hence be suboptimal for code size, such as in the case of very busy expressions, i.e., expressions computed on all paths from some point. Enabling optimal hoisting is future work.

Of these, the last two requirement are new compared to existing work [30].

Formally, the e-graph is represented by a set of e-nodes N and a set of e-classes C. $\{n | n \in c\}$ is the set of e-nodes contained in e-class $c \in C$, and $\chi(n)$ is the set of children e-classes of a particular e-node $n \in N$. We associate with each e-class c a set of e-nodes $p(c)$ defined as $\{n \mid c \in \chi(n)\}$. That is, the set of *parent* e-nodes n that have a child e-class c. Fig. 6 shows an e-graph and CFG skeleton in which the different parts have been annotated with this notation.

We represent the statements contained in the CFG skeleton as the set I. All of these statements cause a different expression to be extracted from the e-graph, rooted at each $i \in I$. For each statement $i \in I$, the statements that dominate i are represented by the set $d(i)$. Each statement i depends on a number of e-class arguments represented by the set $r(i)$.

Finally, we associate a cost $M(n)$ with every e-node n such that our optimization problem becomes[1]

[1]Changes to the formulation by He et al. [30] are highlighted in blue.

$$\min \sum_{i \in I, n \in N} M(n) \cdot w_n^{(i)} \quad \text{subject to}$$

$$\sum_{n' \in c} \left(w_{n'}^{(i)} + \sum_{j \in d(i)} w_{n'}^{(j)} \right) \geq w_n^{(i)}, \forall \; n \in N, c \in \chi(n) \tag{1}$$

$$w_n^{(i)} \leq \sum_{n' \in p(c)} w_{n'}^{(i)}, \quad \forall c \in C - r(i), n \in c \tag{2}$$

$$\sum_{n \in c} \left(w_n^{(i)} + \sum_{j \in d(i)} w_n^{(j)} \right) \geq 1, \quad \forall c \in r(i) \tag{3}$$

$$\left(1 - v_{\Psi,c}^{(i)}\right) + \left(1 - w_n^{(i)}\right) \geq 1, \forall \; \Psi \in \mathcal{A}, c \in \Psi, n \in \mathcal{N}_\Psi^c \tag{4}$$

$$v_{\Psi,c}^{(i)} + \sum_{n \in \mathcal{N}_\Psi} w_n^{(i)} \geq 1, \quad \forall \Psi \in \mathcal{A}, c \in \Psi \tag{5}$$

$$\sum_{c \in \Psi} v_{\Psi,c}^{(i)} \geq 1, \quad \forall \Psi \in \mathcal{A} \tag{6}$$

$$w_n^{(i)}, v_{\Psi,c}^{(i)} \in \{0, 1\}, \quad \forall \Psi \in \mathcal{A}, c \in \Psi, n \in N$$

Where, for each e-node n and statement i from the CFG skeleton, we have introduced a binary variable $w_n^{(i)}$. These variables indicate whether the e-node is selected for the extraction rooted at statement i. Furthermore, following the work by He et al. [30], each $\Psi \in \mathcal{A}$ is a *class cycle*, a collection of classes that contain e-nodes that form a cycle by depending on other classes in the cycle. The e-nodes in an e-class c that participate in a class cycle Ψ are represented by the set $\mathcal{N}_\Psi^c$. For each class c in each class cycle Ψ, the binary variables $v_{\Psi,c}^{(i)}$ enforce acyclicity in the extracted graph. $v_{\Psi,c}^{(i)}$ is assigned 1 if none of the e-nodes in e-class c that participate in the class cycle are selected. For a more complete overview and proof of acyclic extraction, we refer to the work by He et al. [30]. To take into account the dominance information from the CFG skeleton, we adapted their formulation in a few ways.

We adapt constraint (1) to ensure that an e-node can only be selected if for each child e-class an e-node is also selected, *or if an e-node is already selected in a dominating extraction j*. To prevent e-nodes from being selected in dominating extractions when they are not needed there, we introduce constraint (2). This constraint prevents an e-node from being selected if there is no e-node selected at the same extraction that depends on that e-node. This, in turn, prevents suboptimal extractions where the solver finds an optimal solution under the assumption that an extraction would be reused while in reality it is materialized in multiple dominated extractions.

The adapted constraint (3) ensures that at least one e-node in each e-class referenced by a root in the CFG skeleton is truly selected for extraction. Compared to the original ILP formulation by He et al., we added an additional term to account for the fact that an e-node might already have been selected in a dominating extraction.

The other constraints are completely analogous to the constraints by He et al. [30] except for the addition that each constraint is added separately for each possible extraction root $i \in I$.

We implemented this ILP formulation in JuMP [45], a domain-specific language embedded in Julia to express and solve mathematical optimization problems. For a solver, we resorted to HiGHS [32]. By default, our cost function $M(n)$ currently defaults to 1, reducing the optimization problem to choosing the least number of e-nodes. We allow users to manually alter the cost for calls to particular functions to overwrite this behavior.

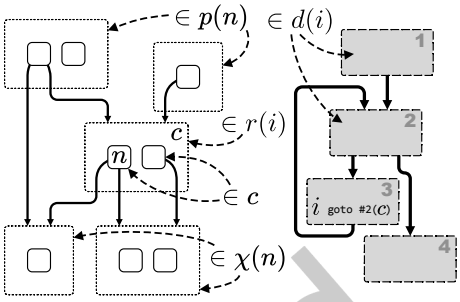

Fig. 6. *Left:* e-graph with 5 e-classes. E-node n has 2 child e-classes. Both e-nodes in c have the same two e-node parents. *Right:* statement i in block 3 of the CFG skeleton is dominated by all statements in dominating blocks, and depends on e-class c in the e-graph.

4 Use Case Demonstration

To demonstrate the capabilities of our work, this section presents a number of use cases, focusing on our main goal of allowing users and domain experts to express and exploit optimization opportunities in a convenient, flexible way. In other words, our use cases aim to demonstrate that our work can give developers control over optimizations without having to acquire knowledge on the compiler's internal operation and code representations, while upholding Julia's high-level generic programming promise of writing high-level code once that works across many types without sacrificing performance. In contrast, our work aims to give developers more flexibility, meaning less restrictions, to write code without incurring a reduction in optimization opportunities.

```
1  translate(p, dx, dy) = p .+ [dx, dy]
2  rotate(p, θ) = [cos(θ) -sin(θ); sin(θ) cos(θ)] * p
3  scale(p, s) = s .* p
4
5  function f(p, dx)
6      p = translate(p, dx, 0)
7      p = rotate(p, π)
8      p = translate(p, sin(dx)^2 + cos(dx)^2 - 1, 0)
9      p = rotate(p, π)
10     return translate(p, -dx, 0)
11 end
```

Fig. 7. Contrived code applying 2D transformations.

