CCLemma: E-Graph Guided Lemma Discovery for Inductive Equational Proofs

COLE KURASHIGE, University of California, San Diego, USA
RUYI JI, Peking University, China
ADITYA GIRIDHARAN, University of California, San Diego, USA
MARK BARBONE, University of California, San Diego, USA
DANIEL NOOR, Technion, Israel
SHACHAR ITZHAKY, Technion, Israel
RANJIT JHALA, University of California, San Diego, USA
NADIA POLIKARPOVA, University of California, San Diego, USA

The problem of automatically proving the equality of terms over recursive functions and inductive data types is challenging, as such proofs often require auxiliary lemmas which must themselves be proven. Previous attempts at lemma discovery compromise on either efficiency or efficacy. *Goal-directed* approaches are fast but limited in expressiveness, as they can only discover auxiliary lemmas which entail their goals. *Theory exploration* approaches are expressive but inefficient, as they exhaustively enumerate candidate lemmas.

We introduce *e-graph guided lemma discovery*, a new approach to finding equational proofs that makes theory exploration goal-directed. We accomplish this by using e-graphs and equality saturation to efficiently construct and compactly represent the space of *all* goal-oriented proofs. This allows us to explore only those auxiliary lemmas *guaranteed* to help make progress on some of these proofs. We implemented our method in a new prover called CCLemma and compared it with three state-of-the-art provers across a variety of benchmarks. CCLemma performs consistently well on two standard benchmarks and additionally solves 50% more problems than the next best tool on a new challenging set.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; *Equational logic and rewriting*.

Additional Key Words and Phrases: Automated Theorem Proving, Equational Reasoning, Verification, Synthesis, Lemma Synthesis

ACM Reference Format:
Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. CCLemma: E-Graph Guided Lemma Discovery for Inductive Equational Proofs. *Proc. ACM Program. Lang.* 8, ICFP, Article 264 (August 2024), 27 pages. https://doi.org/10.1145/3674653

1 Introduction

There are many reasons to want a theorem prover which can automatically verify the equality of two recursive programs. One reason is to ensure an optimized program remains faithful to its original implementation. Suppose we are asked to write a function `mtp` that computes the *maximum*

Authors' Contact Information: Cole Kurashige, University of California, San Diego, USA, ckurashige@ucsd.edu; Ruyi Ji, Peking University, China, jiruyi910387714@pku.edu.cn; Aditya Giridharan, University of California, San Diego, USA, agiridharan@ucsd.edu; Mark Barbone, University of California, San Diego, USA, mbarbone@ucsd.edu; Daniel Noor, Technion, Israel, daniel.noor@campus.technion.ac.il; Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il; Ranjit Jhala, University of California, San Diego, USA, rjhala@ucsd.edu; Nadia Polikarpova, University of California, San Diego, USA, npolikarpova@ucsd.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/8-ART264
https://doi.org/10.1145/3674653

tail product of a list of natural numbers. For example, `mtp [2,0,3,4] = 12`, the product of the suffix `[3,4]`. In a functional language like Haskell, a straightforward and obviously correct way to write `mtp` is as the composition of standard library functions:

```
mtp = maximum . map product . tails
```

(here, `tails` returns all the suffixes of a list, and `product` and `maximum` compute, respectively, the product and the maximum of a list of numbers). The program above, however, is slowed by `tails`, which takes quadratic time. Upon realizing this, we migh write an optimized version `mtp'` which only takes a single pass, using the product of the whole list—`prod`—as an auxiliary accumulator:

```
mtp' = fst . foldr (\x (res,prod) → (max res (prod*x), prod*x)) (1,1)
```

Our second version is more efficient, but less obviously correct. Wouldn't it be nice if a tool could confirm for us that `mtp'` is in fact equivalent to `mtp`? In other words, we would like to automatically find a proof of the following universally-quantified proposition:

$$\forall xs \,.\, \texttt{mtp}\ xs \doteq \texttt{mtp'}\ xs \tag{1}$$

Unfortunately, no existing tools can prove proposition (1), at least not within five minutes. This proof is challenging because it relies on several *auxiliary lemmas*, most notably:

$$\forall xs \,.\, \texttt{snd}\ (\texttt{mtp'}\ xs) \doteq \texttt{product}\ xs \tag{2}$$

Challenge: Lemma Discovery. Clearly, we are not the first ones interested in automating inductive equational reasoning for properties which require auxiliary lemmas—we refer the reader to Johansson [2019] for an excellent survey of the area. At a high level, existing approaches to lemma discovery can be divided into two categories.[1] *Goal-directed* techniques, such as Zeno [Sonnex et al. 2012] and IsaPlanner [Dixon and Fleuriot 2004], use heuristics to generalize proof goals they encounter into lemmas. These techniques are fast, but limited in expressiveness. They cannot discover lemma (2) above because it is not a generalization of (1), but rather relates subterms of the original goal. *Theory exploration* techniques, such as HipSpec [Claessen et al. 2013], TheSy [Singher and Itzhaky 2021], or the inductive prover inside CVC4 [Reynolds and Kuncak 2015], exhaustively enumerate and prove candidate lemmas over a given vocabulary of functions, using evaluation on concrete or semi-concrete values to identify promising candidates. These techniques are expressive, but struggle to scale to large vocabularies and lemma sizes. While they might discover lemma (2) eventually, they get bogged down proving many irrelevant facts about natural numbers and lists from the large vocabulary admitted by our running example.

Our Solution: E-Graph Guided Lemma Discovery. In this paper, we attempt to make theory exploration more goal-directed by *only proposing lemmas that can make progress in the proof of the original goal.* This is easier said that done, however, since there are many ways we can attempt to prove our goal and get stuck, and it is not clear which one of these stuck proof states is the most promising source of lemmas. To address this challenge, we propose to use *e-graphs* [de Moura and Bjorner 2007; Nelson and Oppen 1980] and *equality saturation* [Tate et al. 2009; Willsey et al. 2021] to efficiently construct and compactly represent the space of *all* stuck proof states. We dub this approach *e-graph guided lemma discovery.* At a high-level, the approach works as follows:

(1) it uses equality saturation to compute the set of all terms that are known to be equal to the left- and right-hand sides of the original goal, given the currently available lemmas (including the induction hypothesis);

[1] See Sec. 6 for a more comprehensive and nuanced discussion.

(2) it then identifies equivalence classes of proof terms that are not yet *known* to be equal to each other, but are *likely* equal based on concrete evaluation, and proposes equalities between those classes as candidate lemmas.

We refer to these lemmas as *connector lemmas* because, if proven, they are guaranteed to connect (merge) two distinct equivalence classes of proof terms and therefore make progress in the proof. This approach to lemma discovery attempts to strike a balance between the expressiveness of theory exploration and the efficiency of goal-directed techniques: unlike unguided theory exploration, it is incomplete, but it is able to discover many useful lemmas, including lemma (2) above.

The CCLemma Prover. We implement our approach in an automated theorem prover called CCLemma,[2] which is included in our accompanying artifact [Kurashige et al. 2024]. We evaluate CCLemma on two benchmark suites from prior work as well as a new, more challenging benchmark suite of program optimization problems, like our `mtp` example above. Our experiments show that CCLemma is able to prove many properties that are out of reach for existing tools; for example, it can prove proposition (1) in less than a second. Compared against existing tools, CCLemma is consistently a top performer across all datasets, solving the benchmark suites from prior work faster and solving 50% more optimization problems than the next best tool.

Contributions. In summary, this paper makes the following contributions:

- We show how to perform inductive equational reasoning using e-graphs and equality saturation, and propose a new termination check that works well with this approach.
- We propose a new technique for lemma discovery, *e-graph guided lemma discovery*, which extracts connector lemmas from the e-graph of a stuck proof state.
- We implement our approach in the CCLemma prover, which empirically outperforms existing theory exploration tools both in terms of efficiency and scalability to challenging problems.

2 Overview

In this section we introduce inductive equational proofs, illustrate how to automate them using e-graphs and equality saturation, and finally show how to use e-graphs to guide lemma discovery. We use a simple running example throughout this section, in the interest of clarity.

2.1 Inductive Equational Proofs

Inductive equational reasoning is concerned with proving universally-quantified equalities about inductive datatypes and recursive functions.[3] Fig. 1 shows an inductive datatype `Nat` of natural numbers and two recursive functions over this type, written in Haskell-like syntax. From these definitions it follows that adding any number x to itself and then halving the result yields x back:

$$\forall x : \texttt{Nat}.\ \texttt{half}\ (\texttt{add}\ x\ x) \doteq x \qquad (\texttt{half\_double})$$

Though intuitively true, (`half_double`) needs to be proven by induction: Fig. 2 shows one possible proof. Importantly, this proof assumes that we already know the following lemma about `add`:[4]

$$\forall n, m : \texttt{Nat}.\ \texttt{add}\ n\ (\texttt{S}\ m) \doteq \texttt{S}\ (\texttt{add}\ n\ m) \qquad (\texttt{add\_right})$$

The proof proceeds by case analysis on x. In the base case, we can simply rewrite the left-hand side (LHS), `half (add Z Z)`, to the right hand side (RHS), `Z`, by definition of `add` and `half`. In the inductive case, rewriting the LHS to the RHS requires these definitions again, and additionally the use of the lemma (`add_right`) and the inductive hypothesis (IH).

[2] Its name stands for *Conjecturing Connector Lemmas*.
[3] In section Sec. 4.4 we also consider *conditional* equalities, but for now we focus on purely equational proofs.
[4] Sec. 2.3 will discuss how this lemma might be discovered automatically.

```
data Nat = Z                add :: Nat → Nat → Nat        half :: Nat → Nat
         | S Nat            add n m = case n of           half n = case n of
                              Z    → m                      Z      → Z
                              S n' → S (add n' m)           S Z    → Z
                                                            S (S n') → S (half n')
```

Fig. 1. Inductive datatype `Nat` of natural numbers and two recursive functions `add` and `half` over this type.

```
0  case x of
1    Z →
2          half (add Z Z)            -- def: add Z y = y
3        ≐ half Z                    -- def: half Z = Z
4        ≐ Z
5    S x' →
6          half (add (S x') (S x')) -- def: add (S x') y = S (add x' y)
7        ≐ half (S (add x' (S x'))) -- lemma add_right: add y (S x') = S (add y x')
8        ≐ half (S (S add x' x'))   -- def: half (S (S x')) = S (half x')
9        ≐ S (half (add x' x'))     -- IH: half (add x' x') = x'
10       ≐ S x'
```

Fig. 2. Proof outline of (`half_double`): $\forall x : \texttt{Nat}.\ \texttt{half}\ (\texttt{add}\ x\ x) \doteq x$.

While this proof is relatively easy for a human to come up with, discovering it automatically is not exactly straightforward (even if we assume the lemma (`add_right`) is given). For example, in line 9 of the proof, instead of applying the IH *left-to-right* to rewrite `half (add x' x')` to `x'`, we could have just as well applied it *right-to-left* to expand one of the two occurrences of `x'`; even worse, this incorrect way of applying the IH can be *repeated indefinitely*, leading to *divergence*:

$$\texttt{S}\ (\texttt{half}\ (\texttt{add}\ x'\,x')) \doteq \texttt{S}\ (\texttt{half}\ (\texttt{add}\ x'\,(\texttt{half}\ (\texttt{add}\ x'\,x')))) \doteq \texttt{S}\ (\texttt{half}\ (\texttt{add}\ x'\,(\texttt{half}\ (\texttt{add}\ x'\,(\texttt{half}\ (\texttt{add}\ x'\,x')))))) \doteq \ldots \quad (3)$$

An automated prover is required to make many such choices that may crucially affect the proof construction, its success/failure, and performance; three important ones are:

(1) *Which* equality to apply? (A definition? A lemma? The IH?)
(2) *Where* to apply this equality (to which sub-term and in which direction)?
(3) *When* to give up applying equalities and instead perform a case split?

Because they cannot always know the right choice, automated inductive provers conventionally perform *backtracking proof search*, trying different choices and backtracking when they get stuck.

