An Equality Saturation Tactic for Lean

Marcus Rossel
Supervised by Andrés Goens

June 21, 2024

Masters Thesis at Technische Universität Dresden
Chair for Compiler Construction

TU Dresden Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Master Thesis

For:	**Marcus Rossel**
Degree program:	Informatik (Master)
Matriculation number:	4744651
E-mail:	marcus.rossel@tu-dresden.de

Topic: **An Equality Saturation Tactic for Lean**

Equality saturation is a powerful method in automated reasoning for working with rewrites, allowing one to explore large rewrite spaces and optimize terms in them. Recent work [2] has shown it to be a practical method for several applications. In some cases, fully automatic approaches do not scale well, when the search spaces become too large. Guided equality saturation [1] has shown semi-automatic approaches to be very effective in this case. In particular, it showcases a prototype of a proof tactic in the Lean Theorem Prover, based on the egg library. This allows proof sketches to be written in a manner akin to pen-and-paper proofs, leaving out details and skipping steps. However, this prototype is limited: it supports only a small fragment of Lean and does not take advantage of many other capabilities of the underlying library, egg. The goal of this thesis is to improve on the current implementation to support a larger fragment of Lean, use more of egg, and be overall more useful in practice. In particular, this Master Thesis shall include the following tasks. The student shall:

- Learn and understand equality saturation based on e-graphs, as well as the egg library.
- Learn and understand Lean's expression language and associated metaprogramming functionality.
- Integrate egg with Lean via a proof tactic for automated proofs of equality.
- Evaluate the efficacy of the tactic (and potentially different approaches) by constructing a suitable test suite.
- Optional: Integrate the tactic with other forms of proof automation, like Aesop.
- Optional: Extend the core egg tactic with conditional rewriting capabilities.

References

[1] Thomas Koehler et al. "Guided Equality Saturation." In: *Proc. ACM Program. Lang.* POPL (2024).

[2] Max Willsey et al. "egg: Fast and Extensible Equality Saturation." In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434304. URL: https://doi.org/10.1145/3434304.

Referee:	Prof. Dr.-Ing. Jeronimo Castrillon
Supervisor:	Dr. Andres Goens

Declaration

I declare that I have prepared this thesis independently and that I used only the references and auxiliary means indicated in the thesis.

Marcus Rossel

Dresden — June 21, 2024

Abstract

Recent years have seen a number of milestones in the use of *interactive theorem provers* for formalization and verification of research-level mathematics. The *Lean* theorem prover, in particular, has been at the forefront of this development. Yet, formalization still incurs a large overhead compared to pen-and-paper proofs, thus creating a need for improved proof automation. At the same time, the *e-graph* data structure for representing congruence relations has seen a resurgence in the area of automated term rewriting. In particular, this development has been fueled by the technique of *equality saturation* from the field of program optimization, and a corresponding efficient implementation by the *egg* library. In [KGB+24], this library is used to develop a proof of concept proof tactic for equational reasoning in Lean based on equality saturation. In this thesis, we improve upon this proof of concept and turn it into practical proof automation. This requires various techniques for bridging the gap between the semantics of Lean's λ calculus and the syntax-driven rewriting in e-graphs. We also add new capabilities such as rewriting under binders, conditional rewriting, and guide terms. Our results indicate that our proof automation significantly improves upon the proof of concept, and shows promise for practical applicability.

Contents

1 Introduction

In recent years, machine-checkable formalization of mathematics has seen an increase in popularity. While landmark projects like the computer-assisted proofs of the Kepler Conjecture and the Four Color Theorem have existed for decades, they differ from "normal" mathematics in that they initially required computer-assistance for the mere computational power. In contrast, more recent notable formalization projects target current research mathematics [BCM20, PS23, TT23]. Aside from the immediate upside of automated verification of correctness, these projects aim to reap novel benefits of formalization, like digitization and indexing of mathematical theorems, and large-scale collaboration [Rin24, Mas21]. A major hurdle in this endeavor is the large overhead involved in translating pen-and-paper mathematics into machine-checkable form, as computer systems require meticulously detailed definitions and proofs. A direct means of combating this issue is by improving proof automation. The related field of *automated theorem proving* has been pursued since at least the 1950s [HUW14], and has given rise to many successful decision procedures. Most notably, SAT solvers for deciding satisfiability of propositional formulas, which are also used in various other procedures like answer set programming, satisfiability modulo theories and bounded model checking. While automated theorem proving has thus proven useful for specific problem domains, it has seen limited success in the realm of general mathematical reasoning. The more prominent approach in this area is *interactive theorem proving*, a field aimed at constructing proofs by the interaction and guidance of a human and a computer. Important milestones in this area include the systems *Automath* (1968) [dB68] which introduced proofs as first-class objects, *LCF* (1972) which ensured that proofs can only be constructed by operations corresponding to inference rules, and *HOL* (1980s) which emphasized the use of conservative extensions over axioms [Gor00]. The ideas introduced in these systems, as well as the mathematical field of type theory, lay the foundations for the expressiveness and trustworthiness of modern interactive theorem provers like Isabelle/HOL, Coq, and Lean. However,

> "opinions on the relative values of automation and interaction differ greatly. To those familiar with highly efficient automated approaches, the painstaking use of interactive provers can seem lamentably clumsy and impractical by comparison. On the other hand, attacking problems that are barely within reach of automated methods (typically for reasons of time and space complexity) often requires prodigious runtime and/or heroic efforts of tuning and optimization, time and effort that might more productively be spent by simple problem reduction using an interactive prover." [HUW14]

It should therefore come as no surprise that automated and interactive theorem provers have seen an exchange of ideas in an attempt to mitigate their respective weaknesses. While automated theorem provers are introducing more flexible and expressive languages[1], interactive theorem provers are improving their automation capabilities. This has also led to more symbiotic approaches where interactive theorem provers are used as frontends which rely on automated theorem provers as backends. A prominent example of this is *Sledgehammer* [BBN11], which uses Isabelle/HOL as a frontend and calls various backends like SMT solvers, linear arithmetic solvers, and superposition provers.[2]

> "Sledgehammers and machine learning algorithms have led to visible success. Fully automated procedures can prove 40% of the theorems in the Mizar math library, 47% of the HOL Light/Flyspeck libraries, with comparable rates in Isabelle. These automation rates represent an enormous savings in human labor." [BKPU16]

Other theorem provers, like Lean, have seen a slightly different approach towards proof automation. Instead of a large "hammer tactic", it features an abundance of smaller domain-specific proof automation tools. For example, in this thesis, we consider proof automation for

[1] For example, in TLA$^+$ [Lam99].

[2] `https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf`

equational reasoning in Lean, introduced in [KGB+24]. The tool is based on a single proof automation backend for *equality saturation* [TSTL09], which is a technique for automated term rewriting based on *e-graphs* [NO80]. While e-graphs are usually used for congruence closure inside of tools like SMT solvers, they have seen more widespread use in recent years. This is, in part, a result of an efficient implementation for e-graphs and equality saturation by the *egg* library [WNW+21]. By connecting Lean to egg, [KGB+24] develops a proof of concept for automated equational reasoning, which exhibits capabilities which are currently lacking in other prominent proof automation tools. In particular, by virtue of equality saturation, it is able to perform rewriting without getting stuck at locally optimal solutions, whereas existing proof automation (like simp) performs greedy rewriting up to a local optimum. However, the implementation of [KGB+24] is intended only as a proof of concept, and does not hold up in practical use. Thus, in this thesis, we revisit their approach and try to improve it up to the point of practical usability. A major component of this is handling the differences between Lean's expression semantics and the syntax-driven approach of e-graphs. Understanding these differences requires background in Lean's type theory and implementation, as well as e-graphs and equality saturation. Thus, we start by introducing equality saturation and egg in Section 2, and Lean and its underlying theory in Section 3. Subsequently, we develop our proof automation for equational reasoning in Section 4, and finally evaluate it in Section 5.

2 Equality Saturation with egg

Egg is a Rust library for *equality saturation*, a technique for rewriting expressions while maintaining a congruence relation over them. Equality saturation is used for program optimization [TSTL09, Kœh22, KGB+24] as well as automated theorem proving [DMB08, KGB+24]. In this section, we first review the techniques underlying equality saturation and then show how egg builds on and modifies them.

2.1 Equivalence Closure with Union-Finds

The need for efficient representations of *equivalence* relations in computers goes at least as far back as the 1960s. In [AGG61] equivalence relations are represented as a sequence of *equivalence declarations*. An equivalence declaration is a list of elements which are declared to be equivalent, like $(x, 5)$. A sequence of equivalence declarations then represents an equivalence relation by taking the *equivalence closure* over the equivalence declarations. For example, if we have an equivalence relation defined by $(x, 5)(z, x)(4, w)$, then the elements in the first two equivalence declarations belong to the same equivalence class (as they share x), while the last equivalence declaration forms its own equivalence class. Determining whether given elements belong to the same equivalence class thus requires an algorithm. In the subsequent work [GF64], summarized in [Knu97], equivalence declarations are no longer used to *represent* the equivalence relation but are rather taken to be inputs used to construct directed trees which represent the equivalence relation. Each tree represents a distinct equivalence class with the root node being the (arbitrary) *representative* of the class. An example of this is shown in Figure 1.

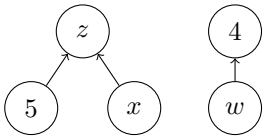


Figure 1: Tree-representation of the equivalence relation $\{\{x, 5, z\}, \{4, w\}\}$.

Based on this representation, checking equivalence of two elements can be reduced to checking whether they belong to equivalence classes with the same representative. The operation of finding the representative of the class of a given element e is aptly called the *find* operation

and is easily implemented by traversing upwards from e until a root node is found. For this approach to work, we need to maintain the property that equivalent elements are members of the same tree. Thus, we need to merge existing trees if any of their members are declared to be equivalent by a new equivalence declaration. As merging trees corresponds to taking the union of their equivalence classes, we call the associated operation *union*. Merging trees is easily achieved by attaching the representative of one tree as a child of the representative of the other tree.

As the data structure we have described is based on a tree structure with *union* and *find* operations it has become known as a *union-find*. The *union* and *find* operations can be made more efficient by performing maintenance steps as outlined in [Tar75]. When performing a *find*, the children visited during the traversal of the tree can be added as direct children of the representative in order to flatten the tree (which leads to faster *find*s). A similar optimization is achieved by ensuring that *union* operations attach the tree with fewer node as a child of the tree with more nodes. With both of these techniques applied, the time it takes to perform u *union*s with $f > u$ intermixed *find*s is bounded by $\Theta(f \cdot \alpha(f, u))$ where α is "related to a functional inverse Ackermann's function and [thus] very slow-growing" [Tar75, NO05]. Thus, *union*s and *find*s are practically constant amortized time operations.

2.2 Congruence Closure with E-Graphs

While union-finds can be used for constructing *equivalence* relations, they cannot maintain *congruence* relations without additional information. The notion of congruence is only meaningful in a context where the elements of the union-find are known to be terms constructed from applications of function symbols. Thus, [NO80] uses a union-find in conjunction with a labeled DAG, called the *term graph*, for representing the structure of terms. Each node in the term graph represents the term obtained by traversing the graph starting at the given node. The union-find then constructs equivalence classes over the nodes of the term graph as shown in Figure 2a.

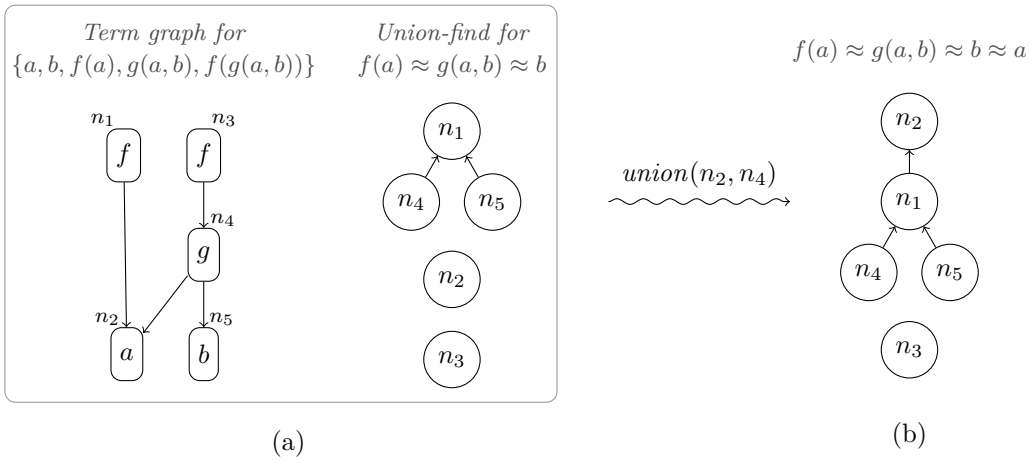


Figure 2: (a) Example of an e-graph as in [NO80]. (b) Example of how *union* does not preserve congruence.

The combination of a union-find and a term graph has become known as an *e-graph* [Nel80]. The equivalence classes of the union-find are then called *e-classes* with the nodes of the term graph being *e-nodes*. By ensuring that a given term is represented by at most one e-node in the term graph, the e-graph's relation maintains reflexivity while also keeping the term graph compact. Symmetry and transitivity of the relation are retained by checking equivalence via *find*. Maintaining congruence, instead of just equivalence, in an e-graph's union-find requires additional work. The reason being that *union*s do not uphold congruence, as shown in Figure 2b. Here we apply *union* to the e-nodes representing a and $g(a, b)$, but the e-nodes of

the now congruent $f(a)$ and $f(g(a,b))$ remain in separate e-classes. Thus, we introduce a new *merge* operation for processing equivalence declarations, which extends *union* such that congruence of the e-graph's relation is maintained.

For this, we first define the notion of congruence over e-nodes. Given an e-node n, let $\lambda(n)$ be its label, $\delta(n)$ its out-degree and $n[i]$ its ith child in the term graph. We call e-nodes n_1 and n_2 *congruent* if

1. $\lambda(n_1) = \lambda(n_2)$
2. $\delta(n_1) = \delta(n_2)$
3. $\forall\, 1 \leq i \leq \delta(n_1),\ n_1[i]$ *and* $n_2[i]$ *are congruent*

Given equivalence declaration (n_1, n_2), where n_1 and n_2 are e-nodes, the *merge* operation then performs the following steps.

1. If n_1 and n_2 are already in the same equivalence class, do nothing.
2. Otherwise, collect all parent nodes P_1 and P_2 pointing to n_1 and n_2 respectively in the term graph.
3. $union(n_1, n_2)$
4. For each pair of parent nodes $p_1 \in P_1$ and $p_2 \in P_2$, if $find(p_1) \neq find(p_2)$, $\lambda(p_1) = \lambda(p_2)$, and all their children are pairwise equivalent, then $merge(p_1, p_2)$.

Thus, the given e-nodes are *union*ed, while all nodes which have become congruent as a result are *union*ed, as well. Accordingly, this operation comes at a higher runtime cost with *merge* taking $\mathcal{O}(m)$ amortized time, where m is the number of edges in the term graph.

Congruence closure with e-graphs has been adopted in the legacy SMT solver Simplify [DNS05] as well as the widely successful SMT solver Z3 [DMB08]. Notably, both of these systems rely on a technique called *e-matching* for enabling equivalence declarations with quantified variables.

2.3 E-Matching

The congruence closure procedure presented in Section 2.2 works only on equivalence declarations relating *ground terms*, that is, terms containing no quantified variables. Thus, they cannot encode universally quantified theorems like $\forall\, x \in \mathbb{N} : x = x + 0$. In the context of SMT solvers, "[t]he typical solution to this problem is to generate ground instances of the quantified [formulas] and hope that the particular instances generated are the ones required" [MŁK08]. To determine *which* ground instances to generate, a common approach was pioneered by [DNS05]. It considers only those instances relevant which instantiate the quantified variables in such a way that the resulting ground terms are congruent to terms already contained in the term graph. Or stated conversely, it deems all instances irrelevant which contain terms not represented in the current e-graph. The problem of finding such relevant instances is called the *e-matching problem* and, following [MŁK08], can be stated as follows.

Given a set of constant and function symbols Σ and a set of variables $\mathcal{V}$, let $\mathcal{T}$ denote the set of terms over Σ and $\mathcal{V}$.[3] For $a \in \mathcal{T}$, let $\mathcal{V}_a$ be the set of variables contained in a. Given an e-graph e, we denote by $\mathcal{G} \subseteq \mathcal{T}$ the set of ground terms represented by e's term graph, with $\mathcal{R} \subseteq \mathcal{G}$ being those ground terms which are the representatives of their e-class (their e-nodes are fixed points under *find*). We write $a_1 \approx_e a_2$ to mean that ground terms a_1 and a_2 are congruent according to e. The solution to the e-matching problem for *pattern term* $p \in \mathcal{T}$ is then the set of substitutions:

$$\mathcal{S} = \{\sigma : \mathcal{V}_p \to \mathcal{R} \mid \exists\, g \in \mathcal{G},\, \sigma'(p) \approx_e g\}$$

[3] More precisely, the set of Σ-*terms over* $\mathcal{V}$ as defined in [FV22].

By σ' we denote the homomorphic extension of σ to a function with signature $\mathcal{T} \to \mathcal{G}$. Having σ only map to terms which are representatives of their e-class has the effect of avoiding redundant substitutions. Namely, those which would result in congruent ground terms. Based on e-matching, we can compute the congruence closure in an e-graph e with respect to a quantified equation $\forall\, x_1, ..., x_n : \mathit{lhs} = \mathit{rhs}$ as follows.

1. Compute $\mathcal{S}$ by e-matching on the pattern term lhs with $\mathcal{V} = \{x_1, ..., x_n\}$.
2. For each equivalence declaration (l, r) in $\{(\sigma'(\mathit{lhs}), \sigma'(\mathit{rhs})) \mid \sigma \in \mathcal{S}\}$, add l and r to e's term graph and $\mathit{merge}(l, r)$.

There are two important caveats to this procedure. First, our choice of e-matching on lhs instead of rhs causes problems as soon as $\mathcal{V}_{\mathit{rhs}} \not\subseteq \mathcal{V}_{\mathit{lhs}}$, because then $\sigma'(\mathit{rhs})$ is not a ground term. Thus, we only match on lhs if $\mathcal{V}_{\mathit{rhs}} \subseteq \mathcal{V}_{\mathit{lhs}}$, and match on rhs if $\mathcal{V}_{\mathit{lhs}} \subseteq \mathcal{V}_{\mathit{rhs}}$. We call the chosen pattern term the *trigger*. For some theorems, variable inclusion is given in neither direction which makes them incompatible with this procedure. Second, the given procedure does not necessarily compute the (full) congruence closure with respect to the given equation. The reason being that applying this procedure may produce new terms which can thus not have been considered during e-matching. Hence, for some equations like $\forall\, x \in \mathbb{N} : x = x + 0$, the procedure would have to repeated infinitely often to construct the full congruence closure.

We have not provided an algorithm for e-matching. In theory, deciding whether $\mathcal{S} = \emptyset$ for a fixed e-graph and pattern is NP-hard [MŁK08]. In practice, this tends not to be a problem as the number of variables in a pattern is usually small. Thus, despite the discouraging runtime complexity, efficient e-matching algorithms have been developed in [DMB07] and more recently combined with techniques from relational databases queries in [ZWWT21]. Aside from use in SMT solvers, congruence closure with e-matching has also enabled a technique called *equality saturation* which lies at the heart of egg.

2.4 Equality Saturation

Equality saturation is a technique which uses congruence closure with e-matching for efficiently rewriting terms. The idea is developed in [TSTL09] for the domain of program optimization with the goal of addressing the *phase ordering problem*: Given a program and a set of optimizations, in which order should the optimizations be applied such that the resulting program is optimal with respect to a given metric? The fundamental problem is that applying an optimization is destructive (it replaces a program with a new one) which may change how effective subsequent optimizations are, or whether they are even applicable. To sidestep this problem, the approach of equality saturation is to simply try all orders of optimizations. Naively, this creates factorially many versions of the program. The solution to this combinatorial explosion is the use of e-graphs and congruence closure to maximize sharing of equivalent subprograms.

In order to apply e-graphs to the problem, two prerequisites need to be fulfilled. First, programs need to be expressed as elements of some term language. Second, optimizations need to be expressed as (quantified) equivalence declarations. The former is achieved in [TSTL09] by introducing an intermediate representation (PEGs), while the latter requirement is partially lifted by generalizing equivalence declarations as follows. Instead of consisting of a simple quantified equality, they are taken to be a pair consisting of a trigger pattern and a callback function which receives the substitutions after e-matching. The callback can then modify the e-graph as needed, which allows for "normal" congruence closure with e-matching, but also custom rewriting logic. We start calling these generalized equivalence declarations *rewrites* or *rewrite rules* from here on.[4]

With these two requirements fulfilled, equality saturation is a rather simple procedure. Given an e-graph e containing initial terms (like a program to be optimized), and a set of rewrites $\mathcal{R}$ (like program optimizations):

[4]These generalized equivalence declarations are called *equality analyses* in [TSTL09] – not to be confused with *e-class analyses* in egg.

1. Apply each rewrite in $\mathcal{R}$ to e.
2. If e is unchanged after Step 1 or if a given exit condition is met, complete. Otherwise, go to Step 1.

When this procedure completes because e remained unchanged after applying rewrites, we say that we have *saturated*. That is, we have reached a fixed point under (the closure operator of) application of rewrites. In this case, we know that e is the full congruence closure with respect to $\mathcal{R}$. As discussed in the previous section, computing the full congruence closure with e-matching may require infinitely many iterations, so equality saturation does not always saturate. Thus, a suitably chosen exit condition, like a bounded number of iterations or a timeout, are generally required to ensure termination. Once equality saturation terminates, the e-graph represents all rewritten (and non-rewritten) versions of terms with a congruence relation over those terms. This result can be used in various ways. In the case of program optimization, the next step would be to extract a concrete program from the e-graph. In particular, a program which is known to be equivalent to the initial program and which minimizes some cost function. Another use case is to approximate the solution to the *ground word problem* in the domain of equational reasoning and term rewriting [BN98], which is the primary use case in this thesis. The ground word problem for a set of (quantified) equations $\mathcal{E}$ and ground terms t_1 and t_2 is the problem of deciding whether $t_1 = t_2$ is a *semantic consequence* of $\mathcal{E}$. Intuitively, this is the case if all assignments of variables which satisfy all equations in $\mathcal{E}$ also satisfy $t_1 = t_2$.[5] The ground word problem is undecidable in general, but is decided by congruence closure when the equations in $\mathcal{E}$ do not contain variables. This coincides with our observation that introducing e-matching to handle quantified equations does not necessarily construct the *full* congruence closure anymore. Solving, or rather approximating, the ground word problem is useful for theorem proving as it automates equational reasoning. In the case of interactive theorem proving, this generally comes with the additional requirement of wanting to know *why* the given equation holds – that is, which steps were taken to prove the equation. To satisfy this, congruence closure needs to be extended with a procedure for constructing proof witnesses.

2.5 Explanations

In the context of congruence closure, proof witnesses are usually called *explanations*. This follows [NO05], which introduced the first procedure for constructing explanations. In its most basic form, the procedure is stated for equivalence relations with atomic terms and solves the following problem. Given terms a and b and a union-find constructed from equivalence declarations $\mathcal{U} = \{(t_1, t_2), ..., (t_{n-1}, t_n)\}$, find the minimal subset $\mathcal{E} \subseteq \mathcal{U}$ such that a is equivalent to b in the equivalence closure of $\mathcal{E}$. Note that $\mathcal{E}$ has no order over its elements, though in practice we do require an order and can easily construct it from $\mathcal{E}$. Two solutions to the explanation problem are given in [NO05]. The first procedure is based on labeling the edges of the union-find with equivalence declarations. This takes $\mathcal{O}(k \cdot \log(k))$ time to produce an explanation with k declarations while maintaining the optimal time bounds for *union* and *find*. In the following, we focus on the second procedure which aims to achieve an optimal $\mathcal{O}(k)$ time bound to produce an explanation with k declarations. For this purpose, we maintain a *proof forest* structure. This is a graph where edges connect exactly those terms which are related by an equivalence declaration. Notably, this graph will always be a collection of trees where equivalent elements are contained in the same tree.[6] Based on such a graph, we can construct an explanation between terms a and b by collecting the equivalence declarations on the path connecting a and b. The explanation problem is thus reduced to efficiently finding a path between given terms a and b. We achieve this by rooting each tree in the forest at an arbitrary node and directing each edge towards the root. The desired path can then be found by traversing to the root from both a and b. Unfortunately, this root directedness needs to be explicitly maintained in *union*s, as shown by the example in Figure 3. When an equivalence declaration

[5] This notion is defined more precisely as Definition 3.5.3 in [BN98].
[6] This property only holds if *union* ignores redundant equivalence declarations.

(t_1, t_2) introduces an edge between previously unconnected proof trees and we choose t_1's root to be the new root of the combined tree, then t_2's tree needs to be rerooted at t_2, thus flipping the direction of all edges connecting t_2 to its root. This slows down *union*s to be amortized $\mathcal{O}(\log(n))$ in a union-find constructed from n *union*s.

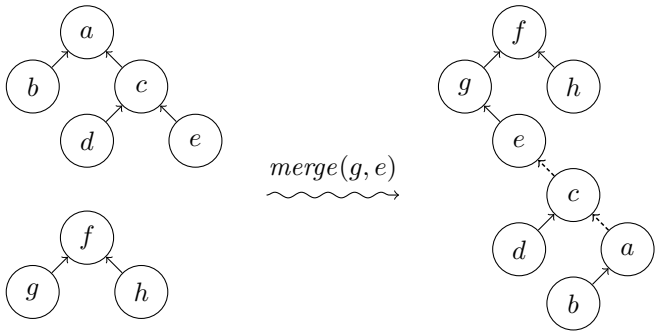


Figure 3: Merging of proof trees which involves flipping (the dashed) edges.

