
# Towards Pen-and-Paper-Style Equational Reasoning in Interactive Theorem Provers by Equality Saturation

MARCUS ROSSEL*, Barkhausen Institut, Germany and TU Darmstadt, Germany
RUDI SCHNEIDER, Technische Universität Berlin, Germany
THOMAS KŒHLER, ICube Lab - CNRS - Université de Strasbourg, France
MICHEL STEUWER, Technische Universität Berlin, Germany
ANDRÉS GOENS, University of Amsterdam, Netherlands and TU Darmstadt, Germany

Equations are ubiquitous in mathematical reasoning. Often, however, they only hold under certain conditions. As these conditions are usually clear from context, mathematicians regularly omit them when performing equational reasoning on paper. In contrast, interactive theorem provers pedantically insist on every detail to be convinced that a theorem holds, hindering equational reasoning at the more abstract level of pen-and-paper mathematics. In this paper, we address this issue by raising the level of equational reasoning to enable pen-and-paper style in interactive theorem provers. We achieve this by interpreting theorems as conditional rewrite rules, and use equality saturation to automatically derive equational proofs. Conditions that cannot be automatically proven may be surfaced as proof obligations. Concretely, we present how to interpret theorems as conditional rewrite rules for a significant class of theorems. Handling these theorems goes beyond simple syntactic rewriting, and deals with aspects like propositional conditions and type classes. We evaluate our approach by implementing it as a tactic in Lean, using the egg library for equality saturation with e-graphs. We show four use cases demonstrating the efficacy of this higher level of abstraction for equational reasoning.

CCS Concepts: • **Computing methodologies** → **Theorem proving algorithms**; • **Theory of computation** → **Equational logic and rewriting**; **Automated reasoning**.

Additional Key Words and Phrases: equality saturation, equational reasoning, e-graphs

**ACM Reference Format:**
Marcus Rossel, Rudi Schneider, Thomas Kœhler, Michel Steuwer, and Andrés Goens. 2026. Towards Pen-and-Paper-Style Equational Reasoning in Interactive Theorem Provers by Equality Saturation. *Proc. ACM Program. Lang.* 10, POPL, Article 25 (January 2026), 30 pages. https://doi.org/10.1145/3776667

## 1 Introduction

In popular culture, equations are synonymous with mathematical knowledge. Schrödinger's and Maxwell's equations capture our knowledge of the universe. Fermat's little theorem, vital for today's cryptography, describes an important equation in modular arithmetic. And his last theorem, disproving an equation, defied mathematicians for centuries. Indeed, equational reasoning is one of the fundamental tools of mathematics.

*Work done at Barkhausen Institut.

Authors' Contact Information: Marcus Rossel, Barkhausen Institut, Dresden, Germany and TU Darmstadt, Darmstadt, Germany, marcus.rossel@barkhauseninstitut.org; Rudi Schneider, Technische Universität Berlin, Berlin, Germany, r.schneider@tu-berlin.de; Thomas Kœhler, ICube Lab - CNRS - Université de Strasbourg, Strasbourg, France, thomas.koehler@cnrs.fr; Michel Steuwer, Technische Universität Berlin, Berlin, Germany, michel.steuwer@tu-berlin.de; Andrés Goens, University of Amsterdam, Amsterdam, Netherlands and TU Darmstadt, Darmstadt, Germany, a.goens@uva.nl.


ACM 2475-1421/2026/1-ART25
https://doi.org/10.1145/3776667

Translating intuitive mathematical reasoning to machine-checkable languages is one of the long-standing goals of interactive theorem provers (ITPs) [Wiedijk 2006]. Arguably, it is the raison d'être of systems like Rocq [Rocq Dev Team 2025], Isabelle [Paulson 1993] or Lean [de Moura and Ullrich 2021]. Decades of research in ITPs has brought innovations that made it easier to express reasoning steps, often by being closer to informal "pen-and-paper" mathematics. Among these innovations is a long tradition of term rewriting techniques. From extensible simplification [Boyer and Moore 1973], to e-graphs [Detlefs et al. 2005], and proof-producing congruence closure [Nieuwenhuis and Oliveras 2005; Selsam and de Moura 2016], rewriting is a critical tool in the ITP toolbox.

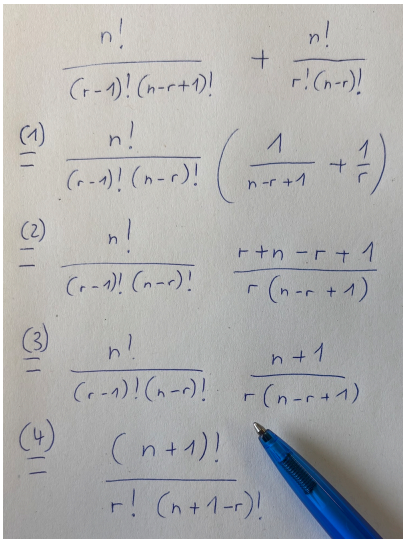
Term rewriting reflects equational reasoning by transforming mathematical equations into rewrite rules. The process of translating equations into rewrites typically involves identifying a class of theorems that describes mathematical equations. These equational theorems are then translated into rewrites rules by orienting the equation and converting universally-quantified variables into (syntactic) pattern variables. This approach, however, is only sound when an equation holds unconditionally, which equations used in pen-and-paper proofs almost never do. For example, the Pythagorean theorem $a^2 + b^2 = c^2$ requires the angle of the triangle formed by the three sides $a, b, c$ to be a right angle. Similarly, Fermat's little theorem $a^p \equiv a \pmod{p}$ requires the exponent and modulus $p$ to be prime. When reasoning with such equations on paper, mathematicians might silently check the conditions in their head. When a condition is not obvious, they might annotate their reasoning with an explanation. While some conditions are explicitly stated in theorem statements, other conditions are often not. For example, the binomial theorem states that $(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$. It would be easy to think of this as an equation that holds unconditionally, but it does not. It has several implicit conditions. For example, we assume that the operations that appear in the equation are all defined for the set $R$ that $x$ and $y$ belong to: addition, multiplication, integer powers and integer coefficients like $\binom{n}{k}$. Even if we do not typically think of it this way, these are preconditions for this equation to hold. Without them, we could not even state the theorem! Additionally, we need these operations to satisfy certain properties: $R$ needs to be a commutative ring, which constrains its algebraic structure.

To reason with such theorems in the same intuitive style as with pen and paper, we need to convince the ITP that all conditions are satisfied. Unfortunately, as we demonstrate in Section 2, ITPs at the moment require us to be explicit about both important preconditions and minor nuances alike. This often makes proofs tedious to formalize and hinders understanding of existing proofs, as the mathematical ideas are hidden among technicalities.

In this paper, we develop a novel approach to facilitate pen-and-paper-style equational reasoning in ITPs, based on a representation of theorems as conditional rewrite rules. We give an explicit characterization of a class of suitable theorems, described in the ITP's underlying logic: in our case, Lean's type theory [Carneiro 2019, 2024]. This involves a classification of preconditions, upon which we build a translation into conditional rewrite rules used for automated term rewriting in an e-graph. In Section 3, we give an overview of our approach and its integration as a tactic in Lean, based on the rewriting engine *egg*. Our approach is enabled by the following contributions:

- An *encoding* to translate theorems into conditional rewrite rules (Section 4), which are then passed to an automated rewrite engine;
- a *decoding* justifying how theorems were instantiated and conditions were satisfied in the automatically-found proof (Section 5);
- a set of *extensions* that further the usefulness of our approach, enabling it to prove more theorems (Section 6).

We demonstrate our technique with multiple use cases in Section 7, showing that we enable the desired pen-and-paper-style equational reasoning.




(a) Reasoning with pen on paper

```
calc
_ = n! / ((r - 1)! * (n - r + 1)!) +
    n! / (r! * (n - r)!) := by
  rw [cast_add, cast_div h₁ h₂, cast_div h₃ h₄, cast_mul,
    cast_mul]
  repeat' rw [← Gamma_nat_eq_factorial]
  rw [cast_sub (by omega), cast_add, cast_sub (by omega),
    cast_one]
_ = n! / ((r - 1)! * (n - r)!) *
    (1 / (n - r + 1) + 1 / r) := by
  rw [Gamma_add_one h₅, mul_comm (n - r + 1 : ℝ),
    ← mul_assoc, div_mul_eq_div_mul_one_div,
    ← sub_add_cancel ↑r (1 : ℝ), Gamma_add_one h₆]
  ring_nf
_ = n! / ((r - 1)! * (n - r)!) *
    ((r + (n - r + 1)) / (r * (n - r + 1))) := by
  rw [div_add_div _ _ h₅ h₇]
  ring
_ = n! / ((r - 1)! * (n - r)!) *
    ((n + 1) / (r * (n - r + 1))) := by
  ring
_ = (n + 1)! / (r! * (n + 1 - r)!) := by
  rw [_root_.div_mul_div_comm, mul_comm,
    ← Gamma_add_one h₈, mul_assoc, mul_comm ((n - r : ℝ)!),
    mul_assoc, ← mul_assoc, mul_comm ((r - 1 : ℝ)!),
    sub_add, sub_self, sub_zero, ← Gamma_add_one h₇,
    ← Gamma_add_one h₅]
  ring_nf
_ = _ := by
  rw [cast_div h₉ h₁₀, cast_mul]
  repeat' rw [← Gamma_nat_eq_factorial]
  rw [cast_add, cast_sub (by omega), cast_add, cast_one]
```

(b) A formalization in Lean using equational-reasoning style with the `calc` tactic. The sub-proofs $h_1$ to $h_{10}$ are elided.

Fig. 1. Comparison of pen-and-paper-style equational reasoning and a direct translation into a Lean proof

## 2 Challenges of Pen-and-Paper-Style Reasoning in Current Interactive Theorem Provers

Our goal is to enable a more informal, pen-and-paper style, equational reasoning in interactive theorem provers. To understand the inherent challenges with reconstructing formal proofs from this style, we discuss an example implemented in the Lean theorem prover. We will use it to identify concrete technical challenges, which we will then address.

Our motivating example is the proof of the binomial theorem. This is a classical theorem that can be found in many entry-level textbooks, such as [Rotman 2006]. To prove the binomial theorem: $(x+y)^n = \sum_{r=0}^{n} \binom{n}{r} x^r y^{n-r}$, Rotman first establishes the notation for binomial coefficients $\binom{n}{r}$ and proves the proposition that *for all* $0 \leq n$ *and all* $r$ *with* $0 \leq r \leq n$, $\binom{n}{r} = \frac{n!}{r!(n-r)!}$. The proof for the binomial theorem then follows from a corollary. We see the proof of the inductive step of this theorem in Fig. 1.

On the left in Fig. 1a we see the reasoning from Rotman reproduced with pen on paper. As typical for pen-and-paper proofs, the individual steps are not justified in detail. Instead, there is a degree of "contextual knowledge" expected of the reader justifying the steps.

In contrast, on the right in Fig. 1b we see the reasoning in the Lean theorem prover. It is not important to understand the details of this proof, which is a direct translation aiming to preserve the same equational reasoning steps using Lean's `calc` tactic, which enables the justification of each proof step individually.[1] Every line from the pen-and-paper proof translates nicely to the lines

[1]Note that this is not the same proof as in the Lean mathematical library Mathlib [mat 2020], which is not written in pen-and-paper style and uses other reasoning steps and tactics.

highlighted in light blue (2-3, 9-10, 15-16, 19-20, and 22) in the `calc` proof. However, after each such line in the proof, there are multiple additional lines of justifications. For this, we use mostly the `rw` tactic that performs rewriting by replacing in the proof goal the left-hand-side of the referenced equality definition by its right-hand-side. In addition, we use specialized tactics, such as `ring` in line 21 that implements a decison procedure for commutative rings. For brevity, we elided about 15 lines of additional proofs for $h_1$ to $h_{10}$, which justify preconditions of theorems. All this noise makes the proof harder to read and understand, but also harder to write and construct.

A key difference between the pen-and-paper proof and the formalized proof is context: in the informal proof, *we know* that we are dealing with numbers and can reason about them with the rules we *know* are true of them. In the formal proof we have to be explicit about the types of our terms and the theorems that we use. For instance, these informal "numbers" are natural numbers, yet we see terms like $\frac{1}{r}$, which is not a natural number for $r \neq 1$. The informal proof silently uses the fact that we can embed the naturals into the rationals (or reals), and, for example, assumes that expressions with factorials remain in the natural numbers and are thus well-defined. In our formalization, on the other hand, we have to be very explicit about this.[2] Ideally, for a direct translation of such a proof into an ITP, the prover should be able to infer, or at least ingest necessary contextual knowledge, such that subsequent reasoning steps can be performed with little or no further justification.

To understand the key technical challenges that need to be overcome to infer or expose the needed contextual knowledge and to enable pen-and-paper-style equational reasoning in ITPs, we inspect the equational reasoning in the binomial theorem in more detail.

*Choosing a Suitable Automated Rewrite Technique.* To justify the equality (1) from Fig. 1a we break it down into multiple smaller steps:

$$\begin{aligned}
&\frac{n!}{(r-1)!(n-r+1)!} + \frac{n!}{r!(n-r)!} \overset{(1.1)}{=} \frac{n!}{(r-1)!}\frac{1}{(n-r+1)!} + \frac{n!}{r!(n-r)!} \\
&\overset{(1.2)}{=} \frac{n!}{(r-1)!}\frac{1}{(n-r+1)(n-r)!} + \frac{n!}{r!(n-r)!} \overset{(1.3)}{=} \frac{n!}{(r-1)!}\frac{1}{(n-r)!(n-r+1)} + \frac{n!}{r!(n-r)!} \\
&\ldots = \frac{n!}{(r-1)!(n-r)!}\left(\frac{1}{n-r+1} + \frac{1}{r}\right)
\end{aligned}$$

Each step is justified by one equation, such as (1.3) by commutativity of multiplication. However, commutativity and associativity are known to be notoriously tricky to automate in the context of rewriting, as these equations are applicable in both directions, and it is often unclear which direction is beneficial. To overcome this challenge, we use a rewrite technique called *equality saturation*, specifically the egg [Willsey et al. 2021] library, which is suited for rewriting with non-directed equations. Equality saturation computes the congruence closure of a set of equations by growing an equivalence graph (*e-graph*) that represents a growing set of equivalent terms. However, equality saturation is a purely syntax-driven approach and does not naturally capture the semantics of Lean's expression language. Therefore, we need to *encode* Lean expressions faithfully into terms in an e-graph to be able to perform equational rewriting over them.

