

CSE 507 Final Report

Noah Huck, Saul Shanabrook, Marko Subotic, Yash Mathur

December 5, 2025

1 Introduction

Egglog [Zha+23] and SMT solvers are both useful tools with a variety of applications in compilers, floating point, and program analysis. However, many subproblems in these fields are efficiently solvable in only one of the two, and they use very different representations of their data. To use egraphs and SMT solvers together to solve a problem, one must convert their intermediate results between two representations, adding computational overhead and making it hard to interleave solver execution effectively.

Our tool solves these problems by extending egglog to support SMT queries and models in a variety of theories as values. Through z3 calling primitives interleaved with egglog execution, users can implement analyses that effectively take advantage of a powerful SMT solver and fast equality saturation at the same time. We developed several example analyses using our system, including polynomial equality checking and factoring, SMT query rewriting, and integer overflow checks.

2 Overview

We implemented our tool in Rust on top of the egglog-experimental repository, adding additional SMT-LIB [BFT16] functionality exposed to the user as egglog primitives. In the backend, we have our own internal representation of SMT queries and models, which interface with the existing smtlib-rs [con25c] bindings for Rust to call the z3 solver. We tried to make our changes to the core of egglog and the smtlib-rs library as minimal as possible, but we occasionally discovered that smtlib-rs had minor bugs or did not expose a feature we wanted to expose, requiring us to fork our own version of it to make small changes.

SMT has been used in conjunction with egg [Wil+20] in the past in the **Enumo** project [Smi+24]. **Enumo** used SMT and equality saturation to infer efficient domain specific ruleset. This project could be re-implemented in egglog using the SMT bindings we have created. Our SMT bindings could also be used to expand the suite of rewrites available in tools like Herbie [Pan+15], which currently uses egglog to improve the accuracy of floating point computations, as described below.

3 Algorithms and Logical Encodings

We exposed custom primitives for queries and models in SMT theories including EUF, integer and real arithmetic, and bitvector arithmetic, as egglog values, as shown in Figure 1. Models returned by SMT queries converted from SMT-LIB can have their values extracted and used to drive egglog rewrite rules, and queries themselves can also be rewritten.

SMT-LIB

```
(set-option :print-success false)
(set-option :produce-models true)
(set-logic QF_UF)

(declare-fun f (Int) Int)
(declare-fun g (Int) Int)

(assert (= (f 0)
           (g 0)))
(assert (= (g 1)
           (f 1)))

(assert
  (not
    (and (= (f 1)
            (g 0))
          (= (g 1)
            (f 0))))))

(check-sat)

(get-model)
(exit)
```

egglog

```
(let f_fun (smt-fn-int "f" "Int"))
(let g_fun (smt-fn-int "g" "Int"))

(let c1
  (smt==
    (smt-call f_fun (smt-int 0))
    (smt-call g_fun (smt-int 0))))

(let c2
  (smt==
    (smt-call g_fun (smt-int 1))
    (smt-call f_fun (smt-int 1))))

(let c3
  (not (and
    (smt==
      (smt-call f_fun (smt-int 1))
      (smt-call g_fun (smt-int 0)))
    (smt==
      (smt-call g_fun (smt-int 1))
      (smt-call f_fun (smt-int 0))))))

(let model (smt-solve c1 c2 c3))

(check (= true (smt-sat? model)))

(extract model)
```

Figure 1: Uninterpreted functions and model inspection in SMT-LIB versus egglog.

We also developed several examples in our extended version of egglog, to demonstrate and explore how our bindings could be used. High-level descriptions of the encodings are given below.

3.1 Symbolic Calculus

We extend an existing symbolic calculus example in egglog to support more general rewrite conditions. This demonstrates how embedding an SMT solver within egglog lets you rewrite more precise checks than would otherwise be feasible. The original example [con25a] is written in egg [Wil+20] and then translated into egglog [con25b]. Many of the rewrite rules fire only if they can prove that a certain term is not zero, such as `(rewrite (Div a a) (Const 1.0) :when ((is-not-zero a)))`. The original implementation is unsound, treating any value that is not the constant zero to be nonzero. In our implementation, we translate the expression first into the theory of reals, converting many operations to their SMT equivalents. Unsupported operations, such as integration, we convert to uninterpreted functions, reducing but not completely eliminating unsoundness. Using the SMT bindings provides us with a safe way to gate rewrites on certain conditionals, without having to compute them manually within egglog.

For future work, we could extend this to cover the checks in the file for differentiation, which

only trigger if an expression is independent of a variable. These currently are only covered for special cases where the variables are different or they are a constant. If we translate this to SMT instead, we could more generally query if the expression is independent of the declared variable.