4.1 Domain Specific Rewrites

Fig. 7 shows a contrived method that applies a sequence of similarity transformations on a 2D point. On close inspection, under the assumption that exact floating point results must not be maintained, application of geometric properties, trigonometric identities, and algebraic simplifications can lead to the reduction of all these transformations to a single identity transform. Indeed, the translation on line 8 can be simplified to `p` by applying the rules shown in Figure 8.

```
translate(translate(~p::Vec, ~dx1, ~dy1), ~dx2, ~dy2)
      --> translate(~p, ~dx1 + ~dx2, ~dy1 + ~dy2)
translate(~p, 0, 0) --> ~p
relaxed rotate(rotate(~p::Vec, ~θ1), ~θ2)
      --> rotate(~p, ~θ1 + ~θ2)
rotate(~p, 2*π) --> ~p
rotate(~p, 0) --> ~p
```

Fig. 8. Rewrite rules to optimize the code from Fig. 7.

This example also illustrates the necessity of CFG skeleton relaxation. The Julia compiler's effect inference fails to reason through the rotation matrix construction in the `rotate` function on line 2 in Fig. 7. As a result, the allocation of the rotation matrix is conservatively considered to have side effects, which means that, by default, even if an optimization makes the rotation matrix obsolete, the

```
A = randn(10, 10)
B = rand(10)
relu(x::T) where T<:Real = max(T(0), x)
relu.(A) .+ B
```

Fig. 9. Broadcasting syntax: the `relu` function is broadcasted over `A` and `B` is added to the result using broadcasting addition.

```
materialize(broadcasted(+, broadcasted(relu, A), B))
```

Fig. 10. Syntactic lowering applied for the broadcasting expression in Fig. 9. `broadcasted` builds a lazy representation of the broadcast operation. `materialize` allocates the result array and computes its values.

matrix will still be materialized by the optimized code. The rewrite rule author is hence required to allow CFG skeleton relaxation for rules with `rotate` by prefixing those rules with `relaxed`.

4.2 Aiding Multiple Dispatch

Instead of using rewrites purely for simplifying code, they can also be used to generate code that provides additional information to the Julia compiler' type system to trigger the execution of better performing methods through multiple dispatch. An example in the domain of linear algebra is to automatically wrap certain matrix expressions in a new, more specific type that allows more efficient implementations for subsequent computations. Take, for example, the expression $A + B^T$ between two regular matrices A and B. For the case where A and B are the same matrix, that is, $A + A^T$, we know that the result is a symmetric matrix. We can encode this fact by applying the following rewrite rule: `~A::AbstractMatrix + transp(~A)` ⟶ `Symmetric(~A + transp(~A))`.

The `Symmetric` function from the `LinearAlgebra.jl` package takes the upper triangle of its argument and uses that to efficiently represent a symmetric matrix. More importantly, it returns a value of type `Symmetric`. Other functions in that linear algebra package, e.g., for solving eigenproblems, have specialized methods for symmetric matrices. After rewriting with the above rule, those specialized methods will now be called automatically thanks to the multiple dispatch system knowing that they are invoked on the type `Symmetric` instead of on a more generic matrix type. Another example is the expression $P^T BP$, which is symmetric if B is symmetric as well, as can be encoded with the rule `transp(~P::AbstractMatrix) * ~B::Symmetric * ~P` ⟶ `Symmetric(transp(~P) * ~B * ~P)`.

Both examples build on the abstractions of the Julia standard library, but the same concepts can also be applied to operations and types in other, external packages.

4.3 Multi-Line Broadcast Fusion

Julia's broadcasting mechanism allows users to apply functions element-by-element on one or more arguments containing multiple elements. Syntactically, this is done by adding a dot (`.`) between the function and its argument list. The top of Fig. 9 shows two broadcast operations. `relu.(A)` applies the scalar `relu` function to each element of the matrix `A`. Next, the `.+` call computes the addition between the result and a vector `B`. Although the two arguments to `.+` have a different shape, the operation can take place because the shapes are compatible.

The Julia compiler transforms expressions containing this special *dot syntax* into calls to the standard library functions `broadcasted` and `materialize`. Fig. 10 shows the result for the expression in Fig. 9. A call to `broadcasted` builds a lazy representation of the broadcast result. Different lazy broadcast objects can be nested; it is only when `materialize` is called that the final result object is allocated and filled with computed values. This is similar to how stream fusion has been implemented in functional languages such as Haskell [46]. As such, different operations within multiple `broadcasted` calls are fused when `materialize` is called. This leads to less memory being used because temporary arrays are not materialized and potentially better run-time performance because there are fewer function calls. This operator fusion is especially beneficial for code that is executed on the GPU, as kernel launches can take a significant amount of time, and because kernel fusion can significantly reduce the required number of memory accesses [15].

LHS pattern of rewriting rule	Count	Distinct Packages
`broadcasted(~f, materialize(~x))`	514	232
`broadcasted(~binop, materialize(~x), ~y)`	1232	348
`broadcasted(~binop, ~x, materialize(~y))`	1044	283
`broadcasted(~binop, materialize(~x), materialize(~y))`	231	93

Table 1. Number of matches for different multi-line broadcasting patterns in an analysis of 3153 Julia packages.

A limitation of Julia's broadcasting lowering is that fusion cannot occur across broadcast expressions on different lines, because replacing the dot syntax with calls to `broadcasted` and `materialize` is a syntactical transformation performed during AST lowering, which operates statement by statement. A potential optimization is hence to eliminate superfluous calls to `materialize`. In practice, this is often beneficial, especially for GPU code, where separate kernel launches can add significant overhead and a fused kernel can thus lead to better performance. Rewrite rules make this easy. For example, the rule `broadcasted(~f, materialize(~x))` → `broadcasted(~f, ~x)`. will remove the intermediary materialization for any call of the form `f.(x)` where `x` is a variable that was defined on a different line with a broadcast expression itself. In the example in Fig. 11, fusion will now be applied despite the two dot operators occurring in two different statements on two separate lines. This illustrates how the system we propose not only supports domain-specific optimization but also at the same time enables optimization of common code patterns involving only core Julia primitives.

CFG skeleton relaxation is required here, as `materialize` ends up calling a foreign C function which the compiler cannot deem effect-free, poisoning the remainder of the effect analysis. It is not possible to mark certain foreign function calls as effect-free in Julia currently. In practice, this means that the rule needs to be prefixed with `relaxed`.

Using PkgEval.jl, a tool to automatically evaluate the tests of Julia packages, we evaluated how many times the above rule's pattern occurs in real-life code in 3153 different packages. In total, we found 514 instances where code containing this pattern was executed, spread over 232 different packages. This pattern occurs, for example, when a broadcasted expression is used in different branches and is factored out by the programmer as a code simplification measure. Applying the rewrite rule does not undo the deduplication but only removes the superfluous call to `materialize`. Note that the discussed rule only matches expressions where the outer broadcasted function is a unary function.

