


# Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory

Joshua Clune   

Carnegie Mellon University, Pittsburgh, PA, USA

Yicheng Qian 

Peking University, Beijing, China

Alexander Bentkamp   

Heinrich-Heine-Universität Düsseldorf, Germany

Jeremy Avigad   

Carnegie Mellon University, Pittsburgh, PA, USA

---

## Abstract

We present Duper, a proof-producing theorem prover for Lean based on the superposition calculus. Duper can be called directly as a terminal tactic in interactive Lean proofs, but is also designed with proof reconstruction for a future Lean hammer in mind. In this paper, we describe Duper's underlying approach to proof search and proof reconstruction with a particular emphasis on the challenges of working in a dependent type theory. We also compare Duper's performance to Metis' on pre-existing benchmarks to give evidence that Duper is performant enough to be useful for proof reconstruction in a hammer.

**2012 ACM Subject Classification** Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Theory of computation → Higher order logic; Theory of computation → Type theory

**Keywords and phrases** proof search, automatic theorem proving, interactive theorem proving, Lean, dependent type theory

**Digital Object Identifier** 10.4230/LIPIcs.ITP.2024.10

**Supplementary Material** *Software*: [https://github.com/JOSHCLUNE/Duper\\_ITP\\_Paper\\_Artifact](https://github.com/JOSHCLUNE/Duper_ITP_Paper_Artifact) archived at [swh:1:dir:ece08eae9c83476bea9fa47d80c266d06d008bed](https://sw.hiclipper.com/1/dir:ece08eae9c83476bea9fa47d80c266d06d008bed)

**Acknowledgements** We thank Mario Carneiro, Rob Lewis, and Gabriel Ebner for their advice in the early stages of the project. We thank Lydia Kondylidou for her input on our evaluation section and Jasmin Blanchette for his feedback on this paper and input on our evaluation section. We also thank Haniel Barbosa and the anonymous reviewers for their feedback on this paper.

## 1 Introduction

Interactive and automated theorem proving offer complementary strengths. Push-button automation is convenient but limited to small problems or restricted problem domains, and generally it does not provide strong correctness guarantees. Interactive theorem provers make it possible to verify just about any theorem with a high degree of confidence in the correctness of the result, but the work required is often unpleasant or impracticable. Most modern proof assistants therefore rely on multiple forms of supporting automation. Domain-specific tactics polish off goals in linear or linear integer arithmetic [17, 37], carry out calculations in any ring [19], and even perform inferences involving abstract geometric structures [39]. Term rewriters carry out general equational simplification [33], and tools like Isabelle's *Auto* [34], HOL Light's *Meson* [20], and Lean's *Aesop* [26] implement general tableau search.

Isabelle's celebrated *Sledgehammer* [31, 32, 35, 36] provides powerful domain-general automation by exporting problems to external provers and then harvesting enough information to reconstruct and verify the results. A proof-producing ordered resolution prover,



© Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad; licensed under Creative Commons License CC-BY 4.0

15th International Conference on Interactive Theorem Proving (ITP 2024).

Editors: Yves Bertot, Temur Kutsia, and Michael Norrish; Article No. 10; pp. 10:1–10:20

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

*Metis* [21], is frequently used for the latter purpose. Similar hammers for proof assistants based on first-order logic or simple type theory include HOL(y) Hammer [22], a hammer for Mizar [23], and a hammer for Metamath [12]. (See also [9] for a survey overview.) Dependently typed foundations like those of Coq [7], Agda [10], and Lean [15] offer better support for algebraic reasoning, but the added expressivity comes at a cost, and developing domain-general automation for dependently typed foundations is notoriously hard. Even applying a single lemma can require complex unification and elaboration processes, including definitional reductions, type class search, and other means of inferring implicit arguments. There is a hammer for Coq, CoqHammer [14], which uses a custom-designed inhabitation procedure for proof reconstruction [13], but that reconstruction procedure is tuned to Coq’s intuitionistic framework. A prototype hammer for Agda [18] uses only equational reasoning for reconstruction. There is currently no hammer for Lean.

This paper introduces *Duper*, a proof-producing theorem prover for Lean based on the superposition calculus [3]. *Duper* is written in the Lean programming language and it directly generates proofs as expressions in Lean’s axiomatic foundation. It can be called as a tactic when writing proofs interactively in Lean, but it is also intended to serve as a means of proof reconstruction for external automation. Most notably, *Duper* is designed to work in a dependently typed setting. Some of the challenges of working in a dependent type theory, such as the need to use type class inference to instantiate algebraic structures, are best handled in a monomorphization and preprocessing phase. For that, we use a tool called *LeanAuto*, developed by the second author, which will be described in detail elsewhere. Other challenges, such as Skolemizing formulas in an axiomatic framework in which quantifiers can, in principle, range over empty types, have to be handled natively. We deal with remaining challenges, such as polymorphic reasoning over types, polymorphic reasoning over type universes, and higher-order reasoning, using a flexible combination of preprocessing and native handling. *Duper* uses methods inspired by the Zipperposition prover [45] to carry out higher-order inferences, and the framework is flexible enough to accommodate, in the main loop, inferences that are specific to dependent types.

Our contributions are as follows:

- We have developed new ways of incorporating important aspects of dependently typed reasoning in a superposition theorem prover.
- We have developed ways of generating proofs directly in a dependently typed axiomatic framework.
- We make effective use of *LeanAuto*’s preprocessing to prove the kinds of goals that arise in practice when working with Lean and its library, *Mathlib*.
- We show that *Duper*’s performance is comparable to *Metis*’ on standard benchmarks in the interactive theorem proving community, specifically the *Seventeen provers* [16] and *GRUNGE* benchmarks [11].

*Duper* is available online at <https://github.com/leanprover-community/duper>.

## 2 Proof Search

*Duper* is a superposition theorem prover implemented in Lean. It accepts as input a goal state of the form  $E(\Gamma \vdash p : \text{Prop})$  where  $\Gamma$  is a local context, or list of hypotheses with corresponding types,  $E$  is a global environment, or list of declarations, and  $p$  is the target proposition. *Duper* also accepts as input a list of lemmas from  $E$  that may be relevant to the provided goal. Since *Duper* is not yet equipped with a relevance filter, these lemmas must currently be supplied manually. Given a goal state and lemma list, *Duper* will attempt to produce a proof term  $t$  such that  $E(\Gamma \vdash t : p)$ .

Although Duper’s input and output formats are tailored for closing Lean goals, its approach to proof search more closely resembles that of standalone automatic theorem provers than other forms of Lean automation. In this section, we describe the core aspects of this approach that are essential to Duper’s functioning as a superposition theorem prover.

## 2.1 Main Saturation Loop

Given a goal state  $E(\Gamma \vdash p : \text{Prop})$ , the first thing Duper does is negate the target  $p$  and add it to the local context  $\Gamma$ , resulting in the modified goal state  $E(\Gamma, \neg p \vdash \text{False} : \text{Prop})$ . Having a goal state of this form allows Duper to enter a main saturation loop in which it attempts to deduce all possible inferences from  $\Gamma$ ,  $\neg p$ , and the user-provided lemmas until either a contradiction is derived or no further inferences can be performed. In the former case, Duper uses the set of hypotheses that yielded a contradiction to produce the desired proof term  $t$ , and in the latter case, Duper informs the user that it was unable to prove the goal with the available lemmas. This high-level approach to proof search makes Duper a *saturation-based* theorem prover.