The given procedure works in the context of equivalence closure, but needs to be adjusted for congruence closure. This follows immediately from the observation that terms may become equivalent by means of congruence and thus can only be explained to be equivalent by equivalence of their children, instead of an equivalence declaration. For example, consider the e-graph from Figure 2. Here, merging the e-classes of n_2 and n_4 entails merging of n_1 and n_3, whose equivalence must therefore be explained by the equivalence of their children. While a solution to this is already described in [NO05], we consider the technique from [FCW+22], which is closest to the one used by egg. It replaces the notion of a proof forest with that of a *c-graph*. A c-graph (V, E, j) is an undirected labeled graph where terms in V are connected by edges in E only if they have been shown to be equivalent either by an equivalence declaration or by congruence. The labeling function j assigns to each edge a *justification* where $j((t_1, t_2))$ is either:

1. The equivalence declaration (t_1, t_2).
2. A pair $(t_1[i], t_2[i])$ for some child term index i.

Thus, if terms t_1 and t_2 are congruent, they may be connected by multiple edges, one for each pair of equivalent children. We call the edges used to show congruence of terms *congruence edges*. As with proof forests, constructing an explanation from a c-graph amounts to finding a path between terms t_1 and t_2. However, now each congruence edge encountered on the path yields an explanation problem of its own. Namely, the equivalence of the associated child nodes must be explained. Thus, explanations become DAGs where congruence steps have children which are themselves explanations. This representation of explanations can still be flattened into a linear sequence of proof steps, though.

Finding explanations which are minimal with respect to the DAG size (the number of non-congruence edges) is NP-complete. As an alternative, [FCW+22] proposes two algorithms which optimize for *tree size*. The tree size of a path connecting terms t_1 and t_2 is the sum of the number of non-congruence edges and the tree sizes of the paths explaining the congruence edges. Optimizing this metric is useful as it relates to the number of proof steps obtained from flattening an explanation. A tree size optimal explanation can be constructed in $\mathcal{O}(n^3)$ time for a c-graph with n nodes.[7] A non-optimal explanation can be constructed using a greedy algorithm in $\mathcal{O}(d \cdot \log(d))$ time, where d is the number of equivalence declarations. This greedy algorithm is currently the only option for explanation size optimization in egg.

[7] Technically it can be $\mathcal{O}(n^5)$ when the number of congruence edges is $\mathcal{O}(n^2)$, whereas it is $\mathcal{O}(n)$ in practice.

2.6 The egg Library

At a high level, egg [WNW+21] is a library for performing equality saturation with support for explanations. However, as it strives to be efficient and applicable in a wide range of domains, it introduces significant modifications to the structures shown above. These warrant closer consideration.

E-Graphs Egg's approach to e-graphs differs from the conventional definitions. We can motivate this approach by considering how we treat terms in an e-graph. In our definition above, each term corresponds to an e-node in the e-graph's term graph. E-classes are then constructed over e-nodes. This makes it easy and efficient to check whether two terms are equivalent, but makes it difficult to consider the entire set of terms represented by a given e-class. To enumerate the set of terms, it does not suffice to consider all e-nodes in a given e-class and then find the terms of those e-nodes in the term graph. As an example of how this fails, consider Figure 4.

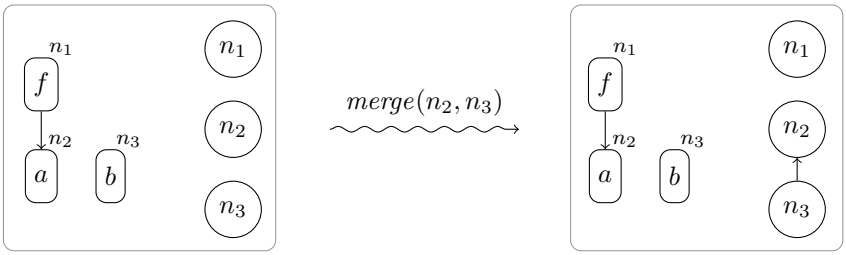


Figure 4: The *merge* operation does not affect term graphs.

Here we start with a term graph containing $f(a)$ and b. We then merge a and b which merges their e-classes, but notably does not change the term graph. As a result, if we consider which terms are represented by f's e-class, we will only discover $f(a)$. A solution to this problem is to consider the children of an e-node as representing their respective e-*class* and not just a concrete e-node. Thus, when enumerating all terms starting at f's e-node, we consider all e-nodes in a's e-class as children and thus find both $f(a)$ and $f(b)$. This view of e-nodes, as representing terms up to equivalence, is fully embraced in egg and changes how we define e-graphs, e-classes and e-nodes. Namely, in egg, an e-node is a constant symbol or a function symbol applied to opaque *e-class identifiers*. An e-class is then a set of e-nodes, which, notably, is not maintained by a union-find anymore, but instead by separate structures in the e-graph. An e-graph is a structure (U, M, H) where:

- U is a union-find over e-class ids.
- M is a map from e-class ids to e-classes with the invariant that for equivalent ids c_1 and c_2 we have $M(c_1) = M(c_2)$.
- H is a map from e-nodes to e-class ids.

That is, the union-find maintains an equivalence over e-class *identifiers*, M holds the actual e-classes (the sets of e-nodes), and H provides an efficient inverse to M. As this construction removes the notion of a term graph, we visualize e-graphs as a single structure as shown in Figure 5.

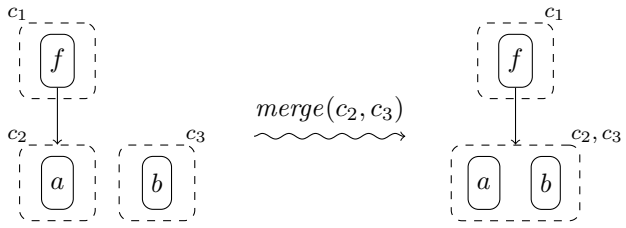


Figure 5: The e-graphs of Figure 4 in egg's e-graph representation.

Dashed lines depict e-classes, while solid boxes depict e-nodes. Note that e-nodes' children (depicted by arrows) are e-classes instead of e-nodes, and thus e-nodes *represent* multiple concrete terms. Namely, an e-node $f(c_1, ..., c_n)$ represents a term $f(t_1, ..., t_n)$, if the e-class identified by c_i represents term t_i for each $1 \leq i \leq n$. And an e-class represents a term t, if it contains an e-node representing t.

E-Matching An immediate consequence of egg's approach to e-graphs is that the substitutions returned by e-matching map variables to e-class ids instead of ground terms. Thus, we can no longer obtain a single ground term by applying a substitution to a pattern term. As a result, e-matching on a pattern term p in egg yields not only a substitution σ but also the e-class id of the e-class which represents the terms obtained by applying σ to p. For an example, consider Figure 6a.

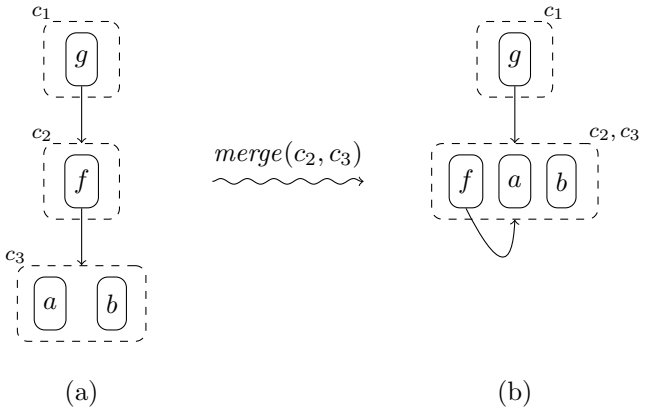


Figure 6: Examples for demonstrating e-matching in egg's e-graphs.

Here, e-matching on pattern term $g(f(x))$ with variables $\{x\}$ yields the substitution $x \mapsto c_3$ and e-class id c_1. As such, we cannot distinguish whether e-matching matched the ground term $g(f(a))$ or $g(f(b))$, as terms are only considered up to equivalence.

Considering terms only up to equivalence also means that applying rewrites can yield loops in the e-graph. For example, in Figure 6b we show the result of applying the rewrite $f(x) = x$ with variables $\{x\}$ to the e-graph from Figure 6a. E-matching on $f(x)$ yields the substitution $x \mapsto c_3$ and the e-class id c_2. Continuing with the regular procedure for congruence closure with e-matching, we merge the e-classes c_2 and c_3 and thus obtain the e-graph shown in Figure 6b. The self-loop of c_2 implies that the e-class represents the infinite set of terms $\{a, b, f(a), f(b), f(f(a)), f(f(b)), ...\}$. Hence, egg can sometimes construct the full congruence closure in a single step, when it would have required an infinite number of steps when using an explicit term graph. A downside of this is that traversing an e-graph requires additional care as it may not be acyclic.

E-Class Analysis In the context of equality saturation, we have established that a rewrite is a pair of a trigger pattern and a callback which can perform operations on the e-graph using the substitutions from e-matching on the pattern term. When substitutions yield ground terms, we can obtain concrete syntactic information from them. For example, we can determine that the term $1+2$ is eligible for constant folding and use this in a rewrite which adds the term 3 to the same e-class. As egg's e-matching yields only e-class ids, we don't have concrete syntactic information. Instead, egg introduces the mechanism of *e-class analysis* for attaching semantic information to e-classes. An e-class analysis operates over a domain $\mathcal{D}$ and associates a value $d_c \in \mathcal{D}$ with each e-class c. For example, we can implement constant folding as above using an e-class analysis where $\mathcal{D} = \mathbb{N} \cup \{\bot\}$ which associates with each e-class c the value $n \in \mathbb{N}$ if c represents a constant value n, and $\bot$ if it does not represent a constant value. Thus, if we e-match on the pattern $x + y$ with variables $\{x, y\}$ and obtain the e-class id c_+ and a substitution $\{x \mapsto c_x, y \mapsto c_y\}$, then we can check whether both $d_{c_x}, d_{c_y} \in \mathbb{N}$ and, if so, add the constant $d_{c_x} + d_{c_y}$ to the e-class identified by c_+.

Defining a concrete e-class analysis requires two operations. The *make* operation constructs the analysis value for a given e-class when it is initially constructed from a new e-node. The *join* operation is used to determine the new analysis value when two e-classes are merged. Notably, $(\mathcal{D}, \mathit{join})$ needs to form a semilattice by *join* being associative, commutative and idempotent. This is necessary to uphold the *analysis invariant*:

$$\forall\, c,\ d_c = \mathit{join}\,\{\mathit{make}(n) \mid n \in M(c)\}$$

It ensures that the analysis value of a given e-class is only a function of its e-nodes and does not depend on the order in which it was computed or by which *merge*s it has been derived. From a more practical standpoint, ensuring these properties of *join* can help uncover cases where the value of an e-class analysis is not actually invariant for all e-nodes in an e-class. For example, a *join* operation for constant values, as in the example above, needs to decide how to reconcile joining analysis values $d_1, d_2 \in \mathbb{N}$ where $d_1 \neq d_2$. Intuitively, such a join should never occur as it means that we are merging e-classes containing e-nodes with non-equal values. Yet, in practice, unsound rewrites could lead to such a scenario, so *join* could decide to either crash or apply an arbitrary function satisfying the semilattice laws like *min* or *max*.

E-class analyses are useful for two categories of rewrites which egg calls *conditional* and *dynamic* rewrites. A conditional rewrite only takes effect if a given condition is met. This is used in the example above, by performing constant folding only if $d_{c_x}, d_{c_y} \in \mathbb{N}$. A dynamic rewrite constructs its result based on the results of e-matching. In the example above, the term $d_{c_x} + d_{c_y}$ could only be constructed after c_x and c_y had become known. Conditional and dynamic rewrites are instrumental in performing rewrites which are not expressible as theorems of the form $\forall\, x_1, ..., x_n : \mathit{lhs} = \mathit{rhs}$.

3 Theorem Proving with Lean

In this section we introduce the Lean theorem prover and its underlying theory. This overview is intentionally incomplete, as we focus on those aspects which are relevant for this thesis.

3.1 Introduction to Lean

Lean is an interactive theorem prover and functional programming language. As such, it allows for creation and compilation of computer programs, while also enabling formalization of mathematical statements and proofs about these programs or any other mathematical objects. The seamless connection between these two aspects of Lean is achieved by its very expressive type system. Most prominently, this type system includes *dependent functions*. These are functions where the output *type* can depend on an input *term*. As an example, consider the following definition of a function which takes a term `a` of some type α, and returns a list containing the single element `a`:

```
def List.singleton : (α : Type) → (a : α) → List α := fun α a => [a]
```

We define the type of this function in curried form, as this is idiomatic and convenient in Lean. As a syntactic convenience, it is common to use the equivalent form:

```
def List.singleton (α : Type) (a : α) : List α := [a]
```

Notably, this function takes the type α as a regular function argument. All subsequent arguments, as well as the return type, can then refer to α. Thus, both a and the return type List α *depend* on α. Dependent functions are especially prevalent in conjunction with *inductive types*, which are Lean's main mechanism for declaring new types:

```
inductive Vector (α : Type) : Nat → Type where
  | nil  : Vector α 0
  | cons : α → (n : Nat) → Vector α n → Vector α (n + 1)
```

This inductive declaration states that for each type α and natural number n, Vector α n is a type (of lists of length n). Its first constructor Vector.nil constructs a list of length 0. The second constructor Vector.cons takes a term of type α and a vector of a given length n and produces a vector of length n + 1. Thus, the type of vectors is parameterized[8] by terms which are not types (natural numbers in this example), which exceeds regular parametric polymorphism [Pie02]. Another common form of polymorphism in Lean are *type classes* [WB89, SO08]. Type classes are similar to what would be called interfaces in Java or traits in Rust:

```
class Add (α : Type) where
  add : α → α → α

instance : Add Nat where
  add := Nat.add

instance : Add String where
  add := String.append

#eval Add.add 40 2       -- 42
#eval Add.add "Le" "an" -- "Lean"
```

In this example, we declare a type class Add which requires conforming types α to provide an add function. We then define *instances* of this type class for Nat and String. Lean stores these instances in a lookup table. When we evaluate Add.add on the values 40 and 2, Lean's *type class synthesis* algorithm constructs an instance of type Add Nat (in this case by simply finding it in the lookup table). The values 40 and 2 are then passed to the add function of the synthesized instance. Thus, the semantics of Add.add depend on the specific instance synthesized for the given type. This is reflected in the type of Add.add, where the brackets around inst indicate that it is a type class argument which will be inferred by type class synthesis:

```
Add.add (α : Type) [inst : Add α] (a₁ a₂ : α) : α
```

Type classes are frequently used in conjunction with notation declarations, such that any conforming type inherits the notation. For example, Lean defines + notation for Add.add:

```
infix " + " => Add.add

#eval 40 + 2       -- 42
#eval "Le" + "an" -- "Lean"
```

Converting between notations based on type classes and their underlying functions will require special care in our proof automation, and is covered in Section 4.5.

[8] We would say *indexed* in this concrete example.

Propositions and Proofs Lean's type system is so expressive that it allows direct encoding of mathematical objects including propositions and proofs:

```
def Nat.IsEven (n : Nat) : Prop :=
  ∃ m : Nat, n = 2 * m

theorem mul_even : ∀ (n k : Nat), IsEven n → IsEven (n * k) := by
  intro n k ⟨m, h⟩
  exists m * k
  rw [h, Nat.mul_assoc]
```

Here, we first define an "is even" predicate for natural numbers. Notably, the definition has no computational interpretation as its return type is a proposition (Prop), not a boolean. We then state and prove the theorem mul_even claiming that multiples of even numbers are also even. The proof uses a sequence of *proof tactics* (intro, exists and rw) to discharge the proof goal. Proof tactics are programs which run at compile time which we therefore call *meta-programs*. Their job is to construct proof terms which Lean can check for correctness. They are also the primary entry point for creating proof automation, as we will do in this thesis.

Compilation Steps In order to implement our own tactic-based proof automation, we need to take a closer look at how Lean works under the hood. Lean is self-hosted and extensible by design [Ull23], and thus its internals are rather accessible. Figure 7 shows how a Lean file is compiled.

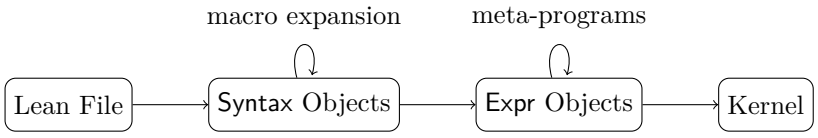


Figure 7: Lean's compilation pipeline.

Compilation starts by parsing a Lean file into terms of the Syntax type, representing a parse tree. After a subsequent macro expansion step, the syntax objects are lowered (*elaborated*) into terms of Lean's expression type Expr. This type is central to meta-programming in Lean. Most notably, Expr is compatible with Lean's *kernel* which performs type checking. Within this compilation pipeline, proof tactics can be implemented in multiple ways. One is by transforming Syntax objects to other Syntax objects via macros. This is common for light-weight tactics, which compose other tactics. More commonly, tactics are implemented as meta-programs, which transform Expr objects to other Expr objects. These meta-programs can build on a variety of monads provided by Lean which give access to important functionality like type inference, type class synthesis, and unification.

3.2 Type Theory

The common currency in Lean meta-programs are Expr objects. These objects are terms of an expression language which derives from the *calculus of constructions* (CoC) [CH88] – a higher-order typed λ calculus. As some aspects of Lean's type theory differ significantly from CoC, we base the following exposition on the canonical reference for Lean's type theory [Car19]. Its terms are given by the following grammar:

$$e ::= x \mid e\,e \mid \lambda x : e, e \mid \forall x : e, e \mid \mathcal{U}_\ell \qquad \text{where } x \text{ are expression variables}$$
$$\ell ::= u \mid 0 \mid S\,\ell \mid max(\ell, \ell) \mid imax(\ell, \ell) \qquad \text{where } u \text{ are universe variables}$$

As expressions are terms of the expression language, we use the terms "term" and "expression" interchangeably throughout this thesis. An expression e can be a variable (x), a function application ($e\,e$), a term- or type-level abstraction (λ and $\forall$), or a *type universe* ($\mathcal{U}_\ell$) at given

universe level (ℓ). We call λ and $\forall$ *binders* and note that the type of the bound variable x can itself be an expression.[9] The type-level abstraction $\forall x : e, e$ represents the type of dependent functions. For nested dependent function types we write $\forall (x_1 \dots x_m : \alpha) \dots (z_1 \dots z_n : \omega), e$ instead of $\forall x_1 : \alpha, ..., \forall x_m : \alpha, ..., \forall z_1 : \omega, ..., \forall z_n : \omega, e$. For example, the declaration List.singleton from above has type $\forall (\alpha : \mathcal{U}_1)(a : \alpha), \mathsf{List}\,\alpha$. The typing judgement $\alpha : \mathcal{U}_1$ states that α is itself a type. That is, type universes like $\mathcal{U}_1$ are types whose members are themselves types. Thus, we can have sequences of typing judgements like $42 : \mathbb{N}$ and $\mathbb{N} : \mathcal{U}_1$. Subsequently, we have the typing judgements $\mathcal{U}_1 : \mathcal{U}_2$, $\mathcal{U}_2 : \mathcal{U}_3$, etc. That is, there exists an infinite hierarchy of type universes $\mathcal{U}_\ell$, where the set of universe levels ℓ contains the standard encoding of natural numbers in Peano arithmetic by 0 and $S\,\ell$.[10] These universes are *non-cumulative*, meaning $e : \mathcal{U}_m$ and $m < n$ does not imply $e : \mathcal{U}_n$.[11] Instead, there exist *universe variables* u which allow definitions to be polymorphic over universe levels. An example of a universe polymorphic definition is Lean's List type, where the type (universe level) of $\mathsf{List}\,\alpha$ depends on the universe of α. The level constructions *max* and *imax* are necessary for cases like the product type $\times$. Given $\alpha : \mathcal{U}_u$ and $\beta : \mathcal{U}_v$, we have $\alpha \times \beta : \mathcal{U}_{max(u,v)}$. We do not consider universe levels in depth in this thesis.

Regarding notation, we use the following conventions from here on out. Non-dependent functions are sometimes written as $e_1 \to e_2$ to mean $\forall x : e_1, e_2$, where e_2 does not refer to x. We use the wilcard character _ to omit terms when they are clear from context or otherwise unimportant. And when the type of a bound variable is irrelevant, we simply drop it completely and write $\lambda x, e$ instead of $\lambda x : \_, e$.

Curry-Howard Correspondence In the infinite hierarchy of type universes, $\mathcal{U}_0$ behaves differently than all its successors. $\mathcal{U}_0$ is the universe of propositions, whereas all subsequent universes contain types of "data". To make sense of this, we need to consider how propositions and proofs are represented according to the *Curry-Howard correspondence*. One of its key ideas is that propositions are types and proofs are terms. That is, a proposition P is represented as a type, and a proof p of P is a term of type P (also called an *inhabitant* of P). In terms of typing judgements, this means that we have $p : P$ and $P : \mathcal{U}_0$. Proving a proposition P thus amounts to finding an inhabitant of the type P, and checking the correctness of a proof p reduces to checking the typing judgement $p : P$. In Lean, type checking is performed by the kernel, and the correctness of all theorems proven in Lean therefore only depends on the correctness of the kernel. When proof tactics are used to prove a theorem, they construct proof terms which are afterwards checked by the kernel. Thus, proof tactics cannot compromise soundness. A notable caveat to this design is that the kernel can never check whether the proposition being proven is the one intended by the user. That is, a malicious macro or elaborator can still trick users into believing that a false statement was proven.

We have already seen two examples of propositions, without explicitly mentioning it: the $\forall$ and $\to$ types. From a computational perspective, they are the dependent and non-dependent function types. Yet, under the Curry-Howard correspondence they have a logical interpretation as universal quantification and implication. A proof of an implication $P \to Q$ is thus a function which takes a proof of P and returns a proof of Q. This corresponds to the traditional mathematical approach of "assuming" P to be true, and then proving Q. Other propositions like $P \vee Q$ and $P \wedge Q$ can be defined as the sum and product types of P and Q respectively. These can both be defined as inductive types, where Prop denotes the type universe $\mathcal{U}_0$:

```
inductive Or (P : Prop) (Q : Prop) : Prop where
  | left  : P → Or P Q
  | right : Q → Or P Q

inductive And (P : Prop) (Q : Prop) : Prop where
  | intro : P → Q → And P Q
```

[9] Lean's typing judgement ensures that this expression results in a type. So an expression like $\lambda x : 1 + 1, x$ can never be well-typed.

[10] This hierarchy is needed, as $\mathcal{T} : \mathcal{T}$ for any type $\mathcal{T}$ is inconsistent by Girard's Paradox [Coq86].

[11] Coq is an example of a theorem prover that uses cumulative universes [Car19].

The structural correspondence between data types like $\times$ and propositions like And is another facet of the Curry-Howard correspondence. Such correspondences even extend to other logics and type theories like modal logic and modal type theory.

Inductive Types In CoC there are exactly two mechanisms for constructing types: $\forall$-types and type universes.[12] Inductive types are not part of CoC, but rather of the extended *calculus of inductive constructions* [CP88]. As inductive types do not have explicit representations in Lean's expression language, we do not consider them in detail here. Instead, we give only an intuition by considering how the type of natural numbers is represented:

```
inductive Nat : Type where
  | zero : Nat
  | succ : Nat → Nat
```

The inductive declaration of Nat introduces new axiomatic constant symbols for the type, its constructors and its *recursor*, with the following typing judgements:

$$\begin{aligned}
\mathsf{Nat} &: \mathcal{U}_1 \\
\mathsf{Nat.zero} &: \mathsf{Nat} \\
\mathsf{Nat.succ} &: \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{Nat.rec}_u &: \forall\, C : \mathsf{Nat} \to \mathcal{U}_u, (C\ \mathsf{Nat.zero}) \to (\forall\, n : \mathsf{Nat}, C\ n \to C\ (\mathsf{Nat.succ}\ n)) \to \forall\, t : \mathsf{Nat}, C\ t
\end{aligned}$$

By their axiomatic nature, these symbols are *uninterpreted*. That is, they have no definitions to unfold to. While constructors define *introduction* rules for the given type, recursors are *elimination* rules. That is, they state how a term of a given inductive type can be destructured into its components. The argument C is usually called the *motive* and determines what the output type should be when calling the recursor. Subsequently, an argument matching the motive type must be provided for each constructor of the inductive type. The semantics of the recursor are then given by *ι-reduction*. Every inductive type has a ι-reduction rule which provides the semantics of recursion (or induction) over a given inductive type. For Nat, ι-reduction is defined by:

$$\begin{aligned}
\mathsf{Nat.rec}\ C\ a\ f\ \mathsf{Nat.zero} &\equiv a \\
\mathsf{Nat.rec}\ C\ a\ f\ (\mathsf{Nat.succ}\ n) &\equiv f\ n\ (\mathsf{Nat.rec}\ C\ a\ f\ n)
\end{aligned}$$

The symbol $\equiv$ denotes *definitional equality*, which defines which expressions should be considered equal without further proof. Thus, the definitional equality rules above ensure that an expression like Nat.rec _ 42 _ Nat.zero is equal to 42 without further proof.

Definitional Equality The definitional equality relation $\equiv$ is explicitly defined only in Lean's underlying theory. Yet, it is used in Lean whenever two terms are checked for equality without proof. Such a notion is sometimes called *judgmental equality*. Definitional equality is important for defining the (expected) semantics of expressions. For example, for the following function f to be well-typed syntactic equality does not suffice:

```
def N := Nat
def f (n : N) : Nat := n
```

For f to be well-typed, the type of n must be "equal to" Nat. Under syntactic equality, this is not the case as the type of n is N which is syntactically distinct from Nat. Yet, under definitional equality we have $\mathsf{N} \equiv \mathsf{Nat}$. This holds by the definitional equality rule called *δ-reduction*, which states that every definition is equal to its body. Other definitional equality rules include β-reduction and η-reduction. We cover all (relevant) definitional equality rules in Section 4.4. For a complete list, see [Car19]. Notably, Lean does not implement full definitional equality,

[12] Technically, CoC's universe of all types $*$ is not itself a type, but in Lean $\mathcal{U}_\ell$ is.

as it is undecidable in general. Instead, it implements *algorithmic* definitional equality, which, for example, is not transitive. Such differences between theory and implementation appear in multiple contexts where the practical aspects of Lean are at odds with its theory. One case where this requires further consideration is Lean's expression language.

3.3 Expressions

Lean's expression language lies at the heart of its underlying theory. At the same time, it is also ubiquitous as Expr objects in meta-programs. Thus, the Expr type contains many constructs which closely correspond to those found in the type theory. For example type universes (also called *sorts*), function application, and term- and type-level abstraction:

```
inductive Expr where
  | sort    : Level → Expr
  | app     : Expr → Expr → Expr
  | lam     : Name → Expr → Expr → Expr
  | forallE : Name → Expr → Expr → Expr
  ...
```

Yet, as Expr needs to be both ergonomic and performant it introduces differences to the theoretical expression language. We consider these differences in the following sections.