For other patterns, e.g., an expression where the outer broadcasted function is a binary function, separate rules need to be written. Table 1 lists how many times our analysis found different multi-line broadcasting patterns that can all be rewritten to `broadcasted(~f, ~x)` and `broadcasted(~binop, ~x, ~y)`. The last three patterns are different configurations of broadcasting a binary function. Note that the matches for these three patterns are not independent: a match against the last pattern, which will result in two invocations of `materialize` being optimized out, implies the two other patterns can be matched and applied as well, each optimizing out only one `materialize` invocation. When the final code is extracted in such cases, the expression from the e-class containing the least materialize calls will be chosen.

4.4 Extending Broadcasting with Domain-Specific Batched Operations

By using specialized libraries and frameworks, programmers can achieve significant performance improvements. However, when their use case evolves to require custom behavior that is not yet supported by their library or framework, or when different competing frameworks with slightly different APIs are to be compared, programmers are required to rewrite parts of their code to either switch to different APIs or to implement the required functionality in the core language themselves. For example, instead of writing deep learning layers and non-linear

```
dropout(x, p_drop) = ...
function forward(σ, W, x, b, training::Bool, p_drop=0.1)
   logits = (W * x) .+ b
   if training
      return σ.(dropout(logits, p_drop))
   else
      return σ.(logits)
   end
end
```

Fig. 11. Straightforward code in the core programming language, without needing to use specialized functions with custom APIs.

```
function forward(σ, W, x, b, training::Bool, p_drop=0.1)
   if training
      logits = (W * x) .+ b
      return σ.(dropout(logits, p_drop))
   else
      return LuxLib.fused_dense_bias_activation(σ, W, x, b)
   end
end
```

Fig. 12. Manually rewritten source code that calls an optimized implementation of certain computations.

activation functions using core language constructs, programmers often resort to using frameworks or libraries that contain optimized implementations of matrix multiplications, activation functions, attention computation, and others [23, 54]. This works well for most common use cases, but falls apart when a programmer requires custom behavior that is not yet supported by the framework. In those cases, the programmer is required to rewrite parts of the program to stop using the framework and instead depend on a different framework or the core language. Instead of having to opt out of framework functionality and rewriting code, one could ideally write simple, readable code and have the frameworks themselves automatically discover opportunities to use their more performant implementations. Our system gives framework authors the option for implementing such discovery and optimization almost for free.

As an example, the code in Fig. 11 shows an implementation of the forward pass of a single fully connected feedforward layer. Several existing Julia packages allow us to compute such operations more efficiently [24, 33, 34]. For example, the programmer could rewrite the code as shown in Fig.12 to use the optimized `fused_dense_bias_activation` function from the Lux.jl deep learning framework [52]. A disadvantage of such manual rewriting is that the code becomes less readable for someone not familiar with the Lux.jl APIs. Moreover, the use of that specific framework's optimized implementation, which is in essence a decision about a non-functional aspect of the code, is then hard-coded in the code that specifies the functionality. This requires the developers and the maintainers of this code to be knowledgeable in the code's application domain as well as in the Lux.jl framework. Switching to other frameworks in the future will require rewriting code. Clearly, hard coding the dependence on Lux.jl and its APIs in the code has a number of downsides.

With our system, instead of rewriting the original code of Fig. 11, it suffices to add a rewrite rule that maps the original code pattern onto the optimized `fused_dense_bias_activation` function:

```
materialize(broadcasted(~σ, broadcasted(+, ~W::AbstractMatrix * ~x::AbstractMatrix, ~b::AbstractVector)))
  ⟶ LuxLib.fused_dense_bias_activation(~σ, ~W, ~x, ~b).
```

Combined with the multi-line broadcast rewriting rule explained above, which rewrites the original code into a single line without the intermediary variable `logits`, this rule maps the single-line broadcast expression onto a call to the library function `LuxLib.fused_dense_bias_activation`, identical to what would be generated for the code in Fig. 12.

This example illustrates how our approach allows for a better separation of concerns. Another advantage is that alternative rewrite rules can be provided for different Julia packages that provide different optimized implementations for similar functionality. Different rewrite rules also do not obfuscate each other. All rewrite rules can be represented in the e-graph. Only during extraction is the final optimized code materialized, optionally based on user-defined extraction costs for different functions. As such, rewrite rules from different authors are fully interoperable.

Similarly, without changing the source code, different rewrites can still occur by applying different sets of rewrite rules that can, for example, be offered by different Julia packages.

This example also shows how our system supports composing high-level, domain-specific optimizations with optimizations of core Julia primitives. In this way, its scope is much broader than recent EqSat optimizers like Cranelift.

4.5 Keeping up with evolving and emerging libraries for GEMMs

As an example of multiple libraries providing complementary functionality, consider General Matrix Multiplications (GEMMs), a cornerstone in many scientific domains. GEMMs are also often combined with element-wise operations or *elops* on their operands or result, for example with non-linear activation functions such as ReLU.

Many libraries exist to perform different GEMM variants [24, 65, 67], each offering different trade-offs and interfaces. For targeting NVIDIA GPUs, e.g., the cuBLAS library provides the highest performance for basic GEMM expressions on matrices with simple datatypes. It also offers some kernels that fuse often-used elops with the GEMM. For elops not supported by cuBLAS, the alternative GemmKernels.jl can be used [24]. This package leverages Julia's multiple dispatch and GPU programming support [15] to generate performant GEMM code on the fly, including for user-defined elops and for GEMMs on less commonly used data types such as dual numbers, which are useful for automatic differentiation [56].

Over time, such libraries evolve. For example, cuBLAS has grown to include ever more kernels with fused elops. Ideally, programmers should not have to rewrite their application code to make use of newly available functionality or when switching from one library to another. Instead, they should be able to write generic code that automatically makes use of the available functionality in libraries. Rewrite rules enable this.

Take for example the expression `C .= relu.(`α `* (f_a.(A) * f_b.(B)) +` β `* C)`, where `A`, `B`, and `C` are matrices and α and β are scalars. The different GEMM and linear algebra packages can each provide a custom set of rewrite rules that rewrite as much as possible of that expression into calls to optimized functions. For example, a rule added to the LinearAlgebra.jl package would allow this expression to be rewritten into a more efficient call to `mul!` that carries out the matrix multiplication, scaling by α and β, and accumulation into `C` at once. Then only the elops (`f_a`, `f_b`, and `relu`) remain to be executed in separate kernels. For particular argument types, such as 16-bit floating-point matrices in GPU memory, rewrite rules in the GemmKernels.jl package would even allow the complete expression to be fused into one single kernel. Other packages could still contain rewrite rules for other GEMM configurations or variants. Depending on which rules are loaded, the same code hence can be rewritten in different ways, exploiting the various and possibly evolving optimization opportunities that complementary libraries offer without having to rewrite the source code.

5 Compilation Time Evaluation

Julia is a just-ahead-of-time compiled language for scientific computing. By default, Julia automatically compiles a native, type-specialized version of a method just ahead of the first time it is to be executed on some combination of argument types. Julia hence positions itself somewhere between typical ahead-of-time compilers such as LLVM and gcc and just-in-time compilers such as Cranelift or those used in a Java VM. Compilation speed is hence important for Julia.

