Aesop: White-Box Best-First Proof Search for Lean

Jannis Limperg
Vrije Universiteit Amsterdam
Department of Computer Science
Amsterdam, The Netherlands
j.b.limperg@vu.nl

Asta Halkjær From
Technical University of Denmark
DTU Compute
Kongens Lyngby, Denmark
ahfrom@dtu.dk

Abstract

We present Aesop, a proof search tactic for the Lean 4 interactive theorem prover. Aesop performs a tree-based search over a user-specified set of proof rules. It supports safe and unsafe rules and uses a best-first search strategy with customisable prioritisation. Aesop also allows users to register custom normalisation rules and integrates Lean's simplifier to support equational reasoning. Many details of Aesop's search procedure are designed to make it a white-box proof automation tactic, meaning that users should be able to easily predict how their rules will be applied, and thus how powerful and fast their Aesop invocations will be.

Since we use a best-first search strategy, it is not obvious how to handle metavariables which appear in multiple goals. The most common strategy for dealing with metavariables relies on backtracking and is therefore not suitable for best-first search. We give an algorithm which addresses this issue. The algorithm works with any search strategy, is independent of the underlying logic and makes few assumptions about how rules interact with metavariables. We conjecture that with a fair search strategy, the algorithm is as complete as the given set of rules allows.

CCS Concepts: • **Mathematics of computing** → *Mathematical software*; • **Theory of computation** → Type theory; *Logic and verification*; *Automated reasoning*.

Keywords: proof search, tactic, Lean, interactive theorem proving, deductive verification, type theory

ACM Reference Format:
Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3573105.3575671

1 Introduction

One of the biggest barriers to a more widespread adoption of interactive theorem provers is the tedium of proving lemmas which are entirely obvious to the human eye. The provers force us to explicitly demonstrate that $n * 2$ is even, that $[z, y, x]$ is a permutation of $[x, y, z]$ or that a homomorphism of groups is also a homomorphism of the underlying semigroups. This adds substantially to the cost of using theorem provers, which is still too high for many applications.

To help address this issue, we present Aesop (Automated Extensible Search for Obvious Proofs), a new proof search tactic for the upcoming version 4 of the Lean theorem prover [7]. In essence, Aesop is a tree-based search procedure which operates on a user-specified set of rules. The rules are arbitrary Lean tactics which, given a goal, either succeed — generating zero or more subgoals — or fail. Aesop applies these rules to the initial goal, then to the subgoals, etc., to build a search tree. On top of this basic setup, we provide the following features:

- Aesop uses a best-first search strategy, prioritising more promising rules (and their subgoals) over less promising ones. Which rules are considered promising is specified by the users themselves, using a simple prioritisation mechanism.
- Aesop distinguishes between safe rules, which are applied eagerly without backtracking, and unsafe rules, which may be backtracked. Safe rules are efficient since the goals to which they apply never need to be revisited.
- Aesop introduces a normalisation phase in which special normalisation rules are applied in a fixpoint loop to normalise the goal, before any other rules are applied. We use normalisation to establish invariants which the subsequent rules can rely on. For example, we split hypotheses of the form $P_1 \wedge \ldots \wedge P_n$ into separate hypotheses P_i, establishing the invariant that no hypothesis is a conjunction.
- The normalisation phase includes an invocation of Lean's simplifier, which performs rewriting with user-specified, possibly conditional equations. This allows us to benefit from the large collection of simplification rules which are typically defined by Lean projects.
- With best-first search, it is not obvious how to deal with metavariables which appear in multiple goals. In search procedures based on backtracking, such as

depth-first search, if a metavariable assignment turns out to be wrong, we can simply backtrack it and try a different one (assuming that the theorem prover's data structures efficiently support this). By contrast, a best-first algorithm must be able to consider multiple assignments in parallel. To address this issue, we present a new algorithm which handles metavariables and is independent of the search strategy.

Users of Isabelle's `auto` [18, 20] will recognise some of these features. More broadly, Aesop stands in the tradition of *white-box* proof automation tools, which also include Coq's `(e)auto`, PVS's `grind` [5] and ACL2's waterfall [11]. White-box tools require users to curate a set of rules which the tool applies. In return, users gain control over the power-performance tradeoff: many explosive rules make the automation stronger but slower; few conservative rules make it weaker but faster. Due to their customisability, white-box tools can also serve as a foundation for domain-specific automation, using domain-specific rule sets.

In contrast, *black-box* or *push-button* tools such as hammers [3], which invoke external automated theorem provers to find proofs, or machine learning systems which write a tactic script [2, 8, 10, 13], aim to operate with little or no user interaction. This makes them very convenient when they succeed, but the complex algorithms which deliver high success rates can be brittle. Sometimes a minor reformulation makes the difference between finding or not finding a proof. When a black-box tool does not find a proof or is slow to find one, it is often unclear how to improve the tool's performance.

White-box and black-box tools thus have complementary strengths and weaknesses, and so we believe it is worthwhile to explore both approaches. Aesop is an attempt to move far to the white-box end of the spectrum while retaining some of the useful features of Isabelle's `auto` and other systems. This is why we choose tree-based search as a base: it is easy to understand and close to interactive proof, which helps users predict how their rules will affect the search. We choose best-first search with customisable prioritisation (rather than some opaque heuristic) to give users more control over Aesop's performance. And we introduce fixpoint-based normalisation as an intuitive and reliable way to establish invariants. Taken together, these features should enable users to design effective and reasonably efficient rule sets for many domains.

Aesop is available as a Lean 4 library.[1] The specific version described here is available as a supplement to this paper.[2]

2 Best-First Proof Search

At its core, Aesop performs a tree-based, best-first proof search. This approach is independent of the underlying logic, so it could also be used as a proof method for, say, first-order or higher-order logic, though we will use the notation of dependent type theory for examples. For now, we assume that goals do not contain metavariables, which simplifies the algorithm considerably.

2.1 Goals and Rules

We assume a set of *goals* given by the underlying logic. These could, for example, be first-order sequents or higher-order formulas. In Lean, they are structures of the form $\vec{h} : \vec{T} \vdash U$, where $\vec{h}$ is a list of hypotheses with types $\vec{T}$ and U is a type. Each hypothesis may depend on earlier hypotheses (so $\vec{h}$ is a telescope) and U may depend on all hypotheses. We call U the goal's *target*.

We also assume a finite set of *rules*, which are partial functions that map a goal to a finite set of goals. When a goal is in the domain of a rule, we say that the rule is *applicable* to the goal. In the Aesop implementation, rules are arbitrary tactics.

When applied to a goal G, a rule produces a set of subgoals $G_1, \ldots, G_n$. Rules should be *provability-reflecting*, meaning that if the subgoals G_i are provable in the underlying logic, then the initial goal G is also provable. For instance, an $\wedge$-introduction rule would map the goal $\Gamma \vdash P \wedge Q$ to the set $\{\Gamma \vdash P, \Gamma \vdash Q\}$. If a rule generates no subgoals, it proves the goal outright.

2.2 Search Tree

Aesop's central data structure is a search tree containing two alternating kinds of nodes: *goal nodes* and *rule application ('rapp') nodes*. The children of a goal node are rapp nodes representing rules which have been applied to the goal. The children of a rapp node are goal nodes representing the subgoals generated by the rule. For example, the goal $\vdash P \wedge Q$ could have a child rapp for $\wedge$-introduction with two subgoals $\vdash P$ and $\vdash Q$.

At any point during the search, a node (goal or rapp) is in one of three states:

- *proved*: the node is proved. For a goal node, this means that *at least one* of its child rapps is proved. For a rapp node, it means that *all* of its child goals are proved. So sibling goal nodes are implicitly conjoined and sibling rapp nodes are implicitly disjoint, making the tree an AND/OR tree.
- *stuck*: the node cannot be proved with the given rules. For a goal node, this means that (a) all rules which can be applied to the goal have been applied and (b) *all* resulting child rapps are stuck. For a rapp node, it means that *at least one* of its child goals is stuck.
- *unknown*: the node is neither proved nor stuck.

The state of a node matters only insofar as it is necessary to determine the state of its parent node, then the parent's parent, etc., until we ultimately learn whether the root goal is

[1] https://github.com/JLimperg/aesop
[2] https://doi.org/10.5281/zenodo.7424818

proved or stuck. This means that nodes can become *irrelevant* during the search. For example, if a goal $\vdash P \lor Q$ has child rapps for left or-introduction (with subgoal $\vdash P$) and right or-introduction (with subgoal $\vdash Q$), and $\vdash P$ is already proved, then $\vdash P \lor Q$ is also proved and there is no point in trying to prove $\vdash Q$. In general, we say that a node is irrelevant if at least one of its ancestors, including the node itself, is already proved or stuck. Incidentally, when the search terminates successfully, the root goal becomes proved and therefore, by our definition, irrelevant. So 'irrelevant' means 'irrelevant for the rest of the search', not 'irrelevant for the proof'.