The procedure that drives the main saturation loop in most saturation-based theorem provers is called the *given clause procedure* [28]. Since different theorem provers implement different calculi and search heuristics, there are multiple variants of this procedure [8], the most common of which are the Otter loop [30] and the DISCOUNT loop [1]. Duper implements a variant of the DISCOUNT loop, described by Vukmirović et al. [45], that is designed to address complications that arise in higher-order unification.

At a high level, Duper’s given clause procedure functions by partitioning its derived facts into a set of fully processed clauses called the active set and a set of unprocessed formulas called the passive set. In each iteration of the main saturation loop, the procedure selects a new formula from the passive set called the “given clause,” clasifies it, simplifies it, and then uses generating rules to produce conclusions from it before transferring it to the active set and proceeding to the next iteration of the loop. Conceptually, this control flow is simple, though there are a variety of complications, such as the fact that some of Duper’s inference rules can generate infinitely many clauses. Solutions to complications arising from higher-order reasoning are described by Vukmirović et al. [45], and our solutions to complications arising specifically from dependent type theory are described in Section 4.2.2.

Aside from implementing a prover’s central control flow, one of the primary purposes of a given clause procedure is to manage redundancy. Since the search space that an automatic theorem prover can consider is so large, reducing it as much as possible is an extremely important part of building a performant prover. Toward this end, modern provers implement a variety of heuristics, strategies, and features to help minimize the time spent reasoning about clauses that are no longer necessary to obtain a contradiction.

One of the main ways the given clause procedure manages redundancy is by performing simplification rules. Simplification rules are like generating rules in that they can produce new conclusions for Duper to reason about, but they are unlike generating rules in that they also eliminate one of the rule’s premises from future consideration. Simplification rules can be applied as forward simplification rules, meaning they use clauses from the active set to modify or eliminate the current given clause, or as backward simplification rules, meaning they use the current given clause to modify or eliminate clauses in the active set.

Typically, in each iteration of the main saturation loop, forward simplification rules are applied first, then backward simplification rules, and finally inference rules. This order is ideal because it ensures that the given clause is simplified before it is used to modify the active set and that the active set is reduced as much as possible before it is used to generate inferences. Although this order is preferred, there are some special cases in which it must be violated. Such cases are discussed in Section 4.2.2.

## 2.2 Core Calculus

The core calculus Duper uses to generate new facts in its main saturation loop is based on Zipperposition’s  $\text{o}\lambda\text{Sup}$  calculus [5]. There are two primary reasons we chose to use this calculus. First, Zipperposition is a state-of-the-art theorem prover with an established track record of high performance, especially on higher-order problems [40, 41, 42, 44]. Second, Zipperposition’s calculus is designed to be a graceful generalization of first-order superposition, rather than a translation-based approach that transforms higher-order problems into first-order logic before calling an essentially first-order procedure. Since Duper operates in a dependently typed setting, we preferred an approach that supports native higher-order reasoning and can be extended to support native dependent type theory reasoning over an approach that requires translating to a less expressive logic.

Duper’s implementation of the superposition calculus has three primary components. First, there is the implementation of each individual rule and its corresponding proof-reconstruction procedure. For the most part, there are few substantive differences between how Duper implements its inference rules and how other provers might implement the same rules.

One aspect of rule implementation worth mentioning is how Duper defines the clauses that its inference rules act on. Most of Duper’s inference rules involve unification. Since that involves mapping variables to specific values, a clause’s variables may take on particular values for the sake of one inference but not others. It is therefore convenient to store clauses both in a permanent format that will be unchanged by inference generation as well as in a temporary format that can be modified by the unification procedure. To avoid confusion, we refer to facts in the first format as clauses and facts in the second format as mclauses. This naming convention arises from the fact that mclauses represent their variables with Lean metavariables whose types and values can easily be modified by the unification procedure.

The second component of Duper’s core calculus is a strict order on terms, literals, and mclauses. Having such an order enables Duper to impose side conditions on its inference rules which drastically improve performance by limiting the number of possible valid inferences. For this, Duper uses the Knuth-Bendix order (KBO) [24, 27] extended to be compatible with higher-order logic [5].

To implement this order for Lean expressions, which come in more categories than just symbols and variables, only a few additions and modifications are necessary. In mclauses, Duper uses Lean metavariables as free variables, and treats Lean constants, free variables, projections, strings, and natural numbers as “symbols.” Of these symbols, Duper treats constants as greatest, followed by free variables, projections, strings, and natural numbers, in that order. Lean expressions with additional metadata are definitionally equivalent to those same expressions without the metadata, so Duper simply removes them from all expressions prior to calling the KBO procedure. To handle  $\lambda$ -expressions, Duper follows the approach described by Bentkamp et al. [6]. This involves exhaustively applying  $\beta$ - and  $\eta$ -reduction rules to get rid of as many  $\lambda$ -expressions as possible, and then treating any remaining  $\lambda$ -expressions of the form  $(\lambda x : t. e[x])$  as expressions of the form  $(\text{LAM } t \ e[\#0])$  where “LAM” is a constant and  $\#0$  is a bound variable with De Bruijn index 0. Finally, to handle let expressions, Duper’s KBO procedure applies  $\zeta$ -reduction exhaustively along with  $\beta$ - and  $\eta$ -reduction which guarantees that no let expressions need to be considered.

The final component of Duper’s core calculus is a higher-order unification procedure. We extend the approach outlined by Vukmirović et al. [46] to dependent type theory. This enables Duper to generate arbitrarily many solutions to unification problems where infinitely many incomparable unifiers may exist. The specifics of how we extend this approach to dependent type theory are described in Section 4.1.

### 3 Proof Reconstruction

Once Duper derives a contradiction in its main saturation loop, it uses that contradiction to construct a proof term  $t$  such that  $E(\Gamma, \neg p \vdash t : \text{False})$ . Since Duper always begins by applying double-negation elimination to transform its input goal  $E(\Gamma \vdash p : \text{Prop})$  into the modified goal  $E(\Gamma, \neg p \vdash \text{False} : \text{Prop})$ , this suffices to close the original goal. We note that although this initial transformation is standard for many automatic theorem provers, it is only sound classically, as are many of the rules in Duper’s underlying superposition calculus. Consequently, Duper only generates classical proofs, even when given a goal that could be proved intuitionistically.

To construct  $t$ , Duper begins by collecting the list of clauses that were used to derive the contradiction it found. Duper then generates proof terms for each clause in this list using proof terms from earlier clauses or facts supplied as input. Since the final clause that Duper generates is the empty clause, the final proof term that Duper generates is  $t$ .

Since the addition of a new clause is justified by a rule and a list of parent clauses, one can construct the proof term for the result of an inference from those of the inference’s premises. The specifics of how each result is justified depend on which rule is used to produce it, so each of Duper’s rules is paired with a unique proof reconstruction procedure. In this section, we discuss two ideas relevant to the proof reconstruction procedure of several of our rules.

#### 3.1 Clause Instantiation

Each of Duper’s clauses is a disjunction of literals preceded by arbitrarily many universal quantifiers. So if conclusion  $D$  is meant to follow from premises  $C_1$  and  $C_2$ , the first step to constructing  $D$ ’s proof term is usually to instantiate all of the universal quantifiers at the heads of  $C_1$  and  $C_2$ . To guarantee that this is possible, if a rule’s proof reconstruction procedure is known to require this first step, the rule itself will output conclusions whose universal quantifiers include all of its parents’ universal quantifiers. Duper can then instantiate  $C_1$ ’s and  $C_2$ ’s universal quantifiers using free variables introduced from the target  $D$ .