Variables In Lean's type theory, we take expression variables to be names from some ambient set of variable names. In Expr, we use a *locally nameless* representation [Cha12]. That is, bound variables are represented with De Bruijn indices, while free variables are represented with names:

```
  ...
  | bvar : Nat → Expr
  | fvar : FVarId → Expr -- `FVarId` is a wrapper around `String`.
```

De Bruijn, or *nameless*, notation of variables identifies them by the distance to their parent binder. For example, $\lambda x : \alpha, x$ is represented by $\lambda \alpha, \hat{0}$ and $\lambda x : \alpha, \lambda y : \beta, x\, y$ by $\lambda \alpha, \lambda \beta, \hat{1}\, \hat{0}$, where $\hat{n}$ denotes the variable with De Bruijn index n. In subsequent sections, we use both named and nameless notation while sometimes omitting the types of bound variables, depending on whether it is contextually relevant. When a variable does not have an associated parent binder, as in $\lambda \hat{1}$, we call it *loose bound*. In *locally nameless* representation, we avoid loose bound variables by replacing them with named free variables. A named expression like $\lambda x, \lambda y, (a\, x)(b\, y)$ is thus represented as $\lambda\lambda(a\, \hat{1})(b\, \hat{0})$, where a and b are free variables. Note that we cannot type check expressions containing free variables without additional context. This context is provided in Lean by programming in a monad which tracks free variables and their types while also ensuring unique naming.

Metavariables *Metavariables* are a special kind of variable which exist solely for the purpose of meta-programming. They represent "holes" in levels and expressions:

```
  ...
  | mvar : MVarId → Expr -- `MVarId` is a wrapper around `String`.
```

Metavariables are useful for constructing expressions incrementally, as is done during elaboration and in (proof) tactic mode. Given a metavariable $?m$ and an expression e, e can be *assigned* to $?m$, which records the assignment in a monadic context. Any expression containing $?m$ can then *instantiate* its metavariables, which replaces $?m$ with e. For example, let a be the expression $\lambda x, ?m\, x$. If we assign $\lambda y, y$ to $?m$ and instantiate all metavariables in a, then we get $\lambda x, (\lambda y, y)\, x$.

Extensions In [Car19], the expression language of Lean's type theory is extended by *named constants* and let-*expressions*. These constructs are also present in Expr:

```
...
| const : Name → List Level → Expr
| letE  : Name → Expr → Expr → Expr → Expr
```

Named constants, also called *definitions*, are tracked by a global context. That is, an expression like Expr.const Nat.add [] represents the function Nat.add, but requires a global context to resolve its type and body. The list of levels holds the universe variables for universe polymorphic definitions. The constructor for let-expressions like let $x := 5$; $x + x$ allows factoring out sub-expressions. While this can also be achieved by λ-expressions in some cases, let-expressions cannot always be replaced by λ-expressions. They can always be eliminated though, by substituting the variable into the body.

Internalizations As the Expr type needs to be performant, Lean introduces two *internalizations* to it. These are constructs which could be represented by other constructors, but are explicitly represented for efficiency:

```
...
| lit  : Literal → Expr
| proj : Name → Nat → Expr → Expr
```

A *literal* is either a natural number or a string, which would otherwise need to be represented (inefficiently) as a sequence of applications of Nat.succ and List.cons respectively. Internalized literals allow the Lean kernel to work with machine-native representations of natural numbers and strings, making common operations much more efficient. The internalization for *structure projections* solves a problem with the size of automatically generated functions for structure types. Structure types are syntactic sugar for inductive types with a single constructor. That is, the following definitions are the same:

```
structure Point where
  x : Nat
  y : Nat

inductive Point where
  | mk : (x : Nat) → (y : Nat) → Point
```

For structure types, Lean automatically generates *projections*, which map from the structure type to its fields:

```
#check Point.x -- Point → Nat
#check Point.y -- Point → Nat
```

Using a naive encoding, each projection has a body of size $\mathcal{O}(n)$ where n is the number of fields in the structure. With internalized structure projections, this is reduced to $\mathcal{O}(1)$ by adding projections as a primitive notion in the expression language. In Expr, this is reflected by representing *applications* of projections like Point.x p as Expr.proj Point x p. More details on internalization are given in [Ull23].

3.4 Equational Reasoning

Equational reasoning is a common mode of reasoning in mathematical proofs. Given the goal of proving an equation $a = z$, one proves a sequence of equations $a = b = ... = y = z$. In this section, we briefly consider different tools and methods for equational reasoning in Lean. Underlying all of these methods are the properties of equality as being an equivalence relation which is a congruence with respect to all functions.[13] In Lean, these properties are a result of the recursor of the inductive equality type Eq, of which we consider the non-dependent version:

[13] We will see that the latter is not completely true for Lean's equality type Eq.

$$\mathsf{Eq.ndrec}: \forall(\alpha : \mathcal{U}_u)(a : \alpha)(C : \alpha \to \mathcal{U}_u), (C\,a) \to \forall(b : \alpha)(h : a = b), C\,b$$

This recursor provides a substitution principle for equality. Namely, given a term of type $C\,a$ and a proof of $a = b$, we obtain a term of type $C\,b$. For example, given $x : \mathbb{N}$ and a proof $h_1 : p\,x$, we can prove $p\,y$ from $h_2 : x = y$ as $\mathsf{Eq.ndrec}\,\mathbb{N}\,x\,p\,h_1\,y\,h_2$. Clearly, using the recursor for actual equational reasoning would be extremely laborious. Thus, as a first simplification, Lean provides a general congruence theorem for equality, proven from Eq.ndrec:

$$\mathsf{congr} : \forall(\alpha : \mathcal{U}_u)(\beta : \mathcal{U}_v)(f_1\,f_2 : \alpha \to \beta)(a_1\,a_2 : \alpha), (f_1 = f_2) \to (a_1 = a_2) \to f_1\,a_1 = f_2\,a_2$$

Based on this, we can prove $p\,x = p\,y$ from $h : x = y$ as $\mathsf{congr}\_\_\_p\,p\,x\,y\,(\mathsf{Eq.refl}\,p)\,h$, where $\mathsf{Eq.refl}\,p$ proves $p = p$. Notably, this congruence theorem only works for non-dependent functions. That is, the output type β of f_1 and f_2 does not depend on the input term of type α. In fact, a dependent version of this theorem only holds when the arguments a_1 and a_2 are definitionally equal:

$$\mathsf{congrFun} : \forall(\alpha : \mathcal{U}_u)(\beta : \alpha \to \mathcal{U}_v)(f_1\,f_2 : \forall x : \alpha,\, \beta\,x)(a : \alpha), (f_1 = f_2) \to f_1\,a = f_2\,a$$

Generalizing this to a_1 and a_2 with $a_1 = a_2$ is not possible. It would not even be clear what the statement of the theorem should be, as then the types of $f_1\,a_1$ and $f_2\,a_2$ are not definitionally equal, and thus cannot be related by Eq. The lack of a general dependent congruence theorem impacts proof reconstruction as explained in Section 4.6.

The rw Tactic Congruence theorems are powerful tools for equational reasoning with proof terms, but are too unwieldy for practical use. Thus, Lean provides the rw proof tactic for rewriting terms using given equations. Notably, these equations can be universally quantified, and the tactic tries to match them against (sub-)terms appearing in the proof goal. For example, let $h_1 : \forall x : \mathbb{N}, x + 0 = x$ and $h_2 : \forall(x\ y : \mathbb{N}), x + y = y + x$. Then the proof goal $2 + (z + 0) = z + 2$ can be solved by $\mathsf{rw}\,[\mathsf{h}_1, \mathsf{h}_2]$. For this, rw first matches h_1 against $z + 0$, creates the specialized proof term $h_1\,z : z + 0 = z$ and uses congruence to turn the proof goal into $2 + z = z + 2$. Then, it matches h_2 against $2 + z$, creates the specialized proof term $h_2\,2\,z : 2 + z = z + 2$ and completes the proof by transitivity of equality.[14] Thus, the tactic performs the steps which we intuitively expect from "applying a rewrite", without having to worry about matching quantified equations to (sub-)expressions or applying congruence theorems. While rw "just works" most of the time, the tactic can easily get stuck if the given equations are provided in the wrong order. For example, the proof goal $(2 + z) + 0 = z + 2$ fails if we switch the order or equations to $\mathsf{rw}\,[\mathsf{h}_2, \mathsf{h}_1]$. In that case, rw first applies h_2 and rewrites the goal to $0 + (2 + z) = z + 2$. Thus, h_1 cannot be applied anymore, and the proof remains incomplete. This problem is akin to the phase ordering problem in that applying a rewrite (destructively) can block the applicability of subsequent rewrites. Another problem arises from the fact that equations are only applied from left to right. That is, given equations $hy : x = y$ and $hz : x = z$, the goal $y = z$ cannot be solved by $\mathsf{rw}\,[\mathsf{hy}, \mathsf{hz}]$, as x does not appear in the proof goal, and thus neither equation matches. This can only be resolved by *explicitly* providing the directions in which equations should be applied, as in $\mathsf{rw}\,[\leftarrow \mathsf{hy}, \mathsf{hz}]$.

The simp Tactic Lean's simplification tactic simp is much more sophisticated than the rw tactic, but is limited by similar problems. It also applies equations only from left to right, but in a different order than rw. Namely, simp tries to match any of the given equations against the proof goal, starting from the most deeply nested expression. It then keeps applying equations in this way until none apply. This does not fix the phase ordering problem though, as trying to prove $2 + z + 0 = z + 2$ from $\mathsf{simp\ only}\,[\mathsf{h}_1, \mathsf{h}_2]$ (with h_1 and h_2 as above) yields the incomplete proof goal $0 + z + 2 = z + 2$ after applying h_1 twice. Using a theorem such

[14] Technically, it uses *modus ponens* (Eq.mp and Eq.mpr) to chain the equalities.

as commutativity (like h_1) with simp is generally a bad idea as it breaks expected semantics. The simp tactic, as its name suggests, expects given equations to *simplify* expressions from left to right. What exactly constitutes "simplification" is up to convention, but generally excludes theorems like commutativity or associativity. A good reason for imposing these semantics, is that simp implements a *term rewriting system* (TRS), and restricting the chosen equations can make the TRS *confluent* [BN98]. A TRS is a set of directed (quantified) equations $l \to r$, where l is not a variable. Given terms x and y, we write $x \to y$ to denote that there exists *some* rewrite in the TRS mapping x to y, and we write $x \xrightarrow{*} y$ do denote that y is reachable from x by applying finitely many rewrites from the given TRS. A TRS $\mathcal{R}$ is called confluent, if $\forall x, y_1, y_2 : (x \xrightarrow{*} y_1) \land (x \xrightarrow{*} y_2) \to \exists z : (y_1 \xrightarrow{*} z) \land (y_2 \xrightarrow{*} z)$. That is, any terms derived from a given term can be further reduced to the same term. This property is useful for proving equalities of terms with *normal forms*. A term y is a normal form of term x, if $x \xrightarrow{*} y$ and $\forall z, y \not\to z$. Depending on the TRS, a term may or may not have a normal form. If it does, and the TRS is confluent, then we can apply rewrites in any order and be assured that we will reach the same normal form. That is, equalities can be proven by mindlessly applying rewrites until a normal form is reached on both sides. While the equations given to simp do not generally form a confluent TRS, they can be made "more confluent" by enforcing certain normal forms of the terms appearing in equations (informally called *simp normal form* for simp). Thus, simp has a much better chance of proving goals than rw. Another significant benefit of simp over rw is its premise selection. The rw tactic rewrites based only on equations passed to it explicitly. The simp tactic, on the other hand, can access a set of *simp lemmas* marked by the user. At the time of writing, Lean's largest library *mathlib* contains more than 30,000 simp lemmas. Determining which of these lemmas to apply by brute force is thus infeasible. Instead, the lemmas are indexed in a *discrimination tree* data structure [McC92], which can efficiently determine which indexed terms may unify with a given term. Thus, at every step, the simp tactic can efficiently select from a large pool of lemmas to apply to the proof goal.

Other Tactics The rw and simp tactics are ubiquitous in Lean proofs. Yet, as the need for better proof automation is ever present, more involved tactics for equational reasoning have emerged. We briefly highlight two of them here.

The cc tactic solves equality goals using congruence closure based on e-graphs. The notion of e-graphs is that of [NO80], instead of [WNW+21]. Notably, cc can handle *heterogeneous equality* – a form of equality, where the types of the related terms do not have to be definitionally equal. Lean has a corresponding type for this called HEq. It permits a congruence theorem on function arguments for dependent functions, as long as the functions are definitionally equal and the arguments satisfy standard *homogeneous* equality. For dependent functions with a bounded number of arguments, [SdM16] shows a more general congruence theorem where functions need only satisfy homogeneous equality and arguments only heterogeneous equality. The cc tactic employs the approach shown in [SdM16], while also exploiting features specific to Lean's type theory, like injectivity of inductive type constructors.

The rewrite_search tactic takes an entirely more practical approach to solving equality goals. It tries to find a list of equations $e_1, ..., e_n$ such that $\mathsf{rw}[\mathsf{e}_1, ..., \mathsf{e}_n]$ solves the proof goal. Its search space consists of all equational theorems proven up to the point of the tactic's invocation. To choose which equation to apply next, rewrite_search performs a best-first search which minimizes the Levenshtein distance [CH69] of the string-representations of the equation and the proof goal.

In the rest of this thesis, we develop yet another tactic suited for equational reasoning, with the hope of improving upon or filling gaps in the current state of proof automation.

4 Equality Saturation Tactic

In this section, we develop a proof tactic for equational reasoning in Lean, based on equality saturation with egg. We start with a high-level overview of the tactic's components and their interactions. Subsequently, we add techniques for expanding the tactic's feature set and improving its soundness. In particular, a variety of methods are required to bridge the gap between Lean's expression semantics and the syntactic nature of e-graphs. When describing these methods, we try to avoid implementation details and give definitions instead of code where possible. A full implementation of the proof tactic is available at `https://github.com/marcusrossel/lean-egg`.

4.1 Overview

Proofs in Lean are seldom written as explicit proof terms. Instead, they are built by calling a sequence of proof tactics, which construct the relevant proof terms. Thus, we implement proof automation for equational reasoning as a tactic $\texttt{egg}\,[\texttt{eqn}_1, \texttt{eqn}_2, ...]$, which solves proof goals of the form $lhs = rhs$ by using egg to find a sequence of rewrites which transform lhs to rhs. Figure 8 shows a simplified outline of the inner workings of this tactic. The setup is both based on and extends the tactic described in [KGB+24].

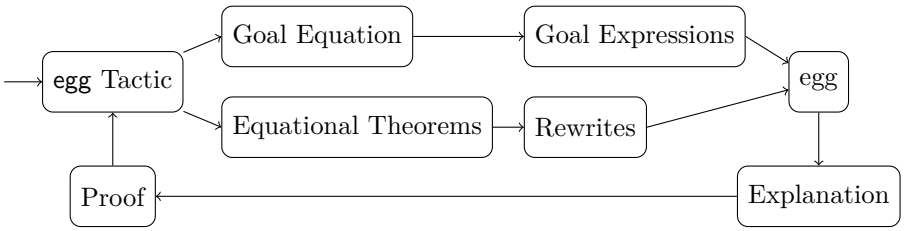


Figure 8: Simplified overview of the egg proof tactic.

When the egg tactic is called, we first obtain the current proof goal and elaborate given equational theorems. We then check that the proof goal is in fact of the form $lhs = rhs$ and that all given equations are (universally quantified) equalities.[15] If these conditions are not met, the egg tactic is not suited for the proof task and aborts. If the conditions *are* met, both the goal and the equations are encoded into an expression language used in egg. Notably, this turns equations into pattern terms used for rewrites. The encoding step produces strings which are passed to the egg Rust library via Lean's and Rust's foreign function interface to C. In Rust, we parse the given strings into objects native to egg, add the goal expressions to an e-graph, and run equality saturation with the given rewrites on that e-graph. When equality saturation completes, we check whether the expressions for lhs and rhs are represented by the same e-class. If they are, egg has proven the required equality, and we generate a corresponding explanation. The explanation is returned to Lean, where we then try to construct a proof term corresponding to the explanation. If this succeeds, we complete the tactic call successfully. If any of these steps fails, we fail the tactic call.

Turning the explanation from egg into a proof in Lean is of particular importance, as it ensures that neither egg, nor our tactic can introduce unsoundness. Any invalid proof term will simply not be accepted by Lean. This constraint creates an interesting trade-off between soundness and completeness in our tactic. As our tactic does not need to focus on maintaining soundness, we can use unsound techniques with the goal of increasing the completeness of our tactic. That is, we can use techniques which work most of the time for many cases instead of all the time for fewer cases. However, we cannot completely ignore soundness either, as unsound techniques can in fact reduce completeness by shadowing correct proofs which may have otherwise been discovered by sound techniques.

[15] We also support the propositional equivalence relation $\leftrightarrow$.

4.2 Representations in egg

To connect Lean and egg, we first need to decide how terms are communicated between these systems. That is, we need a common expression language. Lean already has such a language with its Expr type. In egg, the term language can be chosen freely. We therefore take Expr to be the common language and encode it relatively directly in egg as shown in Figure 9. While this approach roughly matches the one in [KGB+24], they encoded the language generically as S-expressions, whereas we directly match the constructors of our egg expression language to those in Expr.

```
define_language! {
      pub enum LeanExpr {
            // Primitives:
            Nat(u64),
            Str(String),

            // Encoding of expressions:
            "bvar"  = BVar(Id),          // (Nat)
            "fvar"  = FVar(Id),          // (Nat)
            "mvar"  = MVar(Id),          // (Nat)
            "sort"  = Sort(Id),          // (<level>)
            "const" = Const(Box<[Id]>), // (Str, <level>*)
            "app"   = App([Id; 2]),      // (<expr>, <expr>)
            "λ"     = Lam([Id; 2]),      // (<expr>, <expr>)
            "∀"     = Forall([Id; 2]),   // (<expr>, <expr>)
            "lit"   = Lit(Id),           // (Nat | Str)

            // Encoding of universe levels:
            "uvar"  = UVar(Id),          // (Nat)
            "param" = Param(Id),         // (Str)
            "succ"  = Succ(Id),          // (<level>)
            "max"   = Max([Id; 2]),      // (<level>, <level>)
            "imax"  = IMax([Id; 2]),     // (<level>, <level>)
      }
}
```

Figure 9: Encoding of Lean's expression language in egg.

As egg needs to be able to parse terms of this language from strings, most constructors have corresponding string labels. The unlabeled constructors Nat and Str, act as primitive constructors which recognize terms of the underlying type directly. Labeled constructors, like BVar, can only specify how many children they expect, but not of which type. Thus, in a case like Lit, we implicitly expect the child to be a Nat or a Str without being able to encode this explicitly. While this means that ill-formed terms like (lit (bvar 0)) are technically valid terms of the language, this is not a problem in practice as we only ever provide well-formed terms to egg. Another side effect of egg's language mechanism is that we need to specify the language of universe levels as part of the expression language, instead of it being separate. Minding these technicalities, the given language is a relatively direct translations of Lean's Expr type with the following minor adjustments:

- The proj and letE constructors aren't encoded as they are eliminated during preprocessing in Lean. This is discussed in more detail in Section 4.4.
- Free variable and metavariable identifiers are encoded as integers instead of string names. Lean represents these identifiers as wrappers over unique names of the form _uniq.*idx*. Thus, we simply encode the identifiers by their integer index.
- We don't encode Level.zero directly, but use Nat(0) for that instead.

Rewrites Based on the encoding of Expr in egg, equations from Lean are represented as rewrites with a pattern expression for each side of the equality. For example, an equation like $\forall x : \mathbb{N}, x = x + 0$ is represented as the rewrite $?7 \Leftrightarrow \mathsf{app}\ (\mathsf{app}\ (\mathsf{const}\ \mathsf{Nat.add})\ ?7)\ (\mathsf{const}\ \mathsf{Nat.zero})$, where ?7 denotes the pattern variable corresponding to x. The arrow $\Leftrightarrow$ indicates that the rewrite is applicable in both directions. That is, it technically constitutes two rewrites where each side is the trigger pattern for one of the rewrites. The applicable directions of a rewrite are determined in the preprocessing phase in Lean. Namely, when we encode an equation into a rewrite, we first instantiate all quantified variables with fresh metavariables. For example, the above theorem is turned into the Lean expression $?x = ?x + 0$, where $?x$ is a metavariable. These metavariables are encoded as pattern variables in the corresponding rewrite. Thus, to determine the applicable directions of the rewrite, we compute the directions in which the metavariables of the instantiated equation satisfy the $\supseteq$ relation. In the example above, both sides of the rewrite contain only $?x$, so the rewrite is applicable in both directions. In the theorem $\forall x, x \cdot 0 = 0$, we can only apply the forward direction, as the quantified variable only appears on the left. A class of theorems which cannot be applied in any direction are those stating that a given type α is a *subsingleton*, which is a type satisfying the equation $\forall (x\ y : \alpha), x = y$. Here, the variables do not satisfy $\supseteq$ in either direction.

Metavariables Metavariables are special expressions in that they need to be encoded differently depending on the use case of their parent term. When a metavariable appears as part of an equation, it needs to be encoded as a pattern variable. However, when a metavariable appears in the proof goal, it is encoded using the explicit mvar and uvar constructors (for expression and universe level metavariables, respectively). That is, it is treated as an opaque constant. Proof goals contain metavariables when their type is not fully known. A typical source of this is applying transitivity of equality during a proof. This turns a goal like $1 + 1 = 2$ into the goals $1 + 1 = ?m$ and $?m = 2$ with metavariable $?m$. We can't sensibly assign $?m$ in egg, as this would require us to choose what the proof goal should be. Thus, we treat the metavariable as a constant which is left unassigned by our proof procedure.

Unfortunately, the given dichotomy between a metavariable appearing either in an equation or the proof goal is a false one. Equations may contain metavariables which do not correspond to quantified variables. Thus, to decide how to encode a metavariable, we actually check whether it is *ambient*. A metavariable is ambient if it existed (was present in Lean's metavariable context) before the egg tactic was invoked. Metavariables contained in the proof goal are then always ambient, while the fresh metavariables introduced when instantiating equations can never be ambient. A metavariable is then treated as a constant, if and only if it is ambient.

The chosen expression representation in egg is easy to encode from Lean expressions and can easily be turned back into Lean expressions during proof reconstruction. Yet, it introduces challenges when used in e-graphs. The fundamental problem is that Lean expression have associated semantics which is not captured in their syntax. First, there are binders which make it impossible to reason about expressions locally. Second, Lean's notion of definitional equality requires certain syntactically distinct expressions to be interchangeable. And third, Lean's meta-programming contexts track various information, like the types of expressions, and the bodies of definitions, which are not available during equality saturation. Bridging these gaps between syntax and semantics motivates many of the subsequent sections.

We note here that other approaches to encoding could be beneficial. For example, the *lean-auto*[16] project implements a procedure for converting between Lean expressions and a simpler type theory. Using such a procedure to simplify the expressions used in egg may eliminate some of the problems encountered in subsequent sections. Another simpler approach may be to restrict ourselves to a fragment of the expression language as in the encoding of Coq expressions considered in [Bou23].

[16] https://github.com/leanprover-community/lean-auto

4.3 Binders in E-Graphs

Using λ calculus in e-graphs is notoriously difficult due to the presence of binders. Binders make it necessary to consider expressions in context instead of purely syntactically. For example, when using de Bruijn indices, the expression $\hat{1}$ refers to a loose bound variable in $\lambda\hat{1}$, but to a bound variable in $\lambda\lambda\hat{1}$. The problem persists when using named variables where x refers to a free variable in $\lambda y, x$, but to a bound variable in $\lambda x, x$. In this section, we consider three classes of problems resulting from direct representation of bound variables in e-graphs and show how we approach them in the `egg` proof tactic.

4.3.1 Invalid Matching

When applying a rewrite of the form $lhs \Rightarrow rhs$, the first step is e-matching on lhs. This yields a substitution σ which maps every pattern variable in lhs to an e-class. When lhs contains a pattern variable $?x$ whose associated e-class $\sigma(?x)$ represents an expression containing loose bound variables, two problematic scenarios can occur.

1. $?x$ appears under multiple different binders in lhs.
2. $?x$ appears at binder depth d in lhs, but $\sigma(?x)$ represents an expression containing a loose bound variable with index $b < d$.

In both cases we say that an *invalid match* has occurred. An example of Problem 1 occurs when the pattern term $\lambda(\lambda\, ?x)\, ?x$ is matched against $\lambda(\lambda\, \hat{0})\, \hat{0}$. Here, $?x$ is matched against $\hat{0}$, but $\hat{0}$ refers to different variables for the respective occurrences of $?x$. An example of Problem 2 occurs when the rewrite $(\lambda\, ?x)\, 1 \Rightarrow ?x$ is applied to $(\lambda\, \hat{0})\, 1$, which produces the non-equivalent term $\hat{0}$. The reason this fails is that the given rewrite is valid only when $?x$ is not bound. As $?x$ appears under one binder in the trigger pattern, any matched variable with index < 1 is bound in the context of the trigger pattern, and therefore invalid. We can see why matching a bound variable is invalid, by considering the form of the theorem which generated the rewrite. All pattern variables in a rewrite correspond to quantified variables in a theorem, leading to $\forall x, (\lambda y, x)1 = x$. This theorem cannot possibly be applied with $x := y$, as y is not in scope when x is defined. This reasoning applies to any theorem and thus any rewrite generated from a theorem.

Solution We fix invalid matching in `egg` by turning all rewrites into conditional rewrites, and blocking any rewrite initiated by an invalid match. This is achieved in egg by providing a custom *applier*[17] for rewrites, which is a hook into egg's rewriting system. When a rewrite with a custom applier has its trigger pattern matched, the substitution resulting from e-matching is passed to the applier. The applier then runs a user-defined function which also has access to (and can mutate) the e-graph, as well as the trigger pattern. Thus, to block invalid matches, we use a custom applier which performs a depth-first traversal of the trigger pattern while tracking four variables:

- *pos*: the current position in the pattern term. A position is represented as a sequence of indices which describe how to traverse nodes starting at a given root node. Each index dictates which child of a node to visit.
- *depth*: the binder depth at the current position. This starts at 0 and is incremented when visiting the body of a binder.
- *par*: the position of the latest parent binder of the current node.
- *pars*: a partial map of visited variables to their parent binder position.