To capture the explicit reasoning steps of a pen-and-paper proof, we build on our prior work Koehler et al. [2024] where we introduce *guided equality saturation*. This uses equational reasoning steps as intermediate guides, helping the proof of each step using equality saturation. Our prior paper shows how this technique is used to prove simple theorems. However, our prior approach failed specifically on the example of the binomial theorem! This is, among other things, because the proof relies on theorems that have preconditions.

[2]We define a macro to write `n!` for $\Gamma(n+1)$, where $\Gamma$ is a generalization of the factorial to real numbers.

*Syntactic Differences Despite Semantic Equality.* Equality (1.1) follows by rewriting the left term of the addition with the theorem $\frac{a}{b \cdot c} = \frac{a}{b} \cdot \frac{1}{c}$. In Lean, this theorem is written as:

```
theorem div_mul_eq_div_mul_one_div :
  ∀ α [i : DivisionCommMonoid α] (a b c : α), a / (b * c) = a / b * (1 / c)
```

Notably, this theorem is defined over all types $\alpha$ which have a `DivisionCommMonoid` type class instance `i`. As a result, the left-hand side of the theorem: `a / (b * c)` and the target term: `n! / ((r - 1)! * (n - r + 1)!)` do not match syntactically when desugared. Specifically, they differ already when desugaring just the division, with

```
HDiv.hDiv ?α   ?α   ?α   (instHDiv ?α   (DivInvMonoid.toDiv ?α
  DivisionMonoid.toDivInvMonoid ?α (DivisionCommMonoid.toDivisionMonoid ?α ?i)))
```

being the left-hand side of the theorem, where variables preceded by ? represent holes which can be filled by concrete terms. In contrast, the target desugars to:

```
HDiv.hDiv Real Real Real (instHDiv Real (DivInvMonoid.toDiv Real
  Real.instDivInvMonoid))
```

These terms to not match syntactically on the final type class instance. As a result, the equality saturation rewrite technique is not able to apply the theorem directly to solve step (1.1). This is despite the theorem being applicable when *semantically* unifying the terms with the correct choice of `A` and `i`, where by semantic unification we mean Lean's unification algorithm, which underlies its notion of equality. A similar problem occurs when we define the theorem in terms of the function `Real.div` instead of the `/` notation. In that case, the target term would again not match `Real.div` syntactically, despite unifying semantically.

These examples indicate a larger problem of using syntactic rewriting on Lean expressions: the syntax of terms is not sufficient to determine *semantic* equality of terms, and hence insufficient to determine the applicability of theorems. Thus, it is necessary to increase syntactic uniformity of semantically equal terms, and extend equality saturation such that it can perform syntactic conversions between them. Capturing this notion of equality is however complicated by the fact that it is not covered purely by syntactic rewrite rules.

*Ensuring Preconditions of Theorems.* In the previous paragraph, we described the problem of theorems not being applied despite being applicable. The opposite problem, theorems wrongly being applied when they should not, can occur when theorems with conditions are not handled properly. This is relevant in (1.2), which relies on the following theorem:

```
theorem Gamma_add_one : ∀ s, (s ≠ 0) → s! = s * Gamma s
```

Accordingly, rewriting `s!` to `s * Gamma s`, is only valid if `s ≠ 0`. To ensure this condition, a syntactic check is insufficient, as there can be syntactically distinct terms equivalent to 0. Instead, we require a general representation of facts which can be checked during equality saturation. This representation should also allow facts, such as `s ≠ 0`, to be derivable during equality saturation from equivalent facts, such as `s + 0 ≠ 0`. Yet, even then, it is not obvious to decide what even constitutes a condition of a given theorem. For example, in the theorem `div_mul_eq_div_mul_one_div` used in the previous paragraph, the type class argument `i` should be considered a condition, despite appearing in the (desugared) theorem body.

In the following sections, we describe an approach tackling these challenges. Besides focusing on how to handle conditional theorems and *encoding* them for equality saturation, we also describe how we *decode* the information discovered during rewriting to instantiate the conditional theorems in Lean. While we use Lean as an example throughout, and our implementation uses Lean, the ideas should naturally transfer to similar systems like Rocq.

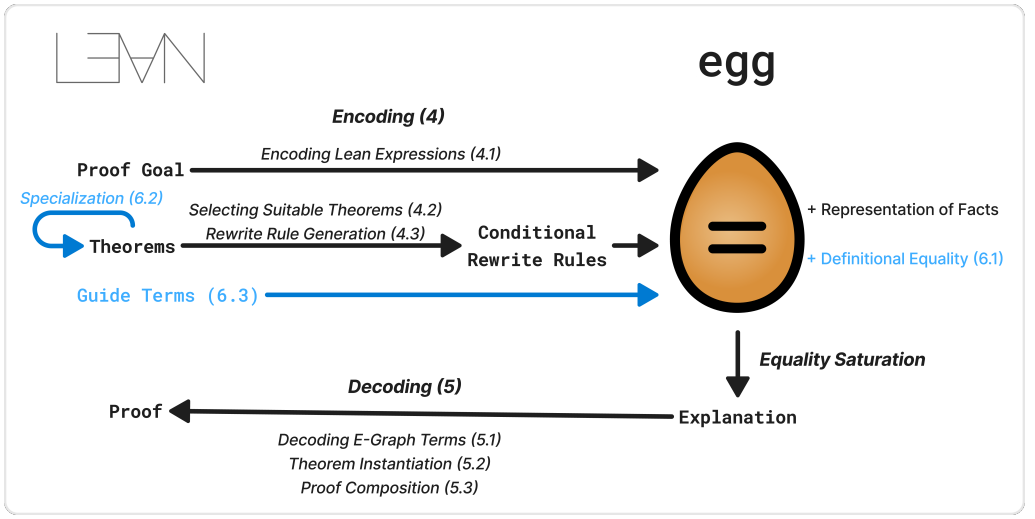



Fig. 2. Overview of our approach: Lean **proof goals** and **theorems** are ***encoded*** into terms and **conditional rewrite rules** in the egg e-graph, which also encodes a representation of facts for reasoning with conditions. After ***equality saturation***, an **explanation** is produced, which is ***decoded*** into a **proof**. Extensions of the core approach are shown in blue. The parenthesized numbers indicate the corresponding sections.

## 3 Overview

Our approach aims at enabling equational reasoning using pen-and-paper style, by automatically leveraging contextual knowledge. To achieve this, we integrate the interactive theorem prover Lean with egg, an efficient implementation of equality saturation, an automated rewriting technique performed over an *e-graph*. The e-graph data structure stores a congruence relation by storing syntactically different, but equivalent terms in the same unique equivalence class (*e-class*). The set of equivalent terms is grown using equality saturation, which relies on *e-matching* to apply rewrite rules directly on the e-graph. A major challenge for encoding Lean expressions into e-graph terms is to ensure that expressions considered equal by Lean a priori are also treated as equal during equality saturation. In Lean's type theory, expressions can be considered *definitionally equal*, even if they are not syntactically equal [Carneiro 2019].[3] This conflicts with e-graphs' syntactic view of equality.

Figure 2 shows an overview of our approach, which consists of three main steps.

- ***Encoding***: A given **proof goal** and **theorems** are encoded as terms and **conditional rewrite rules** and passed to the equality saturation engine egg. This is visualized by the arrows pointing to the right at the top in Fig. 2 and described in Section 4.
- ***Equality Saturation***: The e-graph is initialized with gadgets for a **representation of facts** which enables reasoning with and about the conditions of rewrites. Rewriting is performed, as visualized by the downwards pointing arrow. If successful, it produces an **explanation** consisting of a sequence of rewrite applications.
- ***Decoding***: Finally, the explanation is decoded back into Lean's expression language and used to reconstruct a **proof** of the goal. This is visualized by the arrow pointing to the left and described in Section 5.

[3]Well-known examples of definitional equality rules include $\beta$- and $\eta$-reduction.

We discuss in Section 4.1 how we encode Lean expressions into e-graph terms. In Section 4.2 we discuss how we select suitable theorems for which we can construct sensible rewrite rules. In Section 4.3 we show how these theorems are encoded as rewrite rules.

Once we have found a rewrite sequence using equality saturation, we decode it back into a Lean proof. For this, we first decode e-graph terms back into Lean expressions, which we discuss in Section 5.1. To justify the equivalences between these decoded expressions, we instantiate the theorems previously encoded as rewrite rules, as discussed in Section 5.2. For theorems with preconditions, this involves querying the e-graph for further justifications. Finally, we discuss in Section 5.3 how we compose individual proof steps into a complete proof that is verified by Lean.

The blue parts of Fig. 2 represent extensions which are not necessary to the core procedure, but useful in practice. In Section 6.1, we discuss how to improve our handling of definitional equality by encoding specific definitional equality rules in the e-graph. Section 6.2 discusses how we increase the number of suitable theorems by specializing theorems that we cannot handle in their full generality. And, finally, in Section 6.3 we discuss how we build upon the ideas of [Koehler et al. 2024] to integrate intermediate reasoning steps into our approach.

## 4 Encoding

To leverage e-graphs and equality saturation for conditional rewriting on equational proofs, we need to define a suitable encoding of theorems, and thus necessarily, of terms. We discuss the principles of our encoding for Lean, which should transfer to similar systems, like Rocq.

### 4.1 Encoding Lean Expressions

Terms in Lean are elaborated to terms of a $\lambda$-calculus based on an extension of the Calculus of Inductive Constructions [Coquand and Huet 1988; Coquand and Paulin 1988]. Lean's specific theory was first described in [Carneiro 2019], with subsequent changes in [Carneiro 2024; Ullrich 2023]. Figure 3a shows a slightly simplified syntax of this expression language.

$$
\begin{aligned}
e ::= {} & \mathsf{app}\ e\ e \mid \mathsf{lam}\ e\ e \mid \mathsf{forall}\ e\ e \\
& \mid \mathsf{let}\ e\ e\ e \mid \mathsf{bvar}\ n \mid \mathsf{fvar}\ i \\
& \mid \mathsf{mvar}\ i \mid \mathsf{const}\ i\ \bar{\ell} \mid \mathsf{sort}\ \ell \\
& \mid \mathsf{lit}\ n \mid \mathsf{proj}\ i\ n\ e \\
\ell ::= {} & \mathsf{zero} \mid \mathsf{succ}\ \ell \mid \ldots \\
& \text{where } n \in \mathbb{N}, i \in \mathcal{I} \\
& \text{and } \bar{\ell} \text{ denotes a list}
\end{aligned}
$$

(a) A simplified view of Lean's expressions.

$$
\begin{aligned}
t ::= {} & \mathsf{app}\ t\ t \mid \mathsf{lam}\ t\ t \mid \mathsf{forall}\ t\ t \\
& \mid \mathsf{bvar}\ t \mid \mathsf{fvar}\ t \mid \mathsf{mvar}\ t \\
& \mid \mathsf{const}\ \bar{t} \mid \mathsf{sort}\ t \mid \mathsf{lit}\ t \\
& \mid \mathsf{zero} \mid \mathsf{succ}\ t \mid \ldots \\
& \mid \mathsf{proof}\ t \mid \mathsf{inst}\ t \mid \mathsf{eq}\ t\ t \\
& \mid n \mid i \\
& \text{where } n \in \mathbb{N}, i \in \mathcal{I}
\end{aligned}
$$

(b) The core of our e-graph encoding language.

Fig. 3. Lean's core expression language and our encoding for representing it in e-graphs

The typical constructs of $\lambda$-calculus, such as application, $\lambda$-abstraction, $\forall$-quantification, and let-abstraction, are represented as expected. As Lean uses a locally-nameless representation [Charguéraud 2012], bound variables (bvars) are represented using de Bruijn indices. The first argument of binders declares the *type* of the bound variable. Free variables are represented as named fvars over some ambient set of identifiers $\mathcal{I}$. Metavariables (mvars) represent named holes in expressions, to be filled by further elaboration steps. Constants (consts) denote named definitions, like Bool or Nat.add_comm, known to Lean's environment. For universe polymorphic constants like

the product type Prod, the const construct includes a list $\bar{\ell}$ of universe levels. The sort construct represents type universes of a given level, where sort zero is the universe of propositions which we also denote by Prop. As Lean's type theory relies on an infinite hierarchy of type universes with polymorphism, universe levels require an entire language $\ell$ of their own. For simplicity, we omit most considerations of universe levels in this paper. Finally, Lean's expression language contains two "internalizations" introduced in [Ullrich 2023] to improve the computational overhead of common constructions. First, the lit construct represents natural numbers directly as literals to avoid the overhead of the usual unary encoding $S\ (\ldots\ (S\ 0))$.[4] Second, proj $i\ n\ e$ represents the application of the $n$th projection of the structure type named $i$ to the expression $e$, which avoids the overhead of the usual representation based on calling $i$'s recursor.

For convenience, we denote sequences of (nested) applications with $\overline{\text{app}}$. When an expression is irrelevant or can be inferred from context we sometimes write _, or ... if there are multiple such expressions. Analogously, we may omit type annotations of bound variables and write $\lambda x, e$ instead of $\lambda x : T, e$.

*E-Graph Terms.* Our term encoding, shown in Figure 3b, is largely based on Lean's expression language. However, it has slight modifications to increase syntactic uniformity by representing some definitionally-equal terms using the same syntax: First, we eliminate let and proj, as they can always be reduced to other constructs. Similarly, we could remove lit, yet we keep it to leverage its performance benefits. Second, we add constructs for proof erasure and type class instance erasure. Proof erasure enables us to satisfy Lean's proof irrelevance rule, which states that any two proofs $p_1$ and $p_2$ of the same proposition P are definitionally equal [Carneiro 2024]. We achieve proof irrelevance in an e-graph syntactically by encoding both $p_1$ and $p_2$ as the same term proof P. Type classes are not built into Lean's core expression language, but they are a common occurrence in practice. Thus, we also introduce type class instance erasure to create syntactic equality for different instances $c_1$ and $c_2$ of the same type class C. While there does not exist a rule in Lean's type theory which states that all instances of a given type class are definitionally equal, they almost always are in practice (for example, see [Wieser 2023]). And as our procedure is safeguarded by Lean's proof checking, we can safely apply this heuristic without having to worry about unsoundness. Thus, we encode both $c_1$ and $c_2$ using the syntax inst C. The final construct we add to the term language is the internalization eq $t_1\ t_2$ of equivalence ($=$ or $\leftrightarrow$) between terms $t_1$ and $t_2$. This construct is used for equivalence reflection in the e-graph, as discussed in Section 4.3. Finally, we note that our term language collapses expressions, universe levels, natural numbers, and identifiers into a single language for implementation reasons.