Although this restriction on Duper’s inference rules ensures that Duper will always be able to instantiate  $C_1$ ’s and  $C_2$ ’s universal quantifiers with free variables introduced from  $D$ , there are some instances in which Duper must adopt a different instantiation strategy. This happens primarily when metavariables in mclauses are instantiated with particular values by Duper’s unification procedure. Since metavariables in mclauses correspond to bound variables in regular clauses, instantiating an mclause’s metavariable corresponds to instantiating a clause’s universal quantifier. When this instantiation strategy is used, the term used to instantiate the relevant quantifier is provided by the unification procedure.

This approach has some subtle consequences. The first is that Duper’s inference rules can generate conclusions containing universal quantifiers whose variables do not appear in the conclusion’s clause body. For example, given the premise  $\forall x : \alpha, f x \neq f x \vee a = b$ , Duper will recognize that the literal  $f x \neq f x$  is always false and eliminate it to produce the conclusion  $\forall x : \alpha, a = b$ . Even though  $x$  does not appear in the clause body  $a = b$ , the conclusion includes  $x$ ’s universal quantifier since  $x$  appears in the premise and is not assigned a value by the unification procedure.

To compensate for this behavior, Duper implements an additional simplification rule, called `removeVanishedVars`, which takes an arbitrary premise and attempts to produce a conclusion containing an identical clause body and strictly fewer universal quantifiers. Applying this rule requires knowing that the removed quantifiers range over inhabited types, so Duper must perform additional reasoning to determine which types are provably inhabited. The specifics of this additional reasoning are discussed in Section 4.2.1.

The second subtle consequence of this approach is that although most of Duper’s proof reconstruction can be delayed until after it is known which clauses appear in the final proof, some amount of proof reconstruction must be performed for every generated clause. Specifically, after any rule is carried out, all metavariable instantiations performed by the unification procedure must be recorded before the rule’s mclauses are forgotten. For this reason, Duper constructs partially instantiated parent terms immediately rather than wait until the end of the proof search. Transforming the conclusion’s mclause into a clause also requires taking the unification procedure’s instantiations into account, so performing these partial instantiations immediately does not result in significant additional overhead.

### 3.2 Transfer Expressions

The only input required by most rules’ proof reconstruction procedures is a list of proof terms corresponding to each premise and the type of the desired conclusion. In many cases, a rule’s proof reconstruction procedure amounts to applying a single polymorphic Lean theorem that justifies the rule’s soundness. However, some rules require additional information.

One example of such a rule is `ForallHoist`. This rule takes a clause  $C$  containing an expression  $e$  that can be unified with  $(\forall x : ?m1, (?m2 x))$  and cases on whether  $e$  is true. It does this by replacing  $e$  with `False` in  $C$  and giving  $C$  a new literal of the form  $(?m2 ?m3 = \text{True})$  where  $?m3$  is a fresh metavariable of type  $?m1$ . The idea is that if  $e$  is false, then replacing it with `False` in  $C$  is sound, and if  $e$  is true, then  $(?m2 ?m3)$  will also evaluate to true.

The issue is that although  $?m2$  appears in the parent clause that `ForallHoist` takes,  $?m3$  does not. So it’s not possible to produce a proof term for the correct conclusion by merely passing the parent’s proof term into even a polymorphic Lean theorem. The result type would contain an unassigned metavariable, which is disallowed by Lean’s application procedure. It is therefore necessary to, in addition to supplying the parent’s proof term, supply an expression corresponding to  $?m3$ . In principle, this expression can be derived from the type of the desired conclusion, but in practice, it is both simpler and more efficient to pass the necessary expression to the proof reconstruction procedure directly.

We call expressions that are passed directly to proof reconstruction procedures but are not proof terms for parent clauses “transfer expressions.” Several of the higher-order rules that Duper implements have proof reconstruction procedures that benefit from such transfer expressions, usually because these rules produce more involved unification problems. If Duper were a standalone automatic theorem prover that needed to interface with a separate interactive theorem prover, passing along transfer expressions might be prohibitively difficult. But because Duper is implemented in Lean, has clauses defined in terms of Lean expressions, and makes use of several of Lean’s metaprogramming functionalities, implementing transfer expressions is relatively straightforward.

## 4 Native Dependent Type Theory Reasoning

Lean’s type theory is based on the calculus of constructions (CoC) with a countable hierarchy of non-cumulative universes and inductive types [2]. Lean moreover allows users to mark arguments to be instantiated by type class inference. Developing an automatic theorem prover that can operate in the presence of these features poses a variety of challenges. Here, we discuss the problems that Duper handles natively. Additional issues that are addressed during preprocessing are discussed in Section 5.

## 4.1 Unification

Many of Duper’s inference rules involve unification. To ensure these rules can be applied even when dependent types are at play, it is important that Duper can support unifying dependently typed terms. Consider the following example which involves matrices defined in terms of `Fin n`, the (dependent) type of values less than `n`:

```
example (a b : Nat) (matrix : Fin a → Fin b → Nat)
  (transpose : ∀ n m : Nat, (Fin n → Fin m → Nat) → (Fin m → Fin n → Nat))
  (h : ∀ n m : Nat, (fun x => transpose n m (transpose m n x)) = (fun x => x)) :
  transpose b a (transpose a b matrix) = matrix := by
  duper [h]
```

This example can be proved by using `h` to rewrite `transpose b a (transpose a b matrix)` as `(fun x => x) matrix`. But in order to perform this straightforward rewrite, it is necessary to unify `matrix` of type `Fin a → Fin b → Nat` with `x` of type `Fin ?m → Fin ?n → Nat` (where `?m` and `?n` are assignable metavariables). From this, we can see that even simple examples may require Duper to use a unification procedure that supports dependently typed terms.

The unification procedure in Duper is based on that of Vukmirović et al. [46]. To unify two terms  $s$  and  $t$ , the unification procedure builds a tree with nodes of the form  $(E, \sigma)$ , where  $E$  is a multiset of unification constraints  $\{(s_1 \stackrel{?}{=} t_1), \dots, (s_n \stackrel{?}{=} t_n)\}$  and  $\sigma$  is a substitution. The tree is built by starting from the root  $(\{s \stackrel{?}{=} t\}, \emptyset)$  and progressively creating nodes according to transition rules. A transition rule  $(E, \sigma) \longrightarrow (E', \sigma')$  allows one to attach a new node  $(E', \sigma')$  to an existing node  $(E, \sigma)$ . It is expected that the solutions represented by  $(E', \sigma')$  is a subset of that of  $(E, \sigma)$ , and that either  $\sigma' = \sigma$  or  $\sigma'$  is a refinement of  $\sigma$ .

Among all the transition rules, the `Bind` rule is of great importance because it is the main rule that refines the substitution. The `Bind` rule is defined as

$$(\{s \stackrel{?}{=} t\} \uplus E, \sigma) \longrightarrow (\{s \stackrel{?}{=} t\} \uplus E, \rho\sigma) \quad \rho \in \mathcal{P}(s \stackrel{?}{=} t)$$

where  $\mathcal{P}(s \stackrel{?}{=} t)$  is the set of bindings applicable to  $s \stackrel{?}{=} t$ .

Both the transitions and bindings implemented by Duper are a superset of the ones described by Vukmirović et al., hence Duper’s procedure is complete for the higher-order fragment of Lean. In the following discussion, we use the notations and conventions established by Vukmirović et al., including describing the unification procedure as acting on free variables. Note that because Duper’s unification procedure operates on terms taken from `mclauses`, free variables in the following discussion correspond with metavariables in our implementation.