2.3 Search Algorithm

The search procedure starts with a search tree containing a single goal. It then enters a loop which, in each iteration, picks a goal node G with unknown state and a rule R which has not yet been applied to G. We then apply R to G. If this fails, we continue with the next rule and goal; if it succeeds, we add a rapp node for R with parent G and subgoals $R(G)$ to the tree. We call this operation the *expansion* of G along R. We exit the loop when the root goal becomes proved or stuck (or when one of several configurable limits, e.g. on the depth of the search tree, is reached).

Which goal is expanded first, and along which rule, is determined by a best-first search strategy. Usually, best-first search is realised by a heuristic which ranks goals and rules according to simple numeric properties, e.g. the size of a goal or the number of subgoals of a rule. This goes against Aesop's white-box philosophy since the heuristics tend to be fixed (so users cannot easily change them) and opaque (so users cannot easily predict which goals will be prioritised). Instead, we implement a scheme whereby the rules carry a user-defined priority which is used to rank both goals and rules.

Specifically, Aesop users give each rule a *success probability* between 0% and 100%. This probability is a rough estimate of how useful a rule is, i.e. how likely it is to lead to a proof. For example, left and right $\lor$-introduction could each be given a success probability of 50%.

For rules whose success probability is less obvious, we have found it sufficient in practice to pick probabilities from a six-point scale: last resort (1%), low (25%), medium (50%), high (75%) and almost always (99%). The probabilities could also be determined by automated methods, for example by determining the actual success probability of each rule in an existing corpus of proofs. But while such automated tuning would perhaps improve Aesop's overall performance in a larger library, it would also likely make some previously successful proofs fail, leading to maintenance challenges.

From the rules' success probabilities we derive, for each goal in the search tree, a *priority* between 0% and 100%. The root goal has priority 100%. Then, whenever we apply a rule R to a goal G, the priority of the subgoals is the priority of G multiplied with the success probability of R. In each iteration of the search loop, Aesop picks the highest-priority goal and expands it along the rule with the highest success probability. We could also allow rules to give different priorities to their subgoals, e.g. to prioritise goals which are known to quickly become unprovable if the initial goal is unprovable.

2.4 Safe and Unsafe Rules

So far, we have treated all rules as *unsafe*. An unsafe rule is one that does not necessarily preserve provability: when applied to a provable goal G, it may generate unprovable subgoals. For our search, this means that we must continue to expand both G and the subgoals.

However, in practice there are many rules which preserve provability and are therefore *safe*. For instance, $\land$-introduction is safe: to prove $\Gamma \vdash P \land Q$, it suffices to prove $\Gamma \vdash P$ and $\Gamma \vdash Q$. So after this rule has been applied, the original goal $\Gamma \vdash P \land Q$ does not need to be considered any more, shrinking the search space.

To take advantage of this insight, Aesop, like Isabelle's `auto`, lets users mark rules as safe. To accommodate these safe rules, we split the expansion of a goal G into two phases. First, Aesop tries to apply all safe rules to G. If one of them succeeds, the resulting subgoals are added to the tree as usual. An unsafe rule would then re-insert G into the goal queue which we maintain throughout the search, to give other rules a chance to fire. For safe rules, we simply skip this step, ensuring that G is never expanded again. If no safe rules are applicable to G, Aesop moves to the second phase, in which unsafe rules are applied as explained above.

Safe rules are considered to have success probability 100%, so the subgoals of a safe rule receive the same priority as the parent goal. To control the order in which safe rules are tried, users can give them an integer priority. This order does not affect provability — assuming that rules marked as safe are actually safe — but it does affect the search performance. For instance, suppose we have, in addition to safe $\land$-introduction, a safe rule R that transforms a hypothesis $h : A$ into $h : B$. Then for the goal $h : A \vdash P \land Q$, it is better to apply R before $\land$-introduction; otherwise we would have to apply R twice.

In practice, the distinction between safe and unsafe rules can be tricky since safe rules must preserve provability relative to the whole rule set. When we mark $\land$-introduction as safe, we require the rest of the rule set to maintain the invariant that whenever we can prove $\Gamma \vdash P \land Q$, we can also prove $\Gamma \vdash P$ and $\Gamma \vdash Q$. This invariant can be violated, for example, by registering an unsafe rule R which proves $P \land Q$: the rule will never be applied since any goal $\Gamma \vdash P \land Q$ gets split by "safe" $\land$-introduction before R can be tried. So we must add more rules to ensure that $\Gamma \vdash P$ and $\Gamma \vdash Q$ can also be proved — or consider $\land$-introduction unsafe after all.

2.5 Normalisation

Besides safe and unsafe rules, Aesop introduces a third category of *normalisation* rules. These are rules which normalise and simplify a goal, preparing it for further rule applications. Like safe rules, normalisation rules should preserve provability. Unlike safe rules, they must either prove the goal outright or return a single subgoal. For example, Aesop's default normalisation rules introduce assumptions, unfold certain definitions and prove trivial equations, reducing the goal $\vdash \forall f,$ `map f [] = []` first to $f \vdash$ `[] = []` and then to $f \vdash$ `True`.

Normalisation rules are applied in yet another expansion phase, before the safe and unsafe phases. Like safe rules, they have a user-specified integer priority determining the order in which they are applied. Let $R_1, \dots, R_n$ be the normalisation rules in this order. During the normalisation phase, Aesop then runs a loop which updates the goal. In each iteration, this loop tries to apply first R_1, then, if it fails, R_2, and so on. As soon as one of the R_i succeeds, the goal is set to the subgoal generated by R_i and the loop restarts. (If R_i produces no subgoal, the goal is proved and we are done.) If all the R_i fail, the loop ends. Compared with a simpler fixpoint loop which executes R_1, …, R_n, R_1, …until all the R_i fail, this method has the advantage that the order of rules is always respected, so on each intermediate goal R_1 is tried before R_2.

Like safe rules, normalisation rules must be chosen carefully to ensure that they preserve provability relative to the whole rule set. If we, for example, rewrite with the unfolding rule `[x] ++ xs = x :: xs` during normalisation, rules about concatenation no longer apply to the normalised goal. If this is not desired, the unfolding is better performed as an unsafe rule or added as a local rule when needed. A common pattern is to register unfolding equations as normalisation rules while we prove facts about the respective definition (which almost always requires unfolding) and then remove them again for the remainder of the library.

2.6 Safe Goals

When Aesop fails to prove a goal, it reports the *safe goals*. These are the goals that would remain if we were to run Aesop with only normalisation and safe rules. Since normalisation and safe rules are non-branching (meaning each goal expanded by such a rule has exactly one child rapp), applying them exhaustively results in a single set of safe goals.

The safe goals are interesting because they indicate how much progress Aesop has made in the safe, non-branching part of its search. A typical Aesop proof workflow looks like this:

- Run Aesop on a goal G. If this proves the goal, we are done. Otherwise Aesop produces safe goals $G_1, \dots, G_n$.
- Manually perform some proof steps on each safe goal G_i, producing a goal G_i'.
- For each G_i', apply this workflow recursively.

Once the proof is complete, we collect the manual steps and turn them into Aesop rules. This allows Aesop to prove the initial goal G fully automatically — and hopefully other, similar goals as well.

To report the safe goals, we must address one minor complication. It is possible for the search to terminate before all safe goals have been generated. For example, suppose we register $\wedge$-introduction as a safe rule and search for a proof of the goal $\bot \wedge (P \wedge Q)$. Then the safe goals are $\bot$, P and Q. But Aesop may terminate after the first $\wedge$-introduction, realising that the goal $\bot$ cannot be proved, without ever applying the second $\wedge$-introduction to $P \wedge Q$. So it would wrongly report $\bot$ and $P \wedge Q$ as safe goals. Hence we must expand all relevant safe rules (here: $\wedge$-introduction on $P \wedge Q$) before computing the safe goals.

2.7 Multi-Rules

It is sometimes useful for a rule to add multiple rapps at once. For example, we will shortly see a rule which tries to apply the constructors of an inductive type. If more than one constructor can be applied, it is more natural (and slightly faster) to let the rule add one rapp per applicable constructor, rather than making each constructor a separate rule. We call such rules *multi-rules.*

Unsafe multi-rules are a straightforward generalisation of unsafe regular rules and require almost no changes to the search procedure. Safe and normalisation multi-rules are trickier. Normalisation multi-rules are not allowed at all since normalisation cannot branch. Safe multi-rules could be allowed, but their behaviour would be unintuitive: the whole raison d'être of safe rules is that they, too, do not branch. So we also forbid safe multi-rules.