*Normalization and Encoding Function.* We encode Lean expressions into e-graph terms in two steps. First, a normalization function $\|\cdot\|$ eliminates the let and proj constructs. Then, we map the resulting expression to an e-graph term according to the encoding function $[\![\cdot]\!]$.

*Definition 4.1 (Normalization and Encoding).* Let $e$ be a Lean expression. Then we define its normalization $\|e\|$ as:

$$\|e\| := \begin{cases} \|\zeta(e)\| & \text{if } e = \text{let } e_1\ e_2\ e_3 \\ \overline{\text{app}}\ \|\pi_n^i\|\ \ldots\ \|e'\| & \text{if } e = \text{proj } i\ n\ e' \\ \text{ctr}\ \|e_1\|\ \|e_2\| & \text{if } e = \text{ctr } e_1\ e_2, \text{where ctr} \in \{\text{app}, \text{lam}, \text{forall}\} \\ e & \text{otherwise} \end{cases}$$

[4]The lit construct is also used for string literals, which we omit here for simplicity.

Given a normalized expression $e$, the encoding $[\![e]\!]$ is defined as:

$$[\![e]\!] := \begin{cases} \mathsf{inst}\ [\![C]\!] & \text{if } e : C \text{ where } C \text{ is a type class} \\ \mathsf{proof}\ [\![P]\!] & \text{if } e : P : \mathsf{Prop} \\ \mathsf{eq}\ [\![l]\!]\ [\![r]\!] & \text{if } e = \overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Eq}\ \_)\ \_\ l\ r \\ \mathsf{eq}\ [\![l]\!]\ [\![r]\!] & \text{if } e = \overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Iff})\ l\ r \\ \mathsf{ctr}\ [\![e_1]\!]\ [\![e_2]\!] & \text{if } e = \mathsf{ctr}\ e_1\ e_2, \text{where } \mathsf{ctr} \in \{\mathsf{app}, \mathsf{lam}, \mathsf{forall}\} \\ e & \text{otherwise} \end{cases}$$

We write $\zeta(e)$ for the $\zeta$-reduction of $e$ which inlines let-bound expressions, and $\pi^i_n$ to denote the $n$th projection of the structure type named $i$ as in [Ullrich 2023]. Accordingly, $\|e\|$ is always definitionally equal to $e$. Subsequently, we generally assume expressions to be normalized, and thus simply write $[\![e]\!]$ instead of $[\![\ \|e\|\ ]\!]$. Also, we sometimes use $[\![\cdot]\!]$ on sets of expressions or on terms of Lean's user-facing language. The latter is to be interpreted as first elaborating the term and calling $[\![\cdot]\!]$ on the resulting expression.

*Example 4.2.* Let $e$ be the expression representing $\mathsf{let}\ \mathsf{x} := 0;\ \mathsf{x} = 1$.
Elaborated, $e$ becomes $\mathsf{let}\ (\mathsf{const}\ \mathsf{Nat})\ (\mathsf{lit}\ 0)\ (\overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Eq}\ \_)\ (\mathsf{const}\ \mathsf{Nat})\ (\mathsf{bvar}\ 0))\ (\mathsf{lit}\ 1)$. Then $e$ normalizes to $\|e\| = \|\zeta(e)\| = \overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Eq}\ \_)\ (\mathsf{const}\ \mathsf{Nat})\ (\mathsf{lit}\ 0)\ (\mathsf{lit}\ 1)$. This normalized expression encodes to $[\![\|e\|]\!] = \mathsf{eq}\ [\![\mathsf{lit}\ 0]\!]\ [\![\mathsf{lit}\ 1]\!] = \mathsf{eq}\ (\mathsf{lit}\ 0)\ (\mathsf{lit}\ 1)$.

*Encoding Patterns.* The encoding function $[\![\cdot]\!]$ turns Lean expressions into e-graph *terms*. However, when constructing rewrite rules we need to turn expressions into e-graph *patterns*. The e-graph pattern language extends the term language with named *pattern variables*, denoted $?i$ with $i \in \mathcal{I}$. When encoding the body $e$ of a theorem $\forall \bar{x}, e$ as a rewrite rule, the quantified variables $\bar{x}$ are encoded as pattern variables. We use "quantified variables" for $x \in \bar{x}$ as an opaque term to abstract over how they are represented in Lean's expression language. We define an extension of $[\![\cdot]\!]$ which encodes expressions as patterns given the quantified variables $\bar{x}$:

*Definition 4.3 (Encoding for Patterns).* For all $x \in \bar{x}$, we extend our encoding $[\![\cdot]\!]$ with the following case:

$$[\![x]\!] := ?i_x, \text{ where } i_x \text{ is a unique identifier for } x$$

We also apply additional normalization steps to theorems' expressions to better deal with the fact that terms appearing in rewrite rules cannot themselves be rewritten during equality saturation. Due to this restriction, rewrite rules' terms which would be subject to definitional reductions like $\beta$- and $\eta$-reduction or natural number arithmetic can never be reduced. Thus, we perform reductions beforehand using an extended normalization function:

*Definition 4.4 (Extended Normalization).* We write $\|e\|_{\rightsquigarrow}$ for the expression obtained from $e$ by $\|\cdot\|$, while also applying $\beta$- and $\eta$-reduction on all applicable subterms, and evaluating internalized natural number operators on lits.

As we generally assume expressions to be normalized, we usually write $[\![\cdot]\!]$ instead of $[\![\|\cdot\|_{\rightsquigarrow}]\!]$.

*Example 4.5.* Let $e$ be the expression representing $(\lambda\ \mathsf{x} \Rightarrow \mathsf{x} + y)\ y$, quantified over variable $y$. That is $e := \mathsf{app}\ (\mathsf{lam}\ (\mathsf{const}\ \mathsf{Nat})\ (\overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Nat.add})\ (\mathsf{bvar}\ 0)\ y)\ y$. By $\beta$-reduction, it normalizes to $\|e\|_{\rightsquigarrow} = \overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Nat.add})\ y\ y$. Finally, this normalized expression encodes to $[\![\|e\|_{\rightsquigarrow}]\!] = \overline{\mathsf{app}}\ (\mathsf{const}\ \mathsf{Nat.add})\ ?\mathsf{y}\ ?\mathsf{y}$.

*Transferability.* While the translations given above are expressed in terms of Lean's expression language, the same principles could be transferred to other ITPs, like Rocq. Lean's expression language is sufficiently similar to that of Rocq [Sozeau et al. 2025], such that the following constructs can be translated as above: app, lam, forall, let, bvar, const, sort, proj. Rocq's constructs for inductive types and their constructors can be translated as we do for constants. The pattern matching construct can be translated opaquely, as we currently do for recursors in Lean (which are represented as consts), as we do not yet implement any of their reduction rules. Similar reasoning applies to Rocq's (co-)fixed point constructs.

More relevant differences appear when considering definitional equality. For example, our Lean-based encoding erases proofs to capture proof irrelevance. In Rocq, however, proof irrelevance is not built-in, which necessitates the omission of proof erasure. A more difficult change would be the handling of type class instances. In Lean, type class instances are expected to be unique up to definitional equality. This is not the case in Rocq. Thus, type class instance erasure should be omitted. This, however, causes problems during e-matching and proof reconstruction, for which we are not currently aware of an obvious solution.

### 4.2 Selecting Suitable Theorems

Just as Lean and e-graphs use different languages, they also use different objects for rewriting. In Lean, one uses equational theorems, whereas an e-graph requires rewrite rules. In the following, we introduce a heuristic approach to select and encode suitable equational theorems into rewrite rules. The principle guiding this approach is to construct rewrite rules in a way which makes them amenable to proof reconstruction (see Section 5). In this section, we start by deriving properties required of theorems to be sensibly encoded as rewrite rules. Based on these properties, Section 4.3 describes the actual encoding.

For the rest of this section, let $T$ denote a theorem of the form $\forall \bar{x}, L \sim R$, where $\sim$ represents $=$ or $\leftrightarrow$[5]. We consider only the properties needed to encode a rewrite rule for $T$ in the forward direction, that is, from $L$ to $R$. The reverse direction is analogous. We write $\mathit{type}(e)$ for the type of an expression $e$, which we also apply to sets of expressions. By $e_1 \sqsubseteq e_2$ we denote that $e_1$ is a subterm of $e_2$, with $e_1 \sqsubset e_2$ also requiring $e_1 \neq e_2$. Additionally, we write $\mathit{vars}(p)$ for the set of quantified variables $x \in \bar{x}$ for which a corresponding $?i_x$ appears in the e-graph pattern $p$, and extend this to sets as: $\mathit{vars}(P) := \bigcup_{p \in P}(\mathit{vars}(p))$. Note that generally $x \sqsubseteq e \nleftrightarrow x \in \mathit{vars}(\llbracket e \rrbracket)$ due to aspects of encoding like proof erasure.

*Basic Requirements for Rewrite Rules.* The simplest form of rewrite rule $p_1 \Rightarrow p_2$ matches a given pattern $p_1$, producing a substitution $\sigma$, and equates the terms $\sigma(p_1)$ and $\sigma(p_2)$. However, this only works if the pattern variables in $p_1$ are a superset of the pattern variables in $p_2$. If we naively translate $T$ to a rewrite rule $\llbracket L \rrbracket \Rightarrow \llbracket R \rrbracket$, this translates to the requirement:

$$\mathfrak{R}_1 : \mathit{vars}(\llbracket R \rrbracket) \subseteq \mathit{vars}(\llbracket L \rrbracket)$$

A common, and practical, restriction on rewrite rules is also to disallow $\llbracket L \rrbracket$ from matching *every* possible term. This is the case when $L$ is a quantified variable, which translates to the following requirement:

$$\mathfrak{R}_2 : L \notin \bar{x}$$

[5]Congruence closure and E-graphs work for a more general setting, for any equivalence relation. We could use Lean's quotient types to reason about these with equality, but this is outside of the scope of this paper.

*Recoverability of Variables by Type Inference.* All further requirements are to ensure that $T$ is suitable for proof reconstruction. This is the case if for each rewrite from term $[\![e_1]\!]$ to $[\![e_2]\!]$ by $T$'s rewrite rule, we can recover expressions $\bar{e}$ for all $\bar{x}$, such that $T(\bar{e})$ unifies with $e_1 \sim e_2$. In the naive translation of $T$ as a rewrite rule (as $[\![L]\!] \Rightarrow [\![R]\!]$), the only variables we can recover are $vars([\![L]\!])$, which are matched during equality saturation. This is insufficient for two reasons. First, the encoding function $[\![\cdot]\!]$ does not preserve all quantified variables as pattern variables. For example, quantified variables might be erased as part of proof erasure. Such variables are then not assigned by matching during equality saturation and cannot be recovered. This problem is addressed in subsequent sections. Second, not all quantified variables of $T$ need appear in $L$ in the first place. For example, in the trivial theorem $\forall(\alpha : \mathsf{Type})(l : \mathsf{List}\ \alpha), l = l$, the left-hand side $l$ does not reference $\alpha$, which therefore does not appear in the rewrite rule $?l \Rightarrow ?l$. In this example, we can however recover $\alpha$ by type inference on $l$. Thus, we consider a variable to be recoverable if it is subject to pattern matching, *or* can be recovered from other variables by type inference. Variables recoverable by type inference from an initial set $X$ of variables are captured by the following definition.

*Definition 4.6 (Type Closure).* We define the "type closure" $\omega$ of variables $X$ to be the smallest fixed point satisfying $\omega(X) = X \cup \{x \mid \exists\, y \in \omega(X),\ x \in vars(type(y))\}$.

We can then state the requirement that all of $T$'s variables be recoverable as:

$$\mathfrak{R}_3 : \bar{x} \subseteq \omega(vars([\![L]\!]))$$

*Handling Propositional Conditions.* A common case which is not allowed under the above requirements are theorems with propositional conditions. For example, the equation $\frac{x}{x} = 1$ holds only if $x \neq 0$. Thus, the corresponding theorem is $\forall(x : \mathbb{R})(h : x \neq 0), \frac{x}{x} = 1$, where $h$ violates requirement $\mathfrak{R}_3$. To rectify this, we handle propositional variables specially.

Variables for propositional conditions differ from other quantified variables in that we do not expect them to be recoverable from the theorem's body. For example, in the theorem above, we do not expect the proof $h : x \neq 0$ to be recoverable from the variables in $\frac{x}{x} = 1$. However, to ensure that a proof for a given propositional condition *can* be found during proof reconstruction, we check that the condition is satisfied before applying a rewrite during equality saturation. For example, we only apply the rewrite for $\frac{x}{x} = 1$ after checking that $x \neq 0$. For this purpose, we endow theorem $T$ with a set of propositional conditions $\mathcal{P}(T)$.

*Definition 4.7 (Propositional Conditions).* Let $\mathcal{P}(T)$ be the set of variables $x \in \bar{x}$, where:

(1) $type(x) : \mathsf{Prop}$, and
(2) $(x \sqsubseteq L) \rightarrow \exists\, g, (x \sqsubset g \sqsubseteq L) \wedge (type(g) : \mathsf{Prop})$

The set $\mathcal{P}(T)$ captures those propositional quantified variables which we cannot recover by pattern matching on $[\![L]\!]$ or type inference. That is, a variable $x$ is a propositional condition if (1) it is a proof and (2) if it *does* appear in $L$, then it appears nested in another proof term $g$. Propositional variables appearing nested inside other proof terms in $L$ cannot be recovered by pattern matching, as our encoding erases proof terms and only yields proof $[\![type(g)]\!]$ – thus erasing any reference to $x$ or its type. Variables which are not nested can be recovered by pattern matching and are thus excluded from $\mathcal{P}(T)$. We denote the set of these variables recoverable by pattern matching, satisfying (1) but not (2), as $\tilde{\mathcal{P}}(T)$.