### 4.1.1 Transitions

In dependent type theory, types are also terms, and types can contain  $\forall$  binders. Consider the unification equation  $\forall x. F x \stackrel{?}{=} \forall x. g x$  where  $F$  is a free variable and  $g$  is rigid. It is easy to see that  $F \mapsto g$  is a solution. However, there are no transitions in Vukmirović et al.’s higher-order procedure applicable to this unification equation. To address this issue, we introduce a new transition:

$$\text{ForallToLambda} : (\{\forall x. f x \stackrel{?}{=} \forall x. g x\} \uplus E, \sigma) \longrightarrow (\{\lambda x. f x \stackrel{?}{=} \lambda x. g x\} \uplus E, \sigma)$$

Apart from this, the `Fail` and `Decompose` transitions from Vukmirović et al. are extended to handle dependent type theory features.



### 4.1.2 Bindings

In this section, we explain how *Imitation Binding* of Vukmirović et al. is extended to account for dependent types. Other bindings are extended in a similar way. Apart from these, a new binding *ImitForall* is introduced to deal with  $\forall$  quantifiers.

For unification equations of the form  $\lambda\bar{x}. F \overline{s_n} \stackrel{?}{=} \lambda\bar{x}. a \overline{t_m}$  where  $F : \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta$  and  $a : \gamma_1 \rightarrow \dots \rightarrow \gamma_n \rightarrow \beta$ , Vukmirović et al. allows the binding *Imitation of a for F* to be applied:

$$F \mapsto \lambda\overline{x_n}. a (F_1 \overline{x_n}) \dots (F_m \overline{x_n}).$$

Here  $\overline{F_m}$  are fresh free variables. This binding attempts to find a substitution for  $F$  such that the *head* of the two sides of the equations are identical.

Three issues are introduced when we take dependent type theory into account. Resolving these issues yields an imitation binding suitable for dependent type theory:

- $F$  and  $a$  might be dependently typed. Hence, even when the number of  $\lambda$  binders in the LHS and RHS of the unification equation are different, solutions might still exist. For example, consider unification equation  $\lambda x. F x \stackrel{?}{=} g U$  where  $F$  and  $U$  are free variables and  $F : \gamma \rightarrow \gamma$ ,  $U : \text{Type}$ , and  $g : \forall \alpha : \text{Type}. \alpha$ . Note that both LHS and RHS are  $\eta$ -expanded, and that the number of  $\lambda$  binders are different. However, the substitution  $\{F \mapsto g (\gamma \rightarrow \gamma), U \mapsto (\gamma \rightarrow \gamma)\}$  is clearly a solution to the unification equation. Thus, we need to extend the scope of the imitation binding to unification equations of form  $\lambda\overline{x_u}. F \overline{s_n} \stackrel{?}{=} \lambda\overline{x_v}. a \overline{t_m}$ , where  $u$  and  $v$  are not required to be equal.

- Since  $F$  and  $a$  might be dependently typed, their types should be written as

$$F : \forall (x_1 : \alpha_1) \dots (x_k : \alpha_k). \beta \overline{x_k}$$

$$a : \forall (y_1 : \gamma_1) \dots (y_l : \gamma_l). \delta \overline{y_l}$$

Again, we cannot assert that  $k = n$  or  $l = m$  because types can contain free variables. However, since both  $\lambda\overline{x_u}. F \overline{s_n}$  and  $\lambda\overline{x_v}. a \overline{t_m}$  are  $\eta$ -expanded, we know that  $k \leq n$  and  $l \leq m$ .

- For  $1 \leq i < j \leq n$ , the type of the bound variable  $x_j$  can depend on  $x_i$ , hence the binding for  $F$  should be written as

$$F \mapsto \lambda (x_1 : \alpha_1) \dots (x_k : \alpha_k). a (F_1 \overline{x_n}) \dots (F_h \overline{x_k})$$

Plugging the above binding into the unification equation and comparing the number of  $\lambda$ -binders on the two sides, we get  $h + n - k - u = m - v$ , hence  $h = m + k + u - n - v$ .

## 4.2 Inhabitation Reasoning

As noted in Section 3.1, sound proof reconstruction in a dependently typed setting requires reasoning about the presence of empty types. If any of a clause's universal quantifiers range over an empty type, then the clause is vacuously true and has an irrelevant clause body. One way we might respond to this issue is to observe that in many of the theorems that people actually care to prove, all of the nonpropositional types at play are known to be inhabited. In practice, it is often fine to have Duper temporarily assume that all nonpropositional types it encounters are inhabited and throw an error at the end if this results in an unsound inference. This is exactly what Duper does if passed in the option `inhabitationReasoning := False`.

When Duper is not passed in this option, it takes care throughout the reasoning process to ensure that empty types and potentially vacuous clauses are properly accounted for. In broad strokes, the differences in Duper's behavior when `inhabitationReasoning` is enabled or



disabled can be classified into two categories: there is the additional reasoning needed to determine whether any given type is inhabited, and there are the modifications to the main saturation loop needed to ensure that Duper correctly handles potentially vacuous clauses.

### 4.2.1 Determining Whether a Type Is Inhabited

Lean has a built-in type class `Inhabited`  $\alpha$  that indicates that  $\alpha$  is nonempty and supplies a witness attesting to this fact. When a type is known to be inhabited and is either built-in or part of the popular Mathlib library [29], it will typically already have an instance of this type class. Additionally, user-defined datatypes can often generate instances of this type class automatically with the syntax `deriving Inhabited`, and if a type is composed of inhabited types in certain preapproved ways, then Lean can automatically infer that the produced type is also inhabited. For example, from the instances `[Inhabited  $\alpha$ ]` and `[Inhabited  $\beta$ ]`, type class inference will automatically yield `[Inhabited  $(\alpha \times \beta)$ ]`.

So in most cases, when Duper encounters a type and needs to determine whether it is nonempty, Duper can simply make use of Lean's built-in type class inference. However, there are some goals, particularly those involving polymorphic types, for which this approach is insufficient. Consider the following example:

```
example (f :  $\alpha \rightarrow \alpha$ ) (h :  $\exists x : \alpha, f x \neq x$ ) :  $\exists x y : \alpha, x \neq y$  := by duper [h]
```

If we manually inspect this example, it is clear from hypothesis `h` that  $\alpha$  must be inhabited, otherwise it would be impossible for there to exist a value  $x$  of type  $\alpha$ . But Lean's type class inference on its own is insufficient to make use of this observation. In order for Duper to handle this example, it must explicitly note that  $\exists x : \alpha, f x \neq x$  entails that  $\alpha$  is inhabited.

More generally, when `inhabitationReasoning` is enabled, the simplification rules used to clausify quantifiers must do more than apply Skolemization to eliminate said quantifiers. We note that given a clause of the form  $(\exists x : \alpha, P(x)) \vee R$ , it is not always possible to generate a Skolem symbol of type  $\alpha$  since  $\alpha$  may not be inhabited. So instead, Duper generates a Skolem symbol `skS` of type  $\alpha \rightarrow \alpha$ , and produces the Skolemized clause  $\forall z : \alpha, (P(\text{skS } z) \vee R)$ . This approach is effective in pushing all quantifiers to the head of the clause, but it does not preserve the information that if  $R$  is false, then  $\alpha$  must be inhabited. To ensure that this information remains derivable, the simplification rules Duper uses to clausify quantifiers are expanded from just `Clausify $\exists$ 1` and `Clausify $\forall$ 1` to all of the rules in Figure 1.