The prohibition of safe and normalisation multi-rules is enforced dynamically, meaning that users may register safe and normalisation multi-rules but they fail if they actually generate multiple rapps. This is convenient because, for example, a rule that applies the constructors of an inductive family can be perfectly safe if for any given goal at most one constructor is applicable, which is the case for many inductive predicates and relations. Such multi-rules are effectively non-branching, so we should not ban them outright.

3 Best-First Proof Search in Lean

We now instantiate our best-first search framework to obtain a practical proof method for Lean.

3.1 Rule Builders

Aesop rules are arbitrary tactics, but it would be highly inconvenient if users had to write a tactic whenever they want to add, say, a lemma as a rule. We therefore provide several *rule builders* which register theorems, definitions or types as rules. Rules are registered either locally, i.e. for a single Aesop invocation, or globally in a rule set. Rule sets are collections of rules which can be activated or deactivated for each Aesop invocation. The distinguished `default` rule set is activated by default.

3.1.1 `apply`.

Given a term f of type $\forall \vec{x} : \vec{T},\ P\,\vec{x}$, the `apply` builder creates a rule which applies f to goals $\Gamma \vdash P\,\vec{y}$. The arguments $\vec{x}$ are either inferred (by unification or type-class search) or become subgoals.

When we write $P\,\vec{x}$, P is an arbitrary type-valued function (e.g. $\lambda\, x\, y,\ x = y + 1$), so $P\,\vec{x}$ is essentially an arbitrary type-valued term involving the variables $\vec{x}$. However, due to the inherent limitations of higher-order unification, our `apply` builder cannot support all functions P; it uses the same heuristics as Lean's `apply` tactic to support a useful subset. Similar caveats also apply to some of the following rule builders.

3.1.2 `constructors`.

The `constructors` builder creates a rule which applies the constructors of an inductive type I. The rule has the same effect as if each constructor of I had been added as an `apply` rule, except that these `apply` rules are combined into one multi-rule.

3.1.3 `forward`.

Given a term $f : \forall \vec{x} : \vec{T},\ P\,\vec{x}$, the `forward` builder creates a rule which performs forward reasoning with f. This means that whenever a goal's local context contains hypotheses $\vec{h} : \vec{T}$, the rule adds a new hypothesis $h' : P\,\vec{h}$. For example, the left $\wedge$-elimination lemma $\forall A\, B,\ A \wedge B \rightarrow A$, when used as a `forward` rule, reduces the goal $h : A \wedge B \vdash T$ to $h : A \wedge B,\ h' : A \vdash T$. If there are multiple sets of hypotheses with types $\vec{T}$, one new hypothesis is added for each set.

More generally, users can partition the arguments $\vec{x}$ into *immediate* arguments $\vec{a} : \vec{A}$ and *non-immediate* arguments $\vec{b} : \vec{B}$. Then, Aesop searches only for hypotheses $\vec{h} : \vec{A}$ corresponding to the immediate arguments and, if successful, adds a hypothesis of type $\forall \vec{b} : \vec{B},\ P\,\vec{h}\,\vec{b}$. (This notation suggests that the immediate arguments must precede the non-immediate ones, but in fact they can be interleaved freely.) So the immediate arguments must be "immediately available" as hypotheses while the non-immediate ones remain premises to be proved later. By default — and in our example above — all arguments which cannot be inferred are considered immediate.

In the example, the left $\wedge$-elimination rule is again applicable to the subgoals it generated. This is a general issue with `forward` rules: when a rule applies to a set of hypotheses $\vec{h}$, the subgoals still contain $\vec{h}$, so the rule is still applicable. To prevent this sort of looping, whenever a `forward` rule tries to add a hypothesis $h : T$, we check whether any `forward` rule that was applied earlier on this branch of the search tree already added a hypothesis $h' : T$. If so, the new hypothesis is not added and the rule fails.

There is also a variant of `forward`, `destruct`, which removes any hypotheses that matched the immediate arguments. If we use left $\wedge$-elimination as a `destruct` rule, it reduces the goal $h : A \wedge B \vdash T$ to $h : A \vdash T$. Since the matched hypotheses are removed, `destruct` rules do not generally apply to their own subgoals, so there is no need to prevent cycles.

3.1.4 `cases`.

Given an inductive family I with arguments (parameters and indices) $\vec{x} : \vec{T}$, the `cases` builder creates a rule which performs case analysis on any hypothesis $h : I\,\vec{x}$. For example, the `cases` rule for `Or`, the inductive type behind the notation $P \vee Q$, reduces the goal $h : P \vee Q \vdash T$ to two subgoals $h : P \vdash T$ and $h : Q \vdash T$. To perform this case analysis, we use Lean's built-in `cases` tactic, which uses the standard elimination principle for I.

To perform case analysis according to a non-standard elimination principle, we can use the *view pattern* [15]: define a data type J whose constructors correspond to the desired cases, register a function $f : I \rightarrow J$ as a `destruct` rule and register a `cases` rule for J. With this setup, the goal $h : I \vdash T$ is first reduced to $h : J \vdash T$ and then h is split into the desired cases.

Like `forward` rules, `cases` rules for recursive types, such as lists or trees, can loop. If we register a `cases` rule for the List type, the goal l : `List` $\alpha \vdash P\,l$ is split into two goals $\vdash P\,[]$ and $a : \alpha,\ l :$ `List` $\alpha \vdash P\,(a :: l)$ and the `cases` rule is again applicable to the second goal.

One solution for this problem is to register the `cases` rule as an unsafe rule with very low priority. Aesop then uses it only as a last resort. This method is simple and effective, but it is problematic if Aesop does not find a proof: once there are no other rules left to apply, the `cases` rule is, as before, applied ad infinitum. This sort of `cases` rule is therefore only suitable as an ad hoc rule.

To support global `cases` rules as well, we provide a variant of the `cases` builder which avoids looping in some common cases. Consider the inductive predicate `All P xs`, which encodes the proposition that all elements of the list `xs` satisfy the predicate `P`:

```
inductive All (P : α → Prop) : List α → Prop
| nil  : All P []
| cons : P x → All P xs → All P (x :: xs)
```

When a goal contains a hypothesis `All P (x :: xs)`, we almost always want to perform a case split on this hypothesis, leaving us with two simpler hypotheses `P x` and `All P xs`. Crucially, neither of these hypotheses has the same form as the initial one, so there is no infinite regress. To take advantage of this insight, Aesop allows users to annotate a `cases` rule with a *pattern* which restricts the hypotheses to which the rule is applicable. In our example, we would use the pattern `All _ (_ :: _)` to ensure that an `All` hypothesis is only split if it refers to a non-empty list. Multiple patterns can also be given; the rule is then applied if at least one of the patterns matches a hypothesis.

We also considered a third solution to the infinite regress issue: we could stipulate that once a `cases` rule has been applied to a hypothesis, it cannot be applied again to the descendants of that hypothesis; or, more generally, that it can only be applied to the first n descendants. The vast majority of proofs should still work for, say, $n = 3$. Unfortunately the restriction is somewhat tricky to implement since Lean does not provide a reliable way to associate metadata with a hypothesis, but we want to support this in the future.

3.1.5 `tactic`.

The last and most fundamental rule builder, `tactic`, allows users to register any tactic as a rule. The tactic can generate arbitrary subgoals (justified by a proof term that is later checked by Lean's kernel). We only require that `tactic` rules either change the goal or fail, so they cannot be no-ops.

3.2 Simplifier Integration

Lean's simplifier, which performs rewriting with a user-provided set of conditional rewrite rules, is used heavily in all big Lean projects. In particular, mathlib [4], a large library of formalised mathematics which contains most Lean code written to date, defines an extensive set of simplifier rules. To make Aesop practical, we should leverage this existing automation.

To that end, we integrate simplification into the normalisation process, adding a built-in normalisation rule which runs the simplifier on the entire goal (target type and hypotheses). This invocation of the simplifier uses the default global set of rewrite rules, plus a separate Aesop-specific rule set. Aesop users can add rules to this set by using a special `simp` rule builder.

An important detail of the simplifier integration concerns how we use local hypotheses. Lean's simplifier can be configured to use them in two ways. First, local equations can be used as rewrite rules, transforming the goal $h : x = y \vdash P\,x$ into $h : x = y \vdash P\,y$. This can be dangerous since local equations are not necessarily oriented in a way that works well with other rules. For example, a rule that proves $P\,x$ may not fire any more. Worse, a rogue equation can easily make the simplifier loop.

Second, local hypotheses which are propositions (but not equations) can be rewritten to truth values, transforming the goal $h_1 : P,\ h_2 : \neg Q \vdash P \vee Q$ first into $\ldots \vdash \top \vee \bot$ and then, via a global rewrite rule, into $\ldots \vdash \top$. This functionality allows the simplifier to perform some propositional reasoning. In particular, conditional rewrite rules such as $P \longrightarrow x = y$ are, by default, used only if the antecedent P simplifies to $\top$.