A user can also request ahead-of-time compilation of methods and optionally invoke our rewriting rule-based optimization, based on whether or not the investment in compilation time might yield a positive return on investment. As scientific software often has relatively long execution times, compilation times are hence not as critical as in traditional JIT compilers. Still, we aim for interactive compilation speeds with our work to allow for rapid prototyping.

5.1 Complexity Analysis of Equality Saturation

E-Graph Construction. We reimplemented the algorithm that converts IR into an e-graph introduced for Cranelift [25] in Julia. This algorithm starts by computing the dominator tree of the input SSA code. We reuse the dominator tree construction implementation from the Julia compiler. This implementation is based on an algorithm from Georgiadis's thesis [27], and runs in linear time with respect to code size. IR statements are visited in a pre-order dominator tree traversal to ensure that all dependencies of the current statement have already been visited. For each statement, the e-class dependencies are looked up and an e-node is constructed. If the e-node is not part of the e-graph yet, it is inserted in a new e-class, otherwise an existing e-class is associated with the statement. At the same time that the e-graph is being constructed, the CFG skeleton is being built. This involves pushing the corresponding e-classes of effectful statements in a CFG structure which has a worst-case execution time complexity of $O(N)$, where N is the code size, but an amortized complexity of $O(1)$.

Since no equalities are introduced yet during conversion from IR to e-graph inserting new e-classes in the e-graph does not require running the relatively expensive e-graph rebuilding procedure. As such, e-graph construction only requires inexpensive hash map lookups and insertions for each statement, which have a worst-case execution time complexity of $O(N)$ but an expected worst-case execution time of $O(1)$.

Overall, the worst-case execution time complexity of our e-graph construction is $O(N^2)$ but on average and in practice, as will be discussed in Section 5.2, the worst-case execution time complexity is $O(N)$.

E-Graph Saturation. In general, termination for EqSat is not guaranteed because the application of rewrite rules can introduce new rewrite opportunities indefinitely [61, 70]. In practice, this issue is most often addressed by EqSat timeouts in combination with heuristic schedulers that ensure that all rewrite rules get a chance to be applied by means of a rule application back-off mechanism [17, 72]. The non-terminating behavior of EqSat is due to repeated rule application and rebuilding the e-graph by joining newly discovered equivalences.

Extraction. Constructing the ILP problem entails a few different steps. First, simple cycles are determined using Johnson's algorithm [36], and the dominator tree is constructed. Since the number of cycles in a graph can grow exponentially in the number of nodes [36], in the worst case, the time required to construct the problem is exponential as well. In practice, however, e-graphs typically contain only a few short simple cycles. Next, the minimization objective itself can be constructed in $O(I \cdot N)$, with N the number of e-nodes, and I the number of statements in the CFG skeleton. Assuming a low number of class cycles, the problem construction time is asymptotically dominated by the construction of the class constraint (Equation 1). In the worst case, this takes $O(I \cdot N \cdot N_{c(max)} \cdot \chi_{(\max)} \cdot D_{(\max)})$ time, with $\chi_{(\max)}$ the maximum number of children e-classes of an e-node, $D_{(\max)}$ the maximum number of dominating statements, and $N_{c(max)}$ the maximum number of e-nodes in an e-class.

Solving the ILP problem is an NP-hard problem, but as will be shown in Section 5, solving times are of the same order of magnitude as construction.

5.2 Experimental Compilation Time Evaluation

We conducted our measurements on a single core of a 64-core AMD Ryzen Threadripper PRO 7985WX and 512GiB RAM, with Julia Version 1.11.5 (LLVM 16.0.6). Fig. 13 shows the time taken for the different steps of our optimization scheme for the examples of Fig. 5, Fig. 7, and Fig. 11's `forward` function, with their respective rewrite rules. We report the average time over 1000 runs for Fig. 5 and Fig 11, and 100 runs for the example from Fig. 7. Apart from the first invocation of the EqSat optimizations, which includes compiling the EqSat implementation itself, we see little variance (σ<3%) in the measurements.

Conversion. E-graph conversion, both from and to Julia IR, takes relatively little time for this example. To evaluate the run time of conversions for other code, Fig. 14 shows the time needed to convert between IR and E-Graph for many different Julia methods. These methods were selected arbitrarily by collecting all the methods

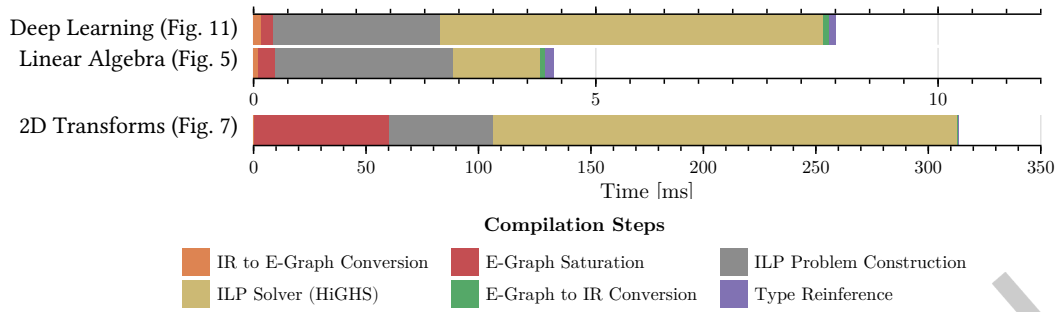

Fig. 13. Time spent in different compilation phases when optimizing different examples discussed in this work.

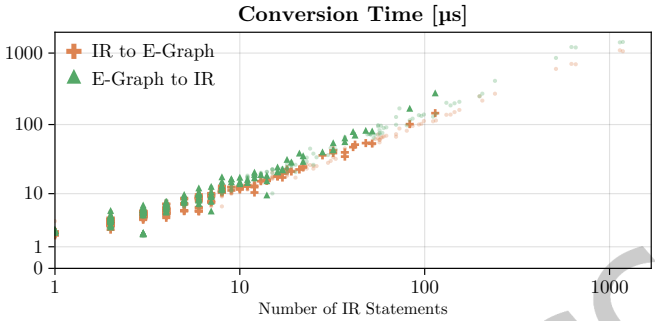

Fig. 14. Conversion times for converting from IR to e-graph and back. Smaller, faint markers indicate measurements on IR after inlining.

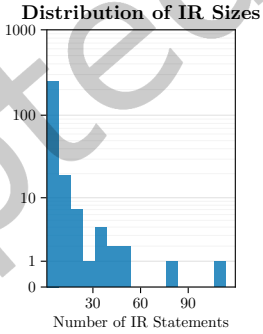

Fig. 15. A histogram of the number of statements before inlining in IR for typical Julia methods.

in the call graphs encountered during the compilation of some methods from Julia's core and linear algebra standard library[2]. While our optimization targets IR without inlined function calls, we also include timing results for IR with inlined calls to capture performance on longer IR. Fig. 15 shows that the vast majority of the evaluated methods consists of only a handful of IR statements. This means that we typically do not expect large e-graphs during optimization.