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\exists y : \beta, p) = \text{True} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \forall z : \beta, p[y/(\text{skS } x_1 x_2 \dots x_n z)] = \text{True} \vee R} \text{Clausify}_{\exists 1}^{\exists}$$

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\exists y : \beta, p) = \text{True} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \text{Nonempty } \beta = \text{True} \vee R} \text{Clausify}_{\exists 2}^{\exists}$$

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\forall y : \beta, p) = \text{False} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \forall z : \beta, \neg p[y/(\text{skS } x_1 x_2 \dots x_n z)] = \text{True} \vee R} \text{Clausify}_{\forall 1}^{\forall}$$

$$\frac{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, (\forall y : \beta, p) = \text{False} \vee R}{\forall x_1 : \alpha_1, \dots \forall x_n : \alpha_n, \text{Nonempty } \beta = \text{True} \vee R} \text{Clausify}_{\forall 2}^{\forall}$$

■ **Figure 1** Quantifier clausification rules. The constant `skS` which appears in `Clausify $\exists$ 1` and `Clausify $\forall$ 1` is a fresh Skolem function of type  $\alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta \rightarrow \beta$ .

Note that although some preprocessing can be done to extract inhabited types from a goal state's original hypotheses, it is not possible in general to derive all type inhabitation information at the preprocessing stage. As an example, if a goal state contains the hypothesis

$(\exists x : \alpha, P) \vee Q$ , then after this hypothesis is classified to  $(\exists x : \alpha, P) = \text{True} \vee Q = \text{True}$ ,  $\text{Classify}_{\frac{3}{2}}$  can be used to derive  $\text{Nonempty } \alpha = \text{True} \vee Q = \text{True}$ . But this does not yet settle the question of whether  $\alpha$  is actually inhabited. Until  $Q$  is refuted, which may not happen until a later stage in the reasoning process, Duper cannot use this clause to determine  $\alpha$ 's inhabitation status. The fact that type inhabitation information must be derived throughout the reasoning process, rather than all at once during preprocessing, has consequences on the structure of the main saturation loop that are discussed in Section 4.2.2.

### 4.2.2 Modifications to the Main Saturation Loop

The main saturation loop described in Section 2.1 accurately reflects Duper's behavior when `inhabitationReasoning` is disabled. As long as all types are known or assumed to be inhabited, a dependently typed setting does not require any substantive changes from this procedure. However, when there are potentially vacuous clauses, meaning clauses with leading universal quantifiers whose types are possibly empty, the approach described in Section 2.1 can cause Duper to remove or ignore clauses that are necessary to obtain a contradiction.

To avoid removing or ignoring clauses that are necessary to obtain a contradiction, Duper must refrain from using potentially vacuous clauses to simplify away nonvacuous clauses. However, Duper cannot avoid reasoning about potentially vacuous clauses entirely, as they may be discovered to be nonvacuous at a later stage in the reasoning process. Additionally, Duper cannot fully defer reasoning about potentially vacuous clauses until they are determined to be nonvacuous because they may be needed to derive certain type inhabitation facts. For example, even if  $\alpha$  is a possibly empty type, it may be necessary to reason about the clause  $\forall x : \alpha, \text{Nonempty } \beta = \text{True} \vee P$  because if  $P$  can be refuted, then the resulting clause will entail that  $\alpha \rightarrow \beta$  is nonempty.

To satisfy these various requirements, when `inhabitationReasoning` is enabled, Duper adopts an alternative given clause procedure that has been modified in the following ways:

- When a given clause is identified as potentially vacuous, forward simplification rules and inference rules are still applied to it, but it is not used for backward simplification rules.
- When a potentially vacuous clause is added to the active set, it is still added to the data structures that allow it to be retrieved for inference rules and backward simplification rules. However, it is not added to any of the data structures that would enable it to be used in future forward simplification rules.
- After any clause is fully simplified, a check is run to see if any type inhabitation facts can be derived from the clause.
- When a new type is discovered to be nonempty, the set of clauses that Duper classifies as potentially vacuous is revisited and updated.
- When a clause previously classified as potentially vacuous is discovered to be nonvacuous, it is immediately used for backward simplification rules and is subsequently added to the data structures that enable it to be used in future forward simplification rules.

### 4.3 Universe Levels

Some problems require Duper to reason about universe polymorphic theorems and inductive types with universe level parameters. In many cases, these complications can be eliminated at the preprocessing stage by way of the monomorphization procedure described in Section 5. When feasible, this is Duper's preferred method of addressing universe polymorphism. Unfortunately, the procedure described in Section 5 cannot be applied to all problems. Even when it can be used, it is still possible for some of Duper's higher-order inference rules to

produce clauses containing universe polymorphic constants such as the equality and inequality predicates  $\text{@Eq}\{u\}$  and  $\text{@Ne}\{u\}$ . So Duper requires the ability to carry out some universe polymorphic reasoning natively.

During the main saturation loop, there are three areas where universe levels are relevant. First, Duper’s unification procedure must take universe levels into account. For the most part, this only requires calling Lean’s built-in level unifier in cases where a level metavariable determines whether two sorts or constants are unifiable. Second, converting between clauses and mclauses requires interchanging parameter names in clauses with level metavariables in mclauses. Finally, the Skolem symbols that Duper generates in Figure 1’s  $\text{Clausify}_1^{\exists}$  and  $\text{Clausify}_1^{\forall}$  are universe polymorphic, so some machinery is needed both to generate the initial Skolem symbols and to keep track of their parameters across inferences.

During proof reconstruction, Duper must occasionally instantiate universe polymorphic facts with specific universe levels. In some cases, Duper must even instantiate the same fact multiple times with different universe levels. Consider the following example:

```
theorem singletonListNotEmpty.{u} : ∀ α : Type u, ∀ z : α, ¬[z].isEmpty := ...

example (t1 : Type 1) (t2 : Type 2) (x : t1) (y : t2) :
  ¬[x].isEmpty ∧ ¬[y].isEmpty := by duper [singletonListNotEmpty]
```

In this example, Duper must instantiate `singletonListNotEmpty` with universe level 1 so that `¬[x].isEmpty` can be derived and with universe level 2 so that `¬[y].isEmpty` can be derived. During proof reconstruction, Duper determines this by examining the children clauses generated via inferences involving `singletonListNotEmpty`. If these clauses have specific universe levels instead of `singletonListNotEmpty`’s universe variable, then those universe levels determine how `singletonListNotEmpty` is instantiated. If the children clauses are also universe polymorphic, then their children clauses are recursively examined until a clause without `singletonListNotEmpty`’s universe variable is found. This is guaranteed to happen because repeatedly examining children will inevitably lead to the empty clause. The information concerning how universe levels must be instantiated can then be propagated from children to parents until it is known how to instantiate each of Duper’s universe polymorphic clauses.

## 5 Monomorphization

An advantage to using dependent type theory as a foundation is that it provides powerful mechanisms to support algebraic reasoning. When a user types an expression like `x + y` in a context where `x` and `y` are inferred to have type `Nat`, the expression is parsed as `HAdd.hAdd Nat _`, where the third argument, represented here by an underscore, is expected to be an *instance* of a type class for the addition notation. Lean will synthesize a suitable value, such as `@instHAdd Nat AddSemigroup.toAdd`, by searching through a database of instances that have been registered with the system. Type classes are similarly used to handle algebraic structures and generic theorems about them.