2.2 Automating Proofs with E-Graphs and Equality Saturation

In recent years, *equality saturation* (EqSat) [Tate et al. 2009; Willsey et al. 2021] has emerged as an efficient alternative to backtracking search for equational reasoning. Instead of choosing which equality to apply and how, EqSat applies *all* possible equalities at once, storing the resulting set of terms in a data structure called an *e-graph* [Nelson and Oppen 1980]. What makes this simple idea work is that e-graphs can represent exponentially large (and even infinite) classes of equal terms

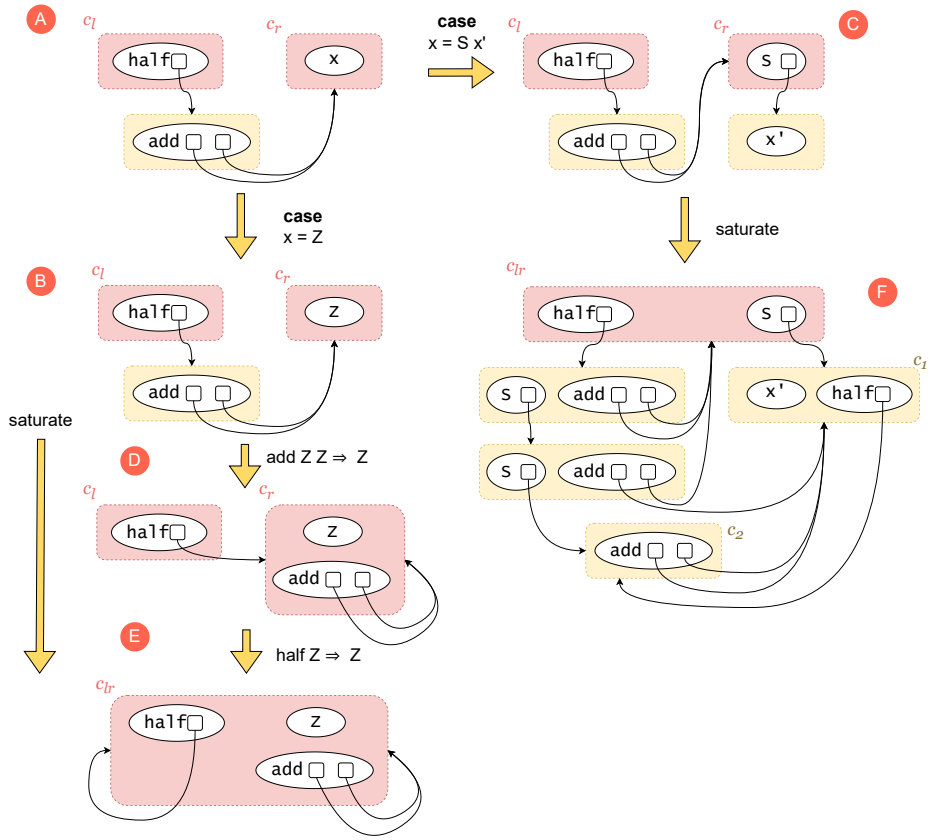

Fig. 3. E-graph-based proof search for (`half_double`).

compactly. While EqSat is traditionally used for equational reasoning in domains that do not require induction [Willsey et al. 2021; Yang et al. 2021], in this paper we show how it can be extended to inductive reasoning.[5] Fig. 3 illustrates how CCLemma uses EqSat to prove (`half_double`).

Proof States as E-Graphs. CCLemma represents the initial proof state as the e-graph A, which stores the LHS and the RHS of (`half_double`), namely `half (add x x)` and `x`. You can think of this e-graph as simply containing the abstract syntax trees (ASTs) of the two terms, but with sharing between common sub-terms (here, just `x`); in particular, the ovals in the figure—called *e-nodes*—correspond to AST nodes. Unlike in an AST, however, edges in an e-graph do not point to other e-nodes, and instead point to *e-classes*—sets of e-nodes proven equal—represented in the figure as colored rectangles. Since we haven't proven anything equal yet, all the e-classes in our initial e-graph A are singletons. We purposefully highlight the two e-classes corresponding to the LHS and the RHS of (`half_double`), marked c_l and c_r in the figure. CCLemma's ultimate goal is to *merge* these two e-classes into one, which would constitute a proof that they are equal.

[5] TheSy [Singher and Itzhaky 2021] uses a similar idea for its inductive proofs, but in this paper we extend it to full-fledged proof search; see Sec. 6 for a more comprehensive comparison.

Rewrite Rules. Equality saturation grows the e-graph via a set of *rewrite rules*. CCLemma distinguishes two kinds of rewrite rules: *reductions* are derived from function definitions, such as:

$$\texttt{add Z}\ ?n \Longrightarrow ?n \qquad \texttt{add (S}\ ?n)\ ?m \Longrightarrow \texttt{S (add}\ ?n\ ?m) \qquad \texttt{half Z} \Longrightarrow \texttt{Z} \qquad \ldots$$

Lemma rewrites include user-provided lemmas and the inductive hypothesis (IH):

$$\texttt{add\_right} : \texttt{add}\ ?n\ (\texttt{S}\ ?m) \Leftrightarrow \texttt{S (add}\ ?n\ ?m) \qquad \text{IH} : \texttt{half (add}\ ?y\ ?y) \Leftrightarrow ?y \mid ?y \prec x$$

Lemma rewrites are *bidirectional*, *i.e.* they can be applied both ways (unless one of the sides of the lemma has variables that do not appear in the other side). The IH rewrite is also *conditional*: it only applies when the term matched by the pattern variable $?y$ is smaller than the universally-quantified variable x of the goal. This check is necessary to ensure *termination*, *i.e.* that the induction is well-founded (see Sec. 3.3 for details of how we define this check).

Case Splitting. Since none of the rewrite rules apply to any terms in the e-graph (A), the only way to proceed is to case-split on x, the only universal variable in our goal. This transforms the original goal into two sub-goals, represented by the e-graphs (B) and (C), where x is replaced by the corresponding constructors of `Nat` with fresh universal variables as arguments. Both of these sub-goals have to be solved in order to prove the original proposition.

Saturation. To prove sub-goal (B), CCLemma invokes equality saturation on its e-graph. This e-graph contains the term `add Z Z`, which matches the reduction `add Z` $?n \Longrightarrow ?n$, instantiated to `add Z Z` $\Longrightarrow$ `Z`. As a result of this rewrite, the e-classes containing `add Z Z` and `Z` are merged into one (if the RHS of the rewrite weren't already in the e-graph, it would be added). Now the e-graph also contains the term `half Z`, which matches the reduction `half Z` $\Longrightarrow$ `Z`. Applying this rewrite merges the e-classes c_l and c_r, thus completing the proof of this sub-goal.

Sub-goal (C) is solved similarly, except that here we also get to apply our two lemma rewrites. Importantly, we need not worry about which rewrites to apply where: EqSat applies all of them in arbitrary order, until the e-graph is *saturated*, *i.e.* further rule applications do not change the e-graph. To see why we need not worry about going down a rabbit hole like in (3), consider what happens when we apply the IH to rewrite `half (add` $x'\ x'$`)` to x'. This places both terms into the same e-class c_1, creating a cycle in the e-graph between c_1 and c_2. This cycle represents an infinite set of terms and equalities, including those described in (3). If we now try to apply the IH backwards to x'—the only term it can be applied to, since only x' is smaller than x—this will not lead to any further changes in the e-graph.

In summary, EqSat gives CCLemma simple answers to the three choices we discussed earlier:

(1) *Which* equality to apply? All of them!
(2) *Where* to apply this equality? Everywhere!
(3) *When* to give up and case-split? When the e-graph is saturated, but the LHS and the RHS have not been merged.[6]

2.3 E-graph Guided Lemma Discovery

In the previous section, our proof of (`half_double`) relied on the lemma (`add_right`); but where would this lemma come from? In the setting of automated theorem proving, we cannot expect the user to provide the necessary lemmas; instead the prover needs to discover them automatically.

The most effective known approach to lemma discovery is *theory exploration* [Claessen et al. 2013; Johansson 2019; Singher and Itzhaky 2021]. Theory exploration essentially enumerates all

[6]One remaining question is *which variable* to case-split on; we discuss this further in Sec. 4.4.

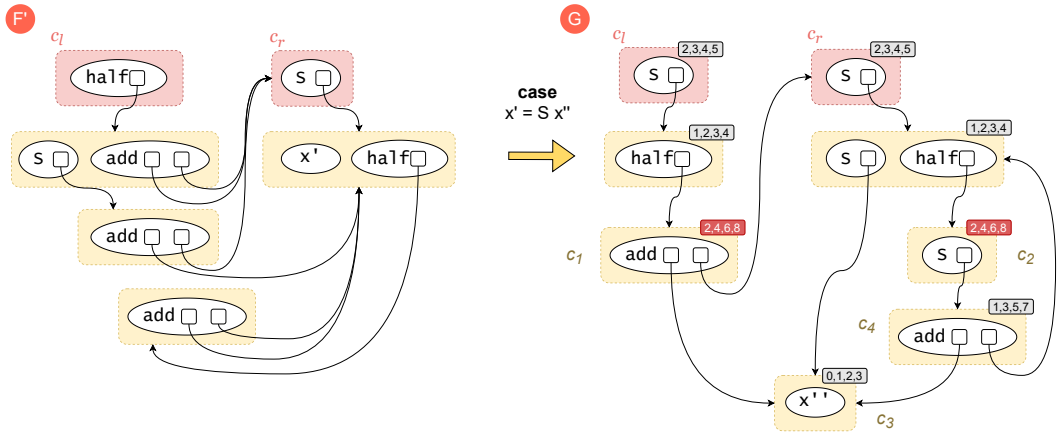

Fig. 4. Discovering the connector lemma (`add_right`) using the e-graph from the proof of (`half_double`). Terms that are not in normal form have been removed—as well as an unnecessary application of the inductive hypothesis—to simplify its presentation.

possible lemmas over the vocabulary of the target property—in our case, `Z`, `S`, `add`, and `half`—and tries to prove them, until one of the proven lemmas helps prove the property.[7]

Although existing theory exploration tools use clever heuristics to prune the space of candidate lemmas, they can still have trouble scaling to large lemmas and large vocabularies. Our key observation is that a significant number of lemmas proven by these tools are irrelevant to the target property, *i.e.* they cannot be applied in its proof to establish new equalities. In our running example, one of the lemmas theory exploration discovers is associativity of addition:

$$\forall x, y, z : \texttt{Nat}.\ \texttt{add}\ x\ (\texttt{add}\ y\ z) \doteq \texttt{add}\ (\texttt{add}\ x\ y)\ z$$

To see why this lemma cannot make progress in the proof of (`half_double`), consider all the proof states in Fig. 3: the only terms this lemma matches are `add Z (add Z Z)` and similar in e-graph D, but all these terms are already in the same e-class, so the lemma does not establish any new equalities.

This observation yields a powerful lemma discovery heuristic for EqSat-based inductive reasoning: *only consider lemmas that can connect (merge) two different e-classes in a proof state.* We refer to such lemmas as *connector lemmas*, and to the overall approach as *e-graph guided lemma discovery.*

Extracting Connector Lemmas. How does CCLemma prove the proposition (`half_double`) if the lemma (`add_right`) is not given? In this case, saturating the e-graph C from Fig. 3 does not actually solve the goal: instead of leading to the e-graph F, saturation results in the e-graph F', shown in Fig. 4. From there, a second level of case-splitting and saturation generates the e-graph G (to avoid clutter, we removed from this e-graph all terms that are not in normal form *wrt.* to the reductions; as you will see in Sec. 3, CCLemma in fact also removes such terms from the e-graph). Note that the e-class c_1 in this e-graph contains the term `add` x''`(S (S` x''`))`, while the e-class c_2 contains the term `S (add` x''`(S` x''`))`. CCLemma conjectures that these two terms should be equal, and then obtains the connector lemma (`add_right`) by *generalizing* their common sub-terms, `S` x'' into a fresh variable y.

[7]Alternatively, theory exploration can be used in an offline mode, to discover all potentially useful lemmas for a given library of functions, without a specific target property in mind. While offline theory exploration has its advantages—*e.g.* we can afford to spend hours searching for lemmas—it is not feasible when the library is large or when new functions are added in parallel with new properties to prove; for this reason, we focus on online lemma discovery in this paper.

Pruning with C-vecs. Even if we restrict lemma discovery only to connector lemmas, this still permits too many candidate lemmas to possibly consider. Fortunately, most of these candidates are easily proven untrue. For example, if we tried to connect the e-classes c_3 and c_4 in the e-graph G, we would get spurious lemmas such as:

$$\forall x : \texttt{Nat.}\ \texttt{add}\ x\ (\texttt{S}\ x) \doteq x$$

This lemma can be shown spurious by evaluating it at $x = \texttt{Z}$: we discover that $\texttt{add}\ \texttt{Z}\ (\texttt{S}\ \texttt{Z}) = \texttt{S}\ \texttt{Z} \neq \texttt{Z}$. In practice, most theory exploration tools use *counterexamples* like these to quickly filter out the vast majority of untrue lemmas—of which there are many more than true lemmas—to avoid wasting time trying to prove them. They generate random concrete values for the variables in a lemma and then check whether the lemma holds for these values. If implemented naïvely, however, the concrete evaluation itself can become a bottleneck.