Code Generation. Not shown in the compilation time overview is the time that LLVM spends to optimize and generate the final executable code from Julia IR. The reason for the omission in the figure is that this step also occurs for regular Julia code compilation. This additional compilation time is dependent on the final IR size after extraction. For the code in Fig 11 and Fig 5, this takes 75%-90% of total optimization time. For the 2D transformation example (Fig 7), the saturation step takes a long time, and the extracted IR is very small (1 statement). In this case further code generation makes up less than 1% of total optimization time. For regular Julia code, much of the compilation cost imposed by LLVM is hidden by caching of compiled code. Tighter integration of our optimization into the Julia compiler could similarly help hide this cost as well.

ILP Construction and Solving. Fig. 13 shows that most of the time is occupied by constructing and solving the ILP extraction problem presented in Section 3.5. Apart from the inherent computational complexity of solving

[2]Concretely, we looked at `factorize(::Matrix{Float64})`, `sin(::Int)`, `print(::Matrix{Int})`, and `svd(::Matrix{Int})` but a different selection produces similar results.

	ILP Construction [ms]	ILP Solve [ms]	Greedy Solve [µs]	Variables	Constraints	E-Classes	E-Nodes
Fig. 5	2.6	1.3	12.5	126	364	26	28
Fig. 7	46.0	206.6	285.1	918	5325	337	1,002
Fig. 11	2.4	5.6	19.1	48	168	22	23

Table 2. Time needed to construct and solve the ILP problems for different examples discussed in this work, and the corresponding number of variables and constraints in the ILP problem, and e-nodes and e-classes in the e-graph. Also shown is the time taken for greedy extraction–expressed in microseconds.

the ILP problem, the conversion takes a large amount of time for converting the e-graph into a collection of variables and constraints in an external ILP solver library. Table 2 shows the time required to construct and solve the ILP problem for several use cases discussed in this paper. With the exception of the code for simplifying 2D transformations in Fig. 7, e-classes do not contain many equivalences, leading to fast problem construction and solving. For the 2D transformation example, particular rewrite rules, e.g., those that combine two subsequent rotations into one rotation, give rise to an infinite number of rewrite opportunities. In these instances, saturation runs until a timeout is reached. If fast compilation times are required for pathological cases like the one shown, we can fall back to a greedy extraction algorithm. In the future, better scheduling of rules and detection of these cases could limit the amount of e-graph blow-up as well.

Greedy Extraction. Above, we evaluated the ILP extraction as this approach generates the optimal solution with respect to a defined cost model. However, in many cases, a greedy extraction algorithm that locally selects the lowest-cost e-node in each e-class produces similar results while being faster. In fact, with the exception of the linear algebra example from Fig. 5, in the examples above, greedy extraction produces the same results as ILP extraction.

Table 2 shows the time taken for greedy extraction as well. Especially for the 2D transformation example, greedy extraction is much faster than ILP extraction (~720×).

6 Performance Evaluation

Any performance optimization that can be obtained with rewrite rules can also be obtained by manually rewriting source code, if necessary by duplicating code (possibly lifting code from libraries) and then special-casing it. Our work hence does not aim to squeeze more performance out of the Julia programming language and its compiler and runtime than what any team of programmers could achieve with unlimited time to rewrite all the source code on which their software depends. Instead, our work aims to support application programmers and library developers to exploit optimization opportunities without requiring extensive rewriting of their own and each other's code.

Still, as the goal of the rewrite rules is to enable optimizations, this section reports the performance gains that can be achieved for some of the previously presented use cases. In all cases, timings were collected by taking the minimum execution time of at least 10 samples. This way, we factor out Julia's ahead of time compilation and the occasional garbage collection, the two main causes of outliers. We observed no significant difference in run time variation across sample runs of original code on the one hand, and across sample runs of optimized code on the other hand.

The use case of Fig. 7, where domain-specific code representing a 2D transformation is optimized into an idempotent function, shows a large speed-up of 16.5×, as the only execution time remaining for the optimized function is its call overhead.

The other examples paint a more interesting picture. For the use case of Fig. 5, where the computation of a matrix determinant is simplified by reusing the already computed eigenvalues, Fig. 16 shows a moderate speed-up across the board (up to 1.05×). For reference, yellow dots show the speed-up when simply omitting the

determinant computation completely. In other words, they show what speed-up can be achieved by reducing the execution time of the determinant computation to zero. The fact that those roughly match with the optimized code shows that in the rewritten code, the determinant value basically comes for free, whereas it costs up to 5% in the original code.

Similarly, for the use case of Fig. 11 where the forward pass of a neural network is replaced with a fused implementation, the plot in Fig. 17 shows a speed-up for small problem sizes (up to 2.5×) but ends up about as fast as the original code for larger problem sizes, as the time spent in the matrix multiplication tends to dominate. [3]

Finally, we look at the example of rewriting the expression `C .= relu.(α * (f_a.(A) * f_b.(B)) + β * C)`, discussed in Section 4.5. Fig. 18 shows results of the basic linear algebra rewrite, which fuses everything except the elops (`f_a`, `f_b`, `relu`) into one GPU kernel. This provides a moderate speed-up across the board over the baseline implementation in which each operation in the whole expression corresponds to a separate GPU kernel. The GemmKernels rewrite, which fuses all operations into a single kernel, is worthwhile for smaller problem sizes (up to 4.8×), and shows a small slowdown for larger problems, where the core GEMM computation dominates.

This performance evaluation of four uses cases demonstrates that our system can indeed be used to achieve significant speed-ups by exploiting domain-specific optimization opportunities, in a programmer-friendly manner.

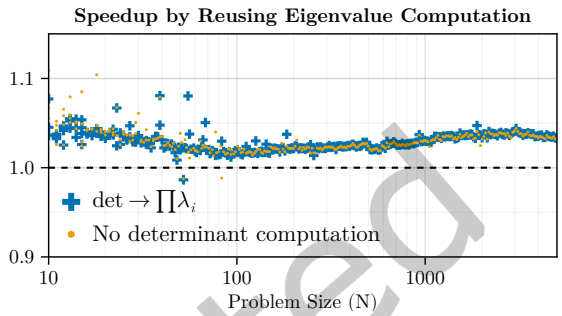

Fig. 16. Speed-up over the original code of Fig. 5. The problem size N refers to the 64-bit floating point input matrix of size $N \times N$. Computation was run on the CPU.