[17] Appliers are egg's terminology for what we called "callbacks" for rewrites in previous sections.

Additionally, we write $d_{var}(c)$ to denote the set of loose bound variables visible from e-class c, as defined in the next paragraph. Based on this, we detect an invalid match, if and only if one of the following conditions holds.

1. An already visited variable $?x$ with $d_{var}(\sigma(?x)) \neq \emptyset$ has $pars(?x) \neq par$. That is, the latest parent binder position of $?x$ does not match the parent binder position of a previous occurrence of $?x$.
2. A variable $?x$ with $b \in d_{var}(\sigma(?x))$ appears at a position where $b < depth$.

These two conditions correspond precisely to the problematic scenarios described above. Thus, if either one of them is satisfied during the traversal, we abort a rewrite.

Loose Bound Variable Analysis We implement d_{var} efficiently, using an e-class analysis. A given bound variable index b is contained in the analysis value $d_{var}(c)$ of e-class c, if there is *some* expression represented by c which contains a loose bound variable whose index is b at c when adjusted for binder depth. To define this analysis, we first need to introduce an e-class analysis for natural number constants. The analysis value $d_{\mathbb{N}}(c)$ tracks whether c contains an e-node which is a natural number (that is, an instance of the Nat constructor in our expression language in egg). The domain of this analysis is $\mathbb{N} \cup \{\bot\}$ with the following *make* and *join*:

$$make(n) := \begin{cases} \mathsf{x} & \text{if } n = \mathsf{Nat(x)} \\ \bot & \text{otherwise} \end{cases} \qquad join(d_1, d_2) := \begin{cases} \bot & \text{if } d_1 = d_2 = \bot \\ d_1 & \text{if } d_2 = \bot \\ d_2 & \text{if } d_1 = \bot \\ max(d_1, d_2) & \text{otherwise} \end{cases}$$

We note that the choice of *max* is an arbitrary choice of function satisfying the semilattice laws. The reason being that if we join e-classes with $d_1, d_2 \in \mathbb{N}$ and $d_1 \neq d_2$, then we cannot sensibly choose an analysis value. In fact, in such a case we know that an unsound rewrite has occurred. Based on $d_{\mathbb{N}}$, we define d_{var} over domain $\mathcal{P}(\mathbb{N})$ with the following *make* and *join*:

$$make(n) := \begin{cases} \{d_{\mathbb{N}}(\mathsf{b})\} & \text{if } n = \mathsf{BVar(b)} \\ d_{var}(\mathsf{fn}) \cup d_{var}(\mathsf{arg}) & \text{if } n = \mathsf{App(fn,\ arg)} \\ d_{var}(\mathsf{ty}) \cup \downarrow d_{var}(\mathsf{bd}) & \text{if } n = \mathsf{Lam(ty,\ bd)} \\ d_{var}(\mathsf{ty}) \cup \downarrow d_{var}(\mathsf{bd}) & \text{if } n = \mathsf{Forall(ty,\ bd)} \\ \emptyset & \text{otherwise} \end{cases} \qquad join := \cup$$

For $\mathsf{BVar(b)}$ we can assume $d_{\mathbb{N}}(\mathsf{b}) \in \mathbb{N}$ as b must be of the form $\mathsf{Nat(\_)}$ in well-formed expressions. We write $\downarrow S := \{i \mid i + 1 \in S\}$ for the function which down-shifts all indices in a given set and removes those which fall below 0. We perform down-shifting whenever we encounter a binder. This implies that the set $d_{var}(c)$ of a given e-class c always contains the indices of loose bound variables as they are called at the binder depth of c. For variable index 0 this implies deletion, as the bound variable $\hat{0}$ is never visible outside its immediate parent binder. As an example, if we assume e-class c to represent the expression $\lambda\,\lambda\,\hat{0}\,\lambda\,\hat{4}\,\hat{5}$, then $d_{var}(c) = \{1, 2\}$, as the bound variables $\hat{4}$ and $\hat{5}$ are called $\hat{1}$ and $\hat{2}$ outside the outermost binder, while the bound variable $\hat{0}$ is not visible there.

4.3.2 Variable Rebinding

When performing a substitution like $s := t[x \mapsto e]$, loose bound variables occurring in e can turn into bound variables in s by being placed under a binder in t. This is called *invalid capture* of loose bound variables in e by binders in t. Invalid capture is problematic as it breaks the expected semantics of substitution. For example, if we apply the rewrite $?x \Rightarrow (\lambda\,?x)\,1$ to the expression $\lambda\,\hat{0}$ by matching $?x$ against $\hat{0}$, we get the incorrect result $\lambda(\lambda\,\hat{0})\,1$, by invalid capture of $\hat{0}$. The reverse problem of variables being "uncaptured" can only occur when t

itself already contains loose bound variables. As such, this problem is usually not considered in the context of λ calculus, but can occur in the context of e-graphs where rewrites do not distinguish between open and closed terms.[18] An example of "uncapturing" occurs if we apply the rewrite $\lambda(\lambda\,?x)\,0 \Rightarrow \lambda\,?x$ to the open term $\lambda(\lambda\,\hat{4})\,0$, which yields the non-equivalent term $\lambda\,\hat{4}$. Here, $\hat{4}$ would need to be shifted to $\hat{3}$ to retain its semantics.

Both of these problems can be solved by having every rewrite be a dynamic rewrite which checks whether a given matched pattern variable refers to loose bound variables, and if so, shifts them as needed. The required shift for an occurrence of a variable $?x$ in a rewrite $lhs \Rightarrow rhs$ is determined by the binder depth of $?x$ in lhs and rhs. Let $depth_{lhs}(?x)$ denote the binder depth of $?x$ in lhs, and $depth_{rhs,p}(?x)$ denote the binder depth of an occurrence of $?x$ in rhs at position p. Then any occurrence of $?x$ in rhs at position p needs to be shifted by $depth_{lhs}(?x) - depth_{rhs,p}(?x)$. Note that $depth_{lhs}(?x)$ is well-defined even when $?x$ appears multiple times in lhs, as long as we ensure valid matches. In fact, in our implementation, we pre-compute $depth_{lhs}(?x)$ for each variable $?x$ in lhs during the check for invalid matches described in the previous section. As such, we always need to check for invalid matches before we can perform the index correction described below.

Index Correction We correct the loose bound variable indices for each pattern variable occurrence in rhs as follows. Let σ be the substitution resulting from e-matching on lhs, and let $\updownarrow_o$ denote a function which shifts the indices of loose bound variables in a given e-class by an offset of o. We construct a shifted substitution $\overline{\sigma}$ and shifted pattern term $\overline{rhs}$, such that $\overline{rhs}$ introduces new pattern variables for different occurrences of the same pattern variable in rhs, and $\overline{\sigma}$ extends σ such that these new pattern variables map to e-classes with the corrected loose bound variable indices. More precisely, we construct $\overline{\sigma}$ and $\overline{rhs}$ such that:

1. $\sigma \subseteq \overline{\sigma}$, meaning we have $\overline{\sigma}(?x) = \sigma(?x)$ for all $?x \in dom(\sigma)$.
2. For each occurrence of pattern variable $?x$ in rhs at binder depth d with $d_{var}(\sigma(?x)) \neq \emptyset$, $\overline{rhs}$ replaces that occurrence with a fresh variable $?x_d$.
3. For each $?x_d \in \overline{rhs}$, let $\overline{\sigma}(?x_d) = \updownarrow_o(\sigma(?x))$ with $o = depth_{lhs}(?x) - d$.

This construction is implemented by a depth-first traversal of rhs, which tracks the current binder depth. To then apply the rewrite $lhs \Rightarrow rhs$ with index correction, we merge the e-classes of $\sigma'(lhs)$ and $\overline{\sigma}'(\overline{rhs})$.[19] We note that shifting the bound variables in e-classes corresponding to variables in rhs can be performed in a more direct manner than shown here. We only use the approach of constructing a shifted substitution, as it allows us to use egg's `union_instantiations` function for merging the final e-classes. This function merges the e-classes of given pattern expressions under a given substitution while, crucially, preserving explanation information. Using egg's `union` or `union_trusted` functions to merge e-classes directly does not preserve the information required to generate correct explanations. Preservation of explanation information is, in fact, an issue which we have yet to solve for the e-classes shifting function $\updownarrow_o$. The implementation of $\updownarrow_o$ is non-trivial and relies on the more general mechanism of e-class substitution, as covered in the following section.

4.3.3 E-Class Substitution

Substitution is commonly one of the first operators which is defined for a λ calculus, as it underpins other constructs like β-reduction. In a de Bruijn representation of λ-expressions, a substitution $t[n \mapsto s]$ replaces all occurrences of variable $\hat{n}$ in t by s. A concise definition of this (for untyped λ calculus) as given in [Pie02] is shown below. Notably, the index n and the indices of variables in s need to be increased as we visit binders (the latter is handled by $\uparrow^1$):

[18] A term is *open* if it contains free (or loose bound) variables, and *closed* if it does not.
[19] As $dom(\sigma) = vars(lhs)$ and $\sigma \subseteq \overline{\sigma}$, we get $\sigma'(lhs) = \overline{\sigma}'(lhs)$.

$$\hat{b}[n \mapsto s] := \begin{cases} s & \text{if } b = n \\ \hat{b} & \text{otherwise} \end{cases}$$
$$(\lambda\, t)[n \mapsto s] := \lambda\, t[n+1 \mapsto \uparrow^1(s)]$$
$$(t_1\, t_2)[n \mapsto s] := t_1[n \mapsto s]\ t_2[n \mapsto s]$$

In the context of e-graphs, substitution cannot be defined this easily and is in fact notoriously difficult. We cannot simply apply the given substitution function, as e-graphs do not contain concrete expressions. Instead, we must consider how to perform substitution on e-nodes and e-classes. Multiple different approaches have been developed for this:

1. Make substitution an explicit constructor of the expression language and introduce rewrites which encode its semantics with respect to other constructors, as described in [WNW+21].[20]
2. Perform substitution by extracting a single expression e from the target e-class and performing substitution on e before adding it back into the e-graph, as described in [Kœh22].
3. Traverse the subgraph of the target e-class and construct an equivalent subgraph where all relevant e-nodes are substituted.

Approach (1) introduces subtleties with respect to correctly handling rewrite rules and makes scaling harder by increasing the number of nodes in the e-graph [Kœh22]. Approach (2) compromises completeness by potentially never extracting all represented expressions of a given e-class, and thus missing potential substitutions. Approach (3) has the highest implementation cost and requires explicit handling of explanations, which is implicit in the other approaches. Yet, in the context of our proof tactic, we use (3) as it offers better completeness than (2) and avoids the increase in nodes of (1).

For a traversal-based approach to e-class substitution, we define the substitution function subst as follows. Let $\sigma : \mathbb{N} \times \mathbb{N} \to \mathit{expr}$ be a substitution mapping from a variable index and binder depth to term of our egg expression language. Let g be an e-graph containing an e-class c. Then $\mathsf{subst}(g, c, \sigma)$ extends g with an e-class s such that for every expression e represented by c, s represents $\sigma'(e)$. We use σ' to denote the lifting of σ to apply to expressions with the binder depth being derived starting at c. As an example of using subst, we define the variable shifting function $\updownarrow_o$ from the previous section as follows (while assuming the e-graph g is known from context):

$$\updownarrow_o(c) := \mathsf{subst}(g, c, \sigma_\pm(o))$$
$$\sigma_\pm(\mathit{offset})(\mathit{idx}, \mathit{depth}) := \begin{cases} \mathsf{BVar}(\mathsf{Nat}(\mathit{idx} + \mathit{offset})) & \text{if } \mathit{idx} > \mathit{depth} \\ \mathsf{BVar}(\mathsf{Nat}(\mathit{idx})) & \text{otherwise} \end{cases}$$

The substitution $\sigma_\pm(o)$ offsets a given variable index idx by a given amount offset, if idx is a loose bound variable (if $\mathit{idx} > \mathit{depth}$). Otherwise, it simply returns an expression representing a bound variable with index idx. Thus, according to the intended behavior of subst, $\updownarrow_o(c)$ extends the e-graph g with a new e-class which represents the same expressions as c, except that all loose bound variables are replaced by variables whose indices are shifted by o. The subst function can therefore loosely be understood as a function which lifts a given substitution function σ over bound variable indices, to a function over e-classes.

[20] An improvement to this approach is given in *Optimizing Beta Reduction in E-Graphs* at EGRAPHS 2023.

E-Graph Traversal Our approach for implementing $\mathsf{subst}(g, c, \sigma)$ is to traverse the subgraph of the target e-class c, and construct an equivalent subgraph where all relevant e-nodes are substituted according to the substitution σ. Traversing an e-graph rooted at a given e-class c entails two kinds of traversal. First, we must traverse all e-nodes in c and, second, we must traverse all child e-classes of each e-node. For this, we use a recursive depth-first traversal which additionally iterates over each e-node of a given e-class on visit, as sketched in Figure 10.

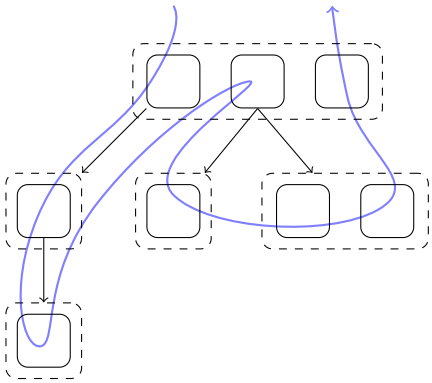


Figure 10: Example of a depth-first traversal over e-classes and e-nodes.

As e-graphs can contain cycles, we make sure to avoid infinite recursion by remembering which e-classes have already been visited. Additionally, we track the current binder depth of an e-class relative to c, as this information is required by the substitution function σ. While traversing the e-graph, we construct substitutes for each visited e-class and e-node. A substitute for a given e-class c containing e-nodes $n_1, ..., n_k$ is a new e-class which contains exactly the substitutes for e-nodes $n_1, ..., n_k$. For a *leaf* e-node n, like a bound variable or named constant, constructing the substitute is simple. If n is a bound variable with index i at binder depth d, we obtain the substitute by $\sigma(i, d)$. If n is not a bound variable, it remains as it. Difficulties only arise once we try to construct substitutes for e-nodes with children: function applications, λ-abstractions and $\forall$-abstractions. Namely, if an e-node n has child e-classes $c_1, ..., c_m$, then in order to construct the substitute of n we first need to construct the substitute e-classes $s_1, ..., s_m$ for $c_1, ..., c_m$. However, as e-graphs can contain cycles, the construction of some s_i may itself depend on the substitute of n. That is, we can have cyclic dependencies. The way in which these cyclic dependencies are resolved is one of the key aspects of traversal-based e-class substitution algorithms.

Breaking Cyclic Dependencies A cyclic dependency occurs when an e-class c contains an e-node n, which (transitively) contains c as a child. When such a dependency exists, we cannot construct a substitute s for c, as this depends on constructing a substitute for n, which itself requires s. Luckily, there are two properties of e-graphs which allow us to escape this cycle. First, the substituted e-class s does not have to be constructed "all at once". That is, during our substitution algorithm, we can first construct an incomplete version of s, and then add the necessary substitute e-nodes as they become available. This results in the following intuitive approach for breaking cyclic dependencies. Let *sub* be a partial map from e-class ids to e-class ids of substitute e-classes. That is, when a substitute e-class s for c is created, we record $c \mapsto s$ in *sub*. Then, to construct the substitute of an e-node n, check if $sub(c_i) = s_i$ (for some s_i) for each child c_i of n. If this is the case, use s_i as the ith child in the substitute e-node for n. Otherwise, if $sub(c_i)$ is not defined, create an empty e-class s_i, add $c_i \mapsto s_i$ to *sub*, and use s_i as the ith child in the substitute e-node for n. Thus, any cyclic dependency between c and n is resolved by first creating an empty e-class s for c upon visiting n. Unfortunately, egg

does not allow the creation empty e-classes.[21] As a simple workaround, Rudi Schneider has implemented a version of this approach which adds a constructor for placeholder expressions to the expression language and uses it to construct pseudo-empty e-classes by populating them with unique placeholder e-nodes. While this allows for an elegant implementation of e-class substitution, it has similar downsides as an explicit constructor for substitution (Approach 1). To avoid such a workaround, our implementation of e-class substitution uses an approach similar to data driven scheduling in process networks [Par95]. When visiting an e-node, we postpone the construction of its substitute until substitute e-classes are available for each of its children. Thus, substitute e-nodes and e-classes are potentially constructed in an order which does not follow the depth-first traversal order. Crucially, the correctness (specifically the termination) of this approach relies on what we call the *Cycle Breaking Property* of e-graphs: Every cycle in an e-graph contains at least one e-class which contains at least one e-node n where each child e-class of n is either (1) the root of an acyclic subgraph or (2) the root of a subgraph satisfying the Cycle Breaking Property. Intuitively, this corresponds to the fact that an e-graph can never contain an e-class which does not represent any (finite) term. This property ensures that the construction of a substitute e-node cannot be postponed indefinitely. This is because if an e-node n satisfies case (1), then its substituted children will always be constructed without being postponed, so the construction of the substitute for n will not be postponed. If it satisfies case (2), then by induction its substituted children will eventually be constructed, thus allowing the substitute of n to be constructed.

Data Driven Scheduling Based on the Cycle Breaking Property, we use a data driven scheduling approach for our substitution algorithm. In this section, we describe how we implement it in the context of a depth-first traversal over an e-graph. The procedure relies on the following variables:

- *wip*: The set of visited e-classes in the depth-first traversal.
- *sub*: A partial map which maps a given e-class to its corresponding substituted e-class.
- *todo*: A partial map from postponed e-nodes to the e-class from which it originated.
- *wait*: A map from postponed e-nodes to the number of children they are waiting on.
- *deps*: A partial map from e-classes to the postponed e-nodes waiting on the given e-class.

These variables are used for data driven scheduling as follows. Let c be an e-class containing an e-node n which has child e-classes $c_1, ..., c_k$. To construct a substitute for n, we try to construct substitutes for all $c_1, ..., c_k$. If $sub(c_i) = s_i$ for some s_i, then c_i already has a substitute. Otherwise, if $c_i \in wip$, then we have a dependency cycle and we call c_i *pending*. If $c_i \notin wip$, then we have not yet visited c_i, so we recursively call our substitution procedure on c_i. If this procedure returns some e-class s_i, then this is the substitute for c_i. Otherwise, if the procedure was unable to construct a substitute for c_i, then we also have a dependency cycle, and again call c_i pending. Thus, by iterating over all children of n, we obtain a subset P of children which are pending. If $P = \emptyset$, then the substitute of n can be immediately constructed. Otherwise, the construction of n's substitute has to be postponed. For this we add the entry $n \mapsto c$ to the map *todo* of postponed e-nodes, we add an entry $n \mapsto |P|$ to the *wait* map, and for each $c_i \in P$, we add n to c_i's entry in the map *deps* of dependencies. Thus, we have registered n as waiting on $|P|$ many children with the *deps* map indicating which children are missing. Afterwards, we proceed to the next e-node in c, according to our depth-first traversal over e-graphs. If all e-nodes in an e-class are postponed, the substitution on the e-class returns with an indication that it is pending. The reason we store information about postponed e-nodes across different maps is that it allows us to efficiently track the progress on its dependencies. Namely, let p be one of the pending children of e-node n. Then, when a substitute s for p

[21] This is understandable from a mathematical standpoint, as equivalence classes should form a partition and can therefore not be empty. Accordingly, the developers of egg do not intend to add empty e-classes to egg.

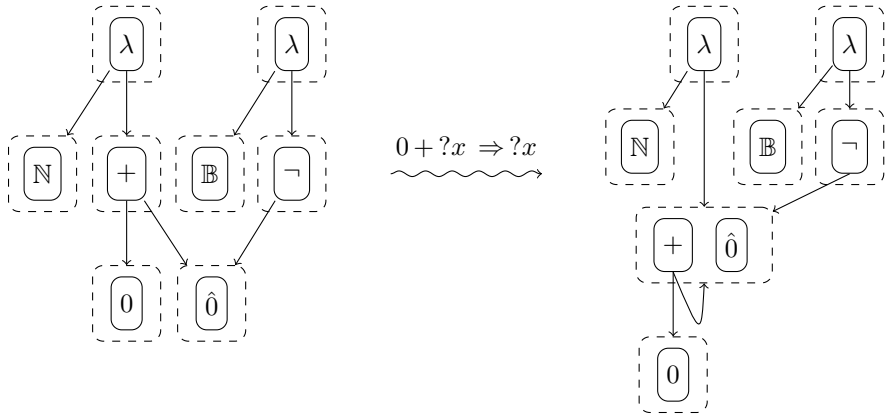
is created, we update the maps as follows. First, we add an entry $p \mapsto s$ to the substitution map *sub*. Next, we obtain the set $W := deps(p)$ of e-nodes which are waiting on a substitute for p. For each e-node $w \in W$ we decrease $wait(w)$ by 1, indicating that w is waiting on one fewer child (namely p). If this causes the count to become 0, then a substitute for w can be constructed. In particular, we know that for each child c_i of w, $sub(c_i)$ is defined. Thus, we construct the substitute e-node s_w for w using the children $sub(c_1), ..., sub(c_k)$. Now, if there already exists a substitute e-class s for w's e-class of origin $todo(w)$, then we simply add s_w to s. Otherwise, if there does not yet exist a substitute e-class s for $todo(w)$, then we construct it by adding the e-node s_w to the e-graph, which yields an e-class. This is precisely the point where we are avoiding the necessity for empty e-classes, as we have waited to construct s until some substitute e-node was available for its construction. As constructing this new e-class might enable other substitute e-nodes to be constructed, the described procedure is recursive. That is, constructing a single substitute e-class can allow all postponed e-nodes in a given dependency cycle to be transitively resolved.

Caveats The described substitution procedure is a slight simplification of the actual implementation.[22] For example, proper handling of binder depth is ignored. When binder depth is considered, a given e-class may have multiple substitutes which correspond to the given e-class at different binder depths. This can have the effect of creating many copies of a given e-class, when the e-class is contained in a cycle with a binder. However, the most notable simplification is in the description of how we add substitute e-nodes to the e-graph. Namely, we stated that when a substitute e-node is created and a corresponding substitute e-class already exists, we simply add the e-node to the e-class. In actuality, we avoid adding any e-nodes to any existing e-classes. Doing so could cause non-termination by continuous construction and subsequent traversal of substitute e-nodes. Thus, instead of adding new e-nodes to existing e-classes in the e-graph, we simply record which e-nodes should be added to which e-classes. Once traversal completes, we process the recorded e-nodes and add them to their target e-classes.[23] While this approach does not affect the correctness of the substitution algorithm, we currently do not propagate explanation information correctly in this final step. As a result, rewrites involving substitution (like for β- and η-reduction in Section 4.4.4) can produce broken explanations, which causes our proof tactic to fail during proof reconstruction. While we believe this problem to be solvable without major changes to the substitution algorithm, we have yet to resolve it.

4.3.4 Bound Variable Aliasing

Of the three classes of problems resulting from direct representation of bound variables in e-graphs, *bound variable aliasing* is the most deep-rooted. An example of aliasing is shown in the e-graph in Figure 11a. It represents two root terms $\lambda\,\mathbb{N}, 0 + \hat{0}$ and $\lambda\,\mathbb{B}, \neg\hat{0}$, where $\mathbb{B}$ denotes the type of Booleans.

[22] https://github.com/marcusrossel/lean-egg/blob/main/Rust/src/subst.rs
[23] Thanks to Rudi Schneider for this approach.



(a) (b)

Figure 11: Examples of bound variable aliasing (a), and ill-typed terms by aliasing (b).

As each e-node can be contained in at most one e-class, the e-node for the bound variable $\hat{0}$ is shared by both λ-expressions. This can cause issues when the e-node for $\hat{0}$ becomes equivalent to other expressions. For example, consider the application of the rewrite $0 + ?x \Rightarrow ?x$ as shown in Figure 11b. This rewrite only applies to the term in the λ-expression over $\mathbb{N}$, and merges the e-classes of $0 + \hat{0}$ and $\hat{0}$. However, as $\hat{0}$ is also referenced by the λ-expression over $\mathbb{B}$, this merge affects which terms are represented by the entirely unrelated e-class of $\neg\hat{0}$. Specifically, it now also represents the ill-typed expression $\neg(0 + \hat{0})$.

The problem of bound variable aliasing cannot be solved by techniques like dynamic rewriting, as it is rooted in the very representation of e-graphs. Each e-node in an e-class can be contained in at most one e-class, and thus, syntactically equal bound variables must be shared. Our only option is therefore to change the representation of bound variables, such that different variables are syntactically distinct. However, this is not possible in general, as e-graphs can contain cycles and would therefore require representing infinitely many syntactically distinct bound variables. Additionally, such an approach would greatly reduce the value of using an e-graph, by creating syntactically distinct copies of semantically equivalent terms.

A novel technique for approaching this problem is introduced in [SKS24] by means of *slotted e-graphs*. Slotted e-graphs modify the concept of an e-graph to include binders and variables as first-class citizens. E-classes can then be parameterized over variables which need to be instantiated by any parent e-node which wants to refer to the given e-class. Thus, naming of variables becomes local and aliasing is not possible. Unfortunately, this work is still at an early stage and does not have a practical implementation which includes explanations. Thus, in our proof tactic, we resort to a different approach, which is to simply accept that bound variable aliasing is a potential source of unsoundness during equality saturation. Recall that this does not compromise the soundness of our tactic, as the Lean kernel will not accept an invalid proof produced by egg. Thus, any problems due to bound variable aliasing manifest in a failure during proof reconstruction. We are, however, currently not aware of any examples where proof reconstruction fails do to this issue. Therefore, we believe bound variable aliasing to be of little concern in practice.

