Save Time and Mechanize

GEORGE ZAKHOUR, University of St. Gallen, Switzerland

1 INTRODUCTION

The gold standard for programming languages semantics are equational theories: operational or denotational. Some languages offer constructs such as typeclasses that encapsulate equational systems while others (e.g., Agda, Lean, and Rocq) offer an engine to manipulate equalities within the language to state and prove properties. Code correctness arguments rely on these equations and optimizers use them to search for optimal programs that compute *equal* values.

Storing and querying equalities is thus crucial. Here two standard data structures help: the union-find [10, 28] which efficiently represents equivalence relations, and the e-graph [18, 29] which extends the union-find with congruence. Both are used in tools such as Z3 [7] and CVC5 [2] (solvers), Simplify [8], TheSy [26], and CCLemma [13] (theorem provers), Babel [3], Herbie [21], and Ruler [17] (synthesizers), and Cranelift [1] (optimizer).

I am developing Propel [33]—an automated inductive theorem prover for programs satisfying algebraic (equational) laws. My co-authors and I demonstrated its effectiveness for Conflict-Free Replicated Datatypes in [32]. The use of e-graphs was natural for Propel's implementation but it was not possible since Propel tracks *disequalities* (unequal expressions) to prune the proof space earlier and thus prove more faster. To integrate e-graphs, we extended them to disequalities in four variants [31], we formally studied the properties of each variant, and discussed the approach that some tools take to reason with disequalities. The next challenge is one that every prover—Z3 and CVC5 included—must address. **How to guarantee the correctness of the e-graph's answer?**

Existing Work. To gain trust in the e-graph's response, provers such as Z3, CVC5, and others use an e-graph that produces proofs derived from **metadata stored alongside the e-graph which doubles its memory footprint** [19]. While proofs can be examined manually, they are often not as **proofs are large**. For example, Flatt et al. [9] tell of a circuit optimizer that optimized a circuit in under a minute but the proof needed 4 hours to verify. **Generating smaller proofs is time consuming** and takes between $O(n \log n)$ and $O(n^5)$ with n being the number of equalities [9].

For example, whenever the state-of-the-art Lean tactic `egg` [23, 24] is invoked, it serializes the goal for a Rust backend that uses the egg library [29], receives back new equalities and proofs, translates these into Lean, and checks the proofs before admitting the new equalities.

Can We Do Better? Using **a mechanically verified e-graph** rids us of the memory and runtime penalties of generating proofs. Put another way, the proof of the e-graph's correctness *is the proof certificate of any equality* which can only be checked once.

Contributions. Towards a verified efficient e-graph, I mechanized a generic yet efficient union-find in Lean [16]. The implementation is publicly available [30] and performs well compared to a Rust implementation (Section 3) demonstrating that verified code does not need to be slow.

2 BACKGROUND: UNION-FINDS AND E-GRAPHS

Union-Find. The union-find [10, 28] allows storing and querying an equivalence relation. It is a forest where two elements are equal whenever they are in the same tree. Each element has a representative: the root of its tree, and to decide equivalence, the representatives are compared. To equate two elements, the representative of one is added to the children of the other.

Author's address: George Zakhour, george.zakhour@unisg.ch, University of St. Gallen, Switzerland.

```
structure UFData τ where parent: α τ τ

abbrev root ufd x := ufd.parent x == x

structure UF τ extends d: UFData τ where
  depth : Erased (τ → Nat)
  dz: ∀ r, d.root r → depth r = 0
  dlt: ∀ r, ¬d.root r →
    depth (d.parent r) < depth r

def ∅ := ( α.id, λ _ ↦ 0 }

def of | []=>∅ | (a,b)::e=>(of e).union a b
```

```
theorem reflexive (uf: UF τ): ∀ x, uf ⊢ x≃x
theorem symmetric (uf: UF τ):
  ∀ x y, uf ⊢ x≃y → uf ⊢ y≃x
theorem transitive (uf: UF τ):
  ∀ x y z, uf ⊢ x≃y → uf ⊢ y≃z → uf ⊢ x≃z
theorem closure (eqs: List (τ × τ)):
  ∀ eq ∈ list, of eqs ⊢ eq.1 ≃ eq.2
theorem smallest (eqs: List (τ × τ)):
  ∀ (R: τ → τ → Bool),
    (∀ x, R x x) → (∀ x y, R x y → R y x) →
    (∀ x y z, R x y → R y z → R x z) →
    (∀ ab ∈ eqs, R ab.1 ab.2) →
   ∀ a b, (of eqs ⊢ a ≃ b) → R a b
```