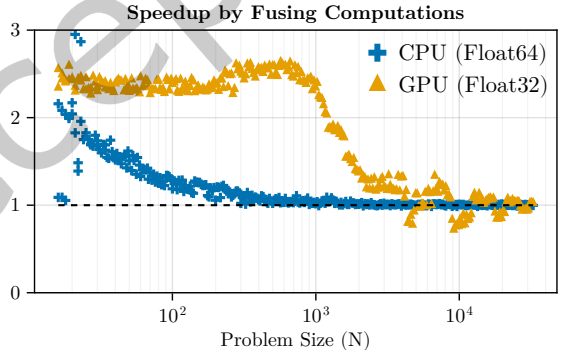

Fig. 17. Speed-up gained over the original code of Fig. 11. The problem size N corresponds to the size of the matrices used in the computation: $\sigma\left(\square^{N\times N} \times \square^{N\times 32} + \square^{N\times 1}\right)$

7 Limitations

Control Flow and Inlining. As discussed in Section 3, our approach to converting IR to an e-graph using the CFG skeleton currently does not support rewrites that alter control flow. Similarly, since we apply rewrites on code where function calls have not yet been inlined, rewrite opportunities where part of an expression is hidden behind a function call are not considered. We optimize each method separately. Other intermediate representations, discussed in Section 8, could be used to represent the entire program in an e-graph and allow rewriting across call boundaries and control flow.

Saturation and Extraction. In this work, we have proposed an ILP formulation for optimal e-graph extraction.

[3] For problems involving GPU computation (Fig 17, Fig. 18), we measured execution time on a machine with a 16-core Intel Xeon E5-2637 v2, NVIDIA GeForce RTX 2080 Ti (12GB), and 64GB RAM, with Julia Version 1.11.5 (LLVM 16.0.6). For the other problems, we use the same setup as in Section 5.

The problem of extracting e-nodes from a general e-graph is NP-hard [28], which means that there is no guarantee that an optimal solution can be found in polynomial time. Currently, when the ILP solver is unable to find a solution within a given timeframe, we fall back to a greedy extraction algorithm that is built into Metatheory.jl. In this work, we chose a simple cost model that treats the cost of nodes independent of other nodes. A more complex cost model could take into account more complex behavior that depends on multiple nodes at once.

Rule Expressiveness. As discussed in Section 3, our rule syntax takes one Julia expression and transforms it into another. This is an intuitive syntax, but does not fully cover all the possible rules a developer could possibly want to express. For one, the rewrite rules can only be used to rewrite a *single* expression into another expression. Multi-pattern rewrite rules to match against and to rewrite multiple expressions at once are currently not supported. We also do not yet support rewrite rules matching against function calls taking an unknown, variable amount of variables. A developer may hence have to write multiple similar rules for the same function with different argument counts.

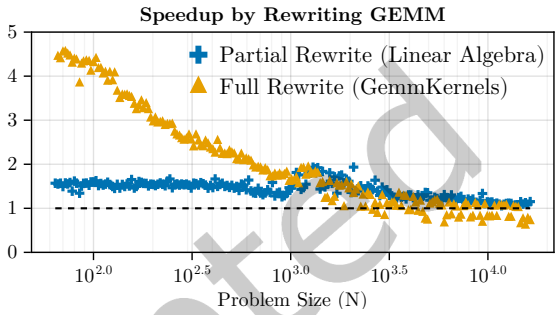

Fig. 18. Speed-up when rewriting the GEMM expression from Sec. 4.5 into a combination of elops and an optimized `mul!` operation (blue), and a single fused GemmKernels.jl kernel (yellow). The problem size N refers to the 16-bit floating point input matrices of size $N \times N$. All computations on GPU.

Rule applicability. Section 6 shows that certain rewrites do not always yield positive returns. Consequently, the burden is currently on the programmer to select the rule sets for their particular problem. In the future, this could be automated using an improved cost model, in combination with more detailed program analysis to gauge the effectiveness of rewrite rules. Alternatively, the rule applicability could be made more extensible, allowing package developers to decide programmatically when a rule can be applied. Lastly, development is currently ongoing in the Julia language to support statically sized arrays, opening up the possibility to use this information in the e-graph to determine rule applicability in the future.

8 Related Work

8.1 Equality Saturation

EqSat and its use as a code optimization tool was first introduced by Tate et al. [64], where it was used to optimize Java bytecode. Willsey et al. [70] introduced algorithmic improvements for EqSat providing asymptotic speed-ups. This has led to an explosion of new work exploring different applications for EqSat, from optimizing tensor programs [59, 69, 72], to compiler verification [38, 60], optimizations for specialized hardware [47, 66, 68], and other purposes [50, 53].

Metatheory.jl [18] provides a framework for EqSat in Julia. It has already been used for code optimization on symbolic representations of Julia code to speed up PDE solvers [29]. That work, however, only handles pure, symbolic programs, as it operates on a symbolic representation of Julia code obtained through manual expression building or tracing through heavily restricted, straight-line code. In contrast to our work, the existing use of Metatheory.jl does hence not allow the presence of general control flow or side effects.

To the best of our knowledge, by introducing type propagation in the e-graph, we are the first to apply EqSat to a dynamic programming language. This allows rewrites to coexist and complement Julia's multiple dispatch feature, as discussed in Section 4.

8.2 Intermediate Representations and E-Graphs

Cranelift, a compiler back-end for WebAssembly and Rust, is among the first to use e-graphs and EqSat in a production-grade compiler [25]. While we reuse ideas and methods first introduced in the Cranelift compiler, we implemented our framework as a standalone project, not reusing any Cranelift components. Our code is integrated in the Julia compiler using user-land compiler extensions.

The Cranelift compiler does not do full EqSat, but only recognizes equalities the first time a node is inserted in the e-graph. This implies that it is possible for equalities to remain undiscovered, but it obviates the fixed-point loop present in full EqSat and ensures there are no cycles in the e-graph which allows for a more efficient representation of the graph in memory and leads to easier extraction. We use the same e-graph representation of code as used in Cranelift, but by utilizing an extraction scheme that enforces acyclic extraction, the optimizer is able to carry out full EqSat. Whereas Cranelift and our work operate on IR that is complemented by a CFG, other work has sought to find alternative IR formats that allow representing control flow fully in the e-graph. Tate et al. [64] have introduced Program Expression Graphs (PEGs) which represent constructs such as loops as special expressions and show that PEGs can be used to do EqSat. Laird et al. [39] raise supported LLVM IR into a functional IR that also allows control flow transformations. Similarly, Regionalized Value State Dependency Graphs (RVSDGs) [14, 55] represent control flow as (nested) expressions. A research prototype already exists using RVSDGs in conjunction with EqSat for a simple toy language [58]. Our approach remains closer to the original SSA-based IR of the source language, allowing less expensive conversion routines. Lastly, there are multiple works applying EqSat on MLIR code [48, 73]. These do not keep track of side effects and control flow in a CFG skeleton and thus can only operate on code that is pure. Additionally, they operate on statically typed IR in contrast to our work.