4.4 Definitional Equalities

When Lean checks whether two expressions should be considered equal without proof, it uses the definitional equality relation $\equiv$. For example, $(\lambda x, x + 0)\, 1 = 1$ holds without proof, as both sides are definitionally equal. In an e-graph, two expressions are equal without proof only if they are syntactically equal. Thus, the previous equality is not provable without rewrites. These differing notions of equality require consideration, as they affect what we can and cannot prove during equality saturation. For example, we might not identify a goal as being proven, simply because we could only reach a term which was definitionally but not syntactically equal

to the goal. Further, we might fail to apply a rewrite when its trigger pattern matches a term only definitionally, but not syntactically. For example, the equation $(\lambda\, l, l+1)\, l_1 = (\lambda\, l, l+1)\, l_2$ does not suffice to prove $l_1 + 1 = l_2 + 1$ in an e-graph, as it does not match any goal term syntactically. To overcome these differences, we employ various techniques which allow egg to convert between different definitionally equal representations of the same expression. Some of the rules of definitional equality are already intrinsic to the way e-graphs work in conjunction with our encoding of Lean expressions in egg. Namely, those which express that definitional equality is an equivalence relation which is a congruence with respect to application, abstraction and quantification given below. The relation $\Gamma \vdash e : \alpha$ denotes that e has type α in the context Γ, where a context is a list of typing judgements. To express that a context contains a specific typing judgement $a : b$, we write $\Gamma, a : b$.

$$\frac{\Gamma \vdash e : \alpha}{\Gamma \vdash e \equiv e}\ \textit{refl} \qquad \frac{\Gamma \vdash e_1 \equiv e_2}{\Gamma \vdash e_2 \equiv e_2}\ \textit{symm} \qquad \frac{\Gamma \vdash e_1 \equiv e_2 \quad \Gamma \vdash e_2 \equiv e_3}{\Gamma \vdash e_1 \equiv e_3}\ \textit{trans}$$

$$\frac{\Gamma \vdash \alpha_1 \equiv \alpha_2 \quad \Gamma, \hat{0} : \alpha_1 \vdash e_1 \equiv \alpha_2}{\Gamma \vdash \lambda\, \alpha_1, e_1 \equiv \lambda\, \alpha_2, e_2}\ \textit{congr}_\lambda \qquad \frac{\Gamma \vdash \alpha_1 \equiv \alpha_2 \quad \Gamma, \hat{0} : \alpha_1 \vdash e_1 \equiv \alpha_2}{\Gamma \vdash \forall\, \alpha_1, e_1 \equiv \forall\, \alpha_2, e_2}\ \textit{congr}_\forall$$

$$\frac{\Gamma \vdash e_1 \equiv e_1' \quad \Gamma \vdash e_2 \equiv e_2'}{\Gamma \vdash e_1\, e_2 \equiv e_1'\, e_2'}\ \textit{cgr}_{app}$$

Other forms of definitional equality, and the techniques used to implement them, are outlined in the following sections.

4.4.1 Normalization

Some definitional equalities can be implemented by eliminating their associated syntactic constructs from all expressions – that is, by partially normalizing expressions. The most prominent syntactic construct we eliminate in this way are let-expressions, which have the following syntax and definitional equality rule:

$$e ::= \ldots \mid \mathsf{let}\, x : e := e;\, e \qquad \frac{\Gamma \vdash e_x : \alpha \quad \Gamma \vdash e[x \mapsto e_x] : \beta}{\Gamma \vdash (\mathsf{let}\, x : \alpha := e_x;\, e) \equiv e[x \mapsto e_x]}\ (\zeta)$$

The definitional equality rule ζ shows how we can eliminate let-expressions. Namely, by substituting its variable x by its value e_x in the body e. This is called ζ-reduction, and is one of the normalization steps we perform on all expressions before they are encoded to egg. Thus, the expression language in egg does not even have a construct for let-expressions, and we do not need to consider their definitional equality rules. Note that while ζ-reduction can theoretically lead to an exponential size increase of the reduced expression, this is not a problem in practice as it is uncommon to have deeply nested let-expressions where each variable appears multiple times – especially as part of a theorem statement.

Another case of eliminating syntactic constructs regards Lean's internalized structure projections. Recall that structure projections are functions which map a member of a structure type to one of its fields. For efficiency, Lean defines an internalized construct for them:

$$e ::= \ldots \mid \pi_i^P\, e \qquad \frac{\Gamma \vdash \pi_i^P\, e_1 \equiv \pi_i^P\, e_2}{\Gamma \vdash e_1 \equiv e_2} \qquad \frac{\Gamma \vdash \pi_i^P\, (c_P\, b_1 \ldots b_n) : \alpha}{\Gamma \vdash \pi_i^P\, (c_P\, b_1 \ldots b_n) \equiv b_i}$$

Given a structure type P and a member $e : P$, the application of the structure projection for the ith field of P is written $\pi_i^P\, e$. The first associated definitional equality rules state that $\equiv$ is a congruence with respect to structure projections. The second rule states that applying the ith structure projection to an element $c_P\, b_1 \ldots b_n$, where c_P is the constructor of P, yields precisely the ith argument b_i. As applying the naive encoding of structure projections (via recursors) yields the same behavior, naive and internalized structure projections are definitionally equal. When encoding expressions to egg, we therefore eliminate all occurrences of internalized structure projections by replacing them with their recursor-based counterpart. This again means that the language encoded in egg does not have a construct for internalized structure projections, and we do not need to consider their definitional equality rules.

4.4.2 Proof Irrelevance

The definitional equality rule for *proof irrelevance* states that all proofs of the same proposition are definitionally equal:

$$\frac{\Gamma \vdash p : \mathcal{U}_0 \quad \Gamma \vdash h_1 : p \quad \Gamma \vdash h_2 : p}{\Gamma \vdash h_1 \equiv h_2}$$

For example, we can prove the proposition $\exists\, n : \mathbb{N}, n > 0$ with infinitely many different choices of n. Yet, all proofs of the proposition are considered definitionally equal. For our proof tactic, this rule is relevant when the proof goal or an equation contains proof terms. A common case of this are safe array accesses written as $\mathsf{arr}[\mathsf{i}]'\mathsf{h}$, where arr is an array, i is an index into the array, and h is a proof of $\mathsf{i} < \mathsf{as.size}$. A proof goal like $\mathsf{arr}[\mathsf{i}]'\mathsf{h}_1 = \mathsf{arr}[\mathsf{i}]'\mathsf{h}_2$ should be provable, as $\mathsf{h}_1 \equiv \mathsf{h}_2$ by proof irrelevance. To achieve proof irrelevance in egg, we erase proof terms in our encoding and replace them with their types. For this, we introduce a special proof constructor to the expression language with which we wrap the type (the proposition) of the proof, instead of the proof term itself. Thus, different proofs h_1 and h_2 of the same proposition p are encoded syntactically equally as $\mathsf{proof}\ \overline{\mathsf{p}}$, where $\overline{\mathsf{p}}$ denotes the encoding of p. The proofs in the safe array access expression from above are therefore both encoded as $\mathsf{proof}\ \overline{\mathsf{i} < \mathsf{arr.size}}$. This makes the proof of $\mathsf{arr}[\mathsf{i}]'\mathsf{h}_1 = \mathsf{arr}[\mathsf{i}]'\mathsf{h}_2$ immediate, as both sides of the equation have the same encoded syntax. While this approach to proof irrelevance is easily implemented during encoding, it comes at a cost during proof reconstruction. Namely, as we erase the proof terms in our encoding, we need to find a way to reintroduce them during proof reconstruction. This is discussed in Section 4.6.

4.4.3 Natural Number Literals

Natural number literals are one of Lean's internalizations which also benefits our encoding in egg. Normally, natural numbers are represented as applications of the zero and succ constructors of the Nat type. As this results in a unary encoding, even moderately large natural numbers result in enormous expressions of the form $\mathsf{app}\,(\mathsf{const\ Nat.succ})(...(\mathsf{const\ Nat.zero}))$. Thus, Lean introduces an explicit representation of natural number literals in its expression language:

$$e ::= ... \mid l_n \text{ where } n \in \mathbb{N} \qquad \frac{}{\Gamma \vdash l_0 \equiv \mathsf{zero}} \qquad \frac{}{\Gamma \vdash l_{n+1} \equiv \mathsf{succ}\ l_n}$$

A natural number like 42 is then efficiently represented as l_{42} , or equivalently, $\mathsf{lit}\ (.\mathsf{natVal}\ 42)$ as a term of Expr. We use this same encoding in our expression language in egg to reap the same efficiency benefits. Yet, as a result, we have to implement the definitional equality rules for natural number literals. These rules state that literals are definitionally equal to their unary representations. The first rule can simply be expressed as the rewrite $(\mathsf{lit}\ 0) \Leftrightarrow (\mathsf{const\ Nat.zero})$. The second rule, on the other hand, cannot be expressed as a simple rewrite, as the result of the rewrite is determined based on the value of the literal that was matched. That is, we need to use a dynamic rewrite in conjunction with an e-class analysis. Based on the e-class analysis $d_{\mathbb{N}}$ for natural numbers defined in Section 4.3.1, we implement the forward direction of the second definitional equality rule using the trigger pattern $(\mathsf{lit}\ ?\mathsf{n})$ and an e-matching callback which performs the following steps.

1. Given the substitution σ resulting from e-matching, obtain the e-class $c := \sigma(?n)$ of pattern variable $?n$.
2. Obtain the analysis value $d_{\mathbb{N}}(c)$ which we can assume not to be $\bot$.[24]
3. If $d_{\mathbb{N}}(c) = 0$, then the rewrite is not applicable, and we leave the e-graph unchanged. Otherwise, add the expression $\mathsf{app}\ (\mathsf{const\ Nat.succ})\ (\mathsf{lit}\ d_{\mathbb{N}}(c)-1)$ to the trigger's e-class.

[24] The reason is that terms coming from Lean must always have a natural number value under a lit constructor, and this is preserved under rewrites.

The backward direction of the definitional equality rule is very similar with pattern term app (const Nat.succ) (lit ?n) being the trigger. The callback only differs in Step 3 where we unconditionally add the expression lit $d_{\mathbb{N}}(c) + 1$ to the e-graph.

It is important that we implement rewrites both for the forward and the backward directions of these definitional equality rules. The backward direction suffices to ensure provability of equivalence of terms in the e-graph which are equal up to definitional equality of natural number literals. Yet, we also need the forward direction to ensure that rewrites can apply, despite varying representations of natural numbers. For example, a rewrite of the form app (const Nat.succ) ?x ⇒ y does not apply to lit 5. This is solved by first expanding lit 5 to app (const Nat.succ) (lit 4) using the forward direction of the second definitional equality rule. Unfortunately, this sort of expansion introduces the problem of blowing up the number of nodes in an e-graph by unfolding natural numbers to their unary representation. In practice this has not been an issue, as theorems tend not to contain large natural number constants. Yet, it is easy to construct examples which demonstrate this blowup. In the future, we hope to find a way to bound the depth of this expansion.

Computations In addition to efficient representation of natural numbers, Lean also internalizes rules for certain computations on natural numbers. This allows theorems like $123 \cdot 456 = 56088$ to be proven simply by Lean's kernel computing whether both sides evaluate to the same value. Without an internalization, the kernel would need to unfold the definitions of multiplication (and subsequently addition) to check the equality, which quickly becomes infeasible for larger numbers. Formally, these internalized computations are defined by definitional equality rules of the form:

$$\frac{}{\Gamma \vdash \mathsf{Nat.add}\, l_n\, l_m \equiv l_{n+m}} \qquad \frac{}{\Gamma \vdash \mathsf{Nat.mul}\, l_n\, l_m \equiv l_{n \cdot m}} \qquad \ldots$$

Among others, the set of internalized computations includes addition, subtraction, multiplication, exponentiation, and division. We implement rules for them following a similar approach as for natural number literal conversion.

4.4.4 β- and η-Reduction

The definitional equality rules β and η establish relationships between application, abstraction and substitution.

$$\frac{\Gamma, \hat{0} : \alpha \vdash e : \beta \quad \Gamma \vdash e_0 : \alpha}{\Gamma \vdash (\lambda \alpha, e)\, e_0 \equiv e[\hat{0} \mapsto e_0]}\ (\beta) \qquad \frac{\Gamma \vdash e : (\forall \alpha, \beta)}{\Gamma \vdash (\lambda \alpha, e\, \hat{0}) \equiv e}\ (\eta)$$

The β-rule states that application of a λ-expression corresponds to substitution. The η-rule allows us to remove unnecessary abstraction around an expression. Note that η requires e to be well-typed, independent of the type of $\hat{0}$. That is, the context of the typing judgement $\Gamma \vdash e : (\forall \alpha, \beta)$ does not have to contain a judgement $\hat{0} : \alpha$. This can only be satisfied if e does not refer to $\hat{0}$, which is an important precondition as otherwise e would contain a loose bound variable after η-reduction.

Directionality Both the β- and η-rule induce bidirectional rewrites. However, we only consider the forward direction for each. That is, we neither perform η-expansion by adding an abstraction around an expression, nor any kind of reverse β-reduction. Adding such rules would entail unnecessary blowup of the e-graph as, for example, η-expansion applies to any expression. However, restricting ourselves to the forward direction means that we need to take some care to ensure that rewrites can apply even when we cannot β- or η-expand the expressions they should match. For example, consider the equation $(\lambda\, l, l+1)\, l_1 = (\lambda\, l, l+1)\, l_2$ and the goal $l_1 + 1 = l_2 + 1$. Even with β- and η-reduction rules, egg cannot solve this goal. The reason being that β- and η-reduction can only be applied to terms in the e-graph, but not to terms appearing in rewrites. As such, the given example gets stuck because the terms in the goal are *more* reduced than the terms in the equation. If we instead assume $l_1 + 1 = l_2 + 1$

to be a given equation and $(\lambda\, l, l+1)\, l_1 = (\lambda\, l, l+1)\, l_2$ the proof goal, we can easily apply β-reduction to the goal terms and then solve the goal with the given equation. To work around this limitation, we always add expressions to the e-graph in their least β- and η-reduced form, but encode rewrites in their most β- and η-reduced form. We achieve this by performing β- and η-reduction on expression appearing in equations during normalization (as in Section 4.4.1).

Shifting and Substitution When working with de Bruijn indices, both β- and η-reduction necessitate shifting indices of loose bound variables. Given an expression $\lambda\, e\, \hat{0}$, the η-reduced e is only equivalent if all loose bound variables in e are shifted down by 1. This is necessary as we remove one binder above e. For example, $\lambda\, \hat{2}\, (\lambda\, \hat{0}\, \hat{2})\, \hat{0}$ η-reduces to $\hat{1}\, (\lambda\, \hat{0}\, \hat{1})$. β-reduction is similar in that given an expression $(\lambda\, f)\, e$, the reduced $f[\hat{0} \mapsto e]$ also requires shifting all loose bound variables in f down by 1, as we again remove one binder above f. Additionally, all loose bound variables in e need to be shifted up by the number of binders appearing above the corresponding occurrence of $\hat{0}$ in f. This is necessary to compensate for the fact that we are potentially adding binders above e. For example, $(\lambda\, (\lambda\, \hat{0}\, \hat{1})\, \hat{0}\, \hat{3})(\lambda\, \hat{0}\, \hat{1}\, \hat{2})$ β-reduces to $(\lambda\, \hat{0}\, (\lambda\, \hat{0}\, \hat{2}\, \hat{3}))\, (\lambda\, \hat{0}\, \hat{1}\, \hat{2})\, \hat{2}$. We implement this shifting and substitution based on the e-class substitution function subst from Section 4.3.3. Thus, β- and η-reduction are defined entirely based on the choice of substitution functions σ:

$$\begin{aligned}
\mathsf{eta}(g, c) &:= \mathsf{subst}(g, c, \sigma_{\pm}(-1)) \\
\mathsf{beta}(g, c, \mathit{arg}) &:= \mathsf{subst}(g, c, \sigma_{\beta}(\mathit{arg})) \\
\sigma_{\beta}(\mathit{arg})(\mathit{idx}, \mathit{depth}) &:= \begin{cases} \mathsf{BVar}(\mathsf{Nat}(\mathit{idx} - 1)) & \text{if } \mathit{idx} > \mathit{depth} \\ \mathsf{BVar}(\mathsf{Nat}(\mathit{idx})) & \text{if } \mathit{idx} < \mathit{depth} \\ \mathsf{subst}(g, \mathit{arg}, \sigma_{\pm}(\mathit{depth})) & \text{if } \mathit{idx} = \mathit{depth} \end{cases}
\end{aligned}$$

By $\sigma_{\pm}$ we denote the function for shifting indices of loose bound variables defined in Section 4.3.3. That is, $\mathsf{eta}(g, c)$ adds an e-class to g which is equal to c except that all loose bound variables' indices are shifted down by one. $\mathsf{beta}(g, c, \mathit{arg})$ adds an e-class to g which is equal to c except that all variables corresponding to the top-level variable $\hat{0}$ are replaced by an appropriately shifted version of arg. The shifting of arg is itself achieved by using subst.[25] The dynamic rewrite for β-reduction in egg then proceeds as follows. Given the trigger pattern $\mathsf{app}\, (\lambda\, ?\mathsf{t}\, ?\mathsf{b})\, ?\mathsf{a}$, let the result of e-matching be a substitution containing $?\mathsf{b} \mapsto b$ and $?\mathsf{a} \mapsto a$ with e being the e-class corresponding to the entire trigger pattern. Then, obtain the shifted e-class $s := \mathsf{beta}(g, b, a)$ and finally merge s and e in g. The dynamic rewrite for η-reduction is very similar. Given the trigger pattern $\lambda\, ?\mathsf{t}\, (\mathsf{app}\, ?\mathsf{f}\, (\mathsf{bvar}\, 0))$, let the result of e-matching be a substitution containing $?\mathsf{f} \mapsto f$ with e being the e-class corresponding to the entire trigger pattern. Then, obtain the shifted e-class $s := \mathsf{eta}(g, f)$ and finally merge s and e in g. Notably, to match the semantics of η-reduction we need to restrict this procedure to only be performed when f does not refer to variable $\hat{0}$. That is, we check whether $0 \in d_{var}(f)$, and if so, abort the rewrite. A downside of this approach is that expressions which would be eligible for η-reduction are blocked from reduction if they are equivalent to an expression referring to $\hat{0}$.

4.4.5 Remaining Rules

We have covered approaches for handling the most important, but not all, definitional equality rules. In this section, we briefly cover the remaining rules and consider how they might be implemented, or why they should not be.

Universe Levels We have largely omitted universe levels in this thesis, as they tend not to cause problems in our tactic. For those cases where they do, we have taken the approach of subsequently implementing definitional rules which solve the issues at hand. Currently, these rules are the following:

[25] This last case in σ_{β} returns an e-class instead of just a single expression. This does not match our definition of a substitution, but subst can easily be adapted to handle this, too.

$$
\begin{aligned}
max(\ell_1, \ell_2) &\equiv max(\ell_2, \ell_1) \\
max(S\,\ell_1, S\,\ell_2) &\equiv S\,max(\ell_1, \ell_2) \\
imax(\ell, 0) &\equiv 0 \\
imax(\ell_1, S\,\ell_2) &\equiv max(\ell_1, S\,\ell_2)
\end{aligned}
$$

We choose this incomplete approach as full definitional equality of universe levels is based on an inequality relation defined by 15 rules [Car19]. It is not immediately clear to us what a complete corresponding set of equalities should look like.

ι-Reduction ι-reduction is the definitional equality rule which encodes the semantics of applying an inductive type's recursor. For example, ι-reduction for Nat is defined by:

$$
\begin{aligned}
\mathsf{Nat.rec}\ C\ a\ f\ \mathsf{Nat.zero} &\equiv a \\
\mathsf{Nat.rec}\ C\ a\ f\ (\mathsf{Nat.succ}\ n) &\equiv f\ n\ (\mathsf{Nat.rec}\ C\ a\ f\ n)
\end{aligned}
$$

Adding such rules as rewrites might help with rewriting on match-expressions, which ultimately reduce to recursor applications. A problem with this approach is that match-expressions tend not to elaborate directly to recursor applications, but instead to generated intermediate definitions. Thus, we would also need to add unfolding rewrites for all intermediate definitions. Additionally, adding these rules for every inductive type appearing in the proof goal or an equation may explode the number of rewrites. Notably, the simp tactic supports ι-reduction which warrants further investigation into its approach in the future.

δ-Reduction δ-reduction states that definitions (or *global constants*) are definitionally equal to their bodies. That is, if we define $f : \mathbb{N} \to \mathbb{N} := \lambda\, n, n \cdot n$, then we get $f \equiv \lambda n : \mathbb{N}, n \cdot n$. Implementing full δ-reduction in egg is neither easy nor desirable. The most straightforward approach would be to add a δ-reduction rewrite for every global constant which may possibly appear during equality saturation. These constants can be collected by traversing the proof goal and every given equation. However, once we add the δ-reduction rewrites to our rewrite set, we then also need to repeat the process on those newly added rewrites. This process would reach a fixed point, as there can only exist finitely many global constants. The major problem with this approach is that it would likely blow up the set of rewrites. And even if the set of rewrites remains manageable, it is generally not desirable to just unfold all definitions during equality saturation, as this quickly blows up the e-graph. Instead, we generally assume that if a definition requires unfolding, then an equation will be explicitly provided for this by the user. While this assumption holds naturally for most definitions, there are certain classes of definitions for which unfolding is expected. Namely, those relating to type classes. Thus, we treat type classes specially, as covered in Section 4.5.

4.5 Type Classes

Type classes are a concept which is at odds with the syntactic nature of e-graphs, as they create syntactic uniformity for context-dependent semantics. For example, recall the Add type class from Section 3:

```
class Add (α : Type) where
  add : α → α → α

instance : Add Nat    where add := Nat.add
instance : Add String where add := String.append

#eval Add.add 40 2       -- 42
#eval Add.add "Le" "an" -- "Lean"
```

The syntactically equal calls to Add.add have different semantics depending on the types of the provided arguments. In the first case Add.add, reduces to Nat.add, while in the second case it reduces to String.append. This polymorphism is achieved by means of type class synthesis. That is, the type of Add.add is $\forall\,(\alpha : \mathsf{Type})\,[\mathsf{inst} : \mathsf{Add}\,\alpha],\ \alpha \to \alpha \to \alpha$ where $[\mathsf{inst} : \mathsf{Add}\,\alpha]$ is an argument which is automatically synthesized by Lean, and applying Add.add ends up applying inst.add. However, this polymorphism does not immediately pose a problem when it comes to syntactic rewriting. While both calls to Add.add look the same on the surface, they actually elaborate to syntactically distinct expressions which reference the synthesized type class instance explicitly. For example, when Add.add is applied to natural numbers it elaborates to:

```
(app (const `Add.add [Level.zero]) (const `Nat []) (const `instAddNat [])
```

Here, the synthesized instance instAddNat appears explicitly. Thus, even in an e-graph, there is no ambiguity about occurrences of Add.add. Problems only arise once we consider the underlying semantics of type class instances. For example, the expression Add.add 40 2 cannot be rewritten in an e-graph using the equation $\forall(x\,y : \mathsf{Nat}),\ \mathsf{Nat.add}\,x\,y = \mathsf{Nat.add}\,y\,x$, as Nat.add does not correspond to Add.add syntactically. As a user, this is rather unintuitive as we tend to treat $\mathsf{Add.add}\,x\,y$ and $\mathsf{Nat.add}\,x\,y$ as "the same" – and rightfully so, as they are definitionally equal by δ-, β- and η-reduction. Thus, when it comes to type classes, there exists an expectation that Lean and its proof tactics transparently handle syntactic differences. To overcome these differences in syntax, we automatically generate equations for converting between different representations involving type classes.

4.5.1 Projection Reduction

To seamlessly convert between uses of type classes and their underlying semantics, we specifically need to convert between applications of *type class projections* (often shortened to "projection" below) and their underlying definition contained in type class instances. The notion of a type class projection is the same as a structure projection. This is because type classes are just structure types with an associated tag.[26] For example, the Add.add function from above is a type class projection. The type signatures of projections have a specific layout. Given a type class T with type parameters $\tau_1, ..., \tau_n$ and a field p of T with type $\alpha_1 \to ... \to \alpha_n$, the projection $T.p$ has type $\forall(\tau_1)...(\tau_n)[T\,\tau_1...\tau_n], \alpha_1 \to ... \to \alpha_n$. Thus, an application of a projection $T.p$ can only sensibly be reduced (unfolded), if it contains at least $n+1$ arguments. This is because the $n+1$st argument is the type class instance argument, which determines the semantics underlying the application of the projection. For example, in the case of Add.add on natural numbers, it is the type class instance argument which tells us that the underlying function is Nat.add. Thus, the equation which captures this reduction of Add.add is Add.add Nat = Nat.add. We make use of the structured type signature of projections in the following procedure for generating equations for projection reductions. In the egg tactic, we run this procedure for all expressions appearing in the proof goal and given equations.

Equation Generation The basic idea for generating projection reductions is to check a given expression e for reducible sub-expressions of the form $f\,a_1\,...\,a_n$, where f is a projection. For this, we traverse e in depth-first order while tracking a list of arguments *args* when traversing function applications. That is, if e is an application $f\,a$, we traverse f with $\mathit{args} := a :: \mathit{args}$. In contrast, if we traverse a function argument, or the domain or body of a binder, we reset $\mathit{args} := [\,]$. Thus, when we reach the head f of a function application $f\,a_1\,...\,a_n$, we have exactly $\mathit{args} = [a_1, ..., a_n]$. If the head expression f is a global constant (an Expr.const), we check whether it is a type class projection. If it is not, we continue the depth-first search. If f *is* a type class projection, we obtain the number t of type parameters of the associated type class. If $n \geq t+1$, we know that the application $f\,a_1\,...\,a_n$ can sensibly be reduced as a_{t+1} must

[26] This is not actually true in Lean at the time of writing. Yet, very few type classes are not structures, and there are proposals to disallow non-structure type classes.

be the type class instance argument for the projection f. We therefore compute a reduction r of $f\, a_1 \ldots a_{t+1}$ which unfolds the field corresponding to projection f in a_{t+1} by means of Lean's meta-programming API. We only consider the first $t+1$ arguments, as this yields a more general reduction equation than including the arguments $a_{t+2}, ..., a_n$. Note that we said that we compute *a* reduction instead of *the* reduction, as it is sometimes possible to compute multiple reductions $r_1, ..., r_k$. This is possible when a reduction itself yields an application of a projection. An example of this is Lean's heterogeneous addition type class projection HAdd.hAdd, which may first reduce to an application of Add.add. Thus, given the expression $f\, a_1 \ldots a_n$ we generate a sequence of reduction equations $f\, a_1 \ldots a_{t+1} = r_1$, ..., $r_{k-1} = r_k$. Notably, the proof for each of these equations is simply reflexivity of equality, as the involved expressions are all definitionally equal. We also note that the described procedure is a slight simplification of the actual implementation, which requires special handling of bound variables and metavariables.