For theorems $T$ with propositional conditions, we construct a rewrite rule such that it matches both $[\![L]\!]$ and all $\mathsf{P} \in [\![\mathit{type}(\mathcal{P}(T))]\!]$. Thus, we rephrase our restrictions as:[6]

$$\mathfrak{R}_1 : \mathit{vars}([\![R]\!]) \subseteq \mathit{vars}([\![L]\!]) \cup \mathit{vars}([\![\mathit{type}(\mathcal{P}(T))]\!])$$
$$\mathfrak{R}_2 : (\{L\} \cup \mathit{type}(\mathcal{P}(T))) \setminus \bar{x} \neq \emptyset$$
$$\mathfrak{R}_3 : \bar{x} \subseteq \omega(\mathit{vars}([\![L]\!]) \cup \mathcal{P}(T)) \cup \tilde{\mathcal{P}}(T)$$

*Handling Type Class Conditions.* Much like propositional conditions, type class instances can be considered as conditions. Unlike propositions, type class instances usually *do* appear in the body of a theorem. For example, consider the theorem $\forall\ \alpha\ (i : \mathsf{Add}\ \alpha)(x : \alpha), x + x = x + x$, where Add is a type class which requires $\alpha$ to have a $+$ operator. Here, the instance $i$ appears in the body, as $x + x$ is syntactic sugar for $\mathsf{HAdd.hAdd}\ \alpha\ \alpha\ \alpha\ (\mathsf{instHAdd}\ \alpha\ i)\ x\ x$. However, we erase the instance and encode the subterm $\mathsf{instHAdd}\ \alpha\ i$ as the term $\mathsf{inst}\ [\![\mathsf{HAdd}\ \alpha\ \alpha\ \alpha]\!]$. This way, the encoded term only requires matching terms to have an $\mathsf{HAdd}\ \alpha\ \alpha\ \alpha$ instance, but not necessarily an $\mathsf{Add}\ \alpha$ instance as required by the theorem.[7] To ensure correctness, we need to check that an instance for $\mathsf{Add}\ \alpha$ can be synthesized before applying the theorem's rewrite rule during equality saturation. For this purpose, we endow theorem $T$ with a set of type class conditions, analogous to how we defined the set of propositional conditions.

*Definition 4.8 (Type Class Conditions).* Let $\mathcal{C}(T)$ be the set of all variables $x \in \bar{x}$, where:

(1) $\mathit{type}(x)$ is a type class, and
(2) $(x \sqsubseteq L) \to \exists\, j, (x \sqsubset j \sqsubseteq L) \wedge (\mathit{type}(j)$ is a type class$)$

Again, Condition (2) excludes instances which appear non-nested in the body, as these *can* be recovered by matching. We denote this set of variables, satisfying (1) but not (2) as $\tilde{\mathcal{C}}(T)$. The restrictions we impose for type class conditions differ from those of propositional conditions, as we do not match on type class conditions during equality saturation. Thus, only $\mathfrak{R}_3$ is extended while $\mathfrak{R}_1$ and $\mathfrak{R}_2$ remain unchanged. We add a fourth restriction to ensure that all variables appearing in the type class conditions are resolved by matching:

$$\mathfrak{R}_3 : \bar{x} \subseteq \omega(\mathit{vars}([\![L]\!]) \cup \mathcal{P}(T)) \cup \tilde{\mathcal{P}}(T) \cup \mathcal{C}(T) \cup \tilde{\mathcal{C}}(T)$$
$$\mathfrak{R}_4 : \mathit{vars}([\![\mathit{type}(\mathcal{C}(T))]\!]) \subseteq \mathit{vars}([\![L]\!]) \cup \mathit{vars}([\![\mathit{type}(\mathcal{P}(T))]\!])$$

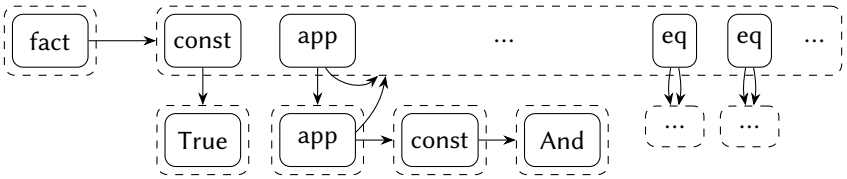
Restriction $\mathfrak{R}_4$ is necessary to ensure that we obtain complete terms for type classes and can run synthesis on them *during* equality saturation.

### 4.3 Encoding Theorems as Conditional Rewrites

With the requirements for suitable theorems established, we now describe how to construct their conditional rewrite rules. The main challenge is checking satisfaction of propositional conditions during equality saturation. We cover this by addressing two questions. (1) How do we represent proven propositions in the e-graph? (2) How do we check satisfaction of a propositional condition based on the previous representation.

[6] In $\mathfrak{R}_3$ we write just $\mathcal{P}(T)$, as $\omega(\mathcal{P}(T)) = \omega(\mathcal{P}(T) \cup \mathit{vars}([\![\mathit{type}(\mathcal{P}(T))]\!])$.
[7] This is similar to the issue of proof variables appearing nested as covered by Condition 2 of $\mathcal{P}(T)$.

*Representation of Facts.* We call proven propositions "facts" and represent them as special terms in the e-graph using the gadget depicted below:




At the heart of this gadget lies an e-class which contains the term const True and is used to represent facts. The e-class is depicted as the dashed box at the top right, which we refer to as $\top$. This representation of facts follows from the theorem $\forall P : \text{Prop}, P \leftrightarrow (P = \text{True})$ also used in [Bourgeat 2023]. To mark a proposition as proven, it suffices to add it to $\top$. A special class of facts which we mark in this way is the conjunction of arbitrary facts. That is, for any given facts $P_1$ and $P_2$ we also make $P_1 \wedge P_2$ a fact by adding an e-node representing $\top \wedge \top$ to $\top$, as shown above.[8] We also equip $\top$ with e-nodes for explicitly reflecting equalities which are otherwise implicit in the e-graph. Namely, for each e-class $c$ in the e-graph, we add an e-node eq $c$ $c$ to $\top$. Thus, for any equivalent terms $t_1$ and $t_2$, $\top$ represents eq $t_1$ $t_2$. Finally, we introduce a fact construct used to *syntactically* mark terms which are facts.[9] Specifically, we maintain an e-class containing the single e-node fact $\top$. Thus, for any proposition $P$, if the e-graph contains fact $[\![P]\!]$, then $P$ must be a fact.

*Conditional Rewrite Rules.* Our encoding of theorems without conditions is straightforward. Let $T : \forall \bar{x}, L \sim R$ be a theorem satisfying the conditions described in Section 4.2. If $\mathcal{P}(T) = \mathcal{C}(T) = \emptyset$, then we construct a corresponding rewrite rule simply as $[\![L]\!] \Rightarrow [\![R]\!]$.

*Propositional Conditions.* If $\mathcal{P}(T) \neq \emptyset$, then a rewrite for $T$ must check for satisfaction of all $P \in \mathit{type}(\mathcal{P}(T))$. We check for satisfaction of $P$ syntactically by matching the pattern fact $[\![P]\!]$ for each $P$. Thus, a rewrite for $T$ must match both $[\![L]\!]$ and fact $[\![P]\!]$ for each $P \in \mathit{type}(\mathcal{P}(T))$, and merge the e-classes found for $L$ and $R$. E-matching on a *set* of patterns is usually covered by "multipatterns". However, the egg framework does not support proof producing multipatterns. Thus, we exploit the properties of our gadget and construct the single pattern: fact $[\![L = L \wedge \bigwedge_{P \in \mathit{type}(\mathcal{P}(T))} P]\!]$. The subterm $L = L$ is used solely to e-match on $L$, and relies on the fact that for any term $L$ in the e-graph, $[\![L = L]\!] \in \top$. Thus, the pattern matches $L$ and all propositional conditions of $T$, but only if the conditions are facts.

*Type Class Conditions.* If $\mathcal{C}(T) \neq \emptyset$, then we also need to check that an instance can be synthesized for each $C \in \mathit{type}(\mathcal{C}(T))$. For this, we call Lean's type class synthesis during equality saturation. Specifically, after e-matching $T$'s rewrite rule's pattern we obtain a substitution $\sigma$. Restriction $\mathfrak{R}_4$ ensures that all quantified variables appearing in each $C$ are covered by $\sigma$. Thus, $\sigma([\![C]\!])$ is guaranteed to yield an e-node. As we cannot run type class synthesis on an *e-node*, we must first obtain a concrete *term* represented by the e-node. We arbitrarily choose the e-node's representative which we denote $\mathit{repr}(\sigma([\![C]\!]))$. This choice relies on the heuristic that $C$ represents a type, which, aside from propositions and dependent types, are usually the only members of their e-class. To finally check the synthesis condition, we decode $\mathit{repr}(\sigma([\![C]\!]))$ into a Lean expression and run Lean's type class synthesis.

[8] More precisely, this is app (app (const And) $\top$) $\top$ above.
[9] We omitted the fact construct from the syntax in Figure 3b, as it is only relevant here.

Based on these approaches, we define the conditional rewrite rule for a given theorem.

*Definition 4.9 (Conditional Rewrite Rule).* A conditional rewrite rule "$M$ if $G \Rightarrow L = R$", consists of the following objects:

- $M$ is a pattern that we match on, yielding a substitution $\sigma$ over $vars(M)$.
- $G$ is a decidable proposition over $\sigma$ which guards the application of the rewrite.
- $L$ and $R$ are patterns that will be equated under $\sigma$ if $G$ holds.

For a theorem $T : \forall \bar{x}, L \sim R$ satisfying the restrictions $\mathfrak{R}$, we generate the conditional rewrite rule $M$ if $G \Rightarrow [\![L]\!] = [\![R]\!]$, with:

- $M :=$ fact $[\![L = L \wedge \bigwedge_{P \in type(\mathcal{P}(T))} P]\!]$, and
- $G(\sigma) := \forall C \in type(\mathcal{C}(T)), synthesizable(repr(\sigma([\![C]\!])))$

Note that $M$ contains the propositional conditions and $G$ represents the type class conditions. By *synthesizable* we denote the predicate of a given term having a type class instance, which we decide by calling Lean's type class synthesis.

*Corollaries.* Our approach to conditional rewriting has several consequences. First, as facts are terms in the e-graph, they can be rewritten. Thus, if a rewrite requires condition $P$, but we only have fact $P'$, then a theorem like $P \leftrightarrow P'$ can derive $P$ during equality saturation. However, it is possible that a condition which would be provable fails to become a fact simply because it is not a term in the e-graph. Second, as propositions can be reasoned about equationally as $P =$ True, we can extend the set of allowed theorems to any of the form $\forall \bar{x}, P$ for $P :$ Prop. When $P$ is not an equivalence, we interpret it as $\forall \bar{x}, P =$ True. In particular, this means that ground facts like $0 < \pi$ are added to the e-graph by the corresponding ground rewrite rule $[\![\text{True}]\!] \Rightarrow [\![0 < \pi]\!]$. Finally, the explicit equality construct eq entails that equivalence of terms cannot purely be checked by comparing their e-classes. This follows as we reflect equivalences from e-classes to eq facts, but not vice versa. Thus, equivalence of terms $t_1$ and $t_2$ must be checked by eq $[\![t_1]\!]\ [\![t_2]\!] \in \top$. This is contrary to [Bourgeat 2023] who "materializes" eq nodes into e-node equivalences via $\forall\ x\ y, ((x = y) = \text{True}) \rightarrow x = y$. This does not suffice for our setting, as we require explicit eq nodes for e-matching of facts.

## 5 Decoding

As a proof checker, Lean only deems a theorem proven if a well-typed proof term is provided. Thus, to use equivalences discovered by egg, we must reconstruct Lean proof terms from egg "explanations". Explanations are proof witnesses for e-graph equivalences.

*Definition 5.1 (Explanation).* An *explanation* for an equivalence between terms $t_1$ and $t_n$ is a sequence $t_1, j_1, t_2, j_2, ..., j_{n-1}, t_n$, where each term $t_i$ is equivalent to $t_{i+1}$ according to justification $j_i$. A justification $j_i$ is a triple $(r_i, d_i, p_i)$ of a rewrite rule (identifier) $r_i$, a rewrite direction $d_i$, and a subterm position $p_i$ in $t_i$ at which the rewrite is applied.

We reconstruct a proof term for a given explanation in three steps. (1) We decode the e-graph terms $t_i$ into Lean expressions $e_i$. (2) We instantiate the theorems corresponding to the rewrite rules $r_i$, such that they justify the equality between the subterms of $e_i$ and $e_{i+1}$ at position $p_i$. (3) We extend the previously instantiated theorems to proofs of the full equalities $e_i = e_{i+1}$ and combine them by transitivity to obtain the final proof.

### 5.1 Decoding E-Graph Terms

The encoding of Lean expressions to e-graph terms via $[\![\cdot]\!]$ erases some subterms. Thus, we define a decoding function $(\!|\cdot|\!)$ from e-graph terms to Lean expressions, which construct expressions containing holes (mvars), which are filled in by later steps of proof reconstruction.

*Definition 5.2 (E-Graph Term Decoding).* Let $t$ be an e-graph term. Then, we define the decoding function $(\!|\cdot|\!)$ to a Lean expression as follows:

$$(\!|t|\!) := \begin{cases} \text{mvar } i & \text{if } t = \text{inst } C, \text{ with fresh } i \text{ and mvar } i : (\!|C|\!) \\ \text{mvar } i & \text{if } t = \text{proof } P, \text{ with fresh } i \text{ and mvar } i : (\!|P|\!) \\ \overline{\text{app}}\ (\text{const Iff})\ (\!|t_1|\!)\ (\!|t_2|\!) & \text{if } t = \text{eq } t_1\ t_2 \text{ and } \mathit{type}(t_1) : \text{Prop} \\ \overline{\text{app}}\ (\text{const Eq } \_)\ \_\ (\!|t_1|\!)\ (\!|t_2|\!) & \text{if } t = \text{eq } t_1\ t_2 \text{ and not } \mathit{type}(t_1) : \text{Prop} \\ \text{ctr}\ (\!|t_1|\!)\ (\!|t_2|\!) & \text{if } t = \text{ctr } t_1\ t_2, \text{where ctr} \in \{\text{app}, \text{lam}, \text{forall}\} \\ t & \text{otherwise} \end{cases}$$

Both type class instances and proof terms are erased during encoding and are turned into (typed) holes. Equivalences (eq) are turned into $\leftrightarrow$ or $=$, depending on the type of their terms. This choice is heuristic, as propositions need not be related by $\leftrightarrow$ instead of $=$, even if it is idiomatic. In our implementation, we backtrack when this choice was incorrect.