3.2 Factoring Polynomials

Another analysis enabled by our tool is polynomial factoring. Our implementation rewrites input polynomials in a single variable with rational coefficients, which can be represented in a list-like sum-of-monomials form in egglog, to a product of factors whenever all of their factors are rational numbers. This analysis could be used in a system such as Herbie, where it would enable chains of rewrites like $\frac{x^3-x^2-x+1}{x-1} \mapsto \frac{(x-1)^2(x+1)}{x-1} \mapsto (x-1)(x+1)$ when it is otherwise known that $x \neq 1$.

The analysis relies on both egglog and SMT features running together to be effective. It begins by creating egglog functions to track the degree and coefficient of a polynomial. It iteratively finds roots with egglog-computed SMT queries - we want to find rational roots, so for a polynomial $P(x)$ we use the constraint $x = p/q \wedge q \neq 0 \wedge P(x) = 0$ for integers p, q , using a model's assignments to those variables to determine one. Once a root has been found, derivatives of the polynomial are successively taken in egglog using the power rule, enabling us to use the fact that the minimum n such that $P^{(n)}(x)|_{x=r} \neq 0$ is the multiplicity of the root r . While the sum of multiplicities of roots found is less than a polynomial's degree, it adds an additional constraint that $x \neq r$ for the newest root, repeating this process until all roots are found and merging it with its factored form in the egraph. If at any point the solver returned UNSAT when trying to find a root, we stop and do not perform the rewrite, as there is no factored form with all rational roots.

3.3 Comparing Polynomials

Our tool can be used to improve equality checking of polynomials. When rewriting polynomials, egglog uses rules such as associativity and commutativity (AC), resulting in an egraph of exponential size in the size of the polynomial. Given two candidate polynomials, our SMT bindings can directly check equality after straightforward conversion. Such an analysis could be useful in verifying the results of a tool that is known to provide useful rewrites, but may be unsound, generating polynomials that are not actually equivalent. Rather than filling up the egraph with intermediate terms, using the SMT rewrite allows for direct comparison, quickly validating or rejecting candidates from an unsound tool.

3.3.1 Overflow Checking

We implemented the overflow checking example from class in egglog. The overflow check detects signed 64-bit overflow in the sum `hi0 + hi1 + c`, where `c` is a carry-in from lower-word arithmetic. In two's-complement arithmetic, signed overflow occurs exactly when the operands have the same sign and the sign of the sum differs from that sign. A common low-level formulation expresses this condition using bitwise operations as an SMT query:

$$\text{sign}(\neg(\text{hi0} \oplus \text{hi1}) \ \& \ (\text{hi0} \oplus (\text{hi0} + \text{hi1} + \text{c}))).$$

Egglog's expressiveness allows this bit-level overflow pattern to be matched by a semantic rewrite rule. Using equality saturation, the expression can be proven equivalent to the higher-level Boolean condition

$$(\text{sign}(\text{hi0}) = \text{sign}(\text{hi1})) \ \wedge \ (\text{sign}(\text{hi0} + \text{hi1} + \text{c}) \neq \text{sign}(\text{hi0})).$$

4 Summary

4.1 Results of Polynomial Equality Checking

While most of our examples were intended to explore the added program design space allowed by our tool, polynomial equality checking is an analysis that can be implemented in standard egglog. Figure 2 compares results for vanilla rewrite rules and SMT bindings in checking equality between polynomials differing only by associativity and commutativity. SMT achieves runtime on the order of milliseconds even for large polynomials, while AC rewrites timeout (120+ seconds) on polynomials with 20 or more terms.

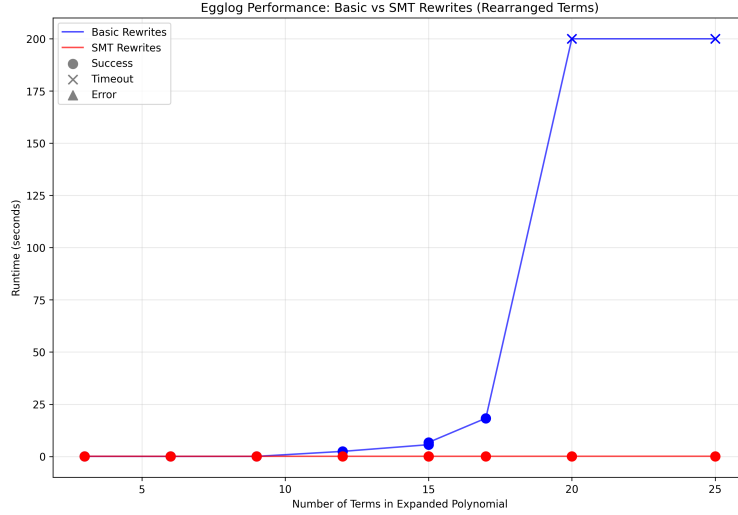


Figure 2: Runtime of equality checking for polynomials differing by AC rewrites. Egglog ran only until the equality was verified, not until saturation.