4.5.2 Specialization

Reductions for type class projections can be generated only when then type class instance arguments are baked into the expressions. That is, for expressions like Add.add 40 2, where type class synthesis constructs an instance of type Add Nat during elaboration. Yet, there exist many theorems where the type class instance cannot be synthesized during elaboration, as the theorem is itself parameterized over the type class instance. Common examples of such theorems come from algebra, as algebraic structures are nicely formalized as type classes. For example, we can define a semigroup as:

```
class Semigroup (G : Type) extends Mul G where
  mul_assoc : ∀ a b c : G, (a * b) * c = a * (b * c)
```

That is, a semigroup structure for a given carrier type G has a binary operation $*$ which is associative. The multiplication operator mul is inherited by extending the Mul type class, which also defines the corresponding $*$ notation. The full type of mul_assoc looks as follows:

```
∀ (G : Type) [self : Semigroup G] (a b c : G) :
  mul G (toMul G self) (mul G (toMul G self) a b) c =
  mul G (toMul G self) a (mul G (toMul G self) b c)
```

The function toMul : (G : Type) → Semigroup G → Mul G casts the type class instance self of type Semigroup G to an instance of type Mul G, such that it can be used with mul. The theorem mul_assoc can thus be applied for any type for which a semigroup structure is provided. However, the syntactic structure of the theorem means that, in an e-graph, it only applies to expressions where multiplication has the syntactic form mul _ (toMul _ _). This is generally not the case for expressions involving concrete semigroups. For example, if we assume Nat to have a semigroup instance, then for a, b, c of type Nat we cannot prove $(a*b)*c = a*(b*c)$ by mul_assoc in egg. The problem is that the goal equality does not have a form which matches mul_assoc, as the multiplication type class instance instMulNat for Nat does not have the form toMul _ _ required by mul_assoc:

```
mul Nat instMulNat (mul Nat instMulNat a b) c =
mul Nat instMulNat a (mul Nat instMulNat b c)
```

This incompatibility is one of the major hurdles we face when using type classes with syntactic rewrites. Namely, even with automatic generation of type class projection reductions, we cannot know that we should generate a reduction like toMul Nat instSemigroupNat = instMulNat, as is not clear that the semigroup carrier type G will be matched against Nat until equality saturation has already started. In an attempt to generate such reductions *before* equality saturation has started, we use heuristic approaches. We assume that rewrites will generally be applied to expressions whose types we are already aware of beforehand. Using this assumption, we can generate *type class specializations* of equations. That is, versions of equations where otherwise generic type class instance arguments are assigned concrete instances. These specializations come in two forms for achieving different goals.

Goal Type Specialization Goal type specialization aims to solve the problem established in the previous paragraph. For this, we choose the heuristic approach of assuming that rewrites will generally be applied to expressions whose type matches the type of the proof goal. Thus, given an equation e which is parameterized over type class instances, we try to generate a corresponding equation $e_{\vdash}$ which is applicable to terms of the proof goal's type T. Let $lhs = rhs$ be the statement of e with all quantified variables instantiated as metavariables. Then, we first generate the type specialized equation e_T by unifying both $type(lhs)$ and $type(rhs)$ with T, where we denote by $type$ a function which maps an expression to its type. Given the theorem mul_assoc from above, and assuming the goal type to be Nat, this step results in a theorem of the following form with the type of ?self being Semigroup Nat:

```
mul Nat (toMul Nat ?self) (mul Nat (toMul Nat self) ?a b) ?c =
mul Nat (toMul Nat ?self) ?a (mul Nat (toMul Nat self) ?b ?c)
```

Next, to generate the specialized equation $e_{\vdash}$, we try to replace all type class metavariables in e_T with concrete instances. For efficiency, we first check the local context for an appropriate instance, but otherwise resort to type class synthesis. Performing this step turns the previous example into a theorem of the form:

```
mul Nat (toMul Nat instSemigroupNat) (mul Nat (toMul Nat instSemigroupNat) ?a ?b) ?c =
mul Nat (toMul Nat instSemigroupNat) ?a (mul Nat (toMul Nat instSemigroupNat) ?b ?c)
```

Thus, we obtain a theorem which is no longer parameterized over the type class instance argument ?self, as this argument could be synthesized under the assumption that the result type is Nat. Note that this theorem is still not syntactically applicable to a proof goal of the form $(a*b)*c = a*(b*c)$ for a, b, c of type Nat, as multiplication is still expressed as mul _ (toMul _ _) instead of instMulNat. However, this type class specialized equation is now amenable to projection reduction, which generates the missing equation toMul Nat instSemigroupNat = instMulNat. Thus, the theorem over natural numbers can be proven in egg by first rewriting all occurrences of instMulNat to toMul Nat instSemigroupNat, and subsequently applying either the equation mul_assoc, or our type class specialization thereof.

As the previous example shows, type class specialization and projection reduction work in conjunction. The specialization enabled a projection reduction which was necessary to apply the specialized theorem. Thus, in our proof tactic, we run specialization and projection reduction in a loop until a fixed point is reached. That is, the equations generated by specialization are passed to the procedure for projection reduction, and vice versa. A finite fixed point must always be reached as both procedures produce "smaller equations" according to suitably chosen metrics. Namely, specialization always reduces the number of type class metavariables, while projection reduction decreases the "depth" of the referenced type classes, where we consider type class T_1 to be deeper than type class T_2, if T_2 was defined before T_1.[27]

Direction Specialization Direction specialization is another form of type class specialization. It enables equations to be applied in directions in which they did not originally apply. Recall that a given equation $lhs = rhs$, where quantified variables are instantiated as metavariables, is only applicable in those directions where the metavariables satisfy the $\supseteq$ relation. For example, $?n \cdot 0 = 0$ is only applicable in the forward direction, as the metavariables in $?n \cdot 0$ are a superset of the metavariables in 0, but not vice versa. For equations involving type class instances, the set of applicable directions is sometimes smaller than expected. Take, for example, the following axiom of groups: $x \cdot 1 = x$. Intuitively, this equation should be applicable in both directions, as the only quantified variable x appears on both sides. Yet, in a standard formalization of groups as a type class, the equation has the following type and is thus only applicable in the forward direction:

```
∀ (G : Type) [self : Group G] (x : G), mul G (toMul G self) x 1 = x
```

[27] Thus, T_2 cannot possibly reduce to anything referencing T_1.

The principal reason for this theorem only being applicable in the forward direction is that the left side contains a multiplication which relies on a type class instance $\mathsf{self} : \mathsf{Group}\,\mathsf{G}$, which does not appear on the right side. This is problematic, as a rewrite from x to $x \cdot 1$ should be possible, and is a well-known "trick" in algebraic reasoning. It is thus important to extend the applicable directions of the equation to include the backward direction. For this, we use the following procedure.

Given an equation e which is parameterized over type class instances, we try to generate corresponding equations $e_{\Rightarrow}$ and $e_{\Leftarrow}$ which are applicable in the forward and backward directions respectively. In the following, we consider only the procedure for generating $e_{\Rightarrow}$ with the reverse direction being analogous. Let $lhs = rhs$ be the statement of e with all quantified variables instantiated by metavariables. Then, we compute the set $B_{\Rightarrow} := tcvars(rhs) \setminus tcvars(lhs)$ of type class metavariables which are blocking e from being applicable in the forward direction, where *tcvars* maps an expression to its contained non-ambient metavariables whose type is a type class. If $B_{\Rightarrow} = \emptyset$, then either e is already applicable in the forward direction, or the reason for this direction being blocked does not relate to type class instances. In that case, we do not generate a specialization for this direction. Otherwise, if $B_{\Rightarrow} \neq \emptyset$, we generate a specialization rhs_1 of rhs by trying to assign all metavariables in $B_{\Rightarrow}$. For this, we first consider instances in the local context, and if none apply, try assignment by type class synthesis. This step may entail the assignment of metavariables which are not type classes instances. For example, if we assign an instance of type $\mathsf{Group}\,\mathsf{Nat}$ to a metavariable $?\mathsf{g}$ of type $\mathsf{Group}\,?\mathsf{G}$, then both $?\mathsf{g}$ and $?\mathsf{G}$ will be assigned. The assignment of metavariables which are not type class instances can in turn enable synthesis of other type class metavariables. Thus, we generate specializations $rhs_1, ..., rhs_n$ until we reach a fixed point. A fixed point must be reached, as the number of assigned metavariables decreases on every iteration. From rhs_n, we construct the specialized equation $e_{\Rightarrow} := lhs = rhs_n$. This equation holds definitionally, and is thus proven by reflexivity of equality. Finally, we check whether the constructed $e_{\Rightarrow}$ is in fact applicable in the forward direction, and if not, discard it.

Applying direction specialization to the equation $x \cdot 1 = x$ from above can yield different results depending on context. If the equation is to be applied in a proof over general groups, then the local context must contain a type G and an instance $\mathsf{inst} : \mathsf{Group}\,\mathsf{G}$. In that case, the specialization yields a theorem of the form $\forall\, x : \mathsf{G},\, \mathsf{mul}\,\mathsf{G}\,(\mathsf{toMul}\,\mathsf{G}\,\mathsf{inst})\,x\,1 = x$, which can be applied in both directions. If, instead, the equation is to be applied in a proof for a concrete group like $\mathbb{Q}$, then direction specialization fails, as the local context need not necessarily contain an instance of type $\mathsf{Group}\,\mathbb{Q}$. In this case, goal type specialization is necessary, and yields an equation which is applicable in both directions.

4.6 Proof Reconstruction

Up to this point, this thesis has only considered preprocessing of expressions and equations in Lean, as well as techniques for improving the applicability of rewrites in egg. In this section, we take a look at what happens after equality saturation completes. That is, how the result of equality saturation is communicated back to Lean, such that Lean accepts the result as a proof. This process, called *proof reconstruction*, involves generating an explanation in egg, and turning that explanation into a proof term in Lean. We motivate our current approach to proof reconstruction by showing which approaches did *not* work along the way.

4.6.1 Explanation Construction

When equality saturation completes, the immediate result is simply an e-graph – no more, no less. However, from this e-graph we can extract various information depending on our use case. In the context of our proof tactic, the goal of equality saturation is to prove the equivalence of two given expressions *lhs* and *rhs*. Thus, when equality saturation completes, we check whether the e-class ids for *lhs* and *rhs* are equal, which implies the desired equivalence. If this check is successful, we ask egg to generate an explanation for the equivalence. As covered in Section 2.5, explanations from congruence closure are generally DAGs. Yet, they can be

flattened into a sequence of sequential rewrite steps. An example of a flat explanation proving $x = x + (0 + 0)$ is shown in Figure 12.

```
(fvar 42)
(Rewrite ⇐ add0 (app (app (const Nat.add) (fvar 42)) (const Nat.zero)))
(app (app (const Nat.add) (fvar 42)) (Rewrite ⇐ add0 (app (app (const Nat.add) (const Nat.zero)) (const Nat.zero))))
```

Figure 12: Flat explanation proving $x = x + (0 + 0)$.

The first line corresponds to the left-hand side (x) of the equality to be proven. Every subsequent line consists of a term containing a Rewrite with the name of the rewrite rule (add0), the direction in which it was applied ($\Leftarrow$), and the position at which it was applied within the resulting term. The final term therefore represents the right-hand side ($x + (0 + 0)$) of the equality. We pass a string representation of such an explanation back to Lean where we construct a native explanation representation from it. This native representation is also a sequence of expressions, but now of Lean's Expr type. Parsing Exprs from our egg expression representation is almost immediate, with two caveats. First, free variables and metavariables need to be reconstructed from their integer index, which is trivial. Second, and more importantly, erased proofs are constructed as metavariables whose type is the proposition contained in the erased proof expression. That is, the erased proof term still needs to be specified, which will occur in later steps. Each expression e in the native explanation representation has the following data associated with it: the uniquely identifying name of the rewrite r which leads to e, the direction in which r was applied, and the position within e at which r was applied. The position is represented by Lean's SubExpr.Pos type, which locates a position within an expression as a sequence of "coordinates" describing which child of a sub-expression to visit, starting at the root expression. Sub-expression positions do not translate to universe levels, which is why we do not record a position for rewrites on them. Luckily, any rewrite on universe levels must necessarily be one of our definitional equality rules over universe levels (Section 4.4.5) which do not require a rewrite position during proof generation.

One piece of information which is notably absent in our explanation representation is the assignment of quantified variables in rewrites. That is, if we have a rewrite from e_1 to e_2 using the equation $\forall(x\ y : \mathbb{N}), x + y = y + x$, then the explanation does not explicitly state what x and y are supposed to be. Instead, the only indication of the required variable assignment are e_1 and e_2. For example, if e_2 is $2 + 3$, then the quantified variables of the previous rewrite can be inferred as $x := 2$ and $y := 3$. Thus, one important task during proof generation will be to determine this assignment. This is a notable difference compared to the tactic prototype of [KGB+24], where the assignment is computed in Rust using a different approach.

4.6.2 Proof Generation

In Lean, proving a theorem means finding a term whose type is the proposition to be proven. That is, if we want to prove a goal of the form $lhs = rhs$, we need to find an expression e of type $lhs = rhs$. We use the explanation steps obtained in the previous section to build such a proof expression step by step. We start with the most straightforward approach, and refine it subsequently as we determine insufficiencies.

Recursor-Based Generation Let e_1, e_2 and e_3 be the expressions at three consecutive steps in an explanation, with r_1, r_2, p_1 and p_2 being the respective equations and positions used to rewrite from one step to the next. As a first attempt at proof generation for $e_1 = e_3$, we follow the procedure sketched in Figure 13.

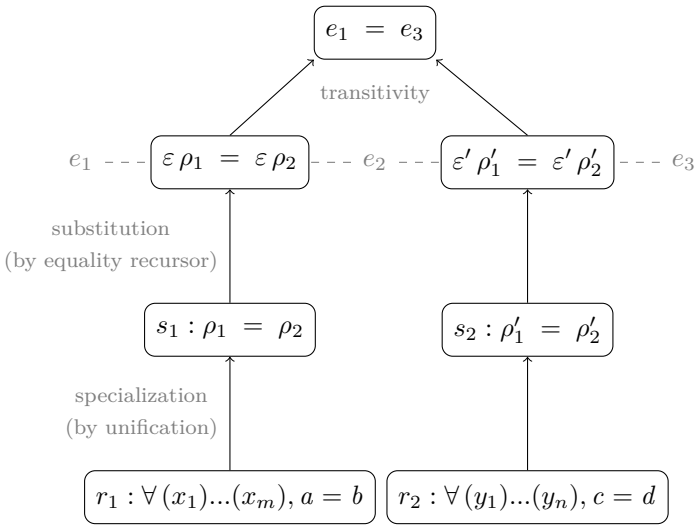


Figure 13: Recursor-based proof generation for two steps of an explanation.

We start by proving $e_1 = e_2$. We know that e_1 and e_2 must differ only at position p_1. That is, exactly where the explanation claims the rewrite r_1 to have occurred. Thus, if ε_1 and ε_2 are functions which take an expression and place it at position p_1 in e_1 and e_2 respectively, then $\varepsilon_1 = \varepsilon_2$. We therefore refer to this function simply as ε. Then, if ρ_1 is the sub-expression of e_1 at position p_1, and ρ_2 is the sub-expression of e_2 at position p_1, we get $e_1 = \varepsilon\,\rho_1$ and $e_2 = \varepsilon\,\rho_2$. To prove the equality of $\varepsilon\,\rho_1$ and $\varepsilon\,\rho_2$, we first prove $\rho_1 = \rho_2$. This should be simple as the explanation tells us that this must precisely be the result of equation $r_1 : \forall\,(x_1)...(x_m), a = b$. The only problem is that we do not know how r_1 should be specialized (how its quantified variables should be assigned) in order to obtain the required proof term. Luckily, the required specialization is completely determined by ρ_1 and ρ_2. By instantiating r_1's quantified variables with metavariables and then unifying ρ_1 with a and ρ_2 with b, we obtain a specialization s_1 of r_1 which proves $\rho_1 = \rho_2$.[28] Based on the specialization s_1, we prove $\varepsilon\,\rho_1 = \varepsilon\,\rho_2$ by using the substitution principle of equality. That is, its recursor:

$$\mathsf{Eq.ndrec} : \ \forall(\alpha : \mathcal{U}_u)(x : \alpha)(C : \alpha \to \mathcal{U}_u), (C\,x) \to \forall(y : \alpha)(h : x = y), C\,y$$

The crux in applying the recursor is constructing the correct motive C. As the result $C\,y$ should have the form $\varepsilon\,\rho_1 = \varepsilon\,y$, we let $C := \lambda y, \varepsilon\,\rho_1 = \varepsilon\,y$. Thus, we get a proof of $\varepsilon\,\rho_1 = \varepsilon\,\rho_2$ by $\mathsf{Eq.ndrec}\ \_\ \rho_1\ C\ (\mathsf{Eq.refl}\,(\varepsilon\,\rho_1))\ \rho_2\ s_1$. The same approach can then be used to prove $e_2 = e_3$ based on a different ε' for the position p_2, and the equation r_2. Finally, to prove $e_1 = e_3$ we combine the previous proofs by applying transitivity of equality.

Two cases of this procedure need to be considered separately. First, we noted in the previous section that rewrites on universe levels do not have a corresponding sub-expression position, but do not require one as the rewrite can only be a result of definitional equality. This only works, because we treat rewrites by definitional equality differently in our proof generation. Namely, given explanation expressions e_1 and e_2, where e_2 follows from some definitional equality rewrite on e_1, we construct a proof of $e_1 = e_2$ directly by unifying e_1 and e_2 and constructing the proof term $\mathsf{Eq.refl}\,e_1$.[29] This works, because definitional equality is a congruence with respect to all expression constructors. That is, if $\rho_1 \equiv \rho_2$, then necessarily $\varepsilon\,\rho_1 \equiv \varepsilon\,\rho_2$. In the future, we hope to skip these proofs by reflexivity altogether, to reduce proof size. Yet, we are currently unsure whether this may cause problems due to a lack of transitivity in Lean's algorithmic definitional equality.

[28] Thanks to Mario Carneiro for this approach.
[29] We could have just as well chosen $\mathsf{Eq.refl}\,e_2$.

The second case that needs to be considered separately is that of erased proofs terms. These terms are represented by metavariables where the *type* is known. In order to reconstruct the actual proof *term*, we need to resort to information contained in the proof goal and equations. If the left-hand side *lhs* of the proof goal contains a proof term, then the first explanation expression e_1 must contain a corresponding metavariable for the erased proof. To assign this metavariable, we begin the proof generation procedure by unifying *lhs* with e_1. Thus, e_1 no longer contains proof erasure metavariables. This property is preserved under definitional equality rewrites from e_i to e_{i+1}, as we unify e_i and e_{i+1} for such rewrites. For non-definitional rewrites, proof erasure actually breaks the property of e_i and e_{i+1} being representable as $\varepsilon\,\rho_i$ and $\varepsilon\,\rho_{i+1}$, as they may require different εs. This is because e_i's ε may contain proof terms for which e_{i+1}'s ε only contains a metavariable. Thus, to propagate proof terms from e_i to e_{i+1}, we simply choose to represent e_{i+1} using e_i's ε in the motive of the equality recursor. For proof terms contained in ρ_i, instead of ε, a similar logic applies. The main difference is that the proof term does not originate from the proof goal, but instead from the equation used to rewrite ρ_i.

Binder-Compatible Generation A major caveat of recursor-based proof generation is that it cannot handle rewriting under binders. The problem is that if e_2 is constructed from e_1 by rewriting with equation r under a binder, then it is not possible to assign the quantified variables in r by unification with the relevant sub-expressions in e_1 and e_2. Namely, let $e_1 := \varepsilon\,\rho_1$ and $e_2 := \varepsilon\,\rho_2$, where ρ_1 and ρ_2 appear under a binder. Then the bound variables in ρ_1 are not the same bound variables as those in ρ_2, as they originate from different binders. This is reflected in the fact that Lean's locally nameless representation represents them with different free variables. Thus, even if ρ_1 and ρ_2 were syntactically identical, they would not unify, as the free variables are distinct. To solve this problem, we need a way to refer to "the" bound variable of two different expressions. This is possible using the theorem of *function extensionality*:[30]

$$\mathsf{funext} : \forall(\alpha : \mathcal{U}_u)(\beta : \alpha \to \mathcal{U}_v)(f\ g : \forall x : \alpha, \beta\,x), (\forall x, f\,x = g\,x) \to f = g$$

It states that any two functions f and g which agree on all inputs are equal. We use this to fix our proof generation procedure as follows. Let e_1 and e_2 be as above with p_1 being the position at which rewrite r_1 was applied. Then we prove $e_1 = e_2$ step by step while simultaneously traversing e_1 and e_2 towards position p_1. We denote by s_1 and s_2 the head symbols of sub-expressions of e_1 and e_2 encountered during the traversal. If ever $s_1 \neq s_2$, then an error has occurred, and we abort the proof reconstruction.[31] When $s_1 = s_2 = \mathsf{Expr.app}$ applied to functions f_1 and f_2 and arguments a_1 and a_2, we apply the theorem of congruence congr and are thus left to prove $f_1 = f_2$ and $a_1 = a_2$. We prove the equality of those sub-expressions which do *not* need to be traversed to reach position p_1 by reflexivity of equality, as they must be syntactically equal by our assumption that e_1 and e_2 differ only at position p_1. When $s_1 = s_2 = \mathsf{Expr.lam}$ with bound variables x_1 and x_2, and bodies b_1 and b_2, we apply function extensionality and are thus left to prove $\forall y, (\lambda x_1, b_1)\,y = (\lambda x_2, b_2)\,y$. By introducing y and β-reducing both λ-expressions, the proof goal becomes $b_1[x_1 \mapsto y] = b_2[x_2 \mapsto y]$. Thus, both bodies of the λ-expressions refer to the same free variable y. When $s_1 = s_2 = \mathsf{Expr.forallE}$, we use the same approach as for $\mathsf{Expr.lam}$, but use the theorem $\mathsf{forall\_congr}$ which lifts function extensionality to the type level. We repeat this traversal until we reach position p_1, at which point we proceed to specialize the rewrite r_1 by unification as in the recursor-based approach. As any bound variables appearing in e_1 and e_2 now correspond to the same free variables, this unification also succeeds for rewrites under binders. A downside of this approach is that the proof terms can be much larger than those generated by the recursor-based proof generation, as the size of the proof term now corresponds to the size of the expressions e_1 and e_2.

[30] While this is a theorem (instead of an axiom) in Lean, it does not follow from the definition of equality, but is instead proven from an axiom for the existence of quotients.

[31] Special care needs to be taken when encountering proof erasure metavariables, which are resolved by intermittent unification.

Mathlib-based Generation The attentive reader will have noticed that the use of congruence theorems in the binder-compatible proof generation brings with it an important caveat: we cannot rewrite in the arguments to dependent function types. Recall from Section 3.4 that for regular (homogeneous) equality, a congruence theorem for $f\,a = g\,b$ is only possible when f and g are non-dependent, or a and b are definitionally equal. A common example where neither is the case is when using the ite function which underlies if-then-else expressions:

$$\mathsf{ite} : \forall(\alpha : \mathcal{U}_u)(c : \mathcal{U}_0)[h : \mathsf{Decidable}\ c], \alpha \to \alpha \to \alpha$$

An expression like if (IsEven n) then n else 0 is syntactic sugar for ite Nat (IsEven n) n 0. Note that the conditional expression c is not a boolean, but a proposition. In order to have a computational interpretation of ite, this proposition needs to be decidable. This is ensured by requiring the type class instance h of type Decidable c. However, as a result, the type of the argument h depends on the conditional proposition c. Thus, if we try to generate a proof for a rewrite from if (IsEven 2) then 2 else 0 to the equivalent if True then 2 else 0, we will try to apply congruence for a dependent function and fail. At the time of writing, we do not have a full understanding of how to solve this problem. Thus, we resort to engineering our way to an acceptable solution. That is, we copy about 500 lines of code from the implementation of *congruence quotations* in the mathlib library:

> "While in simple cases it might be possible to use [regular congruence theorems], congruence quotations are more general, since for example [a given function] could have implicit arguments, complicated dependent types, and subsingleton instance arguments such as Decidable [...]."[32]

We spare the reader from a description of how we use congruence quotations for our proof generation, as this is purely an implementation detail. However, we note that the overarching proof generation procedure still follows that of Figure 13, and we only swap the use of the equality recursor for the congruence quotation mechanism. While congruence quotations expand the capabilities of our proof generation procedure, they are not a silver bullet. For example, our tactic can still not generate a proof for the rewrite from if (IsEven 2) then 2 else 0 to if True then 2 else 0. If we attempt the same rewrite using the rw tactic, it also fails stating that the "motive is not type correct". That is, it tried to rewrite only IsEven 2 to True, which yields an if-then-else expression which is not well-typed as the instance h still has type Decidable (IsEven 2). Interestingly, if we attempt the same rewrite with simp, it succeeds. It does so by using the special congruence theorem ite_congr:

$$\begin{aligned}\mathsf{ite\_congr} : \forall(x\,y\,u\,v : \alpha)(s : \mathsf{Decidable}\ b)[\mathsf{Decidable}\ c],\\ b = c \to (c \to x = u) \to (\neg c \to y = v) \to \mathsf{ite}\,b\,x\,y = \mathsf{ite}\,c\,u\,v\end{aligned}$$

This congruence theorem includes the crucial replacement of the instance of type Decidable b with the correct instance of type Decidable c, which is obtained by type class synthesis. There exist a variety of specialized congruence theorems like this, which are tagged with an @[congr] tag. The simp tactic has a mechanism for using these theorems when need be. Thus, in the future, we hope to use more of the existing infrastructure of simp for our proof generation.

4.7 Conditional Rewriting

In previous sections, we introduced various techniques for syntactic rewriting of expression using equational theorems. Some of these techniques aimed at improving the soundness of applying rewrites (as in Section 4.3). Others, namely the extensions for handling definitional equality and type classes, improved the completeness of our proof tactic. In this section, we introduce another extension which allows our tactic to use *conditional* equations for rewriting. While conditional rewriting is also supported by the simp tactic, our approach to conditional rewriting in e-graphs is new to the best of our knowledge.