To implement concrete evaluation efficiently, CCLemma borrows an idea from Ruler [Nandi et al. 2021]: it leverages an e-graph mechanism called *e-class analysis* [Willsey et al. 2021], to incrementally propagate concrete values through the e-graph, taking advantage of sharing to avoid recomputation. More concretely, CCLemma annotates each e-class with a *characteristic vector* (or *c-vec* for short) as shown in the e-graph G in Fig. 4. The c-vecs in the figure were computed by first initializing the universally-quantified variable x'' with a random vector—here $[0, 1, 2, 3]$—and then incrementally computing the c-vecs of other e-classes bottom-up.

With the c-vecs in place, CCLemma avoids conjecturing many spurious lemmas by only connecting e-classes whose c-vecs match. For example, CCLemma will extract lemmas from the e-classes $\langle c_1, c_2 \rangle$ (by enumerating all pairs of terms from these e-classes, following cycles up to a fixed depth), but it will not extract lemmas from $\langle c_3, c_4 \rangle$. Another benefit of the c-vecs is that if we discover a mismatch between the c-vecs of c_l and c_r, we can immediately rule the property invalid.

Summary. In summary, the key insight of e-graph guided lemma discovery is to only conjecture lemmas that can establish new equalities between terms already explored by the proof search. E-graphs make this idea feasible for three reasons:

(1) E-graphs compactly represent the terms and equalities reachable by proof search, allowing us to conjecture many connector lemmas simultaneously.
(2) Even if lemma discovery finds a large number of lemmas, applying them to the proof of the main goal and other lemmas is efficient, thanks to equality saturation.
(3) Incremental c-vec analysis enables efficient pruning of spurious lemmas.

CCLemma combines the techniques described in Sections 2.2 and 2.3 into a proof search algorithm that processes goals by applying the following steps:

(1) **Saturate.** Run equality saturation and check whether the LHS and RHS are in the same e-class.
(2) **Case Split.** Select a variable in the goal and perform a case split according to its datatype, creating one or more new sub-goals.
(3) **Conjecture** new connector lemmas, which themselves become new goals. Successfully proven lemmas are introduced as rewrite rules.

Proof search is then just scheduling which goal to try next to ensure the original goal is discharged.

3 Equational Reasoning

In this section we introduce the proof system used by CCLemma and formalize equality saturation. Our formalization of inductive equational reasoning in the context of a higher-order rewriting system is adopted from CycleQ [Jones et al. 2021] (although our proofs are not cyclic).

$$\textsc{Refl}\ \frac{}{\mathcal{R};\Gamma \vdash M \doteq M} \qquad \textsc{Rewr}\ \frac{\mathcal{R};\Gamma \vdash L' \doteq R' \quad L \to^*_{\mathcal{R}} L' \quad R \to^*_{\mathcal{R}} R'}{\mathcal{R};\Gamma \vdash L \doteq R}$$

$$\textsc{Ind}\ \frac{\overline{x_i} = \mathsf{vars}(\Gamma) \quad \theta = [\overline{x_i \mapsto ?y_i}] \quad \mathcal{R}, (L\theta \Leftrightarrow R\theta \mid \overline{?y_i} \prec \overline{x_i}); \Gamma \vdash L \doteq R}{\mathcal{R};\Gamma \vdash L \doteq R}$$

$$\textsc{Case}\ \frac{\forall k : \overline{\tau_i} \to d \in \Sigma_{\mathsf{cons}}(d) \quad \theta = [x \mapsto k\ \overline{y_i}] \quad \mathcal{R}\theta; \Gamma, \overline{y_i : \tau_i} \vdash L\theta \doteq R\theta}{\mathcal{R};\Gamma, x : d \vdash L \doteq R}$$

Fig. 5. The inference rules of inductive equational reasoning.

3.1 Preliminaries

Types and Terms. We assume a fixed *signature*, which consists of algebraic datatypes D and function symbols Σ. The *types*, τ, σ of the system include datatypes and functions:[8]

$$\tau, \sigma ::= d \in D \mid \tau \to \sigma$$

The signature assigns types to all function symbols, written $f : \tau \in \Sigma$. Function symbols are divided into two categories: *constructors* Σ_{cons} (such as `Z` and `S`) and *defined functions* Σ_{def} (such as `add`). *Terms* are constructed from variables, function symbols, and applications:

$$L, M, N, R ::= x \mid f \in \Sigma \mid M\ N$$

Terms that have no variables or defined functions are called *values*. A *type environment* (Γ or Δ) is a set of type bindings $x : \tau$; we write Γ, Δ for a disjoint union of two environments. The typing rules are standard; in the following we restrict our attention to well-typed terms.

Term Rewriting. A *(conditional) rewrite rule* is written $M \Rightarrow N \mid \phi$, where M and N are terms that have the same type $d \in D$ in some environment Γ and ϕ is a formula over $\mathsf{vars}(M)$. The exact logic used in ϕ is not important, as long as the term-rewriting engine can check its validity. We write $M \Rightarrow N$ as shorthand for $M \Rightarrow N \mid \top$. A *bidirectional* rewrite notation $M \Leftrightarrow N$ stands for a set of at most two rewrites rules, which includes $M \Rightarrow N$ if $\mathsf{vars}(N) \subseteq \mathsf{vars}(M)$, and includes $N \Rightarrow M$ if $\mathsf{vars}(M) \subseteq \mathsf{vars}(N)$.

A *context* is either a hole or an application with a context on one side:

$$C[\cdot] ::= \cdot \mid C[\cdot]\ M \mid M\ C[\cdot]$$

A *substitution* $\theta = [\overline{x_i \mapsto M_i}]$ is a partial function from variables to terms.[9] Applying a substitution to a term is denoted $M\theta$.

The rewrite rule r of the form $M \Rightarrow N \mid \phi$ induces a *one-step reduction* relation $M' \to_r N'$ if M' is of the form $C[M\theta]$, N' is of the form $C[N\theta]$, and the formula $\phi\theta$ holds. In this case, θ is called the *match substitution*. For a set of rewrite rules $\mathcal{R}$, we define the *one-step reduction* $M \to_{\mathcal{R}} N$ if $M \to_r N$ for some $r \in \mathcal{R}$. The *reduction* relation $M \to^*_{\mathcal{R}} N$ is the reflexive transitive closure of one-step reduction $\to_{\mathcal{R}}$. We omit the subscript if the rule set is clear from context.

3.2 Proof System

Our proof system operates over *equations* of the form $\mathcal{R};\Gamma \vdash L \doteq R$, where L and R are terms that have the same type d in Γ. The rewrites $\mathcal{R}$ encode the reductions and lemmas available to the proof. We assume that *pattern variables*, $\mathsf{vars}(\mathcal{R})$, are disjoint from the variables used in L and R, $\mathsf{vars}(\Gamma)$; to make this explicit, we use a naming convention $?x, ?y$ for pattern variables.

[8] This formalization is simply-typed in the interest of brevity; CCLemma supports polymorphic datatypes.

[9] Hereafter we write $\overline{X}$ to denote zero or more occurrences of a syntactic element X.

$$\textsc{Ind}\ \dfrac{\textsc{Case}\ \dfrac{\textsc{Rewr}\ \dfrac{\textsc{Refl}\ \dfrac{}{\mathcal{R}';\cdot \vdash \mathtt{Z} \doteq \mathtt{Z}}}{\mathcal{R}';\cdot \vdash \mathtt{half}\ (\mathtt{add}\ \mathtt{Z}\ \mathtt{Z}) \doteq \mathtt{Z}} \qquad \textsc{Rewr}\ \dfrac{\textsc{Rewr}\ \dfrac{\textsc{Refl}\ \dfrac{}{\ldots \vdash \mathtt{S}\ x' \doteq \mathtt{S}\ x'} \quad x' \prec \mathtt{S}\ x'}{\ldots \vdash (\mathtt{S}\ (\mathtt{half}\ (\mathtt{add}\ x'\ x'))) \doteq \mathtt{S}\ x'}}{\ldots \vdash \mathtt{half}\ (\mathtt{S}\ (\mathtt{add}\ (\mathtt{S}\ x')\ x')) \doteq \mathtt{S}\ x'}}{\mathcal{R}'';x' : \mathtt{Nat} \vdash \mathtt{half}\ (\mathtt{add}\ (\mathtt{S}\ x')\ (\mathtt{S}\ x')) \doteq \mathtt{S}\ x'}}{\mathcal{R};\mathtt{half}\ (\mathtt{add}\ ?y\ ?y) \Leftrightarrow ?y \mid ?y \prec x; x : \mathtt{Nat} \vdash \mathtt{half}\ (\mathtt{add}\ x\ x) \doteq x}}{\mathcal{R};x : \mathtt{Nat} \vdash \mathtt{half}\ (\mathtt{add}\ x\ x) \doteq x}$$

Fig. 6. A proof derivation for (half_double) (with the lemma (add_right) given).

The proof system has four inference rules, shown in Fig. 5:

- Refl is the usual reflexivity axiom, which concludes that proof once the two sides of the equation are syntactically equal.
- Rewr allows rewriting both sides of the equation using the rewrites in $\mathcal{R}$ for any number of steps; note that this rules does not require L' and R' to be in normal form, since $\mathcal{R}$ is not necessarily normalizing.
- Ind adds the induction hypothesis (IH) to $\mathcal{R}$. The IH is a bidirectional conditional rewrite $L\theta \Leftrightarrow R\theta \mid \overline{?y_i} \prec \overline{x_i}$, where θ simply renames the original variables $\overline{x_i}$ used in L and R to fresh variables $\overline{?y_i}$. The condition $\overline{?y_i} \prec \overline{x_i}$ ensures termination and is discussed in more detail later in this section. Ind is meant to be applied once, at the root of the derivation; although it is possible to allow multiple applications, simulating cyclic proofs [Jones et al. 2021], we have not found a need for this in practice.
- Case picks a variable $x : d$ in Γ to case-split on, and creates a subgoal for each constructor k of the datatype d; we assume that the variables $\overline{y_i}$, used as constructor parameters, are fresh. Note that Case substitutes the scrutinee x not only in the equation, but also in the rewrites $\mathcal{R}$; the only place where x can appear in $\mathcal{R}$ is in the termination condition of the IH, and this substitution enables a simple syntactic definition of the $\prec$ order.

Example derivation. Fig. 6 shows a proof derivation for our example from Sec. 2:

$$\mathcal{R};x : \mathtt{Nat} \vdash \mathtt{half}\ (\mathtt{add}\ x\ x) \doteq x$$

Here $\mathcal{R}$ includes the reductions for `add` and `half`, as well as the lemma (add_right):

$$\mathtt{add}\ ?n\ (\mathtt{S}\ ?m) \Leftrightarrow \mathtt{S}\ (\mathtt{add}\ ?n\ ?m)$$

The proof starts with an application of Ind, which adds the IH rewrite. Next, we apply Case to obtain two sub-goals, where x is replaced with $\mathtt{Z}$ and $\mathtt{S}\ x'$ respectively; let us focus on the second, more interesting sub-goal. Here, the rewrite rules $\mathcal{R}''$ include the original $\mathcal{R}$ plus the following IH, which has been modified by Case:

$$\mathtt{half}\ (\mathtt{add}\ ?y\ ?y) \Leftrightarrow ?y \mid ?y \prec \mathtt{S}\ x'$$

The following sequence of Rewr application could be merged into a single step, but we split them to separate lemma applications from reductions: the first step applies the lemma (add_right), the second one applies two reductions, and finally the third step applies the IH to the sub-term $\mathtt{half}\ (\mathtt{add}\ x'\ x')$. The match substitution for this last rewrite is $\theta = [?y \mapsto x']$, and so the IH's termination condition becomes $x' \prec \mathtt{S}\ x'$, highlighted in red in the figure. Let us now discuss termination checking in more detail.

3.3 Checking Termination

Our proof system uses a simple yet surprisingly effective termination checking mechanism based on two key ideas: (i) when the equation has multiple parameters, for each IH invocation, the termination check only needs to consider those parameters that *have been case-split on*; (ii) for each parameter, we use the *substructure relation* as the descreasing order. In the rest of this section, we formally define the $\prec$ order used in the IH and prove that this order is well-founded, *i.e.* there is no infinite descending chain $\overline{M_i} \succ \overline{M_i^1} \succ \overline{M_i^2} \succ \cdots$.

Order on Terms. We start by defining an order on single terms, and then lift it to tuples of terms.

Definition 3.1. Given two terms M and N, we say that $M \prec N$ iff M is a strict subterm of N.

This order is obviously well-founded because M has a strictly smaller size than N.