In practice we have found that, despite the danger of rewriting with local equations, letting the simplifier use local hypotheses substantially increases Aesop's utility. We therefore enable this behaviour by default, but users can disable it for specific Aesop invocations.

3.3 Rule Indexing

So far we have been pretending that when a goal is expanded, we run all registered Aesop rules in order of priority. But Aesop is intended to be used with a large rule set, so this naive approach would be prohibitively slow. We therefore introduce a *rule index* which, given a goal G, efficiently determines a small subset of rules that may apply to G.

The index offers several *indexing schemes*. An indexing scheme determines, given a rule and a goal, whether the rule is potentially applicable to the goal. We currently implement three schemes:

- *Target*: the rule specifies a pattern expression T, which may contain holes. It is considered potentially applicable when the goal has the form $\Gamma \vdash U$ and U unifies with T. We use this scheme for `apply` rules.
- *Hypothesis*: the rule again specifies a pattern expression T. It is considered potentially applicable when the goal has the form $\Gamma,\ h : U,\ \Delta \vdash V$ and U unifies with T. We use this scheme for `cases` and `forward` rules. For `forward` rules, we take as the pattern T the last immediate argument of the rule, since later arguments are often more specific than earlier ones.
- *Disjunction*: the rule specifies a list of indexing schemes. It is considered potentially applicable when any of the schemes match the goal. We use disjunctive indexing for `constructors` rules (one by-target scheme for each constructor) and for `cases` rules with multiple patterns (one by-hypothesis scheme for each pattern).

The first two schemes are implemented by one discrimination tree each. A discrimination tree is a trie-like data structure that maps expressions T to arbitrary data (here: rules) and enables efficient retrieval of all values in the map whose key T may unify with a query expression U [16]. (In Lean 4, discrimination trees are also used to index typeclass instances and simplifier lemmas.) For the by-target scheme, we query the discrimination tree with the goal's target. For the by-hypothesis scheme, we query the discrimination tree once per hypothesis. The disjunction scheme is implemented by inserting the rule into the relevant discrimination trees multiple times with different keys.

Most rule builders have a natural indexing scheme. The exception is the `tactic` builder, which wraps arbitrary tactics. For `tactic` rules, users can specify a suitable indexing scheme themselves, if there is one.

When an indexed rule matches a goal, we communicate to the rule the set of *match locations*. Each match location is either the goal's target or a specific hypothesis. Using the match locations, a `cases` rule, for example, does not need to scan the hypotheses of the goal to find those of the right type. Instead, it can immediately focus on the hypotheses that were matched by its indexing scheme.

Like other Lean proof methods, notably the simplifier, our indexing schemes perform unification up to *reducible* computation. Each Lean definition is annotated with one of several *transparency modes*, which govern how eagerly the definition is unfolded during unification. Most definitions have `default` transparency and are not unfolded by the unification methods used by automation tactics; only those with `reducible` transparency are. Aesop's indexing follows this scheme. This, along with a convention that only non-recursive definitions are tagged as `reducible`, ensures that discrimination tree indexing does not miss any possible matches (with rare exceptions), but it also weakens certain rules. For example, a rule which proves the goal `a :: as = b :: bs` could also prove `[a] ++ as = [b] ++ bs` since the two goals unify once we unfold the list concatenation operator `++`. But `++` has `default` transparency, so our index does not unfold it and the rule is never tried on the second goal. To compensate, we could register a simplification rule which normalises `[a] ++ as` to `a :: as`.

3.4 Default Rules

Aesop's default rules perform uncontroversial reasoning steps, mostly pertaining to the logical connectives. Hypotheses $h : P \wedge Q$ are eliminated during normalisation, yielding separate hypotheses $h_1 : P$ and $h_2 : Q$, and similar for products $P \times Q$. Goals of the form $\Gamma \vdash P \wedge Q$ are reduced to $\Gamma \vdash P$ and $\Gamma \vdash Q$ by registering $\wedge$-introduction as a low-priority safe rule. For sum-like types such as disjunction, the respective elimination rule, which splits the goal into two subgoals, is safe with low priority. The respective introduction rules, which select one branch of the sum, are unsafe with 50% success probability.

Universally quantified goals $\Gamma \vdash \forall \vec{x} : \vec{T},\, P\,\vec{x}$ are normalised to $\Gamma,\, \vec{x} : \vec{T} \vdash P\,x$. When a goal with target $P\,\vec{x}$ contains a hypothesis $h : \forall \vec{y}, P\,\vec{y}$, h is applied as an unsafe rule. We give this rule 75% success probability, assuming that when a local hypothesis can be applied, it is usually a good idea to do so. In the special case where h has no premises, it is applied safely and proves the goal.

Existentially quantified hypotheses are split eagerly. For goals with an existentially quantified target, we register $\exists$-introduction, which creates a metavariable for the witness, as an unsafe rule. (See the next section for details on how we handle metavariables during the search.) It is important that this rule is unsafe because the goal's context determines which hypotheses can be used in the assignment of the witness metavariable. Thus, if we create this metavariable too eagerly, hypotheses which are added afterwards, e.g. by an unsafe `cases` rule, cannot be used in the metavariable's assignment.

Goals whose target is an equation $t = u$ are proved by reflexivity if t and u are already definitionally equal. Equational hypotheses $h : t = u$ are by default rewritten left-to-right during normalisation, as described in Sec. 3.2. In the special case where t is a local hypothesis, we substitute u for t everywhere in the goal and remove both t and the equation h. This is safe since t, having been removed from the goal, can never appear in a subgoal again, so the equation h has become superfluous. Symmetrically, if u is a local hypothesis, we substitute t for u and remove u and h.

Goals of the form $\Gamma \vdash P \leftrightarrow Q$ are split into subgoals $\Gamma \vdash P \rightarrow Q$ and $\Gamma \vdash Q \rightarrow P$. Hypotheses of type $P \leftrightarrow Q$ are treated like equalities $P = Q$ by appealing to propositional extensionality, an axiom which Lean uses pervasively.

The only default rule which does not pertain to logical connectives (apart from some rules for technicalities) is a low-priority safe case-splitting rule. If a goal's target contains an expression of the form `if t then ... else ...` or `match t with ...`, then this rule performs a case split on t, producing a simpler goal for each possible case. A similar rule applies to hypotheses containing `if` or `match` expressions, with even lower priority.

Designating so many default rules as safe can lead to unintuitive results. For example, as mentioned in Sec. 2.4, splitting a goal with target $P \wedge Q$ into goals with targets P and Q is unsafe if the rule set contains an unsafe rule which proves $P \wedge Q$, but not rules which prove P and Q. However, we believe it would be worse to make these rules unsafe, both for performance and because the printing of safe goals, which is an important debugging aid, becomes less useful if our safe rules are overly conservative.

4 Best-First Proof Search with Metavariables

We now extend the search algorithm to support goals containing metavariables. A *metavariable* (sometimes called schematic variable, existential variable or just free variable) is an expression which represents a typed term to be determined later. For instance, the goal $?m > 0 \wedge ?m < 3$, where $?m$ is a metavariable of type $\mathbb{N}$, can be proved if $?m$ is assigned the value 1 ($?m := 1$) or if $?m$ is assigned the value 2.

In interactive proofs, metavariables are created when we use a tactic without specifying all relevant information. A typical example is $\exists$-introduction, which reduces a goal $\exists w,\, P\,w$ to $P\,?w$, leaving the witness $?w$ to be determined later. Of course, we can also specify the witness up front, but using a metavariable can be convenient: perhaps we can reduce $P\,?w$ to $?w = 0$, in which case we can appeal to the reflexivity of equality to prove the goal, assigning $?w := 0$ as a side-effect.

Mirroring the interactive use of metavariables, Aesop allows rules like $\exists$-introduction to create and assign metavariables. This way of handling existential quantification is obviously incomplete since only witness terms induced by a subsequent rule application are considered. But it is also cheap, reasonably effective and familiar to users from their interactive proofs.

Another important class of rules which create metavariables are transitivity rules, which reduce a goal $x \leq z$ to subgoals $x \leq ?y$ and $?y \leq z$. These rules illustrate the main challenge of dealing with metavariables: they couple goals. A metavariable represents the same term everywhere it appears. So when we apply, say, reflexivity to the first subgoal $x \leq ?y$, we assign $?y := x$ as a side-effect and the second subgoal becomes $x \leq z$. *How* we prove the first subgoal now determines how, and indeed whether, we can prove the second.

This is a problem because our search procedure assumes that goals are independent. When we apply the reflexivity rule to $x \leq ?y$, we do not intend to commit to the resulting assignment $?y := x$ for the remainder of the search. We may, after all, have an assumption $x \leq a$ in the context which induces another instance of the second subgoal: $a \leq z$ with $?y := a$. And since we are doing best-first search, we may visit the second subgoal first and apply a rule which assigns $?y := b$, changing the first subgoal to $x \leq b$. Our search procedure should consider all these possibilities.