[32] https://leanprover-community.github.io/mathlib4_docs/Mathlib/Tactic/TermCongr.html#congr-congruence-quotations

4.7.1 Conditional Equations

Intuitively, a conditional equation is an equation which depends on preconditions. As such, it should only be applied when the preconditions are satisfied. For example, a theorem like $\forall n, n \neq 0 \to \frac{n}{n} = 1$ is a conditional equation where the body $\frac{n}{n} = 1$ should only be applied if the condition $n \neq 0$ is satisfied. However, it is not easy to pinpoint what exactly constitutes a precondition. In [BN98], a *conditional identity* is a formula $s_1 = t_1 \wedge ... \wedge s_n = t_n \to s = t$. This matches the definition used for conditional rewrites in [Bou23]. For our purposes, this is too strict of a definition as it requires preconditions to be equalities. Instead, we want conditional equations to be of the form $\forall (x_1)...(x_n), lhs = rhs$, where $x_1, ..., x_n$ are arbitrary preconditions and quantified variables in $lhs = rhs$. This makes it difficult to determine which x_i are preconditions and which are "basic" quantified variables. We use the following approach.

Let $X := \{x_1, ..., x_n\}$, and let $B := vars(lhs) \cup vars(rhs)$, where $vars$ returns the set of variables contained in an expression. Then, as a first approximation, we could define the set P of preconditions by $P := X \setminus B$. The reason being that any $x_i \in B$ will be assigned by e-matching during rewriting, and thus does not need to be satisfied before the rewrite is applied. Therefore, any $x_i \in P$ remains unassigned during e-matching and must be provided as a precondition to the rewrite. This definition of preconditions works for $\forall n, n \neq 0 \to \frac{n}{n} = 1$, which we shall write as $\forall (n : \mathbb{N})(h : n \neq 0), \frac{n}{n} = 1$. Here, we get $P := \{h\}$, as h does not appear in the equation's body, whereas n does. However, an example where this definition of preconditions fails is the equation $\forall (\alpha : \mathsf{Type})(l : \mathsf{List}\,\alpha), l = l$. Here, α is determined to be a precondition, as it does not appear in the body expression $l = l$. This is, of course, undesirable as α is completely determined by the type of l and can thus be inferred from the term assigned to l. Thus, in the egg tactic we also consider type information when determining the set of preconditions. Namely, we define $P := \{x \in X \setminus B \mid \nexists b \in B, x \in vars(type(b))\}$. Using this criterion, we do not consider α to be a precondition in the previous example, as l appears in the equation body and α appears in the type of l. Another, rather constructed, example where this approach correctly determines the set of preconditions is Fix.fix in the following:

```
class Fix (α : Type) where
  fix : ∀ (f : α → α) (a : α), f a = a
```

Here we define the Fix type class with requires any conforming type α to satisfy the property that every $\mathsf{a} : \alpha$ has to be a fixed point under any $\mathsf{f} : \alpha \to \alpha$. The type of Fix.fix is $\forall (\alpha : \mathsf{Type})[\mathsf{self} : \mathsf{Fix}\,\alpha](\mathsf{f} : \alpha \to \alpha)(\mathsf{a} : \alpha) : \mathsf{f\,a} = \mathsf{a}$. By our precondition criteria, we correctly determine that self is the only precondition. Notably, the type of self is not a proposition. That is, our precondition selection is not restricted to propositions.

While the given criteria for preconditions have worked well in practice so far, they are still fundamentally incomplete. The defining characteristic of a precondition p to an equation e should be that assigning the variables in e does not allow us to infer a value for p. Unfortunately, we do not currently know how to determine whether such a condition is satisfied. However, a potential improvement over our current procedure could be to compute the set of variables appearing in other variables' types up to a fixed point, instead of just for one step.

A more practical restriction that we currently need to impose on conditional equations is that we do not allow *unbound preconditions*. We call a precondition p of an equation e unbound, if e-matching on e does not yield a substitution which entirely resolves the type of p. That is, if $vars(type(p)) \setminus vars(e) \neq \emptyset$. An example of this is Fix.fix from above. Here, the type $\mathsf{Fix}\,\alpha$ of the precondition self refers to the variable α, yet α does not appear explicitly in the body of the equation. As a result, e-matching on the body does not yield an assignment for the variable α. Thus, we cannot resolve the type $\mathsf{Fix}\,\alpha$ *during* equality saturation, even though it is possible by type inference in Lean. However, as we will see in the following section, our conditional rewriting procedure needs to be able to resolve the types of preconditions during equality saturation, and can therefore not handle unbound preconditions. In contrast, a conditional equation like $\forall n, n \neq 0 \to \frac{n}{n} = 1$ does not have unbound preconditions, as e-matching on the body determines n, which determines the type of the precondition $n \neq 0$. Supporting unbound preconditions is not unreasonable, and we have ideas for how to achieve this in the future.

4.7.2 Rewriting Procedure

In this section, we give an overview of the steps required for rewriting with conditional equations. We start at the invocation of the `egg` tactic. When using a conditional equation like `h`, one can (and should) also provide terms which satisfy its preconditions ($\mathsf{x} \wedge \mathsf{y}$). We call such terms *facts* and provide them after the list of equations as follows:

```
example (f : x ∧ y) (h : x ∧ y → 1 = 2) : 1 = 2 := by
  egg [h; f]
```

We encode the types of these facts just as any other expression, and pass them to egg along with the rewrites and proof goal. In egg, we add each fact to the e-graph and thus obtain associated e-class ids $f_1, ..., f_k$. Thus, for the e-classes identified by $f_1, ..., f_k$ we can assume their contained terms to be inhabited (that is, proven, in the case of propositions). We make use of this when applying conditional equations as dynamic rewrites, as follows. Let e be a conditional equation with preconditions $p_1, ..., p_n$, and body $lhs = rhs$, which is applicable in the forward direction. Then, we let the trigger pattern for e's rewrite be lhs, as usual. When lhs is matched during e-matching with resulting substitution σ, we need to ensure that $p_1, ..., p_n$ are satisfied before we apply the rewrite. For this, we first need to assign the variables contained in $p_1, ..., p_n$ by applying the substitution σ. That is, we turn a parameterized precondition like $?n > 0$ into a specific one like $5 > 0$ (assuming $\sigma(?n) = 5$). Notably, applying the substitution to the preconditions does not yield concrete terms for each precondition, but only e-class ids $\varphi_1, ..., \varphi_n$. Thus, to check whether each condition p_i is satisfied, we check whether its corresponding e-class id φ_i is contained in the set of fact e-class ids $f_1, ..., f_k$.[33] If all conditions are satisfied, the rewrite can proceed as usual. Otherwise, the rewrite is not applied.

A useful side effect of this approach to conditional rewriting is that we add the facts to the e-graph, and can therefore also apply rewrites to them. That is, we can prove examples like the following:

```
example (f : p) (h₁ : p = q) (h₂ : q → (p = r)) : p = r := by
  egg [h₁, h₂; f]
```

Here, the only fact is `f` of type `p`. When `egg` gets called, `p` is added to the e-graph which yields some e-class id φ. However, the only rewrite which can solve the proof goal is h_2, which has a precondition of type `q`. Thus, if equality saturation tries to apply h_2 immediately, it does not succeed as the precondition is not satisfied, as `q` is not contained in a fact e-class. However, the rewrite h_1 can be applied to `p`, which adds `q` to the e-class φ. Thus, h_2 can subsequently be applied as its precondition `q` is now part of a fact e-class.

While we get this behavior for free in egg, it comes at a cost during proof reconstruction. Namely, explanations involving rewritten facts require constructing subproofs of equivalences between facts, as discussed in the following section.

4.7.3 Proof Reconstruction

When generating proofs for explanations containing rewrites of non-conditional equations, we determine the assignment of an equation's quantified variables by unification with the target (sub-)expressions. For conditional rewrites, this same approach works to determine the assignment of its quantified variables, yet not its preconditions. Instead, to obtain an assignment for preconditions, we have to record which fact was used for which precondition *during* equality saturation. As each rewrite has an associated name which can be dynamically assigned in dynamic rewrites, we record an assignment for preconditions by storing unique identifiers for the used facts in the names rewrites. For example, let an equation with name r have preconditions p_1 and p_2. Also, let v_1 and v_2 be unique identifiers for facts with corresponding e-class ids φ_1 and φ_2. If the rewrite for r successfully satisfies its preconditions p_1 and p_2 by their membership in φ_1 and φ_2, then we set the name of the rewrite in the resulting explanation

[33] More specifically, we need to check whether there exists an f_j whose *canonicalized* id matches the given φ_i.

to rv_1v_2. We can then use this information during proof reconstruction, as follows. Let the explanation produced by egg show that expressions e_1 and e_2 are equal by application of rv_1v_2, where the rewrite named r has the body $lhs = rhs$, and v_1 and v_2 are the identifiers of facts f_1 and f_2. We first assign the quantified variables of r as usual by unification of e_1 with lhs and e_2 with rhs. Thus, all variables of r except p_1 and p_2 are assigned or inferred. By the name rv_1v_2 of the rewrite step, we know that p_1 was satisfied by f_1 and p_2 was satisfied by f_2. Thus, to assign each precondition p_i, we try unification with the corresponding fact f_i. If unification succeeds, the precondition is successfully proven. If unification fails, this means that the precondition was proven in egg by some fact equivalent to f_i, but not f_i itself. That is, by some fact g_i which was obtained by rewriting on f_i. To obtain a proof of g_i, we need to establish a proof of $f_i = g_i$ and then use modus ponens with f_i. Luckily, we know that f_i and g_i must be in the same e-class, as f_i was recorded as the proof of p_i in the rewrite's name. Thus, we can obtain a proof of $f_i = g_i$ by querying the e-graph for an explanation of this equivalence, and subsequently calling proof reconstruction on the resulting explanation. As such, proof reconstruction with conditional equations is recursive.

4.8 Guidance

As an automated procedure, equality saturation has limitations with respect to the (sizes of) problems it can solve. In [KGB+24], these limitations are investigated for the use cases of program optimization and equational reasoning. Their results point to "a general characteristic of equality saturation: either a successful rewrite sequence is found relatively quickly, or, computational costs explode." That is, long sequences of rewrites tend to be infeasible as the size of the e-graph grows too quickly. This is problematic for two reasons. First, it makes equality saturation fragile in the sense that slight changes to the initial conditions can cause the procedure to fail its objective. For example, adding a single equation in a call to the `egg` tactic sometimes causes equality saturation to time out by a significant margin. Second, the rapid growth of an e-graph makes it very difficult to comprehend *why* a given run of equality saturation failed. Thus, when the procedure times out, it tends to be difficult to meaningfully debug the failure. Both of these problems can be addressed by reducing the length of runs of equality saturation. To achieve this, [KGB+24] introduces the notion of *guided equality saturation*. The basic idea of guided equality saturation is simple. Instead of trying to rewrite from term t_1 to t_n in a single run of equality saturation, we introduce intermediate goals $t_2, ..., t_{n-1}$ called *guides*. We then only perform equality saturation from each t_i to t_{i+1}, thus replacing a single long run of equality saturation with multiple short runs as shown in Figure 14.[34]

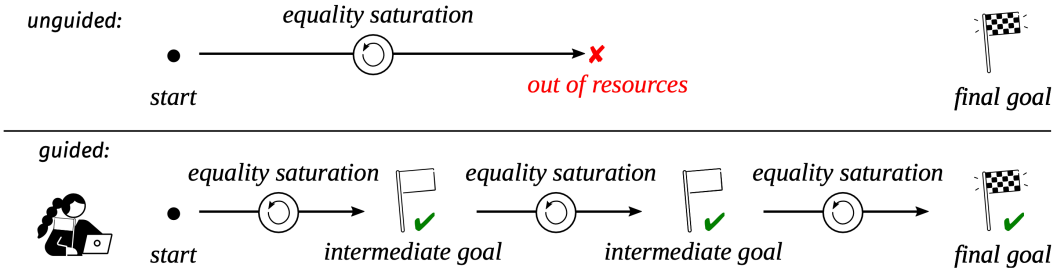


Figure 14: Depiction of guided equality saturation from [KGB+24].

The major trade-off of this approach is that the guides are not automatically inferred, but must be provided by the user. This is, however, a very acceptable trade-off in the context of interactive theorem proving, as proof construction usually involves many manual steps.

[34] A similar idea is used in the field of constraint satisfaction problems, where a single search over n nodes is split into multiple searches with $n_1, ..., n_k$ nodes each by *tree decomposition*. This reduces the runtime from $\mathcal{O}(d^n)$ to $\mathcal{O}(d^{n_1} + ... + d^{n_k})$.

Thus, if a difficult proof goal is not solved automatically by equality saturation, users can specify increasingly easily attainable proof goals by using guides, until the procedure succeeds. Notably, in the context of interactive theorem proving, specifying guides is equivalent to simply splitting the proof into multiple steps. Take, for example, the following theorem over groups where g_1, ..., g_5 denote the axioms of groups:

```
example [Group G] (a b : G) : a⁻¹ * (a * b) = b := by
  egg [g₁, g₂, g₃, g₄, g₅]
```

If this call to the egg tactic were to time out, we could instead perform guided equality saturation by introducing intermediate proof steps:

```
example [Group G] (a b : G) : a⁻¹ * (a * b) = b := by
  calc a⁻¹ * (a * b)
    _ = 1 * b := by egg [g₁, g₂, g₃, g₄, g₅]
    _ = b     := by egg [g₁, g₂, g₃, g₄, g₅]
```

The purpose of the calc tactic is to enable stepwise reasoning over transitive relations with a natural syntax, where each step must be justified by a proof provided after the :=. Notably, for guided equality saturation, every step is proven by the same call egg [g_1, g_2, g_3, g_4, g_5]. Thus, we extend the syntax of our egg tactic to more naturally support guided equality saturation:

```
egg calc [g₁, g₂, g₃, g₄, g₅]
  a⁻¹ * (a * b)
  _ = 1 * b
  _ = b
```

When additional equations are required for single steps, they can be provided by writing `with [eqn₁, ...]` on the relevant line.

Guide Terms When the expressions involved in equality saturation get large, specifying guides can be quite tedious and even error-prone. Moreover, large guide terms can detract from the essential insight provided by the guide, which may only come from a small subterm. As an example, consider the following theorem over groups:

```
example [Group G] (a : G) : a⁻¹⁻¹ = a := by
  egg calc [g₁, g₂, g₃, g₄, g₅]
    a⁻¹⁻¹
    _ = a⁻¹⁻¹ * (a⁻¹ * a)
    _ = a
```

The essential insight in the guide $(a^{-1})^{-1} * (a^{-1} * a)$ is the fact that we can write 1 as $a^{-1} * a$. Yet, we have to embed this insight into a guide which must be a proper intermediate step for guided equality saturation. It would be much more convenient to provide only the relevant subterm, especially when the expressions are larger. Moreover, we may not know exactly what a complete guide should look like, except for the fact that it should contain $a^{-1} * a$. To solve these problems, [KGB+24] introduces the notion of *sketch guides*. Sketch guides are incomplete guide expressions which classify families of terms according to a given sketch language. For example, the sketch guide $\mathit{contains}(a^{-1} * a)$ describes the family of terms containing $a^{-1} * a$ as a subterm. While we do not implement sketch guides directly, we take inspiration from them for a related approach in the egg tactic. We allow users to specify *guide terms* which are added to the e-graph before equality saturation. This is particularly useful when the provided guide terms would have not otherwise been reachable by rewriting. For example, the group axioms do not allow equality saturation to rewrite from 1 to $a^{-1} * a$, as the corresponding group axiom only applies in the reverse direction. Yet, adding the guide term $a^{-1} * a$ to the e-graph allows equality saturation to rewrite from $a^{-1} * a$ to 1, and thus solve the previous theorem:

```
example [Group G] (a : G) : a⁻¹⁻¹ = a := by
  egg [g₁, g₂, g₃, g₄, g₅] using a⁻¹ * a
```

5 Evaluation

To show that our proof tactic is reasonably proficient at equational reasoning, we consider multiple examples of using the tactic with more and less constructed proof goals. In Section 5.1 we follow the evaluation performed for the original prototype of this proof tactic in [KGB+24]. For the sake of clarity, we refer to our tactic as egg, and the tactic prototype from [KGB+24] as ges (guided equality saturation). The evaluation of ges includes three non-trivial use cases for which we show significant improvements with egg. In Section 5.2, we show the results of running our tactic on an additional test suite based on Lean's *batteries* and *mathlib* libraries. As ges is intended as a proof of concept, instead of a practically applicable tool, many test cases which egg can handle are either unsupported by ges, or fail for trivial technical reasons. Therefore, we do not investigate failing test cases for ges in any detail. When we provide timing information for tactic calls, these are rough estimates. As the egg tactic is by no means optimized for speed, they are only intended to show general trends.

5.1 Comparison to Tactic Prototype

The evaluation of ges in [KGB+24] consists of test cases for three mathematical subjects: groups, rings, and the binomial coefficient. We evaluate our tactic both on a set of identical and of similar test cases. This is because the tests for ges tend to formalize the required algebraic structures in a way which is tailored to syntactic rewriting – thus, enabling ges to succeed. In contrast, our test cases for egg use a more idiomatic formalization of these structures, at the cost of a resulting increase in difficulty. Appendix A includes a table of all results in this section.

Group Theory The evaluation in [KGB+24] starts with test cases on groups, as they allow for non-trivial equational reasoning while also allowing for a relatively self-contained formalization. We choose a natural formalization of groups as type classes:

```
class One (α) where one : α

instance [One α] : OfNat α 1 where ofNat := One.one

class Inv (α : Type u) where
  inv : α → α

postfix:max "⁻¹" => Inv.inv

class Group (α) extends One α, Inv α, Mul α where
  mul_assoc     (a b c : α) : (a * b) * c = a * (b * c)
  one_mul       (a : α)     : 1 * a = a
  mul_one       (a : α)     : a * 1 = a
  inv_mul_self  (a : α)     : a⁻¹ * a = 1
  mul_inv_self  (a : α)     : a * a⁻¹ = 1
```

We define a group structure over a type α as having a distinguished 1 element, an inverse function $^{-1}$ and a multiplication $*$, which satisfy the listed group axioms. Based on these axioms, we prove the following five theorems, which assume [Group G] and (a b : G):

1. `theorem inv_mul_cancel_left : a⁻¹ * (a * b) = b`
2. `theorem mul_inv_cancel_left : a * (a⁻¹ * b) = b`
3. `theorem inv_one             : (1 : G)⁻¹ = 1`
4. `theorem inv_mul             : (a * b)⁻¹ = b⁻¹ * a⁻¹`
5. `theorem inv_inv             : a⁻¹⁻¹ = a`

For our formalization of groups, ges cannot prove any of the stated theorems – the main problem being that ges cannot handle type classes. Thus, we compare our results from egg to the results of running ges on the simpler formalization of groups used for [KGB+24]. Qualitatively, our results for egg match those for ges, with unguided equality saturation solving Theorems 1-3, and guidance being required for Theorems 4 and 5. More specifically, given the group axioms, egg manages to prove Theorems 1-3 in under 50ms each. Theorems 4 and 5 cannot be solved directly, as their proofs require the "creative step" of multiplying by a term which reduces to 1, like $a \cdot a^{-1}$. Thus, we prove these theorems using guide terms:

```
theorem inv_mul [Group G] (a b : G) : (a * b)⁻¹ = b⁻¹ * a⁻¹ := by
  egg [/-group axioms-/] using b⁻¹ * a⁻¹ * (a * b) * (a * b)⁻¹

theorem inv_inv [Group G] (a : G) : a⁻¹⁻¹ = a := by
  egg [/-group axioms-/] using a⁻¹ * a
```

Our tactic takes under 2s and 250ms, respectively, to solve these theorems. Notably, if we reduce the guide term for inv_mul to just $\mathsf{a}^{-1} * (\mathsf{a} * \mathsf{b}) * (\mathsf{a} * \mathsf{b})^{-1}$, the call takes twice as long at about 4s. This exemplifies how slight changes to initial conditions can have a significant effect when dealing with the rapid growth of e-graphs during equality saturation.

Freshman's Dream As a slightly more challenging test scenario, ges is evaluated on two theorems about commutative semirings of characteristic 2. That is, commutative semirings with the property $\forall x, x + x = 0$. We implement this algebraic structure in a similar fashion to groups, and thus omit the implementation here. The tests for ges again use a definition which is better suited for syntactic rewriting. The characteristic property of these semirings allows for the following case of a theorem called the freshman's dream: $(x + y)^2 = x^2 + y^2$. For ges, this theorem is proven with three intermediate steps in less than 600ms:

```
(x + y)^2 = (x + y) * (x + y)
        _ = x * (x + y) + y * (x + y)
        _ = x^2 + x * y + y * x + y^2
        _ = x^2 + y^2
```

Without guides, the same theorem takes ges around 4min [KGB+24]. In contrast, without guides our tactic proves the theorem in under 600ms. However, if we prove the theorem using the same guides as shown above, we actually *increase* the required time to 11s. The culprit is the proof step from $x \cdot (x + y) + y \cdot (x + y)$ to $x^2 + x \cdot y + y \cdot x + y^2$. While all other proof steps are solved by egg with less than 10 rewrites, this step generates an explanation with over 300 (mostly nonsensical) rewrites. Simply removing this intermediate step reduces the total time back to 500ms. This highlights a practical issue with guided equality saturation. Guides which correspond to obvious steps in human reasoning may be "counterintuitive to egg" for completely opaque reasons.

As an example of a more involved theorem, ges is also tested against the analogous theorem for power 3: $(x+y)^3 = x^3+x\cdot y^2+x^2\cdot y+y^3$. The unguided version takes over 20 minutes, while using a total of 5 guides reduces the time to under 1s [KGB+24]. Our tactic takes about 6s to prove this theorem without guides. Additionally, the single guide term $(x+y)\cdot(x+y)$ suffices to reduce the time to under 1s. This reduction in runtime again highlights the practical challenges of using guided equality saturation. It is entirely unclear to us why the term $(x + y) \cdot (x + y)$ allows egg to find a proof significantly faster. As such, coming up with relevant guide terms is currently more of an art than a science. This result corresponds to the findings in [KGB+24], where the authors show that a single well-chosen guide can have a much greater impact on runtime than multiple poorly chosen guides.

Binomial Coefficient As a final example of using guided equality saturation, [KGB+24] considers a proof about binomial coefficients from [Rot06]. The theorem states that for all $n, r \in \mathbb{N}$ with $r \leq n$, we have $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. At the heart of this theorem lies the following sequence of equations:

$$
\begin{aligned}
&\frac{n!}{(r-1)!(n-r+1)!} + \frac{n!}{r!(n-r)!} \\
&= \frac{n!}{(r-1)!(n-r)!}\left(\frac{1}{n-r+1} + \frac{1}{r}\right) \\
&= \frac{n!}{(r-1)!(n-r)!}\left(\frac{r+n-r+1}{r(n-r+1)}\right) \\
&= \frac{n!}{(r-1)!(n-r)!}\left(\frac{n+1}{r(n-r+1)}\right) \\
&= \frac{(n+1)!}{r!(n+r-1)!}
\end{aligned}
$$

The ges tactic does not manage to prove these equations. A primary reason for this is the necessity for conditional rewriting, which is not supported by ges. Using guided equality saturation with egg, we can formalize a proof of these equations as follows:

```
egg calc [add_comm, sub_add_cancel, mul_comm, mul_assoc, Gamma_add_one; h₄, h₅, h₆]
  (n¡ / ((r - 1)¡ * (n - r + 1)¡) + n¡ / (r ¡ * (n - r)¡))
    _ = n¡ / ((r - 1)¡ * (n - r)¡) * (1 / (n - r + 1) + 1 / r)
      with [div_mul_eq_div_mul_one_div, left_distrib (R := Real)]
    _ = n¡ / ((r - 1)¡ * (n - r)¡) * ((r + n - r + 1) / (r * (n - r + 1)))
      with [_root_.add_div, mul_div_mul_left, mul_one]
    _ = n¡ / ((r - 1)¡ * (n - r)¡) * ((n + 1) / (r * (n - r + 1)))
      with [sub_add_eq_add_sub]
    _ = (n + 1)¡ / (r¡ * (n + 1 - r)¡)
      with [sub_add_eq_add_sub, _root_.div_mul_div_comm]
```

Since the given equations do not hold for $\mathbb{N}$, we reason over their corresponding terms in $\mathbb{R}$. We thus have to use the Γ function which generalizes the factorial function and satisfies $\Gamma(n+1) = n!$ for all $n \in \mathbb{N}$. To retain visual similarity to the equations in [Rot06], we write $n¡$ for $\Gamma(n+1)$. Minding these technicalities, our proof steps correspond precisely to the steps of [Rot06] above. Notably, the required conditional rewriting is enabled by the given facts h_4, h_5 and h_6, which are proven separately:

```
h₄ : (n : ℝ) - r + 1 ≠ 0
h₅ : (r : ℝ) ≠ 0
h₆ : (n : ℝ) + 1 ≠ 0
```

Given these guides and facts, our tactic solves the goal in less than 12s. However, this setup is again very fragile. For example, moving the lemma sub_add_eq_add_sub from the last two steps into the shared set of rewrites (at the top) causes the second step to time out. Additionally, a major caveat of the current approach to conditional rewriting is that one must determine *beforehand* which preconditions are required to enable the necessary rewrites. This is counter to the goal of our tactic, which is to not have to figure out which exact rewrites need to be performed. Thus, we have implemented a prototype of an extension to our tactic, which does not require the preconditions as inputs, but instead produces the necessary preconditions as subgoals to be solved after calling egg. Using this approach, egg produces the following subgoals for the example above:

```
(n : ℝ) + 1 - r ≠ 0
(r : ℝ) ≠ 0
(n : ℝ) + 1 ≠ 0
(n : ℝ) ≠ 0
```

These preconditions are all provable and almost entirely correspond to the manually provided preconditions h_4, h_5 and h_6. However, when using this approach, the second equational reasoning step fails.[35] Thus, the technique is currently unreliable, which we hope to improve in the future.

[35] We believe the reason to be related to the fact that facts can have a similar effect as guide terms, as they are added to the e-graph before equality saturation.