Type class inference is at odds with automated reasoning in a number of respects. First, it renders expressions quite verbose, increasing processing time and storage during search. Second, searching for instances is generally too expensive to carry out in Duper’s main saturation loop. Third, two fully elaborated instances of an expression like `x + y` may be only *definitionally equal* rather than syntactically equal, which means that Lean can determine they are the same only by unfolding definitions and simplifying them. This can happen when the system infers implicit arguments, like the addition function that is appropriate to the natural numbers, in different ways. Testing equivalence of terms up to definitional equality is also generally too expensive to carry out during the main saturation loop.

When using Duper as a tactic in Lean, we therefore rely on a preprocessing phase, implemented in a separate tool called LeanAuto,<sup>1</sup> that does all of the following:

- abstracts the proof goal as a smaller higher-order problem with dependently typed parameters that are kept separate;
- heuristically instantiates types and type classes in generic lemmas in the context; and
- identifies definitionally equal expressions with a single representative.

We refer to this as *monomorphization* although that term is more properly used to describe the second component above. The monomorphization procedure is complex and will be described in greater detail elsewhere. Here we sketch the main ideas.

## 5.1 Reduction to Essentially Higher-Order Problems

The first observation is that many goals that arise in Lean are *essentially higher-order* validities. For example, suppose we have variables  $n : \text{Nat}$  and  $a\ b\ c : \text{Fin } n$ , where  $\text{Fin } n$  is the (dependent) type of values less than  $n$ . Suppose further that we are trying to prove

$$a + (b + c) = (c + b) + a$$

from

$$\forall x\ y : \text{Fin } n, x + y = y + x.$$

If we abstract  $\text{Fin } n$  to a generic type  $\alpha$ , we get a first-order problem of which our original goal is an instance.

► **Definition 1.** Let  $\varphi$  be a proposition in Lean.  $\varphi$  is an *essentially higher-order validity* iff there exists a valid higher-order formula  $\psi$  and a mapping  $\sigma$  such that:

- $\sigma(\psi) = \varphi$ .
- $\sigma$  maps free variables to expressions in Lean.
- $\sigma$  maps constants, function symbols and type symbols to closed expressions in Lean. Moreover, we require that the equality symbol  $=$  in higher-order logic is mapped to  $=$  in Lean.
- $\sigma$  is type consistent, i.e. for each constant, function symbol, type symbol, or free variable  $x$  of type  $T$ ,  $\sigma(x)$  is of type  $\sigma(T)$ .

A Lean goal  $h_1 : T_1, h_2 : T_2, \dots, h_n : T_n \vdash T$  is *essentially higher(first)-order* iff  $\forall (h_1 : T_1) (h_2 : T_2) \dots (h_n : T_n), T$  is *essentially higher(first)-order*.

Given a Lean goal  $h_1, h_2, \dots, h_n \vdash \text{False}$ , our monomorphization procedure will instantiate the quantifiers of the premises and construct a new goal  $G$  with the resulting instances as premises and  $\text{False}$  as conclusion. Then, it attempts to find a higher-order formula  $\psi$  and a substitution  $\sigma$  such that  $\sigma(\psi)$  is equivalent to  $G$ . If successful,  $\psi$  is passed to the main saturation loop. Finally, if the saturation loop manages to find a proof of  $\psi$ , a proof of  $G$  will be reconstructed using  $\sigma$ .

## 5.2 Type Classes and Definitional Equality

Our monomorphization procedure implements a mechanism to check whether functions with different type class instance arguments are definitionally equal. For each function  $f$  with such arguments, the monomorphization procedure keeps a list of mutually (definitionally)

<sup>1</sup> <https://github.com/leanprover-community/lean-auto>

unequal expressions of the form  $f \bar{t}$ . Each time a new expression  $e$  of the form  $f \bar{t}$  is found by the monomorphization procedure, it compares  $e$  to the previously recorded expressions associated with  $f$  and checks whether they are definitionally equal. This mechanism cannot recognize all definitional equalities in Lean, since expressions with different heads can also be definitionally equal. However, this approach is effective in handling the sorts of type classes that tend to appear in Mathlib.

## 6 Evaluation

In this section, we evaluate Duper’s performance on first-order and higher-order benchmarks in the TPTP format [43]. Duper requires a Lean goal state as input, so to run Duper on these benchmarks, we wrote a parser to convert TPTP problems into Lean goals. Said parser is included in the Duper repository. The goal of our evaluation is to answer the following:

1. How does Duper compare against other automatic theorem provers? In particular, how does Duper compare against Metis, an automatic theorem prover frequently used for Sledgehammer’s proof reconstruction? This is a metric for Duper’s current strength as a general automatic theorem prover.
2. What is the impact of using external automation to minimize benchmarks on Duper’s performance? And is this impact dependent on the nature of the external automation? This is a metric for Duper’s current potential as proof reconstruction for a future hammer.
3. Some of Duper’s rules are marked as expensive based on their behavior in Zipperposition. What is the impact of enabling or disabling expensive rules on Duper’s performance? And is this impact dependent on whether Duper is given a problem in a first-order or higher-order format? This is a metric for determining in which circumstances Duper’s expensive rules should be enabled.

### 6.1 Experimental Methodology

#### 6.1.1 Benchmarks

We borrow benchmarks from *Seventeen Provers under the Hammer* [16] and *GRUNGE: A Grand Unified ATP Challenge* [11]. Both of these sources provide TPTP benchmarks in multiple formats, so we evaluate on the same set of benchmarks translated to a first-order form (FOF) and a typed higher-order form (THF). Specifically, we use the TH0<sup>-</sup> encodings from the *Seventeen* benchmarks and TH0-II encodings from the *GRUNGE* benchmarks.

The *Seventeen* benchmarks contain multiple versions of each problem with different numbers of additional facts supplied by Sledgehammer. We test on the same *Seventeen* benchmarks with 16 facts included and 256 facts included. The *GRUNGE* benchmarks are not duplicated in this manner, so for these, we just test on the available benchmarks in their “bushy” format, meaning the format containing exactly the facts needed to prove the original HOL4 [38] theorems from which the benchmarks were generated.

#### 6.1.2 Provers

In addition to Duper, we evaluate the benchmarks with Metis, Zipperposition, and Vampire [25]. Metis was selected as the standard for Sledgehammer-style proof reconstruction, Zipperposition was selected as a powerful automatic theorem prover that implements the same core calculus as Duper, and Vampire was selected as a powerful automatic theorem prover that implements a different core calculus from Duper.

We use Duper version v0.0.9, Metis version 2.4, Zipperposition version 2.1, and Vampire version 4.6.1. All provers except Duper are given arguments indicating they have a 30 second timeout, and all provers are externally terminated after 30 seconds of wall-clock time. Duper is run both with and without expensive rules enabled. Duper (–) is used to refer to Duper without expensive rules and Duper (+) is used to refer to Duper with “expensive” rules. Metis is run with default settings, while Vampire and Zipperposition are run with options replicating those in the *Seventeen* paper as closely as possible. Specifically, Vampire is run in portfolio mode with the “casc” schedule for FOF problems and the “casc\_hol\_2020” schedule for THF problems, while Zipperposition is run with the script `portfolio.lams.parallel.py` available in Zipperposition’s repository. We note that the Python code used to run Zipperposition’s portfolio mode on multiple cores is not Mac-compatible, so Zipperposition is only able to use one core of the computer used to run these experiments. Vampire’s portfolio mode code does not have such issues, so it was run unaltered. The computer used to run these experiments is a Mac with a 3.8GHz processor and 16GB of RAM.

## 6.2 Experimental Results

■ **Table 1** Original benchmark problems solved.