Fig. 1. Simplified snippets from the Lean mechanization [30] with the key theorems on the right.

E-Graphs. In the union-find it may be that $x = y$ without $f(x) = f(y)$. To guarantee congruence, e-graphs use union-finds and hash-conses: maps from elements to unique identifiers. A hash-cons allows sharing sub-expressions, so the f node of $f(x, x)$ points twice to the *same* x, making the AST a directed graph. The domain of the union-find is the unique identifiers and two expressions are equal when their identifiers are equal in the union-find. To equate two expressions their identifiers are equated in the union-find and the dependencies on identifiers are recursively followed.

3 A MECHANIZED AND EFFICIENT COINDUCTIVE UNION-FIND

The Forest. Algebraic data types are famously the go to for modeling tree-shaped data. But implementing the union-find of elements of type τ using ADTs quickly becomes complicated since its operations "recurse upwards" towards roots. Thus an element must always be accompanied with a context [11, 15]. So instead of the ADT $\mu x.\tau + x \times \tau \times x$ we can treat the whole forest as a graph and use $u : \tau \longrightarrow \tau$ with the invariant that a measure on τ exists that decreases after applying u except for its fixed points which are the roots in the forest with measure 0.

The left of Fig. 1 shows simplified definitions from the mechanization [30]. Particularly `depth` is the measure and `dz` and `dlt` specify the invariant. In the empty union-find all elements are roots, i.e. all elements are fixed points, i.e. the union-find is the identity. Due to space limitations `find`, `union`, and the equivalence $\cdot \vdash \cdot \simeq \cdot$ judgement are omitted [30]. The right of Fig. 1 shows the five key theorems: that supplying equalities to the empty union-find generates the smallest reflexive, symmetric, transitive closure. While the proofs of the last four are common, to the best of my knowledge this is the first presentation of the proof of the first.

Coinduction. The union-find is coinductive: it's the terminal coalgebra of the $X \mapsto \tau^\tau \times X^{\tau \times \tau}$ functor. Informally, equivalences exist in an ambient space (the type of its elements) so querying it at an outsider to the equalities it closes over is not an error: it may have just not seen it yet. So if τ is infinite why shouldn't the union-find be too? With u not only can the union-find contain an infinite number of elements, but also an infinite number of equalities. For example the union-find with $u : \mathbb{N} \longrightarrow \mathbb{N} = x \mapsto 2\lfloor x/2 \rfloor$ and the measure $x \mapsto x \bmod 2$ contains all the natural numbers and equates every odd number to its predecessor which is chosen as the representative.

Efficiency. Unioning two elements requires modifying the root of one subtree. To update u the old one must be captured in a closure and later conditionally called as is classically done [22]. The more the union-find is updated the more closures are captured and the more are unwrapped when one is called, thus degrading the linear complexity of the union-find [28] to become quadratic. While functions are easy to reason about, practically programmers use (hash)maps or arrays [20]. The last conceptual contribution is to realize that a mechanization that captures the common interface α (in Fig. 1) between functions and maps is the best of both worlds.