This simple syntactic definition actually admits many useful proofs. As we have illustrated in Fig. 6, this works because of the renaming trick performed by the Case rule: thanks to this trick, the sub-term relation $M \prec N$ subsumes the more common condition that "M came out of a case split of N", more formally: (i) N appears as the scrutinee in a prior Case, and M is one of the fresh variables introduced in its premises, or (ii) transitively, there exists M' such that $M \prec M'$ and $M' \prec N$.

Branch-wise Order on Tuples. Judgements in proofs typically contain more than one variable, and requiring that all of them decrease is much too strict—that would preclude most proofs and result in an ineffective proof system. Often, different IH applications decrease different variables, like in the proof of $\forall x, y : \texttt{Nat}.\ \texttt{add}\ x\ y \doteq \texttt{add}\ y\ x$, where sometimes x decreases and sometimes y does.

A common way to lift a well-founded order to tuples is the lexicographic order, $\prec_{\text{lex}}$, which orders variables and compares them using $\prec$ in that order. However, this requires committing a fixed variable order, which must be consistent across all uses of the IH in the proof. We find this inconvenient for proof search since a choice of ordering made while exploring one branch of the proof may preclude the use of the IH in another branch, leading to backtracking. Moreover, sometimes, a single lexicographic order applicable to all branches might not even exist.

Example 3.2. Consider function `dec` in Fig. 7 (left). This function always returns `True`, but depending on `x` it does so by decreasing either `y` or `z` until it reaches a base case. We elide this proof (because it has the exact same structure as its definition, except with IH invocations for every recursive call to `dec`), but importantly, the proof not only demonstrates a case where the lexicographic ordering of variables matters, it actually is a case where *no* such ordering exists. This is because the function cases on `x` into two main branches: the first where `y` decreases while `z` increases, and second where the opposite happens.

We propose a more robust method to lift $\prec$ to tuples, which we call a *tree order*.

Definition 3.3 (Case tree). Given a tuple of datatypes $\overline{d_i}$, a *case tree* is a tree whose nodes are labeled with indexed variables v_i, and where children of a node v_i are labeled with constructors of d_i. A case tree t has a root $t.\text{root}$ and a map of labeled children $t.\text{children} = \{k \mapsto t', \ldots\}$.

Example 3.4. Given the tuple $\langle \texttt{Nat}, \texttt{Nat}, \texttt{Nat} \rangle$—for example, representing the variables of `dec` from Example 3.2—one possible case tree is shown in Fig. 7 (right), described by $t = \{\text{root: } v_0, \text{children: } \{\texttt{Z} \mapsto \{\text{root: } v_1, \text{children: } \varnothing\}, \{\texttt{S} \mapsto \{\text{root: } v_2, \text{children: } \varnothing\}\}$.

In general, a case tree need not have distinct variables in all the nodes; for example, if we replaced all occurrences of v_1 in t with v_2, it would still be a valid case tree, but the order it induces would coincide with the lexicographic order $\langle v_0, v_2 \rangle$.

```
0 dec :: Nat → Nat → Nat → Bool
1 dec x y z = case x of
2               Z     → case y of
3                         Z     → True
4                         (S y') → dec x y' (S z)
5               (S x') → case z of
6                         Z     → True
7                         (S z') → dec x (S y) S z'
```

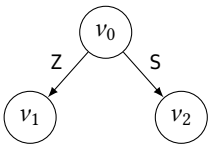

Fig. 7. (Left) Definition of dec, a function that always (eventually) returns True. We will demonstrate that the proof of this fact can be shown to terminate using a tree order. (Right) A case tree for the arguments of dec.

Definition 3.5 (Tree order). Given a case tree t, we define a relation $\prec_t \subseteq \prod \overline{d_i} \times \prod \overline{d_i}$ between tuples of terms as follows:

$$\overline{v_i^1} \prec_t \overline{v_i^2} \quad \iff \quad r^1 \prec r^2 \vee \bigvee_{(k \mapsto t') \in C} r^1 = r^2 \wedge \mathsf{is}(k, r^1) \wedge \left(\overline{v_i^1} \prec_{t'} \overline{v_i^2}\right)$$

where $r = t.\mathsf{root}$, $C = t.\mathsf{children}$. The notation r^1, r^2 refers to the variable taken from the left-hand side v_i^1 and right-hand side v_i^2, respectively. $\mathsf{is}(k, v)$ is a predicate denoting that the discriminator of v is the constructor k. Each variable v may only have a single constructor k; formally, $\mathsf{is}(k_1, v) \wedge \mathsf{is}(k_2, v) \implies k_1 = k_2$.

Example 3.6. Given the case tree from Example 3.4 and two triples $\langle v_0^1, v_1^1, v_2^1 \rangle = \langle \mathsf{Z}, y', \mathsf{S}\ z \rangle$ and $\langle v_0^2, v_1^2, v_2^2 \rangle = \langle \mathsf{Z}, \mathsf{S}\ y', z \rangle$, we have $\langle v_0^1, v_1^1, v_2^1 \rangle \prec_t \langle v_0^2, v_1^2, v_2^2 \rangle$ because

$$\begin{array}{lllll} & v_0^1 \prec v_0^2 & \mathsf{Z} \prec \mathsf{Z} & \text{no} \\ \vee & v_0^1 = v_0^2 \wedge \mathsf{is}(\mathsf{Z}, v_0^1) \wedge v_1^1 \prec v_1^2 & \mathsf{Z} = \mathsf{Z} \wedge \mathsf{is}(\mathsf{Z}, \mathsf{Z}) \wedge y' \prec \mathsf{S}\ y' & \text{yes!} \\ \vee & v_0^1 = v_0^2 \wedge \mathsf{is}(\mathsf{S}, v_0^1) \wedge v_2^1 \prec v_2^2 & \mathsf{Z} = \mathsf{Z} \wedge \mathsf{is}(\mathsf{S}, \mathsf{Z}) \wedge \mathsf{S}\ z \prec z & \text{no} \end{array}$$

Similarly, for the triples $\langle v_3^1, v_4^1, v_5^1 \rangle = \langle \mathsf{S}\ x', \mathsf{S}\ y, z' \rangle$ and $\langle v_3^2, v_4^2, v_5^2 \rangle = \langle \mathsf{S}\ x', y, \mathsf{S}\ z' \rangle$, we have $\langle v_3^1, v_4^1, v_5^1 \rangle \prec_t \langle v_3^2, v_4^2, v_5^2 \rangle$. These two comparisons correspond to the two IH invocations necessary to prove that dec always evalutes to True.

Theorem 3.7. *The tree order $\prec_t$ is well-founded.*

Proof. The proof is by induction on the structure of the case tree. If the tree is a leaf, its tree order is simply the order on terms, which is well-founded. Now let the tree have a root r and children $\{\overline{k \mapsto c}\}$, such that $\prec_c$ is well-founded for all c. Suppose there is an infinite descending chain $\overline{M_i} \succ \overline{M_i^1} \succ \overline{M_i^2} \succ \cdots$. Note that the root term M_r can never increase by definition of $\prec_t$. Hence, the chain must have an infinite suffix where the root term M_r^j does not change; but then, by definition of $\prec_r$, there exists a constructor k and a child $k \mapsto c$ such that $\mathsf{is}(k, M_r^j) \wedge \overline{M_i^j} \prec_c \overline{M_i^{j+1}}$ holds for every j in the suffix, *i.e.* all elements of the suffix are ordered by $\prec_c$ of the *same* child. This is in contradiction to $\prec_c$ being well-founded. This precludes the existence of the infinite descending chain $\overline{M_i^j}$ above. □

Checking Termination Locally using Tree Order. While more expressive than a lexicographic ordering, the main benefit of tree order is that it allows checking termination *locally* in each branch of the proof, while assuring that a consistent order relation can be constructed from the separate branches. More concretely, consider an equation $\mathcal{R}; \Gamma \vdash L \doteq R$ with parameters $\overline{x_i} = \mathsf{vars}(\Gamma)$.

Consider one branch of the proof of this equation, where only some subset of the parameters have been case-split on; let the tuple $\overline{x_j}$ be these parameters in the order they were case-split on, and let $\overline{N_j}$ be the constructor terms they have been case-split into. Now if we apply the IH rewrite with the match substitution θ, it is sufficient to check lexicographic ordering only on the case-split parameters: $\overline{?x_j\theta} \prec_{\text{lex}} \overline{N_j}$. Although the subset and the order of the compared parameters might be different in different branches, it is still sound, because we can assemble all these local checks into a single global check *wrt.* a tree order whose case tree follows the case-split structure of the proof.

Example 3.8. Consider the proof of $\mathtt{dec}(x, y, z) \doteq \mathtt{True}$, which first case-splits on x, and then on y in one branch and on z in another. Consider the branch $[x \mapsto \mathtt{Z}, y \mapsto \mathtt{S}\ y']$ and an invocation of the IH with match substitution $[?x \mapsto \mathtt{Z}, ?y \mapsto y', ?z \mapsto \mathtt{S}\ z]$. This invocation is considered terminating, despite the parameter $?z$ increasing, because in this branch we only need to check that $\langle ?x, ?y\rangle$ decreases: $\langle \mathtt{Z}, y'\rangle \prec_{\text{lex}} \langle \mathtt{Z}, \mathtt{S}\ y'\rangle$.

Now consider a different branch, $[x \mapsto \mathtt{S}\ x', z \mapsto \mathtt{S}\ z']$ and an invocation of the IH with match substitution $[?x \mapsto \mathtt{S}\ x', ?y \mapsto \mathtt{S}\ y, ?z \mapsto z']$. This invocation is also considered terminating because in this branch we only need to check that $\langle ?x, ?z\rangle$ decreases: $\langle \mathtt{S}\ x', z'\rangle \prec_{\text{lex}} \langle \mathtt{S}\ x', \mathtt{S}\ z'\rangle$.

These two checks use different lexicographic orders, but they can be combined into a tree order by a piecewise composition conditioned on $?x$. For the tuple $\langle ?x, ?y, ?z\rangle$, the appropriate well-founded relation is $\prec_t$ shown in Example 3.6.

3.4 E-Graphs and Equality Saturation

For the purposes of this paper, the internal structure of the e-graph is not important; we simply view an *e-graph* E as a collection of *e-classes*, and each e-class c as a (possibly infinite) collection of terms. More formally, we assume the existence of the following operations on e-graphs:

- $\mathsf{classes}(E)$ is the set of e-classes in an e-graph E;
- $\mathsf{terms}(c)$ is the set of terms in an e-class c;
- $E[M]$ is a partial function that, given a term M, returns the e-class c such that $M \in \mathsf{terms}(c)$;
- $\mathsf{eg}(\overline{M_i})$ creates an e-graph that represents all the terms $\overline{M_i}$ in singleton e-classes;
- $\mathsf{saturate}(E, \mathcal{R})$ saturates the e-graph E with the rewrites $\mathcal{R}$, *i.e.* applies the rewrites in $\mathcal{R}$ to terms in E, until E contains all equalities induced by the rewrites.

These operations have the following properties:

- $E[M] = c$ if and only if $\exists c \in \mathsf{classes}(E).M \in \mathsf{terms}(c)$;
- if $E = \mathsf{eg}(M_0, \ldots, M_n)$, then $\forall i \in 0..n.\exists c \in E.\mathsf{terms}(c) = \{M_i\}$;
- if $E' = \mathsf{saturate}(E, \mathcal{R})$, then $\forall c \in E.\exists c' \in E'.\mathsf{terms}(c') = \left\{M' \mid M \to^*_{\mathcal{R}} M', M \in \mathsf{terms}(c)\right\}$.

4 The Proof Search Algorithm

This section describes CCLemma's proof search algorithm, which automatically finds derivations in the proof system of Sec. 3 and also discovers auxiliary lemmas that are necessary to complete the proof. Sec. 4.1 explains how the algorithm works at a high level and the following sections delve into the details of each step.

4.1 Top-Level Algorithm

The top-level algorithm is shown in Fig. 8. The procedure Prove takes as input the top-level equation to prove $\mathcal{R}_0 \vdash L \doteq R$[10] where the rewrites $\mathcal{R}_0$ contain reductions defining the functions in Σ_{def} and (optionally) any user-provided lemmas; the procedure returns Valid or Invalid if the property is proven or disproven, or it might not terminate.

[10]In this section, we omit the type environment Γ from equations and goals to avoid notational clutter.