### 5.2 Theorem Instantiation

Let $t, (r, d, p), t'$ be one step in an explanation, which rewrites the subterm $u$ of $t$ to the subterm $u'$ of $t'$ by rewrite rule $r$, derived from theorem $T : \forall \bar{x}, L \sim R$. We consider here how to construct a proof for the equivalence of the subterms $u$ and $u'$ by instantiating $T$.

Let $s := (\!|u|\!)$ and $s' := (\!|u'|\!)$. To instantiate $T$ such that it produces a proof of $s \sim s'$, we need to assign all variables in $\bar{x}$ with expressions $\bar{a}$, such that $T\ \bar{a} : s \sim s'$. We assign these variables in steps, using the requirements $\mathfrak{R}$ laid out in Section 4.2. Specifically, by $\mathfrak{R}_3$ we know that $\bar{x} \subseteq \omega(\mathit{vars}([\![L]\!]) \cup \mathcal{P}(T)) \cup \tilde{\mathcal{P}}(T) \cup \mathcal{C}(T) \cup \tilde{\mathcal{C}}(T)$. Thus, we can construct an assignment of each variable $x \in \bar{x}$ depending on which of the following cases it belongs to.

(1) If $x$ appears directly in the rule's left-hand side, $x \in \mathit{vars}([\![L]\!])$, then we obtain an assignment for $x$ by unification of $L$ and $s$. They unify as $[\![L]\!]$ matches $[\![s]\!]$ syntactically according to the explanation and by our construction of rewrite rules.

(2) If $x$ is a variable for a propositional condition, $x \in \mathcal{P}(T)$, then we first reconstruct all variables $y$ appearing in the type of $x$. By our construction of rewrite rules, in particular the pattern $M$, all propositional conditions are e-matched during equality saturation, thus producing an assignment for $y$. However, the explanation term $t$ only captures an assignment for the pattern $[\![L]\!]$. Thus, if $y$ does not appear in $\mathit{vars}([\![L]\!])$, we rely on a record of which e-classes matched against which variables during equality saturation. Using this record, we obtain a term for $y$ by querying the e-graph for a representative term corresponding to the matched e-class. With all variables $y$ in the type of $x$ reconstructed, we now denote the type of $x$ as $P$, and reconstruct a proof for it, as follows. By our construction of rewrite rules, we must have matched fact $[\![P]\!]$ during equality saturation.We can therefore obtain an explanation and decode a proof of $P = \text{True}$, and assign this to $x$.

(3) If $x \in \omega(\mathit{vars}([\![L]\!]) \cup \mathcal{P}(T))$, then $x$ is either assigned by the two prior cases, or we assign it by type inference on some previously assigned variable $y$.

(4) If $x$ is a proof term recoverable by unification, $x \in \tilde{\mathcal{P}}(T)$, then we know that $[\![\mathit{type}(x)]\!] \sqsubseteq [\![L]\!]$ and variables in $\mathit{type}(x)$ are assigned by (1). The expression $s$ must contain a subterm of type $\mathit{type}(x)$, with variables assigned, at the same position where $x$ appears in $L$. Thus, we obtain an assignment of $x$ by unifying $L$ and $s$.

(5) If $x$ is a variable for a type class condition, $x \in \mathcal{C}(T)$, then by $\mathfrak{R}_4$ we know that we can use the previous cases to get an assignment for all variables in $\mathit{type}(x)$. To obtain an assignment for $x$ itself, we use type class synthesis on $\mathit{type}(x)$ with all variables assigned, which must have succeeded during equality saturation.

(6) Finally, if $x$ is a type class instance recoverable by unification, $x \in \tilde{\mathcal{C}}(T)$, then we know $[\![type(x)]\!] \sqsubseteq [\![L]\!]$ and variables are assignable by (1). Analogous to (4), $s$ contains an expression of $type(x)$, with all variables assigned, at the same position where $x$ appears in $L$. Therefore, we assign $x$ by unification of $L$ and $s$.

*Example 5.3 (Theorem Instantiation).* To give an example of the above procedure, we choose $T$ to be the theorem stating that for lists whose elements have an additive commutative monoid structure, the sum of those elements is preserved under permutation:

```
∀ {M} [inst : AddCommMonoid M] {l₁ l₂ : List M} (h : l₁.Perm l₂), l₁.sum = l₂.sum
```

Here, $vars([\![L]\!]) = \{\texttt{l}_1, \texttt{M}\}$ (`M` is an implicit argument to `sum`), $\mathcal{P}(T) = \{\texttt{h}\}$, $\mathcal{C}(T) = \{\texttt{inst}\}$, and $\tilde{\mathcal{P}}(T) = \tilde{\mathcal{C}}(T) = \emptyset$. We only cover $\texttt{l}_2$ in $\omega(\mathcal{P}(T))$, as $\texttt{l}_2$ appears in the type of `h`. Now, consider the explanation step $[\![\texttt{[1, 2, 3].sum}]\!], (r_T, \Rightarrow, \top), [\![\texttt{[3, 2, 1].sum}]\!]$, where $r_T$ identifies the rewrite rule of theorem $T$, $\Rightarrow$ indicates that it was applied in the forward direction, and $\top$ denotes that it was applied at the term's root. The assignment of $T$'s variables then proceeds as follows. We assign all variables in $vars([\![L]\!])$ by Case (1). That is, by unification of `[1, 2, 3].sum` with $\texttt{l}_1$`.sum`, we get $\texttt{l}_1 \mapsto$ `[1, 2, 3]` and `M` $\mapsto$ `Nat`. We assign `h` by Case (2). Specifically, we first ensure that all variables in the type of `h` have been assigned. For $\texttt{l}_1$ and `M` this is already the case. As $\texttt{l}_2 \notin vars([\![L]\!])$ we obtain the assignment $\texttt{l}_2 \mapsto$ `[3, 2, 1]` by consulting the e-matching record mentioned in Case (2). Thus, we resolve the type of `h` as `[1, 2, 3].Perm [3, 2, 1]`. By our construction of rewrite rules, the e-graph must contain a proof of $[\![\texttt{[1, 2, 3].Perm [3, 2, 1]}]\!] = [\![\texttt{True}]\!]$, from which we assign `h`. Finally, we assign `inst` by Case (5). First, we ensure that all variables in the type of `inst` have been assigned, which is already satisfied by `M` $\mapsto$ `Nat`. Thus, we assign `inst` by type class synthesis for `AddCommMonoid Nat`, which must succeed, as this was checked during equality saturation.

Above, we assumed that explanation steps are justified by rewrites which follow from theorems. However, we also use rewrites which are not derived from theorems. First, we use rewrites to maintain the gadget for representing facts, by adding equivalences between `eq t t` and $[\![\texttt{True}]\!]$ for all terms $t$. This yields explanation steps between $e = e$ and `True` for some expression $e$. This step is however trivially provable. And second, we use rewrites rules to encode certain definitional equalities, such as $\beta$- and $\eta$-reduction. However, when we know that two terms are definitionally equal, they are provably equal by reflexivity.

### 5.3 Proof Composition

To construct a full proof, we connect the individual proof steps. Let $t, (r, d, p), t'$ be one step in an explanation, with $e := (\!|t|\!)$ and $e' := (\!|t'|\!)$. Then $e$ and $e'$ only differ at position $p$. That is, for a suitable choice of expression context $\epsilon$ and subexpressions $s$ and $s'$, we get $e = \epsilon(s)$ and $e' = \epsilon(s')$. In the previous section, we constructed a proof of the equivalence $s = s'$. We now extend this to a proof of $e = e'$. Intuitively, this extension holds by congruence:

$$\text{congr} : \forall (f_1\ f_2 : \alpha \to \beta)(a_1\ a_2 : \alpha), (f_1 = f_2) \to (a_1 = a_2) \to f_1\ a_1 = f_2\ a_2$$

However, naively applying congruence can fail due to two problems. First, $s$ and $s'$ may appear under a binder in $e$ and $e'$. In that case, the proof of $s = s'$ needs to be parameterized to a proof of the form $\forall x, s = s'$, where $x$ generalizes the bound variable. To "zoom in" on the proof of $s = s'$, congruence must then be interleaved with function extensionality:

$$\text{funext} : \forall\ f_1\ f_2, (\forall x, f_1\ x = f_2\ x) \to f_1 = f_2$$

For example, a proof of $f\ (\lambda x, s) = f\ (\lambda x, s')$, based on the subproof $h : \forall x, s = s'$ would have the form congr ... (rfl : $f = f$) (funext ... ($\lambda x, h\ x$)). Second, the example above only works for non-dependent functions, by definition of congr. A generalization of congruence to dependent functions as in [Selsam and de Moura 2016], where $a_1$ and $a_2$ can have non-definitionally equal types, does not hold in dependent type theory, even when stated over more permissive *heterogeneous* equality. One can however construct congruence theorems specifically for each dependent function, as proposed by the cited work and used in Lean.

As both of these problems are well-known, we rely on the implementation by Kyle Miller for "congruence quotations"[10] to extend proofs of $s = s'$ to $e = e'$. The final step in producing a proof for an entire explanation $t_1, j_1, ..., t_n$ is then simply to connect all proofs $(\!|t_1|\!) = (\!|t_2|\!)$, ..., $(\!|t_{n-1}|\!) = (\!|t_n|\!)$ by transitivity of equality.

*Hole Instantiation and Propagation.* To produce valid proofs, we need to ensure that they do not contain holes. Let $t_1, j_1, ..., t_n$ be an explanation for the proof goal $\mathcal{L} = \mathcal{R}$. When decoding the terms $t_1, ..., t_n$ into expressions $e_1, ..., e_n$, any proof terms or type class instances are turned into holes. For our proof of $e_1 = ... = e_n$ to be valid, we must therefore instantiate these holes. We ensure that our proof reconstruction procedure produces hole-free expressions inductively: As $e_1$ is the start term, it corresponds to $\mathcal{L}$. Therefore, we eliminate holes in $e_1$ by unification with $\mathcal{L}$. Next, we ensure that if $e_i$ is hole-free, then so is $e_{i+1}$ after proof reconstruction. Let $e_i$ rewrite to $e_{i+1}$ by application of theorem $T : \forall \bar{x}, L \sim R$ on the subexpressions $s_i$ and $s_{i+1}$ with $e_i := \epsilon(s_i)$ and $e_{i+1} := \epsilon(s_{i+1})$ for a suitable choice of $\epsilon$. Each hole in $e_{i+1}$ must appear either inside $s_{i+1}$, or in the surrounding term given by $\epsilon$. If a hole appears in $\epsilon$, then we obtain an instantiation for it from $e_i$, as $e_i$ is hole-free and thus has a hole-free $\epsilon$. If the hole instead appears in the subterm $s_{i+1}$, then we instantiate holes by unification with $R$. Thus, after proof reconstruction $e_{i+1}$ is hole-free.

In summary, holes are instantiated by unification with existing terms which have not been obtained from decoding (the proof goal and theorems), or by propagation from previous steps.

## 6 Extensions

The encoding and decoding of Lean expressions and theorems discussed so far comprises the core of our approach. However, to improve the applicability our approach in practice, we add three extensions on top of the core procedure, which we highlight in this section. We have implemented the core procedure and all extensions in our Lean proof tactic egg.

### 6.1 Definitional Equalities

As discussed before, one key challenge of using Lean expressions with e-graphs is different notions of equality. Specifically, e-graphs consider equality of terms only up to syntax, whereas Lean's underlying notion of *definitional* equality includes conversion rules between syntactically distinct terms. It is, therefore, crucial that we make these rules transparent to equality saturation to avoid getting stuck on syntactically distinct but definitionally equal terms. Unfortunately, definitional equality rules cannot generally be encoded as simple rewrite rules. Thus, we handle various definitional equality rules using specialized approaches.

*Erasure.* The simplest approach to handling definitional equality rules is to erase the corresponding syntactic construct. We do this for the let and proj constructs in the normalization function $\|\cdot\|$, which obviates the need for their definitional equality rules. Similarly, Lean's definitionally equality rule for proof irrelevance is implemented by the encoding of proofs as erased proof terms in the encoding function $[\![\cdot]\!]$.

[10] See https://leanprover-community.github.io/mathlib4_docs/Mathlib/Tactic/TermCongr.html.

*Natural Number Literals.* Internalized natural number literals (lit) allow for representing natural numbers avoiding the usual encoding $S\ (\ldots\ (S\ 0))$. However, they introduce syntactically distinct versions for semantically equal terms like lit 0 and const Nat.zero. To bridge this gap, we add (dynamic) rewrite rules which convert between these representations:

$$\text{lit } 0 \Leftrightarrow \text{const Nat.zero} \qquad \text{app (const Nat.succ) (lit } ?n) \Rightarrow \text{lit } ?(n+1)$$
$$\text{lit } ?n \Rightarrow \text{app (const Nat.succ) (lit } ?(n-1)),\ \text{if } 0 < n$$

Lean's definitional equality also includes rules for basic arithmetic operations on natural number literals. We cover these by adding the obvious dynamic rewrite rules.