Case Study. To witness the efficiency of the mechanization, I wrote a program which processes the 163 million connections in the 2018 English WikiLinkGraph dataset [4] and a list of article

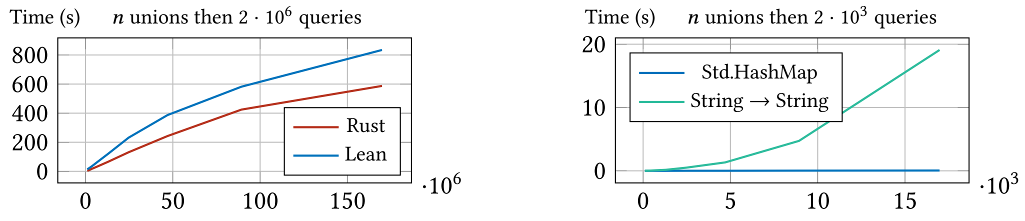

Fig. 2. The time to build a union-find of n unions followed by some queries using a Rust (red), a Lean with hashmaps (blue), and a Lean with functions (green) implementation. Lower is better.

pairs and which prints those which are not mutually reachable. At its heart is a union-find built from the connections encoding the "reachable" relation. The program has three variants: in Rust using a hashmap for the tree, in Lean using its hashmap, and in Lean using function closures.

Fig. 2 first shows that the speed of the first Lean variant is comparable to that of Rust, taking 13 min to build the full union-find and check 1 million pairs while the Rust implementation takes 10 min. Then, it shows that the representation with closures suffers dramatically from a quadratic blowup as each query must unwrap many closures before returning.

4 TOWARDS AN E-GRAPH MECHANIZATION

Coinduction had one more benefit for the mechanization of union-finds: it eliminated `add` and `contains`. Not only was the mechanization effort easier, but the statements of theorems became simpler. For example none of the theorems in Fig. 1 are conditioned by containment nor were the definitions of `union` and `find`. *Having everything present simplified everything.*

E-Graphs are not clearly coinductive. When the hash-cons and the union-find of an e-graph are infinite then its recursive `union` operation never terminates since the proof relies on the decreasing number of roots in the union-find. For example, to union x with y, the e-graph must union $f(x)$ with $f(y)$ then $f(f(x))$ with $f(f(y))$ ad infinitum because all terms are in the e-graph.

Deferred rebuilding is a popular union strategy proposed by egg [29] that separates the loop from `union`. It would also suffer from nontermination for the same reason, but I suspect that it can be adapted to guarantee termination. The key observation is that each equality affects a finite subgraph which—once rebuilt—renders the infinite dependency chain unreachable. In the next steps I wish to explore this strategy and contribute the first coinductive functional e-graph.

5 FUTURE WORK

The mechanized e-graph will be the foundation for the much needed e-graph extensions to integrate e-graphs into Propel. Four extensions are necessary: (1) data-analysis [29] enables useful analyses for optimization and verification such as disequalities [31], (2) e-matching [6] allows efficient search for expressions (a must for applying theorems), (3) conditional reasoning [5, 27] avoids cloning the whole e-graph in every branch of a conditional expression, and (4) α-equivalence as used in slotted e-graphs [25], to enable higher-order reasoning, a hallmark of Propel. Beyond, these mechanizations are useful for the community as they could serve as the new trusted backend for the `egg` Lean tactic, for developing an efficient and verified solver in Lean, or even for the optimizer [12] of a certified compiler such as CompCert [14].

ACKNOWLEDGEMENT

I extend my thanks to Prof. Dr. Guido Salvaneschi for the supervision of the project and my PhD and to my colleagues Dr. Pascal Weisenburger and Jahrim Gabriele Cesario with whom I will be developing e-graph extensions and verify them with the mechanization presented here.

REFERENCES

[1] Bytecode Alliance. 2025. Cranelift. https://cranelift.dev/. Accessed: 15 October 2025.

[2] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442.

[3] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. 2023. babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 396–424.