Input: Equation $\mathcal{R}_0 \vdash L_{\text{top}} \doteq R_{\text{top}}$
Output: Valid or Invalid

1: **function** Prove($\mathcal{R}_0 \vdash L_{\text{top}} \doteq R_{\text{top}}$)
2: $G_{\text{top}} \leftarrow \langle L_{\text{top}} \doteq R_{\text{top}}, \mathsf{IH}(L_{\text{top}}, R_{\text{top}}), \mathsf{eg}(L_{\text{top}}, R_{\text{top}})\rangle$ ▷ Top-level goal
3: $\mathcal{G} \leftarrow [G_{\text{top}}]$ ▷ Initialize proof state
4: $\mathcal{R} \leftarrow \mathcal{R}_0$ ▷ Initialize rewrites
5: **while** true **do**
6: $G \leftarrow$ NextGoal($\mathcal{G}$) ▷ Select next goal to work on
7: **if** $G = \bot$ **then return** Invalid ▷ No more active goals: return invalid
8: $G.E \leftarrow \mathsf{saturate}(G.E, \mathcal{R} + G.IH)$ ▷ Saturate the goal with rewrites and IH
9: **if** $G.E[G.L] = G.E[G.R]$ **then** ▷ Are LHS and RHS in the same e-class?
10: MarkSolved($\mathcal{G}, G$) ▷ Mark goal solved
11: $L \doteq R \leftarrow$ ParentLemma($\mathcal{G}, G$) ▷ Get the lemma this goal belongs to
12: **if** IsProven($\mathcal{G}, L \doteq R$) **then** ▷ Are all lemma's goals solved?
13: **if** $(L \doteq R) = (L_{\text{top}} \doteq R_{\text{top}})$ **then return** Valid ▷ Lemma is the top-level prop: return valid
14: $\mathcal{R} \leftarrow \mathcal{R} + \{L \Leftrightarrow R\}$ ▷ Add proven lemma to the rewrites
15: **else**
16: MarkInactive($\mathcal{G}, G$) ▷ Mark goal inactive
17: $\mathcal{G} \leftarrow \mathcal{G} +$ ConjConnectorLemmas($G, \mathcal{R}$) ▷ Add connector lemmas as goals
18: $\mathcal{G} \leftarrow \mathcal{G} +$ Case-Split(G) ▷ Add case split goals

Fig. 8. The top-level proof algorithm.

The algorithm maintains a *proof state* $\mathcal{G}$, which at a high level can be thought of as a worklist of *goals*. A goal $G = \langle L \doteq R, IH, E\rangle$ is defined by an equality $L \doteq R$, its inductive hypothesis IH, which is a conditional rewrite rule, and an e-graph E, which stores the already proven equalities. We write $G.L$ and $G.R$ to denote the LHS and RHS of the equality, respectively. The proof state $\mathcal{G}$ does some bookkeeping to keep track of the progress of each goal as well as the correspondence between goals and the lemmas they relate to. Sec. 4.3 discusses the data structure behind $\mathcal{G}$ and the bookkeeping in more detail.

The algorithm starts by creating a top-level goal G_{top} from its input equation (line 2) and adding it to the proof state (line 3). Here $\mathsf{IH}(L_{\text{top}}, R_{\text{top}})$ creates the IH rewrite as prescribed by the rule Ind in Fig. 5. The goal's e-graph is initialized to store the two sides L_{top} and R_{top} of the input equation; unless the two terms are syntactically equal, they are placed into two separate e-classes. Apart from the proof state, the algorithm also maintains a global set of rewrites $\mathcal{R}$, which is initialized to $\mathcal{R}_0$ (line 4) and is updated as the algorithm discovers new lemmas.

In each iteration, Prove obtains a goal G from $\mathcal{G}$ and tries to solve it. $\mathcal{G}$ might determine (line 7) that no goals can make progress towards proving the top-level equation, in which case the algorithm returns Invalid. Otherwise, the algorithm first *saturates* the goal G (line 8), and then checks if the saturated goal is solved by Refl (line 9), *i.e.* if $G.L$ and $G.R$ are in the same e-class.

If the goal is solved, Prove marks it as such in the proof state (line 10). It then inspects the goal's parent lemma, which is the lemma this goal's resolution (partially) entails. If all of the lemma's case-split sub-goals are now solved, the lemma is considered proven. If this lemma was in fact the top-level equation, the algorithm returns Valid (line 13), and otherwise the lemma is added to $\mathcal{R}$ as an unconditional bidirectional rewrite (line 14).

If the goal G is not solved, the algorithm marks it in $\mathcal{G}$ as inactive (line 16), and then uses it to generate fresh active goals in $\mathcal{G}$ in two ways: it (1) conjectures connector lemmas from the e-graph of G (line 17), and (2) case-splits G into sub-goals according to the rule Case from Fig. 5 (line 18);

Input: E-graph E, rewrites $\mathcal{R}$
Output: A set of connector lemma goals $\mathcal{G}'$

```
1: function ConjConnectorLemmas(E, R)
2:   G' ← []                                                  ▷ Initialize result
3:   for ⟨c1, c2⟩ ∈ classes(E) × classes(E) do               ▷ For all pairs of e-classes in E
4:     if c1 ≠ c2 ∧ cvec(c1) = cvec(c2) then                 ▷ whose c-vecs match
5:       for ⟨M1, M2⟩ ∈ terms(c1) × terms(c2) do             ▷ For all pairs of terms in the e-classes
6:         for L ≐ R ∈ Generalize(M1 ≐ M2) do                ▷ Create an equation and generalize it into a lemma
7:           if ¬Subsumed(L ≐ R, G', R) then                 ▷ Unless the lemma is subsumed by those already added
8:             G' ← G' + ⟨L ≐ R, IH(L, R), eg(L, R)⟩         ▷ Make a goal out of it
9:   return G'
```

Fig. 9. The sub-routine for conjecturing connector lemmas.

each sub-goal inherits the e-graph of G, only replacing the match scrutinee with a constructor application, hence it preserves all the previously proven equalities.

4.2 Conjecturing Connector Lemmas

As we explain in Sec. 2.3, the key insight behind our approach to lemma discovery is to only consider *connector lemmas*—those that can merge two distinct e-classes in a (saturated) goal in our proof state, and hence make some progress in the proof.

This step of the algorithm is described in the procedure ConjConnectorLemmas, shown in Fig. 9. The procedure takes as input an e-graph E and a set of rewrites $\mathcal{R}$ (s.t. E is saturated *wrt.* to $\mathcal{R}$), and outputs a set of goals $\mathcal{G}'$ corresponding to the connector lemmas of E.

The procedure begins (lines 3-4) by enumerating all pairs c_1, c_2 of e-classes in E that are annotated with the same *characteristic vector* (c-vec); we discuss c-vecs in more detail below, but for now, it suffices to say two e-classes with the same c-vec are likely to be equal, while two e-classes with different c-vecs are definitely not equal. The algorithm then enumerates all pairs of terms M_1, M_2 in c_1 and c_2 (line 5); because the set of terms in an e-class may be infinite due to cycles in an e-graph, the algorithm only considers terms up to a certain depth. The procedure then *generalizes* each equation $M_1 \doteq M_2$ into a set of lemmas $L \doteq R$ (with the original equation included), and adds them to $\mathcal{G}'$ unless they are subsumed by $\mathcal{R}$.

Characteristic Vectors. CCLemma associates every e-class c in a goal's e-graph with a *characteristic vector* $\text{cvec}(c)$, which is a vector of values. Specifically, $\text{cvec}(c)$ stores the result of evaluating the terms in c given many different assignments of values to the variables of the goal. We refer the reader to Fig. 4 in Sec. 2.3 for a concrete example of e-graph annotated with c-vecs. Importantly, if two e-classes have different c-vecs, this serves as a counterexample to any lemma that equates terms from those e-classes.

Although most theory exploration tools use counterexamples to prune obviously false lemmas, our implementation of counterexample generation—and the term c-vec—is borrowed from Ruler [Nandi et al. 2021]. The key idea is to implement c-vec evaluation as an *e-class analysis* [Willsey et al. 2021], which incrementally propagates the c-vecs through the e-graph bottom-up, taking advantage of sharing to avoid re-computing the values of the same sub-terms multiple times. Whenever a new goal e-graph $E = \text{eg}(L, R)$ is created, where $\text{vars}(L, R) = \overline{x_i}$, CCLemma generates a random vector of N values for each variable x_i (in our evaluation, $N = 30$), and initializes each $\text{cvec}(E[x_i])$ with the corresponding vector. Thereafter, whenever new e-classes are created the analysis automatically propagates the c-vecs through the e-graph.

Generalization. When ConjConnectorLemmas enumerates a pair of terms M_1, M_2 from c_1, c_2, it creates not only the lemma $M_1 \doteq M_2$, which directly equates the two terms, but also a set of *generalizations* of this lemma. This is important because $M_1 \doteq M_2$ might not be easily provable. For example, recall the goal G from the proof of (`half_double`) in Fig. 4. Here, from the e-classes c_1 and c_2, we extract a connector lemma:

$$\forall x : \texttt{Nat}.\ \texttt{add}\ x\ (\texttt{S}\ (\texttt{S}\ x)) \doteq \texttt{S}\ (\texttt{add}\ x\ (\texttt{S}\ x))$$

While this lemma would be perfectly useful to us in proving the top-level goal, the issue is that the lemma *itself* cannot be directly proven by induction, unlike its generalization (`add_right`) in which the sub-term $\texttt{S}\ x$ is replaced with a fresh variable:

$$\forall x, y : \texttt{Nat}.\ \texttt{add}\ x\ (\texttt{S}\ y) \doteq \texttt{S}\ (\texttt{add}\ x\ y)$$

To construct the generalizations, function Generalize exhaustively considers all ways to replace common sub-terms of M_1 and M_2 with fresh variables (including the trivial generalization, which leaves M_1 and M_2 unchanged). This is a standard, if a little naïve[11], form of the generalization used by goal-directed lemma discovery tools [Johansson 2019; Sonnex et al. 2012; Yang et al. 2019].

Subsumption. Finally, as ConjConnectorLemmas conjectures many lemmas at once, some of them might be *subsumed* by others. For example, assume that in the process of proving (`half_double`), we have already proven—and added to $\mathcal{R}$—the following lemma:

$$\forall x : \texttt{Nat}.\ \texttt{add}\ x\ \texttt{Z} \doteq x \qquad (\texttt{right\_zero})$$

Now consider a call to ConjConnectorLemmas that generates the following two candidate lemmas:

$$\forall x, y : \texttt{Nat}.\ \texttt{add}\ x\ (\texttt{S}\ y) \doteq \texttt{S}\ (\texttt{add}\ x\ y) \tag{4}$$

$$\forall x, y : \texttt{Nat}.\ \texttt{add}\ (\texttt{add}\ x\ \texttt{Z})\ (\texttt{S}\ y) \doteq \texttt{S}\ (\texttt{add}\ (\texttt{add}\ x\ \texttt{Z})\ y) \tag{5}$$

The first one is our old friend (`add_right`), while the second one is just a "fusion" of (`add_right`) with (`right_zero`). We say that lemma $L \doteq R$ *subsumes* lemma $L' \doteq R'$ under the rewrites $\mathcal{R}$ if the latter is derivable from the former using $\mathcal{R}$, *i.e.* if $L \to^*_{\mathcal{R}} L'$ and $R \to^*_{\mathcal{R}} R'$. Intuitively, a subsumed lemma is redundant: any time we would use lemma (5) in a proof, we could instead use (`add_right`) and (`right_zero`). Hence, to cut down on the number of lemmas conjectured, ConjConnectorLemmas only adds a lemma to $\mathcal{G}'$ if it is not subsumed by any lemma already in $\mathcal{G}'$.

Although filtering based on subsumption is not new—several theory exploration tools [Claessen et al. 2013; Reynolds and Kuncak 2015] use it to avoid redundant lemmas—the EqSat-based architecture of CCLemma makes this check especially efficient and easy to implement. We add all sides $\overline{L}, \overline{R}$ of the conjectured lemmas $\overline{L \doteq R}$ to an auxiliary e-graph and saturate it with $\mathcal{R}$. This reduces our set of lemmas to a set of *equivalence classes* of lemmas, each class represented by an e-class c_L from the auxiliary e-graph for its LHS and a corresponding e-class c_R for its RHS. By extracting the smallest term from c_L and c_R, we pick a representative lemma for the entire class—and if $c_L = c_R$ we can ignore that lemma entirely as it is implied by $\mathcal{R}$.

4.3 Goal Graph

We now explain how CCLemma tracks the progress of each goal and decides which goal to work on next. Recall that procedure Prove (Fig. 8) operates on a proof state $\mathcal{G}$. We implement $\mathcal{G}$ as a data structure we call a *goal graph*. Fig. 10 shows an example goal graph constructed during the proof of (`half_double`). The goal graph stores all generated goals and distinguishes between *lemma goals* (blue)—which are the root goals of connector lemmas and the top-level proposition—and *sub-goals* (gray)—created by case splitting. Edges in this graph track the origin of each goal. Note

[11] Other provers—such as [Sonnex et al. 2012]— use heuristics to prefer certain generalizations and avoid others.