	FOF Format			THF Format		
	Seventeen (16 Facts)	Seventeen (256 Facts)	GRUNGE (Bushy)	Seventeen (16 Facts)	Seventeen (256 Facts)	GRUNGE (Bushy)
Duper (–)	1176/5000	1086/5000	64/1000	1111/5000	934/5000	231/1000
Duper (+)	1167/5000	1050/5000	64/1000	997/5000	670/5000	78/1000
Metis	1195/5000	1120/5000	202/1000	–	–	–
Vampire	1285/5000	2521/5000	262/1000	1331/5000	2341/5000	459/1000
Zipperposition	1277/5000	2209/5000	277/1000	1314/5000	2122/5000	354/1000

Table 1 shows the number of original benchmark problems solved by Duper, Metis, Vampire, and Zipperposition when given a 30 second timeout. Vampire and Zipperposition outperform Metis by a significant margin, especially as more facts are made available, and Metis outperforms Duper by a slimmer margin. We note that except for Metis, which only accepts FOF problems, all provers perform significantly better on *GRUNGE* THF problems than *GRUNGE* FOF problems. The difference in results between *GRUNGE*’s formats is significantly greater than the difference in results between *Seventeen*’s formats. This is likely explained by the fact that the *GRUNGE* benchmarks use a more inefficient encoding of polymorphism into the FOF format, apparently resulting in harder FOF problems [16].

■ **Table 2** Vampire-minimized Seventeen benchmark problems solved.

	FOF Format		THF Format	
	16 Facts	256 Facts	16 Facts	256 Facts
Duper (–)	1246/1285	2372/2521	1214/1331	2121/2341
Duper (+)	1244/1285	2368/2521	1149/1331	2025/2341
Metis	1268/1285	2382/2521	–	–

Tables 2 and 3 show the number of *Seventeen* benchmark problems that Duper and Metis can solve after they are minimized by Vampire and Zipperposition respectively. Minimized benchmark problems are generated by removing all axioms that do not appear in the proofs

■ **Table 3** Zipperposition-minimized Seventeen benchmark problems solved.

	FOF Format		THF Format	
	16 Facts	256 Facts	16 Facts	256 Facts
Duper (–)	1249/1277	2169/2209	1231/1314	2061/2122
Duper (+)	1248/1277	2166/2209	1214/1314	2014/2122
Metis	1267/1277	2165/2209	–	–

output by Vampire and Zipperposition. Thus, a minimized benchmark problem can only be generated if Vampire or Zipperposition solved the original benchmark problem. We only test on minimized *Seventeen* benchmark problems because the original *GRUNGE* benchmark problems already include exactly the axioms necessary to solve them.

Although Metis observably outperforms Duper on original benchmark problems, neither Duper nor Metis significantly outperforms the other on minimized problems. For both Vampire-minimized problems and Zipperposition-minimized problems, there is less than a 2% disparity in the number of reconstructed 16-Facts FOF problems, and less than a 1% disparity in the number of reconstructed 256-Facts FOF problems. Duper appears to do slightly better reconstructing Zipperposition’s proofs than Vampire’s, but the difference is marginal. We take this as a evidence that Metis’ and Duper’s abilities to reconstruct proofs generated by external provers are comparable.

In all benchmark categories, Duper performs better on average with its expensive rules disabled. We note that there are no FOF benchmarks that Duper can only solve with its expensive rules enabled, but there are 46 THF problems across the various categories that Duper requires expensive rules to solve. The fact that expensive rules appear to have some benefit for THF problems but no benefit for FOF problems is explained by the fact that most of Duper’s expensive rules are higher-order.

Overall, we conclude that on raw TPTP problems, Metis performs slightly better than Duper, but that the two tools perform extremely similarly on problems minimized by a more powerful external prover. We note that this evaluation of Duper’s performance is restricted to first-order and higher-order problems. Ideally, the features Duper implements that are oriented toward reasoning in a dependently typed setting would be evaluated using benchmarks from Mathlib, but until Duper is equipped with a relevance filter, such an evaluation is not feasible.

## 7 Related Work

Section 6 provides a quantitative evaluation comparing Duper against other automatic theorem provers. In this section, we discuss some of the qualitative differences between Duper and other general-purpose proof automation in various interactive theorem provers.

In Coq, the tactic most similar to Duper in purpose is `sauto` [13]. Just as Duper was designed with proof reconstruction for a future Lean hammer in mind, `sauto` was created to be CoqHammer’s [14] proof reconstruction procedure of choice. The primary difference between Duper and `sauto` is that Duper produces classical proofs and `sauto` produces constructive proofs. While Duper’s initial goal transformation and underlying superposition calculus are only sound classically, `sauto`’s direct search for type inhabitants in an appropriate normal form is fundamentally intuitionistic. For many Lean users, the benefit of being able to prove more facts outweighs the benefit of staying in Lean’s intuitionistic fragment, since many Lean users are formalizing classical mathematics in any case. On the other hand, many Coq



users are firmly committed to constructive proofs and their computational interpretations, so it makes sense that there would be a greater desire for Coq’s general-purpose automation to remain intuitionistic, even if it means fewer problems can be solved.

In Lean 3, the tactic most similar to Duper is Super.<sup>2</sup> Super is a prototype proof-producing superposition theorem prover implemented with Lean 3’s metaprogramming language. Due to inherent limits of Lean 3 metaprogramming, Super was not performant enough to make it into common use, but it was nonetheless an important proof of concept. The name “Duper” is an homage to that project.

In Lean 4, the most notable general-purpose proof automation tactic currently available is Aesop [26]. The primary difference between Duper and Aesop is that Duper is designed to require as little user input as possible, whereas Aesop is designed to be a highly customizable white-box automation tactic. There are benefits to both approaches. Aesop’s white-box approach gives users more control over how the proof search is performed, while Duper’s black-box approach is more amenable to push-button automation and has a lower barrier to entry for users. Overall, we hope that Duper and Aesop will prove to be complementary tactics well suited to different use cases.

In Isabelle/HOL, HOL4, and HOL Light, the proof automation most similar to Duper is Metis. Although the evaluation given in Section 6 is useful for gauging Metis’ and Duper’s relative performances, it is limited in that it considers only TPTP problems that can be expressed in both first- and higher-order logic. A fact that the evaluation does not capture is that there is a class of problems accessible to Duper but not Metis. This is the class of problems that fundamentally require native higher-order reasoning.

Even though Metis is a first-order prover, it can still solve some higher-order problems by first translating them into first-order logic. For many problems, this approach is sufficient, as evidenced by Metis’ frequent use in higher-order interactive theorem provers. However, as noted in *Mechanical Mathematics* [4], when problems critically involve higher-order constructions, their first-order translations can quickly become intractable. Consider the following problem which is presented both as an Isabelle/HOL lemma and as a Lean example:

```
lemma "( $\sum i::\text{nat}=0..n. i$ ) + ( $\sum i::\text{nat}=0..n. i$ ) = ( $\sum i::\text{nat}=0..n. i + i$ )"
  by (metis sum.distrib)

example :  $\sum i$  in range n, i +  $\sum i$  in range n, i =  $\sum i$  in range n, (i + i) :=
  by duper [Finset.sum_add_distrib]
```

When this problem is given to Isabelle/HOL’s Sledgehammer, Zipperposition is able to find a proof and suggests the above Metis call. However, Metis is unable to reconstruct Zipperposition’s proof because the summation notation involved is significantly more complex when encoded in first-order logic. On the other hand, Duper has no issue with the equivalent Lean example because it does not need to translate the problem into a less expressive logic.