4.2 Implementation Challenges and Future Changes

There are a number of ways we could have chosen to integrate an SMT library with Egglog. We chose to add new custom primitives. This didn't require any changes to the core of egglog and was entirely implemented as extensions to the existing egglog-experimental repo, building on public interfaces exposed by egglog. This means that it integrates nicely with other extensions and should continue to work as egglog evolves. It also minimizes the complexity handled by users writing SMT queries, as we added egglog sorts and primitives for each supported theory, which could be combined into queries and passed to the solver with a single dedicated command.

We also had to choose how to implement the internal data representations and solver interface. There were again a number of options, such as creating an SMT-LIB file directly and calling out to a solver like z3 or cvc5, or using bindings from existing Rust libraries. We opted to use SMT-LIB as our abstraction layer, so that we could swap the solver if we wanted to at a later date. We used the high level bindings exposed by the `smtlib-rs` [con25c] library, which were updated recently and had some usage. After extended work on the project, if we were to further develop this prototype, we would use another strategy, as these bindings did not expose all the SMT functionality we needed, added an unnecessary layer of abstraction and complexity, and contained some minor bugs. If we were to continue on this project or restart it, we would consider either using the low level bindings

to more directly interact with SMT-LIB, or generating SMT-LIB and parsing outputs directly from a solver. We would also want to be more agnostic about how we handle solver output, allowing users to accept or reject different types of output as fits their needs, rather than trying to fit all solver output into a specific representation.

Another issue we dealt with was how to type check uninterpreted functions in SMT. We wanted these to type check under egglog, so that they would return the output sort and only run with the correct input sorts. However, getting this to work with egglog’s existing type checking was difficult. We ended up choosing a middle road, where inputs to uninterpreted functions are only checked at runtime, while return types are checked statically, so that we didn’t have to create a new sort for each function signature. If we were to continue this project, we think that it would make sense to make the type checking more precise for functions. This would require defining functions as a pre-sort and having users instantiate a new one for each function signature, so that it type checks correctly.

Overall, to continue this project we would want to try integrating into some larger real world existing egglog use cases to see how they could use an SMT solver as part of their workflow. Some examples could be using it inside of herbie, which has an egglog backend, or within eggcc to do implement bound checking.

5 Teamwork

We all came in with very different experience using egglog and Rust. By the end of the quarter, we were all comfortable adding new egglog features in Rust and creating examples in egglog. We practiced pair programming to help us explore these topics collaboratively, while also each taking on some areas independently. Noah was focused on the use cases around polynomials. He implemented the theory of real arithmetic and created the example analyses for polynomials. Marko extended Noah’s equality-checking demo to generalize it to arbitrary size polynomials and understand the performance characteristics with scaling, implemented uninterpreted functions, and worked with Yash to implement the Int sort. Yash implemented a BitVec sort and the example of the rewrite from class. Saul created the initial project outline with the Bool sort and implemented model getting logic, as well as helping Marko brainstorm around the UF implementation.

6 Course Topics

We would not have been able to attempt this project without the background offered by the course. In particular, foundational concepts around model checking and verification were instructive in how we designed the SMT bindings in egglog. We also used the background on how SMT combines different theories under one framework. We exposed a number of theories in the bindings and had to understand how they can be composed and the limitations on them in order to use them. More broadly, the lectures on the motivation behind Rosette were instructive in showing how embedding an SMT library in another language can provide a powerful interface. Examples of applications of SMT from lecture guided our example analyses. For example, the last guest lecture gave an example of performing rewrites on a query before sending it the SMT engine, which we reimplemented to show using egglog as a term replacement system can be used in such a context. Some topics that would have been useful to see include detailed engineering examples using SMT, such as how Rosette translates user input to solver calls, and a wider survey of use of egglog in fields like compilers.

Code and Presentation Slides

GitHub

Google Slides

References

- [BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. www.SMT-LIB.org. 2016.
- [con25a] Egg contributors. *math.rs test file in the egg equality saturation library*. GitHub repository file. Tests for the egg library’s mathematical rewrites. 2025. URL: <https://github.com/egraphs-good/egg/blob/main/tests/math.rs>.
- [con25b] Egglog contributors. *Refactor Math tests*. GitHub pull request. Pull request #757, commit 87f5b0c. Aug. 2025. URL: <https://github.com/egraphs-good/egglog/pull/757>.
- [con25c] smtlib-rs contributors. *smtlib Rust library*. GitHub repository. 2025. URL: <https://github.com/oeb25/smtlib-rs>.
- [Pan+15] Pavel Panchekha et al. “Automatically improving accuracy for floating point expressions”. en. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, June 2015.
- [Smi+24] Gus Henry Smith et al. “Scaling program synthesis based technology mapping with equality saturation”. In: *arXiv [cs.PL]* (Nov. 2024).
- [Wil+20] Max Willsey et al. “egg: Fast and Extensible Equality Saturation”. In: *arXiv [cs.PL]* (Apr. 2020).
- [Zha+23] Yihong Zhang et al. “Better together: Unifying Datalog and equality saturation”. en. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023), pp. 468–492.