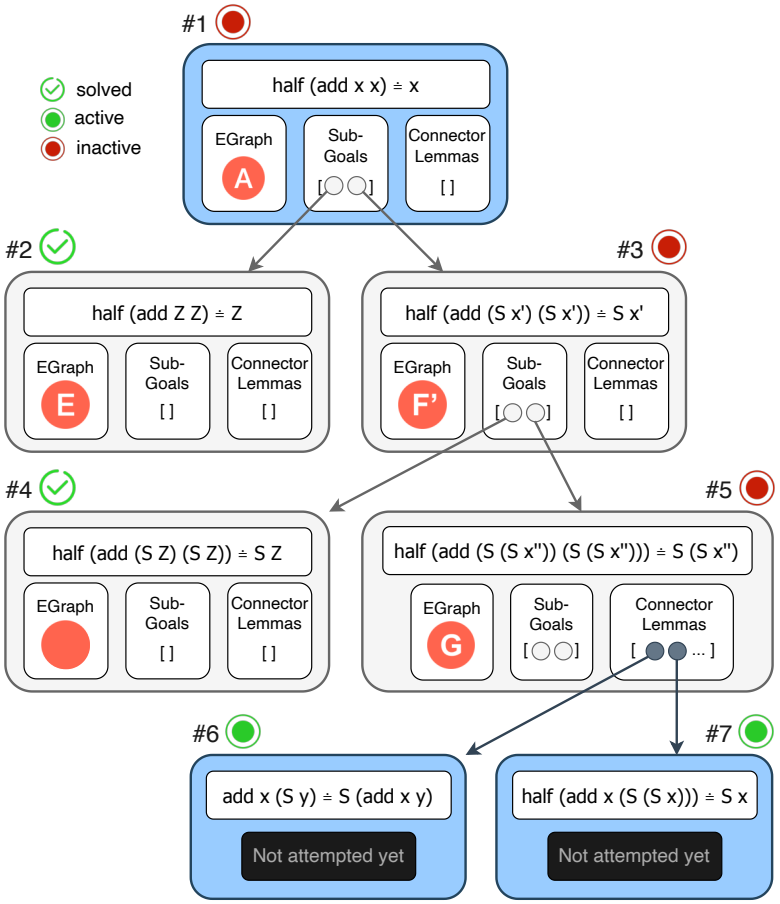

Fig. 10. A depiction of the goal graph for (`half_double`). Each goal that has been attempted has a label corresponding to an e-graph in Fig. 3 or Fig. 4; Goal #4 is in neither figure and thus has a blank label. Some goals are omitted for brevity.

that a lemma goal can have multiple incoming edges (*i.e.* the goal graph is not a tree), because the same connector lemma can be extracted from different goals.

The goal graph maintains a *proof status* for each goal, which has three possibilities.

- *Solved* means the goal's equation has been proven valid.
- *Inactive* means that this goal cannot currently make progress, and should not be attempted. For example in Fig. 10, goal #5 is inactive because it has already been saturated and case-split, so it should not be considered again until any of its children are solved.
- *Active* captures the goals that are worth trying, including newly generated goals (*e.g.*, #6 and #7) and previous goals whose children have been recently solved.

Every time a goal's proof status changes, the goal graph will propagate the change according to the rules below.

(1) A goal becomes *solved* if all its sub-goals are solved, which corresponds to an application of Case in our proof system.

(2) An inactive goal becomes *active* again if any of its child connector lemmas are solved, since now we can apply the lemma to establish new equalities in the goal's e-graph.
(3) An active goal becomes *inactive* if all of its parents are solved, since this goal cannot help with any other unproven goals.

Example 4.1. In Fig. 10, once goal #6 is solved, goal #5 will be marked active by Rule (2). Then, if goal #5 is solved (say, using goal #6 as a lemma), other children of #5 (e.g., goal #7) will be marked inactive by Rule (3). On the other hand, goals #3 and #1 will be marked solved by Rule (1), at which point the top-level proposition is proven.

Prove (Fig. 8) initializes $\mathcal{G}$ with a single active goal, which corresponds to the top-level proposition (Line 3). In each iteration, Prove interacts with the goal graph as follows.

- NextGoal($\mathcal{G}$) selects the next active goal from the goal graph. When multiple active goals exist, we select the smallest one (in terms of AST size); this size heuristic is popular in theory exploration [Claessen et al. 2013], because it prioritizes more general lemmas. For example, in Fig. 10, between goals #6 and #7, we will select the smaller (and more general) goal #6, which happens to be the one we can actually prove directly.
- MarkSolved($\mathcal{G}, G$) and MarkInactive($\mathcal{G}, G$) mark the goal G as solved or inactive, respectively, and propagate this status change through the graph, according to the rules above.

4.4 Implementation and Extensions

We implemented CCLemma in Rust, using the egg e-graph library [Willsey et al. 2021]. In this section we discuss some interesting implementation details and extensions to the core algorithm.

Destructive Rewrites. The key property of EqSat is that its rewrites are *non-destructive*: upon applying a rewrite to a term M, the rewritten term M' and the original term M are *both* in the e-graph. This property allows EqSat to avoid backtracking search, since no information is ever lost. Unfortunately, it also meant that goal e-graphs can grow quite large, blowing up the number of candidate connector lemmas and slowing down the proof search. To mitigate this issue, we leverage an observation from the CycleQ prover [Jones et al. 2021] that, unlike lemma rewrites, *reduction rewrites can be always applied eagerly*, without backtracking and without sacrificing completeness. Based on this observation, we modify EqSat to support both destructive and non-destructive rewrites. CCLemma applies all reductions destructively, removing the e-nodes matched by these rewrites from the goal's e-graph, and hence keeping the e-graph size under control.

Blocking Variables. As discussed in Sec. 2.2, EqSat helps us decide *when* to case split a goal, but not *which* variable to case split on. In principle, redundant case splits need not hurt completeness: they only hurt efficiency by making the goal graph larger. There is a subtlety, however, related to our termination check (Sec. 3.3): because each IH invocation must decrease the tuple of already-case-split variables, the order of case splitting in fact does matter for completeness in our system. For example, to find the proof in Example 3.2 we must case split `x` first; otherwise, we would require the decrease of one of `y` or `z` across all branches, which is not possible because there is always a branch in which one of the two increases.

To avoid these kinds of redundant case splits, we use a common trick [Jones et al. 2021; Singher and Itzhaky 2021; Sonnex et al. 2012] of only case-splitting on variables that are currently *blocking* a rewrite. For example, if a term `add` x y appears in the goal's e-graph, we will know to case split on x and not on y (assuming that only rewrite rules we have for `add` are its reductions, `add Z` $?y \Rightarrow \ldots$ and `add (S` $?x$`)` $?y \Rightarrow \ldots$). To identify blocking variables in an e-graph, we analyze the LHSs of the available rewrite rules and create custom patterns that search for blocked terms in the e-graph (*e.g.* for `add` we would search for terms of the form `add` $?x$ $?y$ $\mid ?x$ is var). While the blocking

variable heuristic does not completely prevent redundant case splits, it handles the common case of *accumulator parameters*, which commonly *increase* in recursive calls but are also never blocking.

Conditional Properties. Many interesting equalities do not hold universally, but rather under certain preconditions, such as

$$\forall x, y : \texttt{Nat}.\ \texttt{isEven}\ x \doteq \texttt{isEven}\ y \implies \texttt{isEven}\ (\texttt{add}\ x\ y) \doteq \texttt{True}$$

We extend CCLemma to handle such properties by making two relatively straightforward changes:

(1) Since we *get to assume* that the precondition holds, we add its sides to the e-graph of the top-level goal and merge their e-classes. In the example above, the e-graph of the top-level goal (and hence, all its sub-goals) will store $\texttt{isEven}\ x$ and $\texttt{isEven}\ y$ in the same e-class.
(2) Since we *need to check* that the precondition holds before we can apply the IH, we add a new condition to the IH rewrite (in addition to the termination check). In the example above, before applying the IH to $\texttt{isEven}\ (\texttt{add}\ x'\ y')$, where $x = \texttt{S}\ x'$ and $y = \texttt{S}\ y'$, we will check that the terms $\texttt{isEven}\ x'$ and $\texttt{isEven}\ y'$ are in the same e-class.

Guard Splitting. Consider, the following equation:

$$\forall x : \texttt{Nat}.\ \textbf{if}\ \texttt{isEven}\ x\ \textbf{then}\ \texttt{Z}\ \textbf{else}\ \texttt{Z} \doteq \texttt{Z}$$

Proving this property should be simple (it in fact does not even require induction!), but our base algorithm from Fig. 8 cannot prove it on its own. The problem here is that the evaluation of the conditional is blocked, and no amount of case splitting on x can unblock it.

Since **`if`**-**`then`**-**`else`** conditionals are very common, we introduce a special heuristic to handle them, which allows *case-splitting on the guard of a conditional*, even if the guard is not a variable. To this end, whenever a goal contains a blocked conditional, the algorithm adds a fresh boolean variable to the e-class of its guard: in the example above, it would add a variable g_0 to the e-class containing $\texttt{isEven}\ x$. When a later proof step case-splits on g_0, it will produce two subgoals corresponding to the cases when $\texttt{isEven}\ x = \texttt{True}$ and $\texttt{isEven}\ x = \texttt{False}$, both of which are easily discharged.

Both conditional properties and guard splitting can generate vacuous goals, *i.e.* goals with contradictory assumptions. CCLemma discharges such goals via an e-class analysis that detect that an e-class contains two different constructors of the same datatype.

5 Evaluation

We evaluate CCLemma by comparing its performance on a diverse set of benchmarks to three state-of-the-art theory exploration tools along the following dimensions:

RQ1) ***Effectiveness***. How *many* properties can CCLemma prove (Sec. 5.2)?
RQ2) ***Efficiency***. How *quickly* can CCLemma prove properties (Sec. 5.3)?

We have released an accompanying artifact containing CCLemma, the tools we compared against, as well as instructions and scripts for running our evaluation [Kurashige et al. 2024].

Baseline Tools. We compare CCLemma against three state-of-the-art theory exploration tools: HipSpec [Claessen et al. 2013], CVC4 [Reynolds and Kuncak 2015], and TheSy [Singher and Itzhaky 2021]. We chose to include HipSpec and CVC4 because they are widely used, and TheSy because it is recent and also uses e-graphs. A more detailed comparison of these tools is provided in Sec. 6. Note that we are using the version of CVC4 that comes with the publication [Reynolds and Kuncak 2015], and not the more recent CVC version 5, because in our preliminary experiments we found that CVC5 was less competitive on lemma discovery benchmarks.

Table 1. Benchmark suite statistics and comparison of the number of properties proved by each tool. **Props** is the total number of properties in the benchmark suite, **Avg vocab** is the average vocabulary size of the properties in the suite. **Proved** is the number of properties proved by each tool, **Them only** and **Us only** is the number of properties proved by the baseline tool and not CCLemma and vice versa.

Benchmark	Props	Avg vocab	Tool	Proved	Them only	Us only
IsaPlanner (easy)	87	7.3	CCLemma	75	-	-
			HipSpec	**77**	4	2
			CVC4	67	3	11
			TheSy	44	1	32
CLAM (medium)	50	6.2	CCLemma	**38**	-	-
			HipSpec	**38**	6	6
			CVC4	22	3	19
			TheSy	34	7	11
Optimization (hard)	96	30.5	CCLemma	**28**	-	-
			HipSpec	11	0	17
			CVC4	18	5	15
			TheSy	6	0	22

Experiment Setup. We ran all the tools on a server with Intel Xeon X5660 CPU, RAM limit of 16GB, and a time limit of 60 seconds per property. We ran all baseline tools using the settings from their respective evaluations; for CVC4, we ran the version that does not use a native SMT encoding of integers, which puts it on a more equal footing with the other tools.

5.1 Benchmarks

We perform our evaluation on three benchmark suites of varying levels of difficulty, summarized on the left size of Tab. 1. The first two of them, IsaPlanner and CLAM, are standard benchmarks for automated equational reasoning; the third one, Optimization, is a new, more challenging, benchmark we constructed from the domain of program optimization.

IsaPlanner. The IsaPlanner benchmark [Dixon and Fleuriot 2003] consists of 87 equational properties over `Nats` and `Lists`. This benchmark is relatively simple because most of its properties are in fact provable without auxiliary lemmas. We consider IsaPlanner as a sanity check for whether CCLemma can handle simple inductive proofs.