## 8 Conclusion and Future Work

We have presented Duper, a proof-producing superposition theorem prover for Lean. Duper’s underlying approach to proof search adapts classical methods in automatic theorem proving to a dependently typed setting using a flexible combination of preprocessing and native reasoning. Since Duper directly generates proofs in Lean’s axiomatic foundation, it can be called as a terminal tactic in interactive Lean proofs.

<sup>2</sup> <https://github.com/leanprover/super>

In the future, we hope to use Duper for proof reconstruction in a Lean hammer. Equipping Duper with a relevance filter and integrating Duper into a full hammer pipeline will drastically increase Duper’s usefulness. The experimental results given in Section 6.2 show that Duper is performant enough to reconstruct proofs from a variety of state-of-the-art automatic theorem provers.

---

## References

- 1 Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: A system for distributed equational deduction. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*, pages 397–402. Springer, 1995. doi: 10.1007/3-540-59200-8\_72.
- 2 Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in Lean 4. URL: [https://leanprover.github.io/theorem\\_proving\\_in\\_lean4/](https://leanprover.github.io/theorem_proving_in_lean4/).
- 3 Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994. doi:10.1093/LOGCOM/4.3.217.
- 4 Alexander Bentkamp, Jasmin Blanchette, Visa Nummelin, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Mechanical mathematicians. *Communications of the ACM*, 66(4):80–90, March 2023. doi:10.1145/3557998.
- 5 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, and Petar Vukmirovic. Superposition for higher-order logic. *J. Autom. Reason.*, 67(1):10, 2023. doi:10.1007/S10817-022-09649-9.
- 6 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirovic, and Uwe Waldmann. Superposition with lambdas. *J. Autom. Reason.*, 65(7):893–940, 2021. doi: 10.1007/S10817-021-09595-Y.
- 7 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.
- 8 Jasmin Blanchette, Qi Qiu, and Sophie Tourret. Verified given clause procedures. In Brigitte Pientka and Cesare Tinelli, editors, *Automated Deduction - CADE 29 - 29th International Conference on Automated Deduction, Rome, Italy, July 1-4, 2023, Proceedings*, volume 14132 of *Lecture Notes in Computer Science*, pages 61–77. Springer, 2023. doi:10.1007/978-3-031-38499-8\_4.
- 9 Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formaliz. Reason.*, 9(1):101–148, 2016. doi:10.6092/ISSN.1972-5787/4593.
- 10 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – A functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9\_6.
- 11 Chad E. Brown, Thibault Gauthier, Cezary Kaliszyk, Geoff Sutcliffe, and Josef Urban. GRUNGE: A grand unified ATP challenge. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 2019. doi:10.1007/978-3-030-29436-6\_8.
- 12 Mario Carneiro, Chad E. Brown, and Josef Urban. Automated theorem proving for metamath. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPICs*, pages 9:1–9:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPICs.ITP.2023.9.
- 13 Lukasz Czajka. Practical proof search for Coq by type inhabitation. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference*,

- IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 28–57. Springer, 2020. doi:10.1007/978-3-030-51054-1\_3.
- 14 Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018. doi:10.1007/S10817-018-9458-4.
  - 15 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5\_37.
  - 16 Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. Seventeen provers under the hammer. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPICs*, pages 8:1–8:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.ITP.2022.8.
  - 17 Jeanne Ferrante and Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.*, 4(1):69–76, 1975. doi:10.1137/0204006.
  - 18 Simon Foster and Georg Struth. Integrating an automated theorem prover into Agda. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2011. doi:10.1007/978-3-642-20398-5\_10.
  - 19 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005. doi:10.1007/11541868\_7.
  - 20 John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 1996. doi:10.1007/3-540-61511-3\_97.
  - 21 Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, pages 56–68, 2003. URL: <http://www.gilith.com/papers>.
  - 22 Cezary Kaliszyk and Josef Urban. Hol(y)hammer: Online ATP service for HOL light. *Math. Comput. Sci.*, 9(1):5–22, 2015. doi:10.1007/S11786-014-0182-0.
  - 23 Cezary Kaliszyk and Josef Urban. Mizar 40 for mizar 40. *J. Autom. Reason.*, 55(3):245–256, 2015. doi:10.1007/S10817-015-9330-8.
  - 24 Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970. doi:10.1016/b978-0-08-012975-4.50028-x.
  - 25 Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013. doi:10.1007/978-3-642-39799-8\_1.
  - 26 Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 253–266. ACM, 2023. doi:10.1145/3573105.3575671.

- 27 Bernd Löchner. Things to know when implementing KBO. *J. Autom. Reason.*, 36(4):289–310, 2006. doi:10.1007/S10817-006-9031-4.
- 28 Ewing L. Lusk. Controlling redundancy in large search spaces: Argonne-style theorem proving through the years. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 96–106. Springer, 1992. doi:10.1007/BFB0013052.
- 29 The mathlib Community. The Lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.
- 30 William McCune and Larry Wos. Otter—the CADE-13 competition incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. doi:10.1023/a:1005843632307.
- 31 Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reason.*, 40(1):35–60, 2008. doi:10.1007/S10817-007-9085-Y.
- 32 Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Appl. Log.*, 7(1):41–57, 2009. doi:10.1016/J.JAL.2007.07.004.
- 33 Tobias Nipkow. Term rewriting and beyond - theorem proving in Isabelle. *Formal Aspects Comput.*, 1(4):320–338, 1989. doi:10.1007/BF01887212.
- 34 Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *J. Univers. Comput. Sci.*, 5(3):73–87, 1999. doi:10.3217/JUCS-005-03-0073.
- 35 Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010. doi:10.29007/36DT.
- 36 Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245. Springer, 2007. doi:10.1007/978-3-540-74591-4\_18.
- 37 William W. Pugh. The test: a fast and practical integer programming algorithm for dependence analysis. In Joanne L. Martin, editor, *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pages 4–13. ACM, 1991. doi:10.1145/125826.125848.
- 38 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Ait Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7\_6.
- 39 Robert Solovay, R. D. Arthan, and John Harrison. Some new results on decidability for elementary algebra and geometry. *Ann. Pure Appl. Log.*, 163(12):1765–1802, 2012. doi:10.1016/J.APAL.2012.04.003.
- 40 G. Sutcliffe. The CADE-27 Automated Theorem Proving System Competition - CASC-27. *AI Communications*, 32(5-6):373–389, 2020.
- 41 G. Sutcliffe. The 10th IJCAR Automated Theorem Proving System Competition - CASC-J10. *AI Communications*, 34(2):163–177, 2021.
- 42 G. Sutcliffe and M. Desharnais. The CADE-28 Automated Theorem Proving System Competition - CASC-28. *AI Communications*, 34(4):259–276, 2022.
- 43 Geoff Sutcliffe. The logic languages of the TPTP world. *Log. J. IGPL*, 31(6):1153–1169, 2023. doi:10.1093/JIGPAL/JZAC068.
- 44 Geoff Sutcliffe and Martin Desharnais. The 11th IJCAR automated theorem proving system competition - CASC-J11. *AI Commun.*, 36(2):73–91, 2023. doi:10.3233/AIC-220244.

- 45 Petar Vukmirović, Alexander Bentkamp, Jasmin Blanchette, Simon Cruanes, Visa Nummelin, and Sophie Tourret. *Making Higher-Order Superposition Work*, pages 415–432. Springer International Publishing, 2021. doi:10.1007/978-3-030-79876-5\_24.
- 46 Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification. *Logical Methods in Computer Science*, Volume 17, Issue 4, December 2021. doi:10.46298/lmcs-17(4:18)2021.