*Bound Variables, $\beta$-Reduction, and $\eta$-Reduction.* Arguably, the most important definitional equality rules concern the interaction between application and $\lambda$-abstraction:

$$(\lambda x, e_1)\ e_2 \longrightarrow_\beta e_1[x \mapsto e_2] \qquad (\lambda x, e\ x) \longrightarrow_\eta e, \text{ if } x \text{ is not free in } e$$

While intuitive, when stated as above, these rules introduce significant challenges when used with an e-graph. The core problem, which is not unique to $\beta$- and $\eta$-reduction, is the presence of bound variables. Handling bound variables in e-graphs is notoriously difficult as their meaning is context-dependent, while any given e-class can be used in many contexts [Koehler 2022; Schneider et al. 2025; Willsey et al. 2021]. As a result, applying rewrites to terms containing bound variables encoded as de Bruijn indices can lead to unwanted shadowing. To compensate for this, we implement rewrite rules such that they explicitly check for collisions, and shift affected bound variables accordingly. For example, to implement a rewrite rule for $\eta$-reduction using de Bruijn indices, we introduce a shifting operator $\downarrow$:

$$\text{lam } ?t\ (\text{app } ?f\ (\text{bvar } 0)) \Rightarrow \downarrow(?f), \text{ if } ?f \text{ does not refer to bvar } 0$$

We implement the semantics of $\downarrow$ using small-step rewrite rules such as:

$$\downarrow(\text{bvar } ?(n+1)) \Rightarrow \text{bvar } ?n \qquad \downarrow(\text{app } ?e_1\ ?e_2) \Rightarrow \text{app } \downarrow(?e_1)\ \downarrow(?e_2)$$

Following [Anaya Gonzalez et al. 2023], we reduce the number of propagated shifts when possible. Yet, for proof goals involving binders, these explicit shifting nodes can contribute overwhelmingly to e-graph explosion. A similar problem occurs for $\beta$-reduction. Implementing a rule for $\beta$-reduction with de Bruijn variables requires both a shifting and a substitution operator. We again implement the semantics of substitution by small-step rewrite rules.

*$\delta$-Reduction and Type Class Projections.* The $\delta$-reduction rule of definitional equality states that any definition is equal to its unfolding. Adding a rewrite rule for each definition would, however, quickly overwhelm the practical limits of equality saturation. The same restriction also applies to other tactics in Lean which, therefore, introduces a notion of *transparency* to indicate how eagerly definitions should be unfolded. Similarly, we restrict ourselves to considering unfoldings only for type class projections, for which unfolding may be expected by the user. For example, consider:

```
example (m n : Nat) : m + n = Nat.add m n := by egg
```

This equality does not follow syntactically, as the `+` notation elaborates to an application of the type class projection `HAdd.hAdd`. However, the equality is evident by unfolding and reducing the definition of `HAdd.hAdd`. Therefore, we generate theorems for reducing applications of type class projections appearing in the proof goal or given theorems. In the given example, this means we generate the following equations:

```
HAdd.hAdd Nat Nat Nat (instHAdd Nat instAddNat) = Add.add Nat instAddNat
Add.add Nat instAddNat = Nat.add
```

*ι-Reduction and Structure Projections.* The $\iota$-reduction rules capture the reduction semantics for recursors of inductive types. Much like with $\delta$-reduction, it is impractical to add rewrite rules encoding $\iota$-reduction for each inductive type. In fact, even within Lean it is generally preferred to reason with equations on definitions, instead of unfolding to recursors. Thus, we again restrict ourselves to special cases of $\iota$-reduction. Specifically, we only consider the case of applications of structure projections to explicitly constructed structure terms. For example, this is required in the following trivial theorem:

```
example (a b : α) : Prod.fst (a, b) = a := by egg
```

Here, we do not run into problems with notations, as in the previous paragraph, as `( , )` is directly elaborated to the constructor `Prod.mk`. However, the projection `Prod.fst` unfolds to an application of `Prod`'s recursor, which by $\iota$-reduction reduces to the left term `a`.Therefore, we generate theorems for reducing applications of structure projections to explicitly constructed structure terms. In the given example, we generate the following equation:

```
∀ α B a b, Prod.fst α β (Prod.mk α β a b) = a
```

Note that we do not generate an equation for the second projection `Prod.snd`. We can omit this rule as the symbol `Prod.snd` does not appear in the proof goal, or any given theorem. Thus, any rule matching on `Prod.snd` could never apply in the first place.

We apply this principle more generally when generating rewrite rules: If a rewrite rule contains a symbol which does not appear in the proof goal or any other rewrite rule, we do not include it. This restriction can significantly reduce the number of generated rewrite rules, for example when considering algebraic structures which can have dozens of projections each.

### 6.2 Theorem Specialization

So far, we ensured applicability of theorems by implementing definitional conversions. This involved generating new theorems entirely. However, in some cases we can extend the applicability of theorems by assigning quantified variables heuristically, thus, specializing it.

*Goal Type Specialization.* Consider the following theorem over additive groups:

```
theorem sub_eq_zero (G : Type) [i : AddGroup G] (a b : G) : a - b = 0 ↔ a = b
```

Encoding the right-hand side of the equivalence yields the pattern `eq ?a ?b`. The encoding of the left-hand side, however, yields a term which also references pattern variables `?G` and `?i`, as part of the `0` and subtraction terms. As a result, the backward direction of this theorem cannot be turned into a rewrite rule. This theorem *should* however be applicable in the backward direction, as the presence of `G` is merely an artifact of our encoding. To enable the backward direction, we heuristically decide that theorems are probably going to be used on terms of certain types, for example the type of the current proof goal. Based on this assumption, we specialize the theorem by unifying the type of the left- and right-hand side with the expected type. For `sub_eq_zero` this unification assigns `G`. Thus, if we are proving a theorem about, for example, the integers, then goal type specialization assigns `G := Int`. This eliminates the pattern variable `?G` from the encoding of the left-hand side, which leaves `?i` as the only variable blocking the backward rewrite rule. However, by specializing `G` to `Int`, we also specialize the type of `i` to `AddGroup Int`. We can therefore eagerly synthesize the instance `i`, which eliminates `?i` from the encoding. In total, this leaves the left-hand side only with pattern variables `?a` and `?b`, which allows the backward rewrite rule.

*Explosion.* For many theorems, goal type specialization does not suffice to enable additional rewrite directions. Take, for example, the following theorem over additive groups:

```
theorem neg_add_cancel (G : Type) [i : AddGroup G] (a : G) : -a + a = 0
```

The encoding of this theorem contains ?G and ?i on both the left- and right-hand side. However, the backward direction is obviously not applicable, as it requires the "creative choice" of `a`. In some cases, it can be useful to automate these creative steps by heuristically specializing all missing variables with all matching terms in the local context. We call this approach *explosion*, due to the combinatorial explosion of specialized theorems which can occur, when applying this technique. If we apply explosion for `neg_add_cancel` in a local context containing additive group elements `x`, `y`, and `z`, we add the equations:

```
-x + x = 0        -y + y = 0        -z + z = 0
```

All of these equations admit rewrite rules in both directions. Explosion must be used with caution as it can, as the name suggests, explode the number of generated theorems, and is not suitable for most proof goals. However, we have found it to be useful in the context of the Equational Theories Project [Bolan et al. 2026]. This project involves proofs using equations over simple objects, which are connected by equations of (bounded) arbitrary shapes, like $x \circ (y \circ z) = (x \circ x) \circ w$. These properties make explosion both suitable and useful. However, when non-trivial creative terms are required, this simple heuristic does not suffice. Instead, we rely on humans to inject creative guidance into the process.

### 6.3 Guidance

As an automated procedure, equality saturation has limitations with respect to the (sizes of) problems it can solve. In [Koehler et al. 2024], 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. As a solution, they introduce the notion of *guided equality saturation*: Instead of trying to perform an ambitious rewrite from term $t_1$ to $t_n$ in a single run of equality saturation, they introduce intermediate goals $t_2, ..., t_{n-1}$ called *guides*. Then only equality saturations from each $t_i$ to $t_{i+1}$ are performed, thus replacing a single long run of equality saturation with multiple short runs significantly reducing the danger of reaching resource limits. To use guidance with our egg tactic, we introduce a syntax reminiscent of Lean's `calc` tactic and resembling the pen-and-paper equational reasoning style:

```
example [AddGroup G] (a : G) : -(-a) = a := by
  egg calc [add_assoc, zero_add, add_zero, neg_add_cancel]
    _ = -(-a) + 0
    _ = -(-a) + (-a + a)
    _ = 0 + a
    _ = a
```

Aside from breaking up long runs of equality saturation, guides provide two additional benefits. First, they allow us to write proofs in the pen-and-paper-style closly resembling textbook proofs, by separating justifications (what we called contextual knowledge in Section 2) from the actually interesting reasoning steps. Second, guides make it possible to inject creative steps into the reasoning, which could not otherwise be derived by equality saturation. In the example above, the creative steps are adding a magic `0`, and rewriting that `0` to `-a + a`. The theorem used to justify the second step is `neg_add_cancel`, which, as discussed in the previous section, can only rewrite from `-a + a` to `0` but not vice versa. Thus, in an unguided attempt at proving the goal, `neg_add_cancel` could never be applied, as the e-graph does not contain the relevant term `-a + a`. However, by providing the guide `-(-a) + (-a + a)` explicitly, we add the relevant term to the e-graph, enabling the rewrite.

*Guide Terms.* It is not uncommon for a proof to require creative terms, such as `-a + a` above, without requiring an *entire* guide to succeed. For this purpose, we introduce *guide terms*: terms which are simply added to the e-graph to enable rewrites rules which might not otherwise apply. Using two guide terms, we can solve the previous example by only supplying the creative terms:

```
example [AddGroup G] (a : G) : -(-a) = a := by
  egg [add_assoc, zero_add, add_zero, neg_add_cancel] using -(-a) + 0, -a + a
```

In fact, as stated by Zucker [2025], *"the ability to seed the e-graph termbank with useful terms is crucial for equality saturation"*. From a theoretical point of view, this is important because equality saturation is not complete for equational reasoning, unless we also enumerate all terms and add them to the e-graph. More practically, a lack of initial seed terms can block rewrite rules which we might expect to apply. For example, by default, trying to use the theorem `sub_eq_zero` : $\forall\, a\, b,\ a - b = 0 \leftrightarrow a = b$ on a goal $c - d = 0$ will fail due to a lack of seed terms. Specifically, we only add terms $[\![c - d]\!]$ and $[\![0]\!]$ to the e-graph, but not $[\![c - d = 0]\!]$, which is necessary for `sub_eq_zero` to match. To improve the seeding of the e-graph, we automatically add *derived guide terms* to the e-graph. These are guide terms which we automatically derive from the proof goal and all rewrite rules, by considering all closed subterms. Derived guide terms, for example, solve the problem of applying `sub_eq_zero`.

## 7 Example Use Cases

Now we consider concrete use cases to show that our approach is useful in practice by raising the level of abstraction at which we can reason about equations. These examples are from different areas of mathematics, and are all based on code from Mathlib, Lean's comprehensive mathematical library [mat 2020] containing over a million lines of formalized mathematics.

### 7.1 Boolean Algebra

We start by considering boolean algebras, an algebraic structure in lattice theory. As is common in modern mathematics, this structure is built by a tower of more general structures that it keeps refining: distributive lattices, lattices, join- and meet- semilattices, partial orders, etc. The details of these structures are not relevant here, but Mathlib builds these definitions on top of each other in a type class hierarchy. Each of the definitions has its own properties and equational lemmas, and reasoning about boolean algebras requires knowing properties of all of these structures. This modular approach to defining algebraic structures relies heavily on type class synthesis when it comes to applying theorems from different parts of the algebraic hierarchy. This is why it is crucial that our encoding represents type class instances uniformly using erasure, and can rely on type class synthesis to check and reconstruct the instances as necessary.

Using algebraic hierarchies with our tactic is additionally aided by a syntax for declaring and hierarchically extending the set of theorems we consider for proofs, which we call the egg baskets. For example, the following lines hierarchically define a basket for generalized boolean algebras with their equations `sup_inf_sdiff` and `inf_inf_sdiff`. We don't want to put all our eggs in one basket, so we define it as an extension of baskets for the underlying algebraic structures:

```
egg_basket lattice extends slattice_sup, slattice_inf with ...
egg_basket distrib_lattice extends lattice with ...
egg_basket bool extends distrib_lattice with sup_inf_sdiff, inf_inf_sdiff
```

Based on this set of theorems, we consider some proofs from Mathlib about boolean algebras. These are based on Stone [1935], but the postulates defining generalized boolean algebra are not purely equational. In Mathlib they slighly adapt them to be equational, and use a different notation, but the equational reasoning is still present in the proofs. An example is shown on the left of Fig. 4.

**calc**

```
calc
  y \ x ⊔ x
  _ = y \ x ⊔ (x ⊔ x ⊓ y) := by rw [sup_inf_self]
  _ = y ⊓ x ⊔ y \ x ⊔ x := by ac_rfl
  _ = y ⊔ x := by rw [sup_inf_sdiff]
```

**egg**

```
egg +bool calc
  y \ x ⊔ x
  _ = y \ x ⊔ (x ⊔ x ⊓ y)
  _ = y ⊓ x ⊔ y \ x ⊔ x
  _ = y ⊔ x
```

Fig. 4. Proof using Lean's `calc` tactic on the left and our egg tactic on the right.

**calc**

```
calc
  z \ (x \ y ⊔ y \ x)
  _ = (z \ x ⊔ z ⊓ x ⊓ y) ⊓ (z \ y ⊔ z ⊓ y ⊓ x) := by
    rw [sdiff_sup, sdiff_sdiff_right, sdiff_sdiff_right]
  _ = z ⊓ (z \ x ⊔ y) ⊓ (z \ y ⊔ z ⊓ y ⊓ x) := by
    rw [sup_inf_left, sup_comm, sup_inf_sdiff]
  _ = z ⊓ (z \ x ⊔ y) ⊓ (z ⊓ (z \ y ⊔ x)) := by
    rw [sup_inf_left, sup_comm (z \ y), sup_inf_sdiff]
  _ = z ⊓ z ⊓ (z \ x ⊔ y) ⊓ (z \ y ⊔ x) := by ac_rfl
  _ = z ⊓ (z \ x ⊔ y) ⊓ (z \ y ⊔ x) := by rw [inf_idem]
```

**egg**

```
egg +bool calc
  z \ (x \ y ⊔ y \ x)
  _ = (z \ x ⊔ z ⊓ x ⊓ y)
    ⊓ (z \ y ⊔ z ⊓ y ⊓ x)
  _ = z ⊓ (z \ x ⊔ y)
    ⊓ (z \ y ⊔ z ⊓ y ⊓ x)
  _ = z ⊓ (z \ x ⊔ y)
    ⊓ (z ⊓ (z \ y ⊔ x))
  _ = z ⊓ z ⊓ (z \ x ⊔ y) ⊓ (z \ y ⊔ x)
  _ = z ⊓ (z \ x ⊔ y) ⊓ (z \ y ⊔ x)
```

Fig. 5. More complex proof using Lean's `calc` tactic on the left and our egg tactic on the right.