CLAM. The CLAM benchmark [Ireland and Bundy 1996] is a more challenging dataset, specifically targeting lemma discovery. To focus on this goal, we consider a subset of the original CLAM dataset, only including the 50 properties (out of 86) that require auxiliary lemmas.[12]

Optimization. To go beyond simple properties of inductive data types (like (`half_double`)), we curate a new, more challenging benchmark suite from the domain of program optimization. To construct this dataset, we collect pairs of reference programs and their optimized versions from the literature on *fusion* [Bird 1989; Gill et al. 1993; Hu et al. 1997; Wadler 1988] and *program synthesis* [Bird and de Moor 1997; Farzan et al. 2022; Ji et al. 2024]. For each pair, we construct an equation between the two programs. Since CCLemma only supports algebraic datatypes, while

[12]The properties we use are referred to as T1-T35, T48-T50, and G1-G12 in [Ireland and Bundy 1996]. Several past evaluations (such as in [Claessen et al. 2013]) were on T1-T50; we draw attention to this difference because the total number of properties is the same, but 12 properties differ between these two evaluations.

many of the original problems are over integers, we encoded all integer expressions into `Nats` and exclude all tasks that cannot be meaningfully encoded (such as those involving subtraction and large integer constants). This results in a benchmark suite of 96 properties, one of which is our motivating example of `mtp` from the introduction. These tasks are challenging for theory exploration because they have a much larger *vocabulary* (*i.e.* the set of functions and constructors used) than the IsaPlanner and CLAM benchmarks: as you can see in Tab. 1, the average vocabulary size of the Optimization properties is 30.5, compared to 7.3 and 6.2 for IsaPlanner and CLAM, respectively.

5.2 RQ1: Effectiveness

The right side of Tab. 1 compares the total number of properties proved for CCLemma vs the baselines across all the benchmarks.

On the simpler datasets, IsaPlanner and CLAM, CCLemma and HipSpec do similarly well. TheSy struggles with IsaPlanner, which is understandable: TheSy's strength is in theory exploration—which often does not help on these benchmarks—while its inductive prover is relatively naïve and, for example, does not support guard splitting. Surprisingly, CVC4 does worse than the rest of the tools on CLAM, for which we do not have a good explanation.

HipSpec and TheSy's undirected approach to theory exploration comes to a head in the Optimization benchmark, which have a much larger vocabulary. Both tools prove significantly fewer properties than CCLemma: 11 and 6, respectively, compared to CCLemma's 28. CVC4 clearly outperforms both TheSy and HipSpec with 18 properties; we attribute it to the fact that CVC4's exploration is more goal-directed thanks to their active conjecture pruning (see Sec. 6). Nevertheless, CCLemma proves 10 more properties than CVC4, more than a 50% increase. Finally, we note that none of the four tools are able to prove more than 30% of the Optimization properties, which is a testament to the difficulty of this benchmark; consequently, we hope that our new dataset will be useful for evaluating future theory exploration tools.

5.3 RQ2: Efficiency

Fig. 11 shows how many properties each tool can solve (y-axis) given a fixed time out for each property (x-axis). Because IsaPlanner and CLAM have many properties that can be proven in under a second, we zoom in these two plots to provide better resolution on the sub-second time range. Sub-second performance is important if a prover is to be used inside a compiler optimization pass, or for auto-completion within an interactive theorem prover.

As we can see from the plot, CCLemma is *consistently the fastest* across the three domains, proving most of CLAM benchmarks in under a second, and most of the IsaPlanner benchmarks in milliseconds. HipSpec is consistently the slowest to get started (although it eventually catches up to CCLemma on IsaPlanner and CLAM); this is not surprising, because HipSpec first generates a large number of lemmas and filters them using randomized testing, before sending the remaining lemmas to an external prover.

5.4 Discussion

At the end of this section, we discuss specific properties where CCLemma's goal-directed approach shines and those where it struggles.

Benefits of Goal-Directedness. CCLemma's goal-directedness wins the day when the space of candidate lemmas is too large. For example, functions like `mult`, `fac` (multiplication and factorial, respectively in Peano arithmetic) are themselves defined in terms of other functions which increases the search space of possible lemmas significantly. In their evaluation, HipSpec has to apply heuristics limiting size to prune the lemma search space. For some CLAM properties (T33-T35, G10-G12)

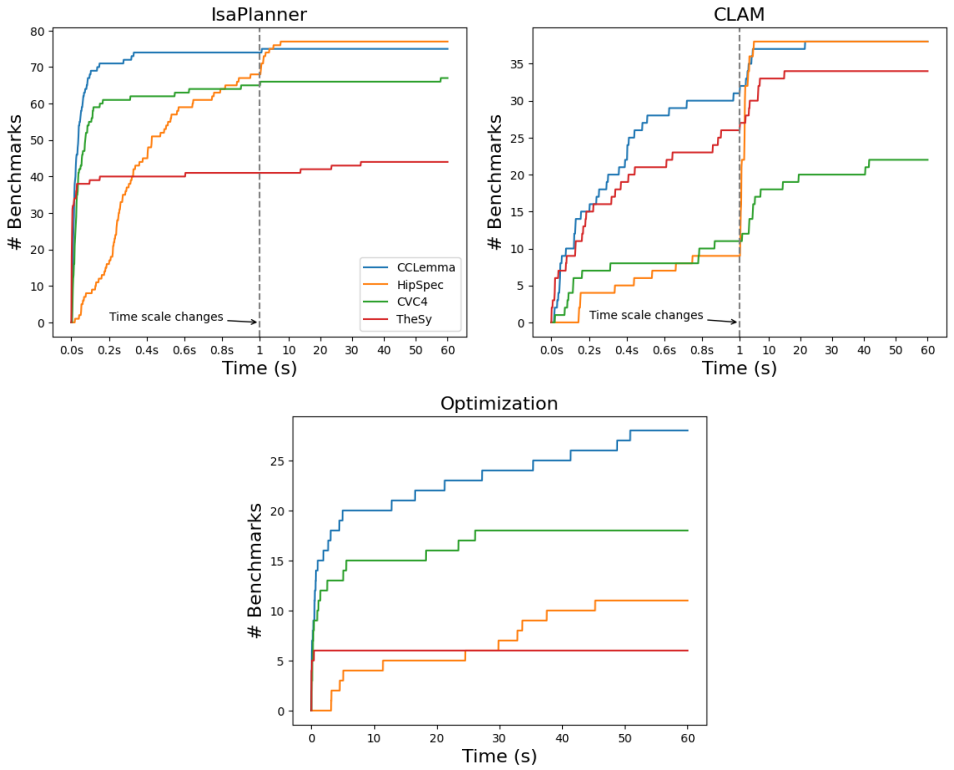

Fig. 11. Properties proved by each tool over time, one plot per benchmark suite. Because many properties are provable in a short period of time for the IsaPlanner and CLAM benchmarks, we adjust the time scale to be from 0-1s on the first half of their plots and 1-60s on the second.

HipSpec limits the size of considered lemmas as otherwise the space of candidate lemmas is too large to handle. For example, consider the property G11 from CLAM

$$\forall x, y : \texttt{Nat.mult}\ x\ y \doteq \texttt{qmult}\ x\ y\ \texttt{Z}$$

which relates the natural definition of multiplication to an accumulator-based multiplier. Given additional time, HipSpec can actually prove this property, but takes well over 60 seconds and proves 28 lemmas in the process. TheSy did not prove it within the same time; we timed it out after about 120 seconds. CCLemma, proves this property in less than a second using only 2 lemmas that it proves; CVC4 too—owing to its hybrid design—is able to prove it, albeit after around 15 seconds.

Benefits of Theory Exploration. While CCLemma's goal-directed nature gives it an overall efficiency advantage over theory exploration (HipSpec and TheSy), it is incomplete, and hence sometimes fails to prove properties where the lemmas cannot be conjectured or generalized from sub-terms. For example, consider the property T31 from CLAM (whose definitions are shown in Fig. 12), which TheSy and HipSpec both prove in a matter of seconds, but CCLemma cannot prove:

$$\forall xs : \texttt{List a.}\ \texttt{qrev}\ (\texttt{qrev}\ xs\ \texttt{Nil})\ \texttt{Nil} \doteq xs$$

This property is essentially the same as T10 ($\texttt{rev}\ (\texttt{rev}\ xs) \doteq xs$)—which CCLemma can prove—but uses `qrev` (an accumulating version of `rev`).

```
data List a = Nil | Cons a (List a)

qrev :: List a → List a
qrev xs acc = case xs of
  Nil         → acc
  Cons x' xs' → qrev xs' (Cons x' acc)
```

Fig. 12. Inductive datatype and function definitions for CLAM property T31.

HipSpec and TheSy both quickly enumerate and prove the lemma

$$\forall xs, ys\ \texttt{List}\ a.\ \texttt{qrev}\ (\texttt{qrev}\ xs\ ys)\ \texttt{Nil} \doteq \texttt{qrev}\ ys\ xs$$

as this lemma is small. This lemma trivially implies the original goal when applied to its LHS.

While CCLemma can prove the above lemma, it cannot find it. This is because CCLemma's primary means of conjecturing connector lemmas is to use evaluation to yield new terms (and thus new candidates for connector lemmas) after case splitting a variable. Case splitting $xs = \texttt{Cons}\ x'\ xs'$ yields the evaluation

$$\forall x' : a.\ xs' : \texttt{List}\ a.\ \texttt{qrev}\ (\texttt{qrev}\ xs'\ (\texttt{Cons}\ x'\texttt{Nil}))\ \texttt{Nil} \doteq \texttt{Cons}\ x'\ xs'$$

But CCLemma cannot evaluate the inner `qrev` further – and it cannot evaluate the outer `qrev` at all. Further case splitting xs' would accumulate more values onto the second argument of `qrev`, but that does not allow for CCLemma to conjecture any interesting or useful connector lemmas. Even clever applications of the IH (e.g. to xs') cannot get us out of this bind, and without a way to evaluate we cannot add interesting terms to conjecture lemmas with.

Interestingly, we observe that if CCLemma were capable of "unevaluating" the RHS from xs to $\texttt{qrev}\ \texttt{Nil}\ xs$, then it would have a chance at finding the necessary lemma by generalizing two of the `Nil`s in the connector lemma

$$\forall xs\texttt{List}\ a.\ \texttt{qrev}\ (\texttt{qrev}\ xs\ \texttt{Nil})\ \texttt{Nil} \doteq \texttt{qrev}\ \texttt{Nil}\ xs$$

"Unevaluation," however, is still incomplete when functions are not one-to-one and CCLemma's generalization mechanisms are too naïve to partially generalize. Both are interesting subjects for future work and offer ways of overcoming (although not fully) its incompleteness.

6 Related Work

Inductive reasoning and lemma discovery have a rich history, so we necessarily narrow our focus, considering only *automated* provers for *equational* reasoning. Specifically, we leave out lemma discovery for most interactive theorem provers [Heras et al. 2013; Sivaraman et al. 2022], as well as non-equational logics, such as first-order logic [Cruanes 2017; Hajdú et al. 2020; Hajdu et al. 2021; Murali et al. 2022; Reger and Voronkov 2019] and separation logic [Enea et al. 2015; Ta et al. 2017]. These tools are not directly comparable to equational inductive provers like CCLemma: while equational provers dedicate the bulk of their proof search to discovering nontrivial uses of induction, first-order and separation logic provers focus on nontrivial reasoning in the corresponding logics, while keeping the uses of induction relatively simple.

6.1 Lemma Discovery for Equational Reasoning

According to the extensive survey by Johansson [2019], lemma discovery for equational reasoning can be broadly divided into two categories: goal-directed and theory exploration.

Goal-Directed Lemma Discovery. Zeno [Sonnex et al. 2012] discovers lemmas by common-subterm generalization. Two things make it effective as a prover despite lacking more complicated heuristics or complete methods for lemma discovery: (1) Through a blocking term analysis that accounts for control flow, Zeno determines when it needs to perform induction, case splitting, or generalization. (2) Zeno's method of performing induction allows it to create conditional lemmas. This allows it to prove some properties (such as T14 from CLAM) that none of the provers we benchmarked can because they require conditional lemma synthesis, a traditionally tricky problem. Despite these strengths, Zeno is ultimately hindered by its inability to discover lemmas that aren't generalizations of the top-level goal and it ultimately performs poorly on benchmarks like CLAM compared to other provers (even though it can prove a property no others can).

Like Zeno, `sem_ind` [Nagashima 2021] observes that one of the major branch points for provers is induction, where they must choose both the variable to induct upon and whether to generalize the goal. It attempts to make this step simpler by enumerating induction and generalization steps, then (1) filtering them based upon plausibility and (2) ranking them according to several heuristics.

