Aesop: White-Box Automation for Lean 4

Jannis Limperg
University of Munich (LMU)
jannis@limperg.de

BICMR and Istituto Grothendieck (online), 18. April 2024

Search Algorithm

Registering Rules

Built-In