If we were to use a search strategy based on backtracking, such as depth-first search, this would be easy. We would merely have to ensure that when a rule application is backtracked, any metavariable assignments it has performed are erased. But for best- or breadth-first search, all assignments must be considered in parallel. So for the above example, the search tree must reflect the fact that we may prove any of the sets of goals $\{x \leq x, x \leq z\}$, $\{x \leq a, a \leq z\}$ and $\{x \leq b, b \leq z\}$. In the remainder of this section, we present an extension of our search algorithm which achieves just that.

4.1 Overview

To see the core issue with metavariables, suppose we have a rapp R with subgoals $G[?x]$ and $H[?x]$ that depend on $?x$. We say that G and H are *m-coupled* ('metavariable-coupled') since they share a metavariable $?x$ such that if G is proved for some assignment $?x := a$, then we must also prove $H[?x := a]$ (i.e. H with a substituted for $?x$) to get a proof of the parent rapp R. We can view $H[?x := a]$ as a "virtual subgoal" of the rule which proves G.

Our solution for this issue is simply to make the virtual subgoal an actual subgoal: when a rule S is applied to G and assigns $?x := a$, then $H[?x := a]$ is added as an additional subgoal of the S rapp. We call this additional subgoal an *m-copy* of H. Symmetrically, when a rule T is applied to H and assigns $?x := b$, then $G[?x := b]$ is added as an additional subgoal of T.

More generally, it is not only the siblings of G which may need copying. Suppose we first apply a rule to G which does not interact with $?x$ and produces a goal $G'[?x]$. We then apply R' to G', assigning $?x := a$. Then $H[?x := a]$ still needs to be copied even though it is not a sibling of G'. Accordingly, we expand our notion of m-coupled goals. Let $G_1, \ldots, G_n$ be the path from G' (so $G_1 = G'$) towards the root of the tree such that G_n is the first goal in which $?x$ appears. Each goal I which depends on $?x$ and which is a sibling of a goal G_i on the path is m-coupled to G' and is therefore copied.

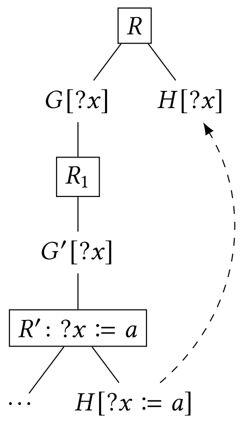

Figure 1. Copying of m-coupled nodes

Fig. 1 visualises this example, showing the incomplete search tree with root R. Here and in the next figure, rapp nodes are displayed as rectangles and are annotated with the metavariables they assign. Goal nodes are annotated with the metavariables they depend on, including the metavariables' assignments. Dashed arrows point from each copied goal to the goal it was copied from.

Once we perform copying, we must also modify our notion of when a goal is proved. Suppose we have three subgoals of a rule R: $G_1[?x]$, $G_2[?x, ?y]$ and $G_3[?y]$. If we prove G_3, then $?y$ must be assigned somewhere in this proof, say to $?y := a$. At this point, G_2 is copied since it also depends on $?y$, so the proof of G_3 contains a proof of the goal $G_2[?x, ?y := a]$. This proof, in turn, must assign $?x$, say to $?x := b$, at which point G_1 is copied, so the proof of G_3 also contains a proof of $G_1[?x := b]$. In general, any goal that is m-coupled to G_3 must already be included in a proof of G_3. To prove R, therefore, it suffices to prove G_3 (plus any other subgoals of R that are not m-coupled to G_3).

Fig. 2 visualises this example. Proved nodes are underlined. The dotted boxes around goals will become relevant shortly. We have added an additional subgoal of R, G_4, which is not m-coupled to G_3 and therefore needs to be proved separately. To keep the figure simple, each goal is proved by a single rule application with one subgoal, but in general, there could be an entire subtree between, say, G_3 and R_1. Moreover, the figure shows a proof attempt in which we happen to apply exactly those rules which lead to a proof. A less fortunate attempt would explore subtrees below the various goals before it finds the closing rapps R_3 and R_4.

Our modified definition of when a goal is proved relies on a crucial assumption: when we apply a rule R to a goal $G[?x]$, then either R must assign $?x$ or at least one of the subgoals

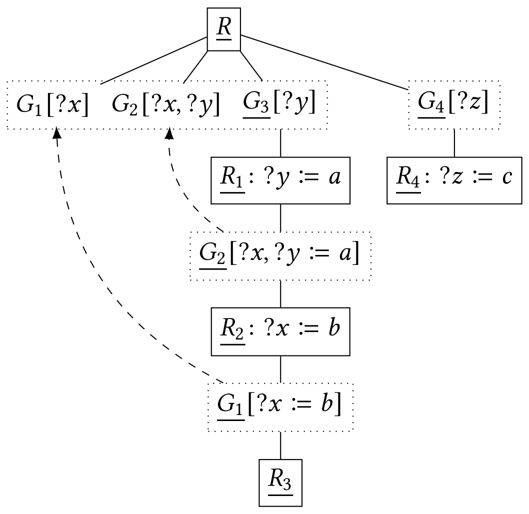
$\underline{R}$
$G_1[?x]$ $G_2[?x, ?y]$ $\underline{G_3}[?y]$
$\underline{G_4}[?z]$
$\underline{R_1}$: $?y := a$
$\underline{R_4}$: $?z := c$
$\underline{G_2}[?x, ?y := a]$
$\underline{R_2}$: $?x := b$
$\underline{G_1}[?x := b]$
$\underline{R_3}$

Figure 2. Proved nodes with copying

generated by R must also contain $?x$. Otherwise, we say that $?x$ has been *dropped*. If we were to allow dropped metavariables, a proof of $G[?x]$ would not necessarily have to assign $?x$ and an m-coupled sibling $H[?x]$ would not necessarily be proved. However, completely disallowing dropped metavariables turns out to be too strict for some applications, so we revisit this restriction in Sec. 4.6.

4.2 Search Tree

To support metavariables, we first augment the search tree to track some metavariable-related information. These data could also be computed on demand; we only cache them for efficiency.

In each goal node, we store the set of metavariables which the goal depends on. These are the metavariables which occur in the goal's hypotheses and in its target type. We assume for simplicity that assigned metavariables are immediately substituted everywhere, so only unassigned metavariables can occur in a goal. Additionally, the metavariables which occur in the goal may in turn depend on other metavariables since the type or context of a metavariable may contain other metavariables. We collect these recursively. The recursion terminates since cyclic dependencies between metavariables are not allowed. Obtaining the metavariables which occur in an expression is cheap since Lean optimises the common case in which an expression does not contain any metavariables.

In each rapp node, we store two additional pieces of information. First, we store the metavariables created by the rule application, which are those metavariables which the reported subgoals depend on and which the initial goal does not depend on. Second, we store the metavariables assigned by the rule application, which are those metavariables which the initial goal depends on and which are assigned after the rule has been run. We thus assume that rules do not assign metavariables which are not reachable from the initial goal, which is true for all (bug-free) Lean tactics.

We also need to keep track of which goals are m-coupled. This requires a more substantial augmentation of the search tree: we partition each rapp's set of subgoals $\{G_1, \ldots, G_n\}$ into *metavariable clusters*, which are, informally, sets of transitively m-coupled sibling goals. For example, suppose we have subgoals $G_1[?x]$, $G_2[?x, ?y]$, $G_3[?y]$ and $G_4[?z]$ as in Fig. 2. Then we partition these subgoals into metavariable clusters $\{G_1, G_2, G_3\}$ and $\{G_4\}$. Note that G_1 and G_3 do not have a metavariable in common, but they are still transitively m-coupled via G_2. In the figure, metavariable clusters are indicated by dotted boxes around sets of goals, but all clusters except for one are trivial, containing only one goal each.

Formally, for two goals G and H we write $G \sim H$ if there is a metavariable on which both G and H depend. We define $\approx$ as the transitive closure of $\sim$. Since $\sim$ is already reflexive and symmetric, $\approx$ is an equivalence relation. The metavariable clusters of a rapp are the equivalence classes of the rapp's subgoals with respect to $\approx$.