5.2 Library Tests

The test cases of [KGB+24] provide meaningful insight into the *potential* of proof tactics based on equality saturation, as well as their fundamental shortcomings. However, they do not meaningfully indicate how well these tactics perform "in the wild", as test cases can be tweaked until they succeed. In this section, we cover a much broader spectrum of test cases. Unfortunately, obtaining a large number of test cases is not trivial, as we are not aware of any benchmarks for equational rewriting. A similar issue is mentioned in [LF23] when testing Lean's prominent aesop proof tactic. The authors resort to testing their tactic against hand-picked theorems from the *mathlib* library [mC20], which at the time of writing has amassed over 150,000 theorems.[36] Thus, we also test egg against theorems from mathlib, and additionally consider Lean's extended standard library *batteries*. However, instead of hand-picking theorems, we automatically construct suitable test cases from theorems which use the simp tactic. By using Lean's meta-programming capabilities, we override the simp tactic such that it calls egg instead. In particular, we only override calls to simp only, which is a variant of simp which uses only those equations which are explicitly provided to it at the call site. We also make sure that the calls are *terminal* (they are the final step in a given proof), and that they are solving proof goals of the form $lhs = rhs$ or $lhs \leftrightarrow rhs$. These criteria, and others which we omit here, serve to filter those calls to simp only which should be solvable by egg. Using this approach, we obtain over 2,000 test cases against which we evaluate egg. By setting a 10s time limit for equality saturation and disabling some or all of the techniques developed throughout this thesis, we obtain the results shown in Figure 15. For a complete table of results, see Appendix B.

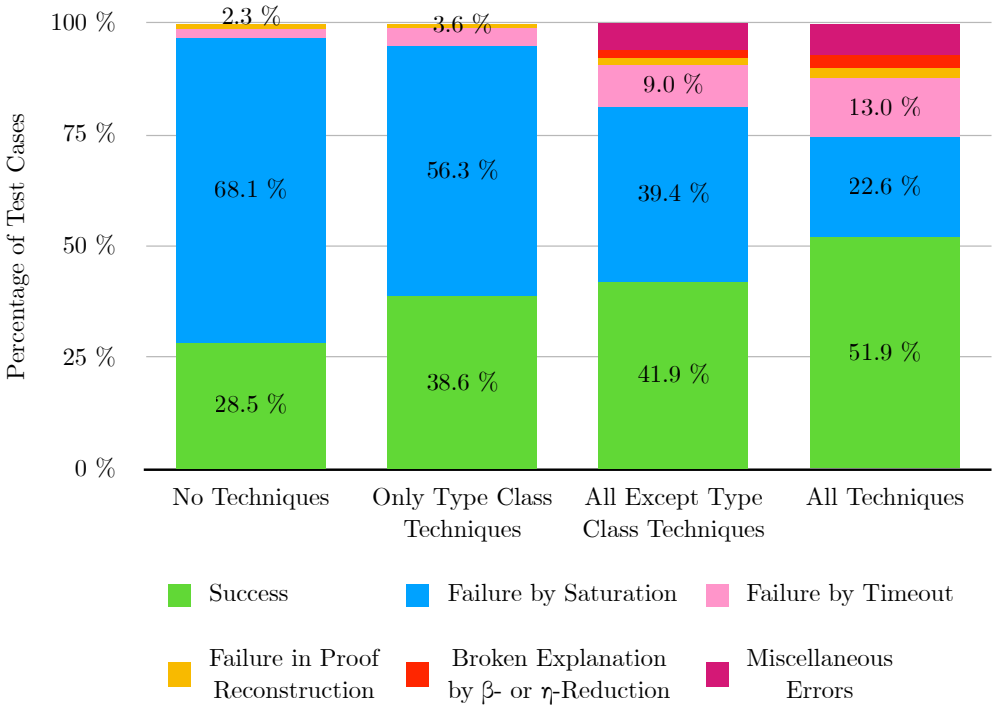


Figure 15: Results of evaluating egg on simp onlys in *mathlib* and *batteries*.

Successful test cases are those where egg manages to prove the proof goal. Applying all techniques discussed in this thesis, we succeed on about half of all test cases, which is a clear improvement over disabling these techniques. We also considered techniques for type classes (Section 4.5) separately, as we assumed them to have a rather large impact. This seems to hold up, as they increase the success rate by about 10%, whereas all other techniques add 13.5%.

[36] See https://leanprover-community.github.io/mathlib_stats.html.

Failing Tests For tests cases where egg fails to prove the goal, we distinguish whether equality saturation saturated, or timed out. Tests which fail by saturation indicate that egg lacks techniques for applying given equations, as a successful proof by simp only implies that the given equations must suffice to prove the goal. As the proportion of failure by saturation significantly decreases as we apply more techniques (from 68.1% to 22.6%), we can deduce that adding more techniques successfully contributes to the applicability of equations. Tests which fail by timeout are harder to interpret. They can again indicate that egg lacks techniques for applying given equations, yet is able to apply enough equations to not saturate. That is, failure by timeout can be thought of as "one step better" than failure by saturation, but not good enough for a successful test. The other interpretation of failure by timeout is simply that the e-graph grew too quickly during equality saturation, and that failure was not caused by our tactic's lack of capability. However, we can virtually entirely rule out this latter interpretation. By collecting timing information, we found that 99.7% of all successful test cases completed equality saturation in less than one second. Thus, it is highly unlikely that a test which fails by timeout did so only because it did not have enough time. Instead, it is much more likely that egg was able to apply enough equations to not saturate, but was not capable enough to prove the goal. For example, given five equations, it may have been able to apply four of them, while not being able to apply the fifth due to missing definitional equality rules. This conclusion about failures by timeout allows for the following interpretation of their distribution in our results. As applying more techniques increases the number of applicable equations, the number of failures by saturation decreases in part because tests succeed, but also because more test cases become failing by timeout. Thus, the increase in applied techniques also sees an increase in failures by timeout (from 2.3% to 13%).

Other categories of failures are of a more technical nature. In some tests, egg finds an explanation, but our tactic fails to construct a proof from it. In this case, we have a failure in proof reconstruction. A significant portion of these failures result from β- and η-reduction, as they rely on our e-class substitution algorithm, which often produces broken explanations. We thus separate these failures involving β- and η-reduction from other failures in proof reconstruction, which indicate either a lack of capability or a presence of bugs in our proof reconstruction. About 2% of test cases make up the latter category, showing that our tactic is quite effective at turning explanations into proofs. The category of Miscellaneous Errors covers failures which we attribute to bugs in our implementation, the biggest source being stack overflows in our e-class substitution algorithm. As e-class substitution (virtually) does not occur in the first two test runs, they do not show these kinds of failures.

Caveats Our testing approach covers a large number of test cases which range across a variety of mathematical fields. However, by the nature of how simp only tends to be used, each test involves only a relatively small number of equations as shown in Figure 16.

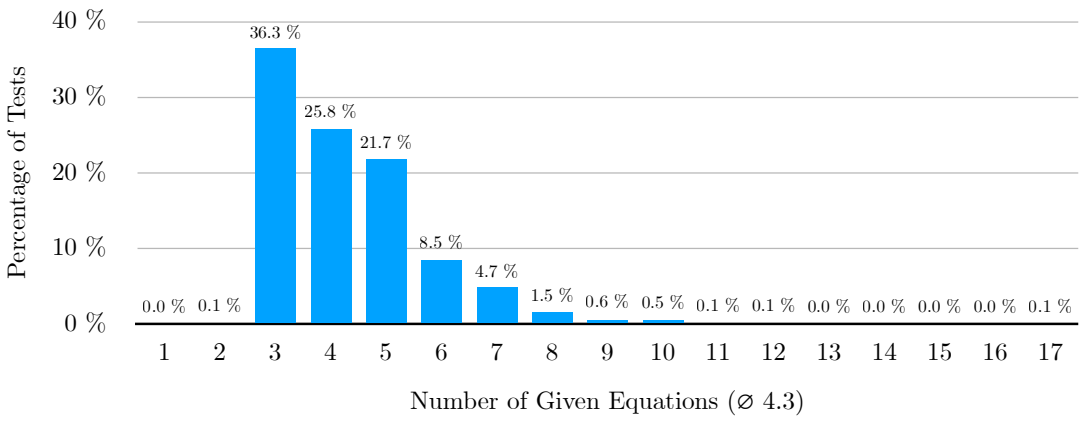


Figure 16: Distribution of the number of equations occurring in simp only tests.

Consequently, we cannot deduce whether our results are representative when a larger number of equations is used. For example, as mentioned above, 99.7% of all successful test cases complete in less than one second. Yet, anecdotally, tests involving more equations can take a couple of seconds to complete. A similar case can be made for the sizes of explanations produced by egg for successful test cases, as shown in Figure 17.

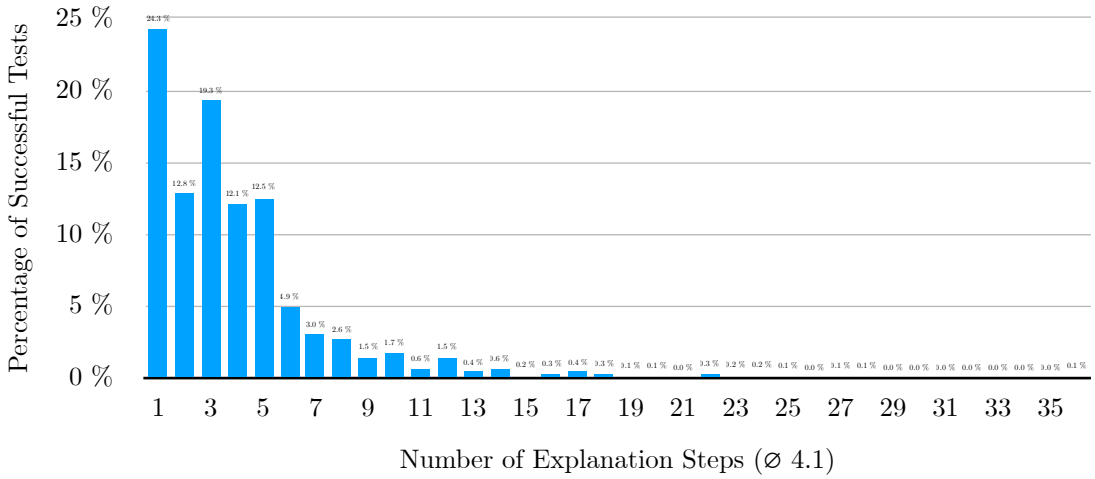


Figure 17: Distribution of the length of explanations for successful simp only tests.

While this distribution suggests that explanations tend to be small, with a bound at 35 steps, we have encountered explanations with hundreds of steps for tests like those in Section 5.1. We believe that these short test durations and explanation sizes also skew the timing distribution shown in Figure 18.

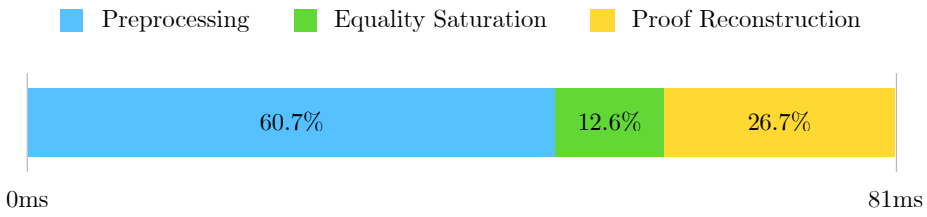


Figure 18: Average timing distribution of phases of the egg tactic in the average successful simp only test.

These averages suggest that preprocessing is a major bottleneck in our tactic. However, as this preprocessing metric includes the parsing and elaboration steps performed by Lean, it also measures a baseline of overhead which we expect to decrease for calls with more equations. A similar argument can be made for proof reconstruction, though our anecdotal evidence suggest that the time required for proof reconstruction can vary widely depending on how short of an explanation egg finds.

An additional caveat of our test setup is that the unique capabilities of equality saturation are not tested, as all tests are solvable by simp only, and thus by greedy rewriting. Also, our approach to conditional rewriting (Section 4.7) is not covered by these test at all. This is because simp only does not separate its list of equations from the list of facts, which means that we cannot simply transform such calls into calls to egg. While the latter problem may be solvable, addressing the former problem requires constructing a test suite for equality saturation.

6 Conclusion

This thesis develops a practically applicable proof tactic for equational reasoning in Lean, by combining various techniques for enabling effective rewriting of Lean expressions in e-graphs. We improve upon the previous work of [KGB+24] by addressing the problems of definitional equality and binders in e-graphs. We also enable new capabilities such as rewriting under binders and conditional rewriting, which require more sophisticated forms of proof reconstruction. Figure 19 summarizes our work, and extends the simplified view of Figure 8.

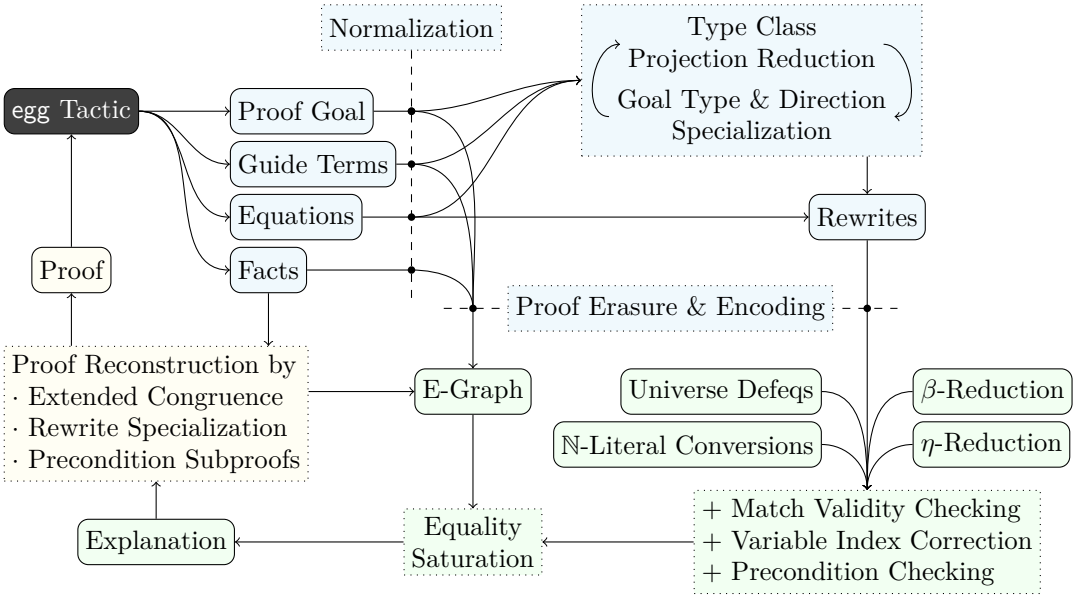


Figure 19: Complete overview of the egg proof tactic, color coded by preprocessing in Lean, equality saturation in egg, and proof reconstruction in Lean.

We address different forms of definitional equality in the preprocessing phase by normalization (Section 4.4.1), proof erasure (Section 4.4.2), and type class projection reduction and specialization (Section 4.5). In the egg backend, we additionally consider β- and η-reduction (Section 4.4.4), as well as natural number literals (Section 4.4.3). To fix the problems of using binders in e-graphs, we validate e-matching results (Section 4.3.1) and correct the indices of captured variables (Section 4.3.2). As multiple techniques require substitution, we develop an algorithm for substitution on e-classes (Section 4.3.3). Notably, this algorithm does not currently support proper handling of explanations, making the associated techniques very unreliable. During proof reconstruction, we rely on an existing implementation for congruence quotations, which allows us to rewrite under binders and dependent functions. We also extend the procedure with recursive subproof generation for conditional rewriting (Section 4.7). Finally, we address the notion of guided equality saturation with a syntactic calc extension of our tactic, and a notion of guide terms (Section 4.8). Our results show that while we outperform the tactic prototype of [KGB+24] on qualitative tests, we can also solve a decent number of existing equational goals "blindly" (Section 5). In particular, we find that the above techniques significantly contribute to our tactic's success rate. However, we also find that using egg can be fragile and hard to debug, as equality saturation is a rather opaque technique and e-graphs do not scale nicely. While we are aware of various implementation problems and inefficiencies in our tactic, we expect to solve these in subsequent work. As such, we hope egg to be useful and appealing to the broader Lean community. The tactic is openly developed at `https://github.com/marcusrossel/lean-egg`.

Future Work Throughout this thesis we have at various points mentioned opportunities for improvements to our proof tactic. We gather these points here, and mention additional avenues for future work.

The most pressing item is fixing the propagation of explanation information during e-class substitution (Section 4.3.3). This would fix broken explanations involving β- and η-reduction, which currently account for a non-negligible number of failing test cases. Additionally, substitution is currently implemented as a recursive function, which comes at a cost for runtime and memory. It should therefore be replaced by an iterative implementation.

As many of the challenges in this thesis stem from using Lean's expression language in e-graphs, it may be worthwhile to reduce these expressions to a simpler language via the *lean-auto* project. Alternatively, advancements in slotted e-graphs [SKS24] may solve some these problems in the long run. As a more immediate measure, we can add type ascriptions to encoded expressions. This solves problems introduced by equations like $\forall u : \mathsf{Unit}, u = \mathsf{unit}$, which cause egg to immediately solve any goal with an invalid explanation.

An existing project which uses egg (that is, the library, not our tactic) is the convex optimization modeling framework *CvxLean* [BFMA23]. Our proof tactic does not currently suffice to satisfy the use cases of this project, as it requires more fine-grained control of egg, for example, for e-class analyses and extraction. A useful extension to our tactic is therefore to provide a more general interface between Lean and egg. As a first step towards an improved interface, we intend to remove the currently explicit interface via C, by using the *lean_sys*[37] library for interfacing directly with Lean from Rust. A second necessary step is to allow for non-terminal calls to egg. That is, calls which do not entirely solve the current proof goal, but instead exit with a remaining subgoal when the goal cannot be proven (like rw and simp). The crux of non-terminal calls is to determine what the resulting proof goal should be. For this, it could be useful to introduce a mechanism similar to sketch-satisfying extraction in [KGB+24], by allowing the user to sketch desirable subgoals.

As a restricted form of non-terminal calls to egg, we have considered generating subgoals for preconditions used by egg for conditional rewriting in Section 5.1. While this mechanism is almost entirely implemented, it requires refinement by, for example, collapsing equivalent subgoals. It is also currently open whether this mechanism is practically useful, as egg may tend to generate unprovable subgoals.

As calls to egg can take multiple seconds, which is rather long for interactive proof tactics, it is desirable to retain the resulting proof, and not rerun the tactic each time a file is recompiled. While there are multiple options for this, an interesting one is the *calcify*[38] project, which turns proofs by rw and simp into proofs by small-step equational reasoning with calc. We hope to easily adjust the project's implementation to work with egg, which may become easier by adjusting our proof generation to use simp's infrastructure. A potential problem with persisting proofs in this way is that egg sometimes produces proofs with hundreds of steps.

Finally, Andrés Goens has already implemented an extension to our tactic which allows theorems to be tagged with @[egg], which builds up an index of theorems to be used by egg. We intend to extend this mechanism with categorization of theorems into named groups, such that a call like egg/ring would call egg with all theorems tagged with @[egg ring]. This is equivalent to the mechanism of *simp sets* for simp, yet bears the superior name of *egg baskets* suggested by Johan Commelin. We think that egg baskets could be extremely useful for user-definable, ad-hoc proof automation. As immediate future work, we therefore intend to investigate the viability of imitating existing tactics such as ring and zify using egg with egg baskets. We expect this to require some form of *premise selection* [BKPU16], as to not overwhelm egg with too many rewrites.

[37] https://github.com/digama0/lean-sys
[38] https://github.com/nomeata/lean-calcify

A Results of Comparison to Tactic Prototype

The following table summarizes the results of Section 5.1, where we compare our egg tactic to the ges prototype from [KGB+24]. Failed test cases are marked with ✗. When results are cited from [KGB+24], we mark them with $*$. For tests involving algebraic structures we distinguish between "idiomatic" and "simplified" versions. The former define algebraic structures using type classes, while the latter use the formalizations of [KGB+24]. We note that the results of the simplified version of freshman's dream for egg are somewhat unintuitive, as we faced difficulties reconstructing these test cases.

Test	egg	ges
Groups (Idiomatic)		
Theorem 1	50ms	✗
Theorem 2	50ms	✗
Theorem 3	50ms	✗
Theorem 4	2s (1 good guide) 4s (1 bad guide)	✗
Theorem 5	250ms (1 guide)	✗
Groups (Simplified)		
Theorem 1	30ms	400ms
Theorem 2	30ms	400ms
Theorem 3	10ms	100ms
Theorem 4	200ms (1 good guide) 200ms (1 bad guide)	2s (4 guides)
Theorem 5	100ms (1 guide)	600ms (1 guide)
Freshman's Dream (Idiomatic)		
$(x+y)^2$	600ms	✗
$(x+y)^3$	1s (1 guide) 6s (unguided)	✗
Freshman's Dream (Simplified)		
$(x+y)^2$	700ms (1 guide)	600ms (3 guides) 4min (unguided) $*$
$(x+y)^3$	800ms (1 guide)	1s (5 guides) 20min (unguided) $*$
Binomial Coefficient		
Binomial Coefficient	12s (3 guides)	✗

B Results of Library Tests

The following table contains the data underlying the results of Figure 15. The number of test cases decreases as we introduce more techniques, as this causes tests to take longer on average and increases the number of stack overflows by e-class substitution, which had to be fixed manually.

Outcome	No Techniques	Only Type Class Techniques	All Except Type Class Techniques	All Techniques
Total Number of Tests	4017	3424	2410	2123
Success	1145 (28.5%)	1320 (38.6%)	1010 (41.9%)	1101 (51.9%)
Failure by Saturation	2736 (68.1%)	1928 (56.3%)	949 (39.4%)	480 (22.6%)
Failure by Timeout	93 (2.3%)	124 (3.6%)	217 (9.0%)	277 (13.0%)
Failure in Proof Reconstruction	40 (1.0%)	49 (1.4%)	41 (1.7%)	47 (2.2%)
Broken Explanation by β- or η-Reduction	0 (0.0%)	0 (0.0%)	39 (1.6%)	67 (3.2%)
Miscellaneous Errors	3 (0.1%)	3 (0.1%)	154 (6.4%)	151 (7.1%)

References

[AGG61] Bruce W Arden, Bernard A Galler, and Robert M Graham. An algorithm for equivalence declarations. *Communications of the ACM*, 4(7):310–314, 1961.

[BBN11] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in isabelle/hol. In *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings 8*, pages 12–27. Springer, 2011.

[BCM20] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 299–312, New York, NY, USA, 2020. Association for Computing Machinery.

[BFMA23] Alexander Bentkamp, Ramon Fernández Mir, and Jeremy Avigad. Verified reductions for optimization. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 74–92. Springer, 2023.

[BKPU16] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge university press, 1998.

[Bou23] Thomas Bourgeat. *Specification and verification of sequential machines in rule-based hardware languages*. Massachusetts Institute of Technology, 2023.

[Car19] Mario Carneiro. The Type Theory of Lean. Master's thesis, 2019.

[CH69] Lorenzo Calabi and WE Hartnett. Some general results of coding theory with applications to the study of codes for the correction of synchronization errors. *Information and Control*, 15(3):242, 1969.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Cha12] Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49:363–408, 2012.

[Coq86] Thierry Coquand. An analysis of girard's paradox. 1986.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.

[dB68] NG de Bruijn. Automath: a language for mathematics. 1968.

[DMB07] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*, pages 183–198. Springer, 2007.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3):365–473, may 2005.

[FCW+22] Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. Small proofs from congruence closure. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pages 75–83. IEEE, 2022.

[FV22] Zoltán Fülöp and Heiko Vogler. Weighted tree automata – may it be a little more? *arXiv preprint arXiv:2212.05529*, page 40, 2022.

[GF64] Bernard A Galler and Michael J Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[Gor00] Mike Gordon. From lcf to hol: a short history. 2000.

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In *Computational Logic*, volume 9, pages 135–214, 2014.

[KGB+24] Thomas Kœhler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. Guided equality saturation. *Proceedings of the ACM on Programming Languages*, 8(POPL):1727–1758, 2024.

[Knu97] Donald E Knuth. *The Art of Computer Programming: Fundamental Algorithms, volume 1*. Addison-Wesley Professional, 1997.

[Kœh22] Thomas Kœhler. *A domain-extensible compiler with controllable automation of optimisations*. PhD thesis, University of Glasgow, 2022.

[Lam99] Leslie Lamport. Specifying concurrent systems with tla+. *Calculational System Design*, pages 183–247, 1999.

[LF23] Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 253–266, 2023.

[Mas21] Patrick Massot. Why formalize mathematics, 2021.

[mC20] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[McC92] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of automated reasoning*, 9(2):147–167, 1992.

[MŁK08] Michał Moskal, Jakub Łopuszański, and Joseph R Kiniry. E-matching for fun and profit. *Electronic Notes in Theoretical Computer Science*, 198(2):19–35, 2008.

[Nel80] Charles Gregory Nelson. *Techniques for Program Verification*. Phd thesis, Stanford University, 1980.

[NO80] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005.

[Par95] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, 1995.

[Pie02] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[PS23] Patrick Massot Peter Scholze, Johan Commelin. Blueprint for the Liquid Tensor Experiment. https://leanprover-community.github.io/liquid/, 2023.

[Rin24] Talia Ringer. Proofs and conversations. 2024.

[Rot06] Joseph J Rotman. A first course in abstract algebra: with applications. 2006.

[SdM16] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8*, pages 99–115. Springer, 2016.

[SKS24] Rudi Schneider, Thomas Kœhler, and Michel Steuwer. Slotted e-graphs. 2024.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[TT23] Yaël Dillies Terence Tao. A digitisation of the proof of the Polynomial Freiman-Ruzsa Conjecture in Lean 4. https://teorth.github.io/pfr/, 2023.

[Ull23] Sebastian Andreas Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Dissertation, Karlsruhe, Karlsruher Institut für Technologie (KIT), 2023, 2023.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, 1989.

[WNW+21] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[ZWWT21] Yihong Zhang, Yisu Remy Wang, Max Willsey, and Zachary Tatlock. Relational e-matching. *arXiv preprint arXiv:2108.02290*, 2021.

Acknowledgements

First and foremost I would like to thank Andrés for continuously taking an interest in my work throughout the last couple of years, while quite significantly guiding what that work is actually about, and nudging me to take some opportunities which I would not otherwise have. Our exchanges always make the process fun and interesting, even during laborious work like writing theses. I would also like to thank Jeronimo for repeatedly supporting my work and fostering a diverse range of research topics. I shall set out into the world as a CCC evangelist for it. I would also like to thank Rudi Schneider for our one meeting which basically lead to all of Section 4.3. On a broader scale, I would like to thank the Lean and egg communities for their amazing support. It still baffles me how virtually any question is addressed so persistently and quickly. In particular, I would like to thank Oliver Flatt for taking the time to discuss explanations, and Mario Carneiro and Kyle Miller for having the patience to dive into rather specific Lean-related topics.