IsaPlanner [Dixon and Fleuriot 2004], and OYSTER-CLAM [Ireland and Bundy 1996] (the latter being the originator of the CLAM dataset) are both capable of using more complicated heuristics for lemma discovery. These provers utilize a technique known as *rippling*, whose chief insight is that most inductive proofs need to apply the inductive hypothesis to succeed. When rippling fails to bring about an IH application, these provers will try to compute lemmas that will allow for the IH to be applied. While effective in some cases, rippling ultimately as a heuristic is not broadly applicable enough to be generally useful [Johansson 2019].

Imandra [Passmore et al. 2020], which takes significant inspiration from Boyer-Moore provers such as ACL2, is an interactive prover which uses the *waterfall* approach discussed in [Johansson 2019] to automate proofs. It uses *bounded verification* (similar to our *characteristic vector* checks) to first filter out invalid proof candidates, and then applies a variety of tactics such as generalization to produce additional goals which are recursively proven. Even though it is capable of complicated proof reasoning (such as the generation of conditional lemmas), Imdara is not fully automatic and requires user guidance for most nontrivial proofs.

Theory Exploration. HipSpec [Claessen et al. 2013] generates lemmas via bottom-up term enumeration and verifies them using an SMT solver. Key to its efficacy are two powerful pruning techniques: (1) *counterexample finding* to filter out obviously invalid lemmas, and (2) *congruence closure* to filter out redundant lemmas. We borrow both of these pruning techniques from HipSpec, but we implement them using equality saturation and e-class analysis.

Reynolds and Kuncak [2015] implement an inductive prover with theory exploration *inside* the CVC4 SMT solver. Their theory exploration algorithm is similar to HipSpec's, except that they add a third pruning technique based on *active conjectures*. At a high level, CVC4 won't propose a lemma unless one of its sides matches a term in the current context that contains a variable from one of the currently unsolved goals; similarly to our connector lemma idea, this restricts candidate lemmas to those applicable to the proof, but only one side of the lemma is restricted.[13]

Similarly to CVC4 and CCLemma, AdtInd [Yang et al. 2019] strives to generate lemmas that can apply to the current proof. Unlike CCLemma, AdtInd explicitly creates a grammar out of proof terms and uses it to enumerate lemmas. However, without strong guidance from the user in the form of lemma templates that it uses to reduce its search space, AdtInd faces the same general problem of theory exploration: scaling. In practice, it has to use heuristics to prune its on-the-fly lemma theorization to keep the number of considered lemmas down—even with user guidance.

[13]Note also, that because HipSpec and CVC4 rely on SMT solving, they might not be able to produce externally checkable proofs, while CCLemma can produce such proofs using the *explanation* mechanism in egg [Willsey et al. 2021].

AdtInd, as well as the TBC [Nagashima et al. 2023] prover, leverage *lemma templates* to significantly cut down on lemma enumeration space. In the case of TBC, these are a predefined set of general mathematical properties which the authors demonstrate often are useful if proven. In the case of AdtInd, these templates are user-provided, although the authors include several in their evaluation which they observed were helpful.

Theory Exploration with E-Graphs. Two closest techniques to ours, although in different ways, are TheSy [Singher and Itzhaky 2021] and Ruler [Nandi et al. 2021].

TheSy is a theory exploration tool, which is similar to HipSpec but uses semi-concrete values to prune the lemmas instead of concrete values. Like CCLemma, TheSy also uses e-graphs, but for the primary purpose of conjecturing lemmas more efficiently than full, naïve enumeration. From TheSy we got the idea to do inductive proofs with EqSat by alternating saturation and case splitting; TheSy's proof search, however, has a simplistic case-splitting strategy: it only case-splits once, on a single variable. CCLemma expands on this idea by adding a more sophisticated case splitting strategy based on blocking variables, and a novel termination check that works well with this strategy and with EqSat-based proof search.

Ruler is an e-graph based theory exploration tool, but for theories that do not require induction. From Ruler, CCLemma borrows the idea of using c-vecs to implement the counterexample finding.

6.2 Termination Checking

We are not aware of any literature that uses exactly the same technique as CCLemma for termination checking. Focusing on termination checking for *structural induction* on datatypes, rather than general induction using, *e.g.*, integers, we find two categories: a global lexicographic termination check per proposition is used in, *e.g.*, dependently-typed languages [Idris Team 2024; Moura and Ullrich 2021; Norell 2007], and the more general automata-theoretic criteria used in cyclic proofs [Jones et al. 2021; Rowe and Brotherston 2017] and *size-change termination* [Lee et al. 2001].

Our technique may be seen as a simple generalization of a the commonly-used lexicographic termination metric, allowing for more flexibility in how induction is done in different branches of a case-split. The automata-theoretic conditions used in cyclic proofs are more general, and thus also more expensive to compute. They are also *global*—in order to check soundness of one use of the inductive hypothesis, we must first complete the whole proof tree. By contrast, our termination metric is *local*, allowing us to immediately reject unsound uses of the induction hypothesis.

We find our termination check to be a nice intermediate between these two approaches, allowing more flexibility than lexicographic ordering while being less expensive than the automata-theoretic check. Although the latter is more general, we conjecture that our criterion is just as good in the following sense: for any inductive proof which passes the automata-theoretic check, we conjecture that it can be "unrolled" into a longer inductive proof which passes our simpler soundness check.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Science Foundation under Grants No. 1943623 and 1911149.

Data Availability

The accompanying software artifact for this paper is available online [Kurashige et al. 2024]. This artifact includes the source code for CCLemma as well as the sources or prebuilt binaries for all of the tools in our comparison. It also includes scripts and instructions for running CCLemma and reproducing all of the experiments in our evaluation. The host for our artifact additionally makes available any future versions and as a link to the accompanying code repository.

References

Richard S. Bird. 1989. Algebraic Identities for Program Calculation. *Comput. J.* 32, 2 (1989), 122–126. https://doi.org/10.1093/comjnl/32.2.122

Richard S. Bird and Oege de Moor. 1997. *Algebra of programming*. Prentice Hall.

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2013. Automating Inductive Proofs Using Theory Exploration. In *Proceedings of the 24th International Conference on Automated Deduction* (Lake Placid, NY) *(CADE'13)*. 392–406. https://doi.org/10.1007/978-3-642-38574-2_27

Simon Cruanes. 2017. Superposition with Structural Induction. In *Frontiers of Combining Systems*, Clare Dixon and Marcelo Finger (Eds.). Springer International Publishing, Cham, 172–188.

Leonardo Mendonca de Moura and Nikolaj Bjorner. 2007. Efficient E-Matching for SMT Solvers.. In *CADE (Lecture Notes in Computer Science, Vol. 4603)*. Springer, 183–198.

Lucas Dixon and Jacques Fleuriot. 2003. IsaPlanner: A Prototype Proof Planner in Isabelle. In *Automated Deduction – CADE-19*, Franz Baader (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 279–283.

Lucas Dixon and Jacques Fleuriot. 2004. Higher Order Rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–98.

Constantin Enea, Mihaela Sighireanu, and Zhilin Wu. 2015. On Automated Lemma Generation for Separation Logic with Inductive Definitions. arXiv:1507.05581 [cs.LO]

Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 244–259. https://doi.org/10.1145/3519939.3523726

Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 223–232. https://doi.org/10.1145/165180.165214

Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. 2020. Induction with Generalization in Superposition Reasoning. In *Intelligent Computer Mathematics*, Christoph Benzmüller and Bruce Miller (Eds.). Springer International Publishing, Cham, 123–137.

Márton Hajdu, Petra Hozzová, Laura Kovács, and Andrei Voronkov. 2021. Induction with Recursive Definitions in Superposition. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. 1–10. https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_34

Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. Proof-pattern recognition and lemma discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings 19*. Springer, 389–406.

Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. 1997. Tupling Calculation Eliminates Multiple Data Traversals. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*, Simon L. Peyton Jones, Mads Tofte, and A. Michael Berman (Eds.). ACM, 164–175. https://doi.org/10.1145/258948.258964

Idris Team. 2024. *Idris 2*. https://www.idris-lang.org/index.html

Andrew Ireland and Alan Bundy. 1996. *Productive Use of Failure in Inductive Proof*. Springer Netherlands, Dordrecht, 79–111. https://doi.org/10.1007/978-94-009-1675-3_3

Ruyi Ji, Yuwei Zhao, Yingfei Xiong, Di Wang, Lu Zhang, and Zhenjiang Hu. 2024. Decomposition-Based Synthesis for Applying Divide-and-Conquer-Like Algorithmic Paradigms. arXiv:2202.12193 [cs.PL]

Moa Johansson. 2019. Lemma Discovery for Induction. In *Intelligent Computer Mathematics*, Cezary Kaliszyk, Edwin Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen (Eds.). Springer International Publishing, Cham, 125–139.

Eddie Jones, C-.H. Luke Ong, and Steven Ramsay. 2021. CycleQ: An Efficient Basis for Cyclic Equational Reasoning. *CoRR* abs/2111.12553 (2021). arXiv:2111.12553 http://arxiv.org/abs/2111.12553

Cole Kurashige, Ruyi Ji, Aditya Giridharan, Mark Barbone, Daniel Noor, Shachar Itzhaky, Ranjit Jhala, and Nadia Polikarpova. 2024. *CCLemma: E-Graph Guided Lemma Discovery for Inductive Equational Proofs*. https://doi.org/10.5281/zenodo.11508050

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/360204.360210

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.

Adithya Murali, Lucas Peña, Eion Blanchard, Christof Löding, and P. Madhusudan. 2022. Model-guided synthesis of inductive lemmas for FOL with least fixpoints. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 191 (oct 2022), 30 pages. https://doi.org/10.1145/3563354

Yutaka Nagashima. 2021. Faster Smarter Proof by Induction in Isabelle/HOL. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Zhi-Hua Zhou (Ed.). ijcai.org, 1981–1988. https://doi.org/10.24963/IJCAI.2021/273

Yutaka Nagashima, Zijin Xu, Ningli Wang, Daniel Sebastian Goc, and James Bang. 2023. Template-Based Conjecturing for Automated Induction in Isabelle/HOL. In *Fundamentals of Software Engineering - 10th International Conference, FSEN 2023, Tehran, Iran, May 4-5, 2023, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 14155)*, Hossein Hojjat and Erika 'Abrah'am (Eds.). Springer, 112–125. https://doi.org/10.1007/978-3-031-42441-0_9

Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (1980), 356–364.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Grant O. Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. 2020. The Imandra Automated Reasoning System (System Description). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12167)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 464–471. https://doi.org/10.1007/978-3-030-51054-1_30

Giles Reger and Andrei Voronkov. 2019. Induction in Saturation-Based Proof Search. In *Automated Deduction – CADE 27*, Pascal Fontaine (Ed.). Springer International Publishing, Cham, 477–494.

Andrew Reynolds and Viktor Kuncak. 2015. Induction for SMT Solvers. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 8931)*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer, 80–98. https://doi.org/10.1007/978-3-662-46081-8_5

Reuben N. S. Rowe and James Brotherston. 2017. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*. ACM, 53–65. https://doi.org/10.1145/3018610.3018623

Eytan Singher and Shachar Itzhaky. 2021. Theory Exploration Powered by Deductive Synthesis. In *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12760)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer, 125–148. https://doi.org/10.1007/978-3-030-81688-9_6

Aishwarya Sivaraman, Alex Sanchez-Stern, Bretton Chen, Sorin Lerner, and Todd Millstein. 2022. Data-driven lemma synthesis for interactive proofs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 143 (oct 2022), 27 pages. https://doi.org/10.1145/3563306

William Sonnex, Sophia Drossopoulou, and Susan Eisenbach. 2012. Zeno: An Automated Prover for Properties of Recursive Data Structures. In *TACAS*.

Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. 2017. Automated lemma synthesis in symbolic-heap separation logic. *Proc. ACM Program. Lang.* 2, POPL, Article 9 (dec 2017), 29 pages. https://doi.org/10.1145/3158097

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 264–276.

Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings (Lecture Notes in Computer Science, Vol. 300)*, Harald Ganzinger (Ed.). Springer, 344–358. https://doi.org/10.1007/3-540-19027-9_23

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

Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. 2019. Lemma Synthesis for Automating Induction over Algebraic Data Types. In *Principles and Practice of Constraint Programming*, Thomas Schiex and Simon de Givry (Eds.). Springer International Publishing, Cham, 600–617.

Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper/2021/file/65ded5353c5ee48d0b7d48c591b8f430-Paper.pdf

Received 2024-02-28; accepted 2024-06-18