We can view metavariable clusters as a third type of node in the tree. The children of a rapp are then metavariable clusters; the children of a metavariable cluster are the goals contained in it; and the children of a goal are (still) rapps. This view leads to a natural generalisation of the node states:

- *proved*: as before, a goal node is proved if *at least one* of its child rapps is proved; a rapp node is proved if *all* its children are proved. But the children of a rapp are now metavariable clusters, and a metavariable cluster is proved if *at least one* of its goals is proved. This is motivated by the observation we made above: if we have a metavariable cluster with goals $\{G_1, \ldots, G_n\}$ and we prove some G_i, then all the G_j with $j \neq i$ must have been proved as part of the proof of G_i.
- *stuck*: as before, a goal node is stuck if *all* its child rapps are stuck and there are no more rules which could be applied to it; a rapp node is stuck if *at least one* of its children is stuck. A metavariable cluster is stuck if *all* its goals are stuck. This is because even if a goal $G[?x]$ is stuck, as long as some other goal H in the same metavariable cluster is non-stuck, it is still possible that the proof of H will discover a new assignment $?x := a$ and we can prove $G[?x := a]$.
- *unknown*: as before, a node is unknown if it is neither proved nor stuck.

The definition of irrelevance also remains unchanged: a node (which can now also be a metavariable cluster) is irrelevant if any of its ancestors, including the node itself, is proved or stuck.

When a search tree contains no metavariables, each goal is only m-coupled to itself, so there is one metavariable cluster per goal. The metavariable-free version of our algorithm from Sec. 2 then emerges as a special case of the metavariable-encumbered one.

4.3 Copying

The search procedure with metavariables is largely the same as without metavariables. The only change is that when we add a rapp which assigns a metavariable, we must copy the m-coupled goals.

To see how, suppose we are adding a rapp R with parent goal $G[?x_1, \ldots, ?x_n, ?\vec{y}]$ such that R assigns $?x_i \coloneqq a_i$ for $1 \leq i \leq n$. Aesop then walks the path from G up the tree towards the root goal, stopping at the topmost rapp node R_m that creates any of the metavariables $?x_i$. (Thus the $?x_i$ can only appear in the subtree below R_m.) Let this path be $G_1, R_1, G_2, R_2, \ldots, G_m, R_m$, where $G_1 = G$ and for each i, R_i is the parent rapp of G_i and G_{i+1} is the parent goal of R_i. Aesop then copies every sibling H of the G_i which depends on any of the $?x_j$, adding $H[?x_1 \coloneqq a_1, \ldots, ?x_n \coloneqq a_n]$ as an additional subgoal of R.

However, there are two special cases in which it is not useful to copy a sibling goal H. First, H may be a copy of one of the G_i on the path. This means we are already in the subtree that will serve as a proof of G_i, so adding H as a subgoal would be pointless. Second, we may discover multiple goals $H_1, \ldots, H_k$ which are copies of the same original goal. In this case, we only need to copy one of them.

Note that we copy only the sibling goals themselves and not their subtrees. This means that any rules which were applied to the sibling goals must be re-applied to their copies. In general, this is necessary because the copied goals have different types and hypotheses (on account of the metavariable substitution we applied to them), so re-applying the rules may yield different results. But it is still somewhat inefficient. We discuss a possible solution to this issue in Sec. 4.7.

4.4 Interaction with Safe Rules

Most safe rules become unsafe if they assign metavariables. This applies even to such unassuming rules as proof by assumption. Suppose we have goals $h_1 : \alpha, h_2 : \beta \vdash ?x$ and $\vdash \beta \longrightarrow ?x$. If we prove the first goal via h_1, the second goal may well become unprovable. If we use h_2 instead, the second goal is trivial. So proof by assumption does not preserve provability in the presence of metavariables.

Accordingly, Aesop treats any safe rule that assigns a metavariable as unsafe. This means that when we expand a goal G, we first run the safe rules applicable to G, as usual. Whenever one of these rules assigns at least one metavariable, we do not add the rule to the search tree. Instead, we treat the rule as failed but store its result (subgoals and some metadata) in a list of *postponed* safe rapps. We then continue to apply the remaining safe rules. If one of them succeeds without assigning metavariables, we apply it directly and throw away the postponed rapps. Otherwise — if all safe rules either fail or assign metavariables — we apply the unsafe rules as usual, but we also add the postponed rapps as unsafe rules with success probability 90%. When Aesop selects a postponed rapp to be applied as an unsafe rule, it does not re-execute the rule but simply adds its stored result to the search tree.

In principle, one could imagine situations in which a safe rule assigns metavariables in a safe manner and thus does not need to become unsafe. But in practice, we have yet to encounter such a situation.

4.5 Interaction with Normalisation Rules

For normalisation rules, we need to restrict metavariable assignments even more than for safe rules. Since normalisation rules must also be safe, we have the same issue as with safe rules. Additionally, normalisation rules are applied in a fixpoint loop, so there is no natural way to postpone a normalisation rule. So we simply forbid normalisation rules from creating or assigning metavariables.

This restriction is, for the most part, unobtrusive in practice, with one unfortunate exception. Our implementation of `cases` rules uses Lean's built-in `cases` tactic to perform case analysis. When a goal contains a metavariable, `cases` may replace this metavariable with a new one, which to Aesop looks as if an existing metavariable had been assigned and a new one created. We have not found a reliable way to detect this situation, so we currently do not allow `cases` normalisation rules (which could otherwise be used to, for example, split a hypothesis $h : A \land B$ into $h_1 : A$ and $h_2 : B$).

4.6 Synthesis of Dropped Metavariables

Recall that when a rule R is applied to a goal $G[?x]$, there must be at least one subgoal of R which depends on $?x$. If this is not the case, we say that $?x$ has been dropped, and so far we have disallowed dropped metavariables.

However, this restriction turns out to be too harsh. One application — Jesse Vogel's Duck tool[3], which aims to use Aesop to find examples of structures with certain properties in algebraic geometry — provided this trivial test goal: under the assumption that the integers form a ring and that every ring R has a ring automorphism $\mathtt{id} : \forall R : \mathtt{Ring},\ \mathtt{RingHom}\, R\, R$, show

$$\exists R : \mathtt{Ring},\ \mathtt{RingHom}\, R\, R.$$

To prove this goal, Aesop first applies $\exists$-introduction, obtaining the goal $\mathtt{RingHom}\ ?R\ ?R$. It then tries to apply `id`, which proves the goal without assigning $?R$, so $?R$ is dropped. Since this is forbidden, the application of `id` fails and the goal cannot be proved.

To address this obvious deficiency, we must allow dropped metavariables. But at the same time, we must take care not to violate the conditions that led us to disallow them in the first place:

[3] https://github.com/jessetvogel/duck

- Dropped metavariables must be assigned eventually. This is necessary in dependent type theory since the type of a metavariable could be uninhabited. An unassigned metavariable of type T corresponds to an assumption that we can inhabit T. (The situation is different for logics in which all types are inhabited, such as the logic of Isabelle/HOL.)
- When we prove a goal $G[?x]$ and drop $?x$ in the process, we must ensure that related goals containing $?x$ are proved as well.

To address the first requirement, when a rule R is applied to a goal $G[?x]$ and drops $?x$, we add an additional subgoal to R which corresponds to $?x$. In our ring example, applying `id` proves the goal `RingHom` $?R\,?R$, but since $?R$: `Ring` is dropped, we add an additional subgoal of type `Ring`, which is then proved by assumption.

To address the second requirement, we modify the procedure for copying metavariable-related goals such that it treats dropped metavariables as assigned for the purposes of copying. So if, in our example, we had an m-coupled goal, say `RingHom` $?R\,\mathbb{Z}$, then the `id` application would copy this goal as an additional subgoal. Thus, the proof of the original goal, `RingHom` $?R\,?R$, again contains proofs for all m-coupled goals.

Importantly, whether a dropped metavariable appears in the subgoals of a rapp — and therefore whether a subgoal for it is added — is determined after copying. This ensures that a subgoal for a metavariable $?x$ is only created once we can no longer obtain an assignment from the proof of any goal in this branch of the search tree. If this were not the case, we could end up with a solution a for the subgoal $?x$ which is different from an assignment $?x := b$ performed by a later rapp.

4.7 Discussion

Our algorithm is conceptually attractive for two reasons. First, it is a strict and fairly simple generalisation of the algorithm without metavariables. Second, it is very general: it works for any search strategy and makes almost no assumptions about how rules interact with metavariables. We only require that rules limit their assignments to metavariables appearing in the goals to which the rules are applied.

The downside of this generality is some inefficiency. In particular, as mentioned in Sec. 4.3, our algorithm treats goals which are m-copies of each other as entirely independent, so a rule applied to one has no effect on the others. For an example of how this leads to inefficiency, suppose the goal $h : n < ?x \vdash A$ appears in the search tree. Then, during the search, we likely create a number of m-copies of this goal with different instantiations for $?x$. Now suppose we have a rule $R : B \longrightarrow A$. When this rule is applied to one m-copy of the goal, we could recognise that R is independent of the instantiation of $?x$ and therefore applies to every m-copy. As it stands, our algorithm does not take advantage of this optimisation opportunity. However, the optimisation is also valid only for certain rules. A rule which searches for contradictory hypotheses $n < 0$ (where n is a natural number) is not independent of the instantiation of $?x$ and therefore cannot be shared between m-copies of the goal.