Willsey et al. introduced e-class analyses, a framework to associate additional metadata with each e-class [70]. The metadata is kept up to date throughout the EqSat process by potential modifications whenever new information is available, for example, when e-classes are merged. An exemplary existing use case for an e-class analysis is constant propagation, where each e-class can optionally carry a constant value. Further work has shown that this technique can be combined with abstract interpretation for more complex program analyses [20]. To track types in the e-graph, we designed a novel e-class analysis, as discussed in Section 3.2.

8.3 Extraction

We based our ILP optimal extraction formulation on the work of He et al. [30]. Recent work proposes other formulations, for example by looking at e-graphs through the lens of circuits [62], or finite state automata [71]. These formulations can lead to faster extraction or provide termination guarantees under particular assumptions. Cranelift operates in a just-in-time context where fast and reliable extraction is crucial. For this reason, they use an extraction algorithm that greedily tries to minimize the total number of nodes, without taking into account node reuse in extracted expressions or from expressions that dominate them. In contrast to Cranelift's greedy algorithm, we have introduced an ILP formulation that aims for node reuse from other nodes within an extracted expression, as well as from nodes in expressions that dominate them in the CFG.

8.4 Julia Code Optimization

Other works exist that aim to improve the performance of Julia code with custom optimizations. Symbolics.jl [29] can be used to transform scientific machine learning code into a symbolic representation on which simplifications can be applied. Finch.jl [12, 13] uses a custom IR to represent iteration patterns of loop nests over sparse or structured arrays. We instead focus on optimizing code in Julia's own IR format, allowing users of our system to target any Julia code.

8.5 Domain-Specific Languages

Domain-specific languages and their optimization has been explored in many other contexts. Jones et al. demonstrated the application of domain-specific rewrites in the GHC compiler [37]. In contrast to EqSat, their rewrites are applied greedily and destructively to the program, hence suffering from the phase-ordering problem. Other works focus on designing extensible, high-level IRs to express optimized domain-specific programs directly [42, 43, 57]. We consider that work complementary to ours, as similar techniques to those discussed in this paper could be used to optimize such high-level IRs using equality saturation.

9 Conclusion and Future Work

We have developed a novel system that allows Julia developers to write domain-specific rewrite rules that are automatically applied using EqSat. Our system works in the presence of control flow and side effects. Through an e-class analysis, we keep track of the most specific type of all equivalent terms and use this information to support type-constrained rewrite rules. Unlike previous work, our system can rewrite dynamically-typed code. We introduce a technique called CFG skeleton relaxation that allows rewrite rules to nullify side effects in the original code. We have adapted an ILP formulation for optimal, acyclic e-graph extraction to take into account value reuse from dominating statements. Finally, we show that our system can be used to optimize a variety of domain-specific code, enabling developers to achieve significant speed-ups by applying rewrite rules.

In the future, our work can be extended to support rewriting control flow and apply rewrites that span different call depths, by integrating function call inlining in the EqSat procedure. Further improvements to our extraction scheme could also lead to more efficient code being generated.

All artifacts will be made available upon acceptance of the paper.

References

[1] 2019. MLIR Rationale. https://mlir.llvm.org/docs/Rationale/

[2] 2025. abstractarray.jl. https://github.com/JuliaLang/julia/blob/9118ea7565feae13d5b47654a3c245c0df36e753/base/abstractarray.jl#L2485-L2497

[3] 2025. CUDA.jl. https://github.com/JuliaGPU/CUDA.jl/blob/860eb88e40053b2709ef949f2eaf593c59bcecf1/lib/cusparse/generic.jl#L48-L70

[4] 2025. CUDA.jl. https://github.com/JuliaGPU/CUDA.jl/blob/860eb88e40053b2709ef949f2eaf593c59bcecf1/lib/cusparse/generic.jl#L304-L335

[5] 2025. float.jl. https://github.com/JuliaLang/julia/blob/9118ea7565feae13d5b47654a3c245c0df36e753/base/float.jl#L320-L328

[6] 2025. float.jl. https://github.com/JuliaLang/julia/blob/9118ea7565feae13d5b47654a3c245c0df36e753/base/math.jl#L756-L766

[7] 2025. float.jl. https://github.com/JuliaGPU/CUDA.jl/blob/860eb88e40053b2709ef949f2eaf593c59bcecf1/lib/cusolver/linalg.jl#L35-L48

[8] 2025. intfuncs.jl. https://github.com/JuliaLang/julia/blob/9118ea7565feae13d5b47654a3c245c0df36e753/base/intfuncs.jl#L592-L606

[9] 2025. Julia Documentation. https://docs.julialang.org/en/v1/base/base/#Base.@assume_effects

[10] 2025. LinearAlgebra.jl. https://github.com/JuliaLang/LinearAlgebra.jl/blob/98a0f1500da3b92a60bd18a11d49e7acf2a2267e/src/bidiag.jl#L1472-L1503

[11] 2025. NNlib.jl. https://github.com/FluxML/NNlib.jl/blob/81e6cd1843dbd2baf2b7b2efd00e942f4f040a2d/src/fold.jl#L186-L191

[12] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. 2024. Finch: Sparse and Structured Array Programming with Control Flow.

[13] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proc. 21st ACM/IEEE Int'l Symposium on Code Generation and Optimization (CGO '23)*. ACM, 41–54.

[14] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. 2015. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Transactions on Architecture and Code Optimization* 11, 4 (Jan. 2015), 1–25.

[15] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (April 2019), 827–841.

[16] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. arXiv:1209.5145 [cs.PL] https://arxiv.org/abs/1209.5145

[17] Alessandro Cheli. 2021. Automated Code Optimization with E-Graphs. arXiv:2112.14714 [cs.PL]

[18] Alessandro Cheli. 2021. Metatheory.Jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation. *Journal of Open Source Software* 6, 59 (2021), 3078.