[4] Cristian Consonni, David Laniado, and Alberto Montresor. 2019. *WikiLinkGraphs: A complete, longitudinal and multilanguage dataset of the Wikipedia link networks.* https://doi.org/10.5281/zenodo.2539424

[5] Samuel Coward, Theo Drane, and George A. Constantinides. 2024. Constraint-Aware E-Graph Rewriting for Hardware Performance Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), 14 pages. https://doi.org/10.1109/TCAD.2024.3483096

[6] Leonardo De Moura and Nikolaj Bjørner. 2007. Efficient E-matching for SMT solvers. In *International Conference on Automated Deduction*. Springer, 183–198.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473.

[9] Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022. Small proofs from congruence closure. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 75–83.

[10] Bernard A Galler and Michael J Fisher. 1964. An improved equivalence algorithm. *Commun. ACM* 7, 5 (1964), 301–303.

[11] Gérard Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (Sept. 1997), 549–554. https://doi.org/10.1017/S0956796897002864

[12] 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 *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3497776.3517781

[13] Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. CCLemma: e-graph guided lemma discovery for inductive equational proofs. *Proceedings of the ACM on Programming Languages* 8, ICFP (2024), 818–844.

[14] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[15] Conor McBride. 2008. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. *SIGPLAN Not.* 43, 1 (Jan. 2008), 287–295. https://doi.org/10.1145/1328897.1328474

[16] Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*. Springer, 625–635.

[17] Chandrakana Nandi, Max Willsey, Amy 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, OOPSLA, Article 119 (Oct. 2021), 28 pages. https://doi.org/10.1145/3485496

[18] Charles Gregory Nelson. 1980. *Techniques for program verification.* Stanford University.

[19] Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*. Springer, 453–468.

[20] Robert Nieuwenhuis and Albert Oliveras. 2007. Fast congruence closure and extensions. *Information and Computation* 205, 4 (2007), 557–580.

[21] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.* 50, 6 (June 2015), 1–11. https://doi.org/10.1145/2813885.2737959

[22] Benjamin C. Pierce et al. 2015 - (revised 2025-08-24). Maps: Total and Partial Maps. In *Software Foundations, Volume 1: Logical Foundations*, Benjamin C. Pierce (Ed.). Electronic textbook, University of Pennsylvania. Available online: https://softwarefoundations.cis.upenn.edu/lf-current/Maps.html (accessed 30-10-2025).

[23] Marcus Rossel. 2024. An Equality Saturation Tactic for Lean (Master's thesis).

[24] Marcus Rossel, Rudi Schneider, Thomas Kœhler, Michel Steuwer, and Andrés Goens. 2026. Towards Pen-and-Paper-Style Equational Reasoning in Interactive Theorem Provers by Equality Saturation. *Proceedings of the ACM on Programming Languages* POPL (2026).

[25] Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Kœhler, and Michel Steuwer. 2025. Slotted E-Graphs: First-Class Support for (Bound) Variables in E-Graphs. *Proceedings of the ACM on Programming Languages*

9, PLDI (2025), 1888–1910.

[26] Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 125–148.

[27] Eytan Singher and Shachar Itzhaky. 2024. Easter egg: Equality reasoning based on E-graphs with multiple assumptions. In *2024 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 70–83.

[28] Robert Endre Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)* 22, 2 (1975), 215–225.

[29] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.

[30] George Zakhour. 2025. UnionFind in Lean4. https://github.com/geezee/union-find-lean

[31] George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 2282–2305.

[32] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2023. Type-Checking CRDT Convergence. *Proc. ACM Program. Lang.* 7, PLDI, Article 162 (June 2023), 24 pages. https://doi.org/10.1145/3591276

[33] George Zakhour, Pascal Weisenburger, and Guido Salvaneschi. 2024. Automated Verification of Fundamental Algebraic Laws. *Proc. ACM Program. Lang.* 8, PLDI, Article 178 (June 2024), 24 pages. https://doi.org/10.1145/3656408