We believe that despite its generality, our algorithm is as complete as possible, in the following sense. Suppose we use a fair search strategy, i.e. one which guarantees that every rule will eventually be applied to every goal. Now take a goal G that can be proved by applying a sequence of rules from the rule set, creating and assigning arbitrary metavariables in the process. Then, we conjecture, our algorithm will also find a proof of G. Intuitively, this is because our algorithm only adds to the search tree, so it is not possible to apply a rule in such a way that another rule cannot be applied any more. Thus, since we assume a fair search strategy, each rule in the proving sequence of rules is applied eventually (unless the goal to which it would be applied is already proved). We plan to prove this conjecture in future work.

5 Case Studies

As evidence that Aesop provides a reasonable level of automation, we present two case studies: one in which we prove a variety of basic theorems about lists and one in which we formalise a simple automated theorem prover for intuitionistic propositional logic. Both case studies are available in the supplement to this paper.[4]

Ideally, we would evaluate Aesop on a standardised benchmark such as the TPTP problem set [21]. But this is conceptually difficult: without an extensive rule set, Aesop is not expected to prove many theorems, and with an extensive rule set, we could game many benchmark problems by providing just the right rules. Perhaps as a result, there is currently no standard benchmark for white-box proof search tools.

5.1 Lists

As a first test of Aesop, we port some lemmas about lists from Lean 3 to Lean 4. We consider a file from mathlib, `data/list/basic.lean`. This file contains a large number of lemmas about basic list functions such as `length`, `append` and `reverse`, and about predicates such as subset and membership. We take the first 200 of these lemmas and port them to Lean 4.

Of the 200 lemmas, we exclude 16 which merely state definitional equations. (Such lemmas are used to register definitional equations with the simplifier.) For lemmas which reference notations or concepts that are not available in Lean 4, we either add the necessary definitions or, in 11 cases, exclude the lemma from our case study. Some of the remaining 173 lemmas are already proved in Lean 4, in which case we

[4]https://doi.org/10.5281/zenodo.7424818

re-prove them. If these lemmas are registered as global simplifier rules, we remove them first; otherwise Aesop's job would be a bit too easy.

Whenever we prove a lemma which makes a good global Aesop rule, we add the lemma to the global Aesop rule set. We also add a small number of lemmas about other concepts (injective/surjective/involutive functions and the `Option` type) which could sensibly be included in a library-wide Aesop rule set.

With this setup, Aesop proves 109 (63%) of the list lemmas outright. If we manually perform induction where necessary (which Aesop by design does not do), Aesop proves 163 (94%) of the lemmas. Specifically, for lemmas which require induction, we either add one or more calls to the `induction` tactic (after possibly unfolding some definitions and introducing hypotheses) or we write the lemma as a `match` statement, use a recursive call to prove the induction hypotheses and let Aesop do the rest. The latter is the most ergonomic way to perform functional induction in Lean.

Of the 10 lemmas Aesop cannot prove, 4 involve existentially quantified statements with non-trivial witnesses, e.g.

```
∀ (a : α) (l : List α), a ∈ l →
  ∃ (s t : List α), l = s ++ a :: t
```

Aesop's quantifier instantiation method, which relies solely on unification, is too weak to determine the proper witnesses for each case of the induction. The other 6 unsolved lemmas fail either because a lemma is missing from the library (2) or because Aesop's rule set misfires in specific situations (4).

Of the 163 lemmas Aesop (plus induction) can prove, 48 (29%) require local rules; the rest are solved using only global rules. By far the most common local rule, with 25 occurrences, is a low-priority unsafe rule which performs a case split on hypotheses of type `List`. Since each such case split produces another hypothesis of type `List`, this rule can loop, so it is not suitable as a global rule. But for lemmas which require such a case split, we can add the rule and due to its low priority, Aesop applies it only as a last resort. This makes sure that if a proof is found, it is found quickly.

Another notable local adjustment involves Aesop's simplifier integration. As we discussed in Sec. 3.2, Aesop by default rewrites with equations in the local context. This can be dangerous because such equations are not necessarily properly oriented. For example, a hypothesis of type $n = n + 0$ would, together with the global rule $n + 0 = n$, send the simplifier into a loop. In our case study, this happens two times; in both cases, Aesop succeeds once we disallow the use of hypotheses during simplification.

To get a broad idea of how fast Aesop is, we also ran a small benchmark involving this case study. For the benchmark, we prepared a version of the case study in which all lemmas are proved by hand. The proofs are written in the runtime-efficient style of mathlib (most proofs are translated from Lean 3), meaning they involve no expensive tactics except the simplifier, which moreover is always given the exact set of lemmas it should simplify with. Thus, we believe that this hand-written version of the case study has close to optimal performance. We then compared the total time Lean takes to typecheck the hand-written version and the Aesop version of the case study, averaging over 10 runs each. On one particular machine, the hand-written version took on average 2.48 seconds to typecheck (min = 2.46, max = 2.50, σ = 0.015) whereas the Aesop version took 6.25 seconds (min = 6.17, max = 6.32, σ = 0.045). This means delegating all proofs to Aesop resulted in a slowdown of 2.53x. When run on other machines, the benchmark yielded slowdowns of 2.59x and 2.60x.

It is perhaps not surprising that Aesop is fairly successful in this case study: most of the lemmas we consider are very simple. But automating trivial goals about basic data structures is still an important part of making interactive theorem provers less onerous to use. And many lemmas which are straightforward consequences of facts known to Lean would not have had to be written if Aesop had been available at the time.

5.2 Propositional Sequent Calculus Prover

As a second test of Aesop, we have programmed a small prover for propositional logic [22] in Lean 4 and used Aesop to verify the soundness and completeness of both the prover and the sequent calculus proof system it is based on.

To that end, we first define the type `Form` of propositional formulas. Given an interpretation `i` of propositional variables Φ, the predicate `Val : Form Φ → Prop` gives the truth value of a formula. Aesop proves, after manual induction over `Form`, that if `i` is decidable, then so is `Val`.

Satisfiability of formulas extends to satisfiability of sequents: a sequent with premises Γ and conclusions Δ is valid if, whenever all premises are true, at least one conclusion is true. Formally, `All (Val i) Γ` implies `Any (Val i) Δ`. We saw `All` earlier; `Any` is similar but encodes the fact that *some* element in the list satisfies the predicate. The `cases` rule for `Any` makes good use of patterns: case analysis on a hypothesis which matches the pattern `Any _ []` is safe since the hypothesis is contradictory; case analysis on a hypothesis which matches `Any _ (_ :: _)` is unsafe but often useful.

With the help of this `cases` rule, we prove some fundamental lemmas about `All` and `Any`, such as a weakening lemma for `All`:

```
(∀ x, P x → Q x) → All P xs → All Q xs
```

After induction on the `All` premise, Aesop finishes the proof. We use this weakening lemma to prove that all elements of a list are members of that list: `All (· ∈ xs) xs`. The application of weakening requires support for metavariables since `P` is unconstrained and becomes a metavariable. Similarly, metavariables are crucial when proving existentially quantified lemmas, e.g.

```
Any P xs ↔ ∃ a : α, P a ∧ a ∈ xs
```

The prover itself, `Cal`, attempts to prove a sequent by breaking down connectives according to the classical sequent calculus rules and collecting lists of positive and negative propositional variables when they appear on either side of the sequent. A branch of the proof terminates successfully when the same variable occurs both positively and negatively, corresponding to the usual *Axiom* rule. We use `Any (· ∈ ys) xs` to check if two lists `xs` and `ys` share a common element. The computational behaviour of `Cal` thus depends on the decidability of `Any`, which is proved using Aesop. We verify soundness and completeness of the prover simultaneously, using induction on the call structure of `Cal`. The main theorem states that `Cal` proves a sequent if and only if the sequent is valid for all decidable interpretations.

Since the prover rearranges formulas, the proof relies on the fact that `All` and `Any` respect list permutations, as encoded by an inductive predicate taken from the Agda standard library.[5] Here, Aesop significantly reduces our workload: that permutations are symmetric, that they are preserved by `map` and that `All` and `Any` respect permutations can be proven automatically after we perform induction.

As a consequence of the soundness and completeness of `Cal`, we additionally obtain soundness and completeness of its underlying proof system, formulated as an inductive predicate `Proof Γ Δ`. A key ingredient of the proof is this weakening lemma:

```
Proof Γ Δ → Proof Γ (δ :: Δ)
```

After induction on the premise, Aesop proves the lemma automatically, apart from one case which requires an explicit application of the induction hypothesis. This is because two of the constructors of `Proof` allow us to apply arbitrary permutations to the sequents, which Aesop's metavariable handling is too weak to find. These constructors also apply to every goal, so it is important that Aesop is not limited to depth-first search, which might get lost in infinite permutations.