The second step of this calculation block uses the tactic `ac_rfl`, which is a tactic specifically for reasoning about associativity and commutativity (AC). This is necessary, as reasoning with AC is notoriously difficult [Benanav et al. 1987], and, thus, more generic tactics like Lean's simplifier cannot generally accommodate these theorems in their procedure. While equality saturation also has scalability issues when reasoning with AC, in practice our egg tactic has no problem brute forcing AC on small terms, as shown on the right of Fig. 4.

We observe that, by continuously growing our `bool` basket with lemmas as we prove them, we can readily build up to more complicated proofs, such as the Mathlib proof in Fig. 5. Using our tactic, we can clarify the proof to a more readable form, skipping tedious bookkeeping steps, and omitting all explanations, as seen on the right in the figure.

We can even omit all equational steps, and prove the theorem using a single guide term:

```
egg +bool using z \ x ⊔ z ⊓ x ⊓ y
```

We choose this specific guide term as it is contained in the longest term of the explicit equational reasoning steps. This tends to be a good heuristic, as expanded terms are likely to contain the creative terms which are required for rewrites to apply.

### 7.2 Lie Algebra

Boolean algebra combines many equational theories that are well-studied and have decision procedures for different aspects, like the `simp_ac` tactic that specifically deals with AC. In fact, a variety of algebraic structures have dedicated tactics, like lattices [James and Hinze 2009] or commutative rings [Grégoire and Mahboubi 2005]. Building a new tactic for every algebraic structure or equational theory, however, will likely not scale. For example, consider Lie algebras, an algebraic structure that is common in physics. Lie algebras are vector spaces with a bilinear map $[\cdot,\cdot]$, adhering to the so-called Jacobi identity:

$$\forall\, x\, y\, z,\ [x,[y,z]] + [y,[z,x]] + [z,[x,y]] = 0$$

As far as we know, there is no decision procedure for words over Lie algebras, nor any specific tactic for them.[11] To evaluate our tactic in this setting, we try to replicate the equational reasoning of a proof from the textbook by Erdmann and Wildon [2006, Ch. 1]:

> As the Lie bracket $[-,-]$ is bilinear, we have
>
> $$0 = [x+y, x+y] = [x,x] + [x,y] + [y,x] + [y,y] = [x,y] + [y,x].$$
>
> Hence condition (L1) implies
>
> $$[x,y] = -[y,x] \text{ for all } x, y \in L. \tag{L1'}$$

After setting up the algebraic definitions and creating a `lie` basket which includes the referenced condition L1 ($[x,x] = 0$), we can directly write:

```
theorem L1' : ⁅x, y⁆ = -⁅y, x⁆ := by
  have h := by egg +lie calc
    0 = ⁅x + y, x + y⁆
    _ = ⁅x, x⁆ + ⁅x, y⁆ + ⁅y, x⁆ + ⁅y, y⁆
    _ = ⁅x, y⁆ + ⁅y, x⁆
  egg +lie [h]
```

This formalized version does differ from the textbook in a few superficial ways: the Lie bracket is written using slightly different notation. Our Lean version of the proof of L1' also has to explicitly reference the proof of the previous identity `h` and mention the `lie` basket in the final call to `egg`. While these are all superficial syntactic differences, the proof remains very close to the textbook. Note that we could have also shortened the proof to just:

```
egg +lie using ⁅x + y, x⁆ + ⁅x + y, y⁆
```

Compare this with the proof of the same identity in Mathlib:

```
theorem lie_skew : -⁅y, x⁆ = ⁅x, y⁆ := by
  have h : ⁅x + y, x⁆ + ⁅x + y, y⁆ = 0 := by rw [← lie_add]; apply lie_self
  simpa [neg_eq_iff_add_eq_zero] using h
```

While it is not the goal of the Mathlib proof to replicate the textbook proof, the justification is non-obvious and hard to read for a simple identity. It uses a manual rewrite with an explicit rewrite direction (denoted by ←), as well as a specific `simpa` incantation of the simplifier tactic. In aggregate, these cloud the important reasoning steps. As a result, it is arguably harder to *write* this proof than our direct translation of the pen-and-paper steps.

Reviewing the Mathlib code of Lie algebras, we find more interesting examples, such as:

```
theorem neg_lie : ⁅-x, m⁆ = -⁅x, m⁆ := by
  rw [← sub_eq_zero, sub_neg_eq_add, ← add_lie]
  simp
```

Again, the Mathlib proof requires explicit rewriting in different directions and separate calls to the explicit rewriting and simplifier tactics. In the textbook this theorem is not mentioned at all, as it is "obvious" by the bilinearity of the Lie bracket. Our tactic also makes this "obvious": it proves it with `egg +lie` without any additional steps. Notably, this proof relies on reasoning over the entire proposition `⁅-x, m⁆ = -⁅x, m⁆` instead of just the individual subterms, as the proof begins by applying `sub_eq_zero : ∀ a b, a - b = 0 ↔ a = b`. Crucially, this theorem only applies as a result of the extensions of goal type specialization and derived guide terms, and it is not solved by other tactics using e-graphs.

[11] In fact, we chose this example precisely because of this, as suggested by a colleague working in Mathlib.

egg

```
theorem map_tail_trans (a : Vec α (m + 1)) (v : Vec (Vec α (m + 2)) (n + 1)) :
    map tl (trans (tl a :: map tl (map tl v))) = trans (map tl (map tl v)) := by
  induction m <;> cases a
  all_goals egg +rise [*]

theorem rule₂ (as : Vec (Vec α m) n) : transpose (transpose as) = as := by
  induction n <;> cases m <;> cases as <;> try cases ‹Vec _ (_ + 1)›
  all_goals egg +rise [*, fill_nil (_ :: _)]

theorem rule₆ (f : α → β) (g : β → γ → γ) (init : γ) (as : Vec α n) :
    reduceSeq (λ a b => g (f a) b) init as = (reduceSeq g init ∘ map f) as := by
  induction as generalizing init
  all_goals egg +rise [*]
```

Fig. 6. Using egg, we can automate the equational steps of the formalization by [Hagedorn et al. 2020].

### 7.3 Functional Array Programs

To investigate the applicability of our tactic in a non-algebraic domain, we consider theorems for functional array programs. Specifically, we consider results from [Hagedorn et al. 2020], which develops and formalizes rules for rewrite-based program optimization. These rules are defined over the *Rise* language, which includes fixed-length vector types. These vectors are formalized in Agda using the canonical representation of fixed-length vectors by a dependent type. The Agda proofs of the developed rules generally involve induction or case splitting, followed by many manual equational proof steps. In our formalization of the same proofs, we find that we can fully automate the equational proof steps with egg. That is, out of the 17 theorems formalized, all take the form of the proofs shown in Figure 6.

The proofs start by `induction` and/or `cases` bashing, and all resulting goals are proven by egg using the accumulated `rise` egg basket. Notably, these proofs involve rewriting over the dependent `Vec` type. In some cases, this involves rewriting vector lengths at the type level – however, only up to definitional equality.[12] Moreover, $\texttt{rule}_6$ involves rewriting over terms with binders, which requires $\beta$- and $\eta$-reduction. This is handled transparently by egg. We find only one case, $\texttt{rule}_2$, where the equational reasoning is not *fully* automated, as the theorem `fill_nil` is not suitable as a rewrite rule and is instantiated manually.

### 7.4 Binomial Theorem

As a final use case, we go back to the motivational example from Section 2, the proof of the binomial theorem from Rotman [2006]. Motivated by Koehler et al. [2024], we consider the equational steps of the proof of Proposition 1.15, which they could not prove with guided equality saturation. The issue are implicit preconditions, discussed in Section 2: the textbook proof uses terms like $\frac{1}{n-r+1}$, which are not in $\mathbb{N}$ in general and require casting to $\mathbb{R}$ for the desired arithmetic. Additionally, and crucial to our proof, many reasoning steps are only valid subject to conditions like $r \leq n+1$. That is, they need conditional rewriting.

As the preconditions of casting and arithmetic are not fundamental to the theorem at hand, it is not immediately clear which conditions need to be justified to solve the goal. In the proof below, we therefore employ a feature of our egg tactic which allows us to postpone the proofs

[12] Our tactic will also rewrite non-definitional equalities at the type level. However, our current implementation of proof reconstruction would not be able to reconstruct proofs for these steps.

of propositional conditions until after equality saturation. That is, the necessary preconditions that cannot be solved automatically during equality saturation are presented to the user as proof obligations. This is possible by a trivial modification of the proof reconstruction procedure. Thus, we can conveniently handle the preconditions required for the rewrites that Koehler et al. [2024] could not. We do, however, still have to be explicit that we are reasoning in $\mathbb{R}$ instead of $\mathbb{N}$ and explicit cast between these types. Aside from these technical differences, our proof follows the structure of the proof in [Rotman 2006].

The first line in the proof enables propositional conditions of rewrites to be surfaced as proof obligations. The line containing the egg invocation references egg baskets with facts about casts between $\mathbb{N}$ and $\mathbb{R}$ and arithmetic on $\mathbb{R}$, as well as lemmas about the $\Gamma$-function which generalizes factorial to $\mathbb{R}$. The rest of the proof proceeds essentially verbatim from the textbook, except that we first and finally cast ($\uparrow$) to and from $\mathbb{R}$. Finally, the last line shows the first steps of proving the proof obligations produced by egg. These are analogous to the sub-proofs $h_1, ..., h_{10}$ in Figure 1b. The proof obligations are conditions like $r \leq n + 1$ for casting subtractions like $n + 1 - r$, or $n \neq 0$ for performing arithmetic with division.

In total, we generate 15 proof obligations, all of which can be discharged by suitable tactics in one or two lines. Combining our methods with specific theory solvers could potentially automatically solve all of these obligations, but that is beyond the scope of this paper.

egg

```
set_option egg.subgoals true

egg +cast +real calc [Gamma_nat_eq_factorial, Gamma_add_one]
  ↑(n ! / ((r - 1)! * (n - r + 1)!) + n ! / (r ! * (n - r)!))
  _ = n! / ((r - 1)! * (n - r + 1)!) + n! / (r! * (n - r)!)
  _ = n! / ((r - 1)! * (n - r)!) * (1 / (n - r + 1) + 1 / r)
  _ = n! / ((r - 1)! * (n - r)!) * ((r + (n - r + 1)) / (r * (n - r + 1)))
  _ = n! / ((r - 1)! * (n - r)!) * ((n + 1) / (r * (n - r + 1)))
  _ = (n + 1)! / (r! * (n + 1 - r)!)
  _ = ↑((n + 1)! / (r ! * (n + 1 - r)!))

all_goals try first | (norm_cast; done) | (norm_cast; omega) | ...
```

### 7.5 Limitations and Comparisons

The previous examples showcase our egg tactic. Naturally, it has practical limitations, which we briefly discuss here. We also compare to the behaviour of related Lean tactics.

When it comes to collecting "contextual knowledge" in the form of egg baskets, it can be difficult to judge which theorems should be included. For example, some proofs about boolean algebras use theorems defined over Coheyting algebras. Without sufficient knowledge about the problem domain and its formalization, discovering such theorems is difficult. This problem is, for example, handled much better by Lean's `grind` tactic, which contains a database of all possibly suitable theorems. In contrast, our current implementation of the egg tactic only handles on the order of $<$ 100 theorems. This can be improved in the future by using smart premise selection procedures, as [Blanchette et al. 2011; Czajka and Kaliszyk 2018], or by more sophisticated e-matching procedures based on discrimination trees.

Discovering missing theorems with our tactic is additionally complicated by it only being suitable for *closing* proof goals, not making partial progress on them, as some other tactics do. While partial progress can be trivially implemented using extraction on e-graphs, the notion of *sketch guides* from [Koehler et al. 2024] presents a more promising approach.

One approach we have found for addressing failing invocations of `egg` is to rely on guided equality saturation. By using `egg calc` to iteratively specify more intermediate steps, users can hone in on problematic reasoning steps and unveil missing theorems or capabilities.

Our practical limit on the number of theorems reflects the fact that equality saturation can easily fail by explosively growing the e-graph, as discussed in [Koehler et al. 2024]. The same holds for the sizes of explanations. To keep the duration of proof reconstruction on the order of seconds, we currently limit the length of explanations we handle to 200 steps. We have various examples where egg exceeds this length, which can vary significantly with slight changes to the initial conditions. Using guides to reduce the length of equality saturation runs, and thus explanation length, is also frail, as discussed in [Koehler et al. 2024]. However, using tree-structured instead of flattened explanations, as we currently do, can significantly reduce the sizes of explanations, thus directly addressing this problem [Flatt et al. 2022].

Finally, in the context of *interactive* theorem proving our tactic is sometimes slow, taking on the order of seconds to complete difficult proofs. Therefore, keeping long-running calls to `egg` in proof scripts is costly and other means of proof persistence should be considered. Other tactics like `simp` and `grind` usually are faster completing in less than one second.

*Comparison.* We tried two other tactics on the three use cases we presented in the subsections before. The "simplifier" tactic `simp`[13] greedily rewrites with given equations. The `grind` tactic [de Moura and Morrison 2025] is a novel proof tactic in Lean using e-graphs, e-matching, congruence closure, and solvers for specific theories. We provided the same inputs to these tactics as to our `egg` tactic. However, in our usage of `grind` we cannot count out user error entirely, as `grind` sometimes requires the user to decide how theorems should be turned into patterns for e-matching, which our tactic does not require.

Running `simp` on our use cases fails on all reasoning steps, besides some steps of the L1' theorem. Most of the time, this is because `simp` exceeds set resource limits by falling into loops on non-oriented theorems like associativity and commutativity.

The `grind` tactic proves some, but not all, reasoning steps. For the Boolean Algebra example in Fig. 5 two steps are not proven by `grind`. Similarly, `grind` fails to prove the final step of the L1' theorem, and fails on the second theorem entirely. For the Binomial Theorem, `grind` fails on three steps and can only prove the other steps given proofs of the necessary preconditions as *input*, whereas `egg` discovers these preconditions automatically.