[19] Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Automating Constraint-Aware Datapath Optimization Using E-Graphs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[20] Samuel Coward, George A. Constantinides, and Theo Drane. 2023. Combining E-Graphs with Abstract Interpretation. In *Proc. 12th ACM SIGPLAN Int'l Workshop on the State Of the Art in Program Analysis*. ACM, 1–7.
[21] Leonardo de Moura and Nikolaj Bjørner. 2007. Efficient E-Matching for SMT Solvers. In *Automated Deduction – CADE-21*. Springer, 183–198.
[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Vol. 4963. Springer Berlin Heidelberg, 337–340.
[23] Radwa Elshawi, Abdul Wahab, Ahmed Barnawi, and Sherif Sakr. 2021. DLBench: A Comprehensive Experimental Evaluation of Deep Learning Frameworks. *Cluster Computing* 24, 3 (Sept. 2021), 2017–2038.
[24] Thomas Faingnaert, Tim Besard, and Bjorn De Sutter. 2022. Flexible Performant GEMM Kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 33, 9 (Sept. 2022), 2230–2248.
[25] Chris Fallin. 2023. Aegraphs: Acyclic e-Graphs for Efficient Optimization in a Production Compiler. EGRAPHS 2023 keynote.
[26] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (Dec. 1999), 381–391.
[27] Loukas Georgiadis. 2005. *Linear-Time Algorithms for Dominators and Related Problems*. Ph. D. Dissertation. Princeton University.
[28] Amir Kafshdar Goharshady, Chun Kit Lam, and Lionel Parreaux. 2024. Fast and Optimal Extraction for Sparse Equality Graphs. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 361 (Oct. 2024), 27 pages.
[29] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwóźdź, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. 2022. High-Performance Symbolic-Numerics via Multiple Dispatch. *ACM Communications in Computer Algebra* 55, 3 (Jan. 2022), 92–96.
[30] Mike He, Haichen Dong, Sharad Malik, and Aarti Gupta. 2023. Improving Term Extraction with Acyclic Constraints.
[31] Kenneth Hoste and Lieven Eeckhout. 2008. Cole: Compiler Optimization Level Exploration. In *Proc. 6th Annual IEEE/ACM Int'l Symposium on Code Generation and Optimization*. ACM, 165–174.
[32] Q. Huangfu and J. A. J. Hall. 2018. Parallelizing the Dual Revised Simplex Method. *Mathematical Programming Computation* 10, 1 (2018), 119–142.
[33] Mike Innes. 2018. Flux: Elegant Machine Learning with Julia. *Journal of Open Source Software* 3, 25 (May 2018), 602.
[34] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. 2018. Fashionable Modelling with Flux. arXiv:1811.01457 [cs.PL] https://arxiv.org/abs/1811.01457
[35] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. ACM, 47–62.
[36] Donald B. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 4, 1 (1975), 77–84. arXiv:https://doi.org/10.1137/0204007 doi:10.1137/0204007
[37] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, Vol. 1. 203–233.
[38] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. 2022. Caviar: An e-Graph Based TRS for Automatic Code Optimization. In *Proc. Int'l Conference on Compiler Construction (CC 2022)*. ACM, 54–64.
[39] Avery Laird, Bangtian Liu, Nikolaj Bjørner, and Maryam Mehri Dehnavi. 2024. SpEQ: Translation of Sparse Codes using Equivalences. *Proc. ACM Program. Lang.* 8, PLDI, Article 215 (June 2024), 24 pages. https://doi.org/10.1145/3656445
[40] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Int'l Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, 75.
[41] 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
[42] 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 Int'l Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14. doi:10.1109/CGO51591.2021.9370308
[43] Roland Leißa, Marcel Ullrich, Joachim Meyer, and Sebastian Hack. 2025. MimIR: an extensible and type-safe intermediate representation for the DSL age. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 95–125.
[44] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2021. MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In *Proc. 2021 ACM SIGSAC Conference on Computer and Communications Security (Ccs '21)*. ACM, 2183–2196.
[45] Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. 2023. JuMP 1.0: Recent Improvements to a Modeling Language for Mathematical Optimization. *Mathematical Programming Computation* 15, 3 (Sept. 2023), 581–589.

[46] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. 2017. Exploiting vector instructions with generalized stream fusion. *Commun. ACM* 60, 5 (April 2017), 83–91. doi:10.1145/3060597
[47] Kazuaki Matsumura, Simon Garcia De Gonzalo, and Antonio J. Peña. 2023. A Symbolic Emulator for Shuffle Synthesis on the NVIDIA PTX Code. In *Proc. 32nd ACM SIGPLAN Int'l Conference on Compiler Construction (CC 2023)*. ACM, 110–121.
[48] Jules Merckx, Alexandre Lopoukhine, Samuel Coward, Jianyi Cheng, Bjorn De Sutter, and Tobias Grosser. 2025. eqsat: An Equality Saturation Dialect for Non-destructive Rewriting. arXiv:2505.09363 [cs.PL] https://arxiv.org/abs/2505.09363
[49] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc.
[50] Chandrakana Nandi, Max Willsey, Amy Y. X. Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5 (Oct. 2021), 1–28.
[51] Charles Gregory Nelson. 1980. *Techniques for Program Verification.* Ph. D. Dissertation. Stanford University.
[52] Avik Pal. 2023. *Lux: Explicit Parameterization of Deep Neural Networks in Julia.* doi:10.5281/zenodo.7808903
[53] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proc. 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1–11.
[54] Aniruddha Parvat, Jai Chavan, Siddhesh Kadam, Souradeep Dev, and Vidhi Pathak. 2017. A Survey of Deep-Learning Frameworks. In *2017 Int'l Conference on Inventive Systems and Control (ICISC)*. IEEE, 1–7.
[55] Nico Reissmann, Jan Christian Meyer, Helge Bahmann, and Magnus Själander. 2020. RVSDG: An Intermediate Representation for Optimizing Compilers. *ACM Transactions on Embedded Computing Systems* 19, 6 (Nov. 2020), 1–28. arXiv:1912.05036 [cs]
[56] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. 2016. Forward-Mode Automatic Differentiation in Julia. arXiv:1607.07892 [cs.MS] https://arxiv.org/abs/1607.07892
[57] Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. 2023. Codon: A compiler for high-performance pythonic applications and dsls. In *Proc. Int'l Conf. on Compiler Construction*. 191–202.
[58] Jamey Sharp. 2022. Optir.
[59] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proc. 5th Int'l Symposium on Machine Programming*. ACM, 21–31.
[60] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-Based Translation Validator for LLVM. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 737–742.
[61] Dan Suciu, Yisu Remy Wang, and Yihong Zhang. 2025. Semantic Foundations of Equality Saturation. arXiv:2501.02413 [cs]
[62] Glenn Sun, Yihong Zhang, and Haobin Ni. 2024. E-Graphs as Circuits, and Optimal Extraction via Treewidth. arXiv:2408.17042 [cs.DS]
[63] swiftlang. 2012. Swift Intermediate Language (SIL).
[64] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, 264–276.
[65] L. A. Torres, Carlos J. Barrios H, and Yves Denneulin. 2024. Evaluation of computational and energy performance in matrix multiplication algorithms on CPU and GPU using MKL, cuBLAS and SYCL. arXiv:2405.17322 [cs.DC] https://arxiv.org/abs/2405.17322
[66] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. 2022. IMpress: Large Integer Multiplication Expression Rewriting for FPGA HLS. In *2022 IEEE 30th Annual Int'l Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1–10.
[67] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (June 2015), 33 pages. doi:10.1145/2764454
[68] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. Vectorization for Digital Signal Processors via Equality Saturation. In *Proc. 26th ACM Int'l Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 874–886.
[69] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endowment* 13, 12 (Aug. 2020), 1919–1932.
[70] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM on Programming Languages* 5, POPL (Jan. 2021), 1–29.
[71] Y. Wang, James Koppel, Altan Haan, and Josh Pollock. 2022. E-Graphs, VSAs, and Tree Automata: A Rosetta Stone.
[72] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268.
[73] 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

Received 3 July 2025; revised 14 January 2026; accepted 21 January 2026