6 Related Work

The closest relative of Aesop is Isabelle's `auto` [18, 20]. Like Aesop, it performs a tree-based search with integrated simplification and it distinguishes between safe and unsafe rules. Aesop adds a best-first strategy (`auto` is depth-first) and normalisation as a separate phase. It also adds a number of rule builders apart from `auto`'s `intro`, `elim` and `destruct` rules, though some of these can be emulated with auxiliary Isabelle tools.

More fundamentally, `auto` is used as a semi-black-box tool in practice. It is essentially undocumented, so it is difficult to understand the details of its search procedure, e.g. how exactly the simplifier is invoked, how it integrates `blast` [19] (a tableau prover) and how metavariables interact with safe rules. Indeed, our conversations with experienced Isabelle users indicate that they are unaware of these details and that as a result, their interactions with `auto` are partly based on trial and error, adding and removing rules until `auto` is able to prove a goal.

Other semi-black-box proof tools include PVS's `grind` [5] and the 'waterfalls' of ACL2 [11] and its descendants. While these tools are based on simple search algorithms and are extensively documented, they use, at least in their default configurations, a large number of proof methods (e.g. several forms of simplification; decision procedures for certain fragments of the logic; several methods for quantifier instantiation) in a fixpoint loop surrounded by pre- and post-processing steps. As a result, it again becomes somewhat difficult for users to predict and adjust their behaviour.

Aesop, by contrast, attempts to remain firmly white-box by limiting itself to a small number of simple concepts (essentially: normalisation, safe and unsafe rules) with no opaque heuristics and no pre- or post-processing. This should make it possible to design predictable special-purpose rule sets for specific domains. With larger rule sets, Aesop may also become somewhat unpredictable, but at least its transparency should make it easier to debug unexpected failures or performance issues. Of course, the downside of Aesop's simplicity is that it is considerably less powerful than, say, `grind`; for example, it does not currently have any support for arithmetic beyond that provided by Lean's simplifier.

Even farther towards the white-box end of the scale lie Coq's `auto` and `eauto`. These tactics perform backtracking depth-first search (up to some configurable depth limit) with arbitrary rules, so they are essentially Aesop without safe or normalisation rules and with a different search strategy. Matita's `auto` [1] augments `eauto` with a superposition calculus for equational reasoning and provides a GUI which allows users to inspect the search tree.

A rare white-box tool not based on tree search is Isabelle's `auto2` [23], which uses a saturation algorithm instead. This means that rules can be applied without backtracking, but the proof procedure is also farther removed from interactive proof and therefore perhaps less easy to customise.

There are also black-box tools based on tree search, notably Coq's `sauto` [6] and the Agsy tool [14] for Agda [17]. These tools use fairly strong default rules, some of which could also be interesting for Aesop. But since they are push-button tools, their rules are also quite opaque.

For Lean 3, mathlib [4] already contains some search tactics which are currently being ported to Lean 4: `continuity`,

[5] https://github.com/agda/agda-stdlib/blob/ebfb8814b4330b314da8fb9cae527e6a6fab01aa/src/Data/List/Relation/Binary/Permutation/Propositional.agda

`measurability`, `tidy`, `tautology` and `finish`. These tactics perform essentially depth-first search with various rule sets, so Aesop should supersede them. However, `finish` uses e-matching and so makes better use of unoriented equations.

Our handling of metavariables is most closely related to that of TH∃OREM∀ [12], which, like Aesop, uses an AND/OR search tree. Variations of the TH∃OREM∀ algorithm are also used for tableaux with metavariables ('free variable tableaux') [9]. However, these algorithms are specific to first-order logic and do not obviously generalise to our setting. In particular, they require that rules behave uniformly for different metavariable assignments.

7 Conclusion

We have presented Aesop, a white-box proof search tactic for Lean. Starting with a straightforward tree search framework, we have added features that increase the power of the search while keeping its semantics simple and transparent: best-first search with customisable prioritisation, which lets us effectively use rules that are only occasionally useful or that may loop; safe rules, which are useful both for performance and for debugging; normalisation, to establish invariants which other rules can rely on; and simplification, which enables equational reasoning. Taken together, these features should allow users to design effective and predictable rule sets.

To support goals with metavariables, we have developed a generic algorithm for tree-based search with metavariables. The algorithm is independent of the search strategy and of the underlying logic and is, we believe, as complete as the given rule set allows.

Acknowledgments

We thank Jasmin Blanchette for advising us throughout the development process and giving detailed comments on drafts of this paper; Sebastian Ullrich for providing excellent feedback on early versions of Aesop; Gabriel Ebner for helping with Aesop's implementation; and the anonymous reviewers for providing comprehensive and insightful reviews.

Limperg was funded by the NWO under the Vidi programme (project No. 016.Vidi.189.037, Lean Forward).

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2011. The Matita interactive theorem prover. In *CADE 2011*. 64–69. https://doi.org/10.1007/978-3-642-22438-6_7

[2] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: an environment for machine learning of higher order logic theorem proving. In *ICML 2019*. 454–463. https://proceedings.mlr.press/v97/bansal19a.html

[3] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *Journal of Formalized Reasoning* 9, 1 (2016), 101–148. https://doi.org/10.6092/issn.1972-5787/4593

[4] The mathlib Community. 2020. The lean mathematical library. In *CPP 2020*. 367–381. https://doi.org/10.1145/3372885.3373824

[5] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. 1995. A Tutorial Introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques 1995*. http://www.csl.sri.com/papers/wift-tutorial/

[6] Łukasz Czajka. 2020. Practical proof search for Coq by type inhabitation. In *IJCAR 2020*. 28–57. https://doi.org/10.1007/978-3-030-51054-1_3

[7] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 theorem prover and programming language. In *CADE 2021*. 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

[8] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. 2021. TacticToe: Learning to prove with tactics. *Journal of Automated Reasoning* 65, 2 (2021), 257–286. https://doi.org/10.1007/s10817-020-09580-x

[9] Martin Giese. 2001. Incremental closure of free variable tableaux. In *IJCAR 2001*. 545–560. https://doi.org/10.1007/3-540-45744-5_46

[10] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. 2021. Proof artifact co-training for theorem proving with language models. https://doi.org/10.48550/ARXIV.2102.06203

[11] M. Kaufmann and J. Strother Moore. 1996. ACL2: an industrial strength version of Nqthm. In *COMPASS 1996*. 23–34. https://doi.org/10.1109/CMPASS.1996.507872

[12] Boris Konev and Tudor Jebelean. 2005. Solution lifting method for handling meta-variables in TH∃OREM∀. *Journal of Mathematical Sciences* 126, 3 (2005), 1182–1194. https://doi.org/10.1007/s10958-005-0090-6

[13] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. 2022. HyperTree proof search for neural theorem proving. https://doi.org/10.48550/ARXIV.2205.11491

[14] Fredrik Lindblad and Marcin Benke. 2004. A tool for automated theorem proving in Agda. In *TYPES 2004*. 154–169. https://doi.org/10.1007/11617990_10

[15] Conor McBride and James McKinna. 2004. The view from the left. *Journal of Functional Programming* 14, 1 (2004), 69–111. https://doi.org/10.1017/S0956796803004829

[16] William McCune. 1992. Experiments with discrimination-tree indexing and path indexing for term retrieval. *Journal of Automated Reasoning* 9, 2 (1992), 147–167. https://doi.org/10.1007/BF00245458

[17] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers University of Technology, Göteborg, Sweden. https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

[18] Lawrence C. Paulson. 1996. *Generic automatic proof tools*. Technical Report UCAM-CL-TR-396. University of Cambridge. https://doi.org/10.48456/tr-396

[19] Lawrence C. Paulson. 1999. A generic tableau prover and its integration with Isabelle. *Journal of Universal Computer Science* 5, 3 (1999), 73–87. https://doi.org/10.3217/jucs-005-03-0073

[20] Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. 2019. From LCF to Isabelle/HOL. *Formal Aspects of Computing* 31, 6 (2019), 675–698. https://doi.org/10.1007/s00165-019-00492-1

[21] Geoff Sutcliffe. 2017. The TPTP problem library and associated infrastructure: from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning* 59, 4 (2017), 483–502. https://doi.org/10.1007/s10817-017-9407-7

[22] Jørgen Villadsen. 2020. A micro prover for teaching automated reasoning. In *PAAR 2020*.

[23] Bohua Zhan. 2016. AUTO2, a saturation-based heuristic prover for higher-order logic. In *ITP 2016*. 441–456. https://doi.org/10.1007/978-3-319-43144-4_27

Received 2022-09-21; accepted 2022-11-21