## 8 Related Work

*Proof Tactics.* The egg-based proof tactics in Lean [Koehler et al. 2024] and Rocq [Bourgeat 2023] are most closely related to our work. We strictly improve upon the former, which also encodes Lean terms, but does not handle binders, type classes, or conditional rewrites. The Rocq tactic imposes restrictions on how symbols can appear in terms, and also does not admit binders. It also allows for conditional rewrites, but does not cover how theorems are encoded as rewrite rules, or how this affects proof reconstruction. Other tactics, focussing more specifically on congruence closure, are Lean's `cc` or Rocq's `congruence` tactics. A more expansive approach is chosen by Lean's recent `grind` tactic [de Moura and Morrison 2025], which uses e-graphs with e-matching, congruence closure and theory solvers (similar to SMT solvers). The tactic is good at deriving facts, and performing case analysis, at the cost of limiting the length of discoverable rewrite sequences. The SMTCoq plug-in [Ekici et al. 2017] provides tactics which rely on existing SMT solvers like CVC4 [Barrett et al. 2011]. Notably, their proof reconstruction uses *computational reflection*, instead of direct proof term construction, by implementing a certified checker of proof certificates. It can

[13] https://leanprover-community.github.io/extras/simp.html

also generate proof obligations for uncertified steps. Hammer for Coq [Czajka and Kaliszyk 2018] includes premise selection and covers a larger fragment of Coq's (Rocq's) calculus, but uses the output of external SMT solvers merely as hints for a hand-crafted, backward reasoning proof reconstruction procedure. Isabelle's Sledgehammer [Blanchette et al. 2011] reconstructs *saturation* proofs from external solvers like Z3 [de Moura and Bjørner 2008] and Vampire [Riazanov and Voronkov 2002]. Agda's user-defined rewrite-rule extensions [Cockx 2019] also have many conceptual similarities with our work. As these extend computational reductions in the core language itself, they are also more conservative: these extensions only support equality constraints as preconditions, and have stronger conditions on the usage of the variables on the LHS of the rewrite for the RHS, which we relax (cf. Section 6.2.)

*Equality Saturation and E-Graphs.* E-Graphs were introduced in the 1980s [Nelson 1980] and have long been used for SMT solvers. They have seen a resurgence with equality saturation [Tate et al. 2009] and the subsequent efficient implementation of egg [Willsey et al. 2021]. Our work builds heavily on the idea of guided equality saturation subsequently introduced by [Koehler et al. 2024]. Handling disequalities and context-sensitive facts efficiently is explored by dis-/equality graphs [Zakhour et al. 2025] and colored e-graphs [Singher and Itzhaky 2024], respectively. These ideas are complementary to our approach in this paper. A different approach is taken by egglog [Zhang et al. 2023], combining datalog with e-graphs. However, it does not yet support proof production, as in e-graphs [Flatt et al. 2022].

*Formalization of Pen-and-Paper Proofs.* The goal of a document that fits both as a human-readable argument and as computer-checkable proof was already formulated in AUTOMATH [de Bruijn 1968]: *"Our system should check a kind of language that comes as close as possible to what we write in ordinary mathematics"* [De Bruijn 1994]. Inspired by this, Mizar [Trybulec and Blair 1985], and later Isar [Wenzel 2002] made further progress at aligning the syntax of their languages with the languages in intuitive handwritten proofs.

In contrast, the systems mathNat [Humayoun 2010] and Naproche [Cramer et al. 2009] work on reading human-written proof texts in a controlled but *natural language*, instead of a programming language typical syntax. And finally, the more recent, *Draft, Sketch, Prove* system [Jiang et al. 2023], intends to use *informal* human-written texts, interpreted by a machine learning model, as guidance of what an automated reasoning engine should explore.

## 9 Conclusion

We presented an approach allowing to write proofs in an interactive theorem prover using a familiar pen-and-paper equational reasoning style found in mathematics. Our implementation as a Lean tactic, encodes suitable theorems as conditional rewrite rules dealing with propositional conditions as well as type class instances. We pass proof goals and rewrite rules to the egg equality saturation engine which attempts to perform the proof by rewriting with equality saturation. Once a proof has been found, we decode the produced explanation and reconstruct a valid Lean proof. We evaluated our approach on three case studies, demonstrating that we enable the desired pen-and-paper style, while proving theorems that could not be proven with the existing `simp` or `grind` tactics.

## Data-Availability Statement

There is an artifact [Rossel et al. 2025] available for reproducing the work presented in this paper. The latest version of the egg tactic implementation is available at reservoir.lean-lang.org/@marcusrossel/egg and its source code at github.com/marcusrossel/lean-egg.

## References


2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. doi:10.1145/3372885.3373824

Emmanuel Anaya Gonzalez, Cole Kurashige, Aditya Giridharan, and Polikarpova Nadia. 2023. Optimizing Beta Reduction in E-Graphs. (2023). https://pldi23.sigplan.org/details/egraphs-2023-papers/12/Optimizing-Beta-Reduction-in-E-Graphs EGRAPHS 2023.

Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 171–177. doi:10.1007/978-3-642-22110-1_14

Dan Benanav, Deepak Kapur, and Paliath Narendran. 1987. Complexity of Matching Problems. *J. Symb. Comput.* 3, 1/2 (1987), 203–216. doi:10.1016/S0747-7171(87)80027-5

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. 2011. Automatic Proof and Disproof in Isabelle/HOL. In *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6989)*, Cesare Tinelli and Viorica Sofronie-Stokkermans (Eds.). Springer, 12–27. doi:10.1007/978-3-642-24364-6_2

Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, Floris van Doorn, Martin Dvorak, Andrés Goens, Aaron Hill, Harald Husum, Hernán Ibarra Mejia, Zoltan Kocsis, Bruno Le Floch, Amir Livne Bar-on, Lorenzo Luccioli, Douglas McNeil, Alex Meiburg, Pietro Monticone, Pace P. Nielsen, Giovanni Paolini, Marco Petracci, Bernhard Reinke, David Renshaw, Marcus Rossel, Cody Roux, Jérémy Scanvic, Shreyas Srinivas, Anand Rao Tadipatri, Terence Tao, Vlad Tsyrklevich, Fernando Vaquerizo-Villar, Daniel Weber, and Fan Zheng. 2026. The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale. In preparation.

Thomas Bourgeat. 2023. *Specification and verification of sequential machines in rule-based hardware languages.* Ph. D. Dissertation. MIT, USA. https://hdl.handle.net/1721.1/150194

Robert S. Boyer and J Strother Moore. 1973. Proving Theorems about LISP Functions. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, Nils J. Nilsson (Ed.). William Kaufmann, 486–493. http://ijcai.org/Proceedings/73/Papers/053.pdf

Mario Carneiro. 2019. *The Type Theory of Lean.* Master's thesis. Carnegie Mellon University.

Mario Carneiro. 2024. Lean4Lean: Towards a formalized metatheory for the Lean theorem prover. *CoRR* abs/2403.14064 (2024). arXiv:2403.14064 doi:10.48550/ARXIV.2403.14064

Arthur Charguéraud. 2012. The Locally Nameless Representation. *J. Autom. Reason.* 49, 3 (2012), 363–408. doi:10.1007/S10817-011-9225-2

Jesper Cockx. 2019. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway (LIPIcs, Vol. 175)*, Marc Bezem and Assia Mahboubi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:27. doi:10.4230/LIPICS.TYPES.2019.2

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. doi:10.1016/0890-5401(88)90005-3

Thierry Coquand and Christine Paulin. 1988. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings (Lecture Notes in Computer Science, Vol. 417)*, Per Martin-Löf and Grigori Mints (Eds.). Springer, 50–66. doi:10.1007/3-540-52335-9_47

Marcos Cramer, Bernhard Fisseni, Peter Koepke, Daniel Kühlwein, Bernhard Schröder, and Jip Veldman. 2009. The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts. In *Controlled Natural Language, Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8-10, 2009. Revised Papers (Lecture Notes in Computer Science, Vol. 5972)*, Norbert E. Fuchs (Ed.). Springer, 170–186. doi:10.1007/978-3-642-14418-9_11

Lukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.* 61, 1-4 (2018), 423–453. doi:10.1007/S10817-018-9458-4

NG de Bruijn. 1968. Automath: a language for mathematics. (1968).

Nicolaas Govert De Bruijn. 1994. A survey of the project AUTOMATH. In *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier, 141–161.

Leonardo de Moura and Kim Morrison. 2025. The Lean Language Reference: The grind tactic. https://lean-lang.org/doc/reference/latest/The--grind--tactic/#grind

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

David Detlefs, Greg Nelson, and James B. Saxe. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52, 3 (2005), 365–473. doi:10.1145/1066100.1066102

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 2017. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 126–133. doi:10.1007/978-3-319-63390-9_7

Karin Erdmann and Mark J Wildon. 2006. *Introduction to Lie algebras*. Vol. 122. Springer.

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022. Small Proofs from Congruence Closure. In *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, Alberto Griggio and Neha Rungta (Eds.). IEEE, 75–83. doi:10.34727/2022/ISBN.978-3-85448-053-2_13

Benjamin Grégoire and Assia Mahboubi. 2005. Proving Equalities in a Commutative Ring Done Right in Coq. In *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3603)*, Joe Hurd and Thomas F. Melham (Eds.). Springer, 98–113. doi:10.1007/11541868_7

Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4, ICFP (2020), 92:1–92:29. doi:10.1145/3408974

Muhammad Humayoun. 2010. Mathnat-mathematical text in a controlled natural language. *Special issue: Natural Language Processing and its…* (2010).

Daniel W. H. James and Ralf Hinze. 2009. A Reflection-based Proof Tactic for Lattices in Coq. In *Proceedings of the Tenth Symposium on Trends in Functional Programming, TFP 2009, Komárno, Slovakia, June 2-4, 2009 (Trends in Functional Programming, Vol. 10)*, Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter W. M. Koopman (Eds.). Intellect, 97–112.

Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timoth ée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. 2023. Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In *The Eleventh International Conference on Learning Representations , ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=SMa9EAovKMC

Thomas Koehler. 2022. *A domain-extensible compiler with controllable automation of optimisations*. Ph. D. Dissertation. University of Glasgow, UK. doi:10.5525/GLA.THESIS.83323

Thomas Koehler, Andrés Goens, Siddharth Bhat, Tobias Grosser, Phil Trinder, and Michel Steuwer. 2024. Guided Equality Saturation. *Proc. ACM Program. Lang.* 8, POPL (2024), 1727–1758. doi:10.1145/3632900

Charles Gregory Nelson. 1980. *Techniques for Program Verification*. PhD thesis. Stanford University.

Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 453–468. doi:10.1007/978-3-540-32033-3_33

Lawrence C. Paulson. 1993. Isabelle: The Next 700 Theorem Provers. *CoRR* cs.LO/9301106 (1993). https://arxiv.org/abs/cs/9301106

Alexandre Riazanov and Andrei Voronkov. 2002. The design and implementation of VAMPIRE. *AI Commun.* 15, 2-3 (2002), 91–110. http://content.iospress.com/articles/ai-communications/aic259

Rocq Dev Team. 2025. *The Rocq Prover*. doi:10.5281/zenodo.15149629

Marcus Rossel, Rudi Schneider, Thomas Koehler, Michel Steuwer, and Andrés Goens. 2025. Artifact: Towards Pen-and-Paper-Style Equational Reasoning in Interactive Theorem Provers by Equality Saturation. Zenodo. doi:10.5281/zenodo.17696648

Joseph J Rotman. 2006. *A first course in abstract algebra: with applications*. Pearson.

Rudi Schneider, Marcus Rossel, Amir Shaikhha, Andrés Goens, Thomas Koehler, and Michel Steuwer. 2025. Slotted E-Graphs: First-Class Support for (Bound) Variables in E-Graphs. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1888–1910. doi:10.1145/3729326

Daniel Selsam and Leonardo de Moura. 2016. Congruence Closure in Intensional Type Theory. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9706)*, Nicola Olivetti and Ashish Tiwari (Eds.). Springer, 99–115. doi:10.1007/978-3-319-40229-1_8

Eytan Singher and Shachar Itzhaky. 2024. Easter Egg: Equality Reasoning Based on E-Graphs with Multiple Assumptions. In *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, Nina Narodytska

and Philipp Rümmer (Eds.). IEEE, 70–83. doi:10.34727/2024/ISBN.978-3-85448-065-5_13

Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1 (2025), 8:1–8:74. doi:10.1145/3706056

Marshall H Stone. 1935. Postulates for Boolean algebras and generalized Boolean algebras. *American Journal of Mathematics* 57, 4 (1935), 703–732.

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 264–276. doi:10.1145/1480881.1480915

Andrzej Trybulec and Howard A. Blair. 1985. Computer Assisted Reasoning with MIZAR. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*, Aravind K. Joshi (Ed.). Morgan Kaufmann, 26–28. http://ijcai.org/Proceedings/85-1/Papers/006.pdf

Sebastian Ullrich. 2023. *An Extensible Theorem Proving Frontend.* Ph. D. Dissertation. Karlsruhe Institute of Technology, Germany. doi:10.5445/IR/1000161074

Markus Wenzel. 2002. *Isabelle, Isar - a versatile environment for human readable formal proof documents.* Ph. D. Dissertation. Technical University Munich, Germany. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf

Freek Wiedijk (Ed.). 2006. *The Seventeen Provers of the World, Foreword by Dana S. Scott.* Lecture Notes in Computer Science, Vol. 3600. Springer. doi:10.1007/11542384

Eric Wieser. 2023. Multiple-Inheritance Hazards in Dependently-Typed Algebraic Hierarchies. In *Intelligent Computer Mathematics - 16th International Conference, CICM 2023, Cambridge, UK, September 5-8, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14101)*, Catherine Dubois and Manfred Kerber (Eds.). Springer, 222–236. doi:10.1007/978-3-031-42753-4_15

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. doi:10.1145/3434304

George Zakhour, Pascal Weisenburger, Jahrim Gabriele Cesario, and Guido Salvaneschi. 2025. Dis/Equality Graphs. *Proc. ACM Program. Lang.* 9, POPL (2025), 2282–2305. doi:10.1145/3704913

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proc. ACM Program. Lang.* 7, PLDI (2023), 468–492. doi:10.1145/3591239

Philip Zucker. 2025. Omelets Need Onions: E-graphs Modulo Theories via Bottom-up E-matching. *CoRR* abs/2504.14340 (2025). arXiv:2504.14340 doi:10.48550/ARXIV.2504.14340

Received 2025-07-10; accepted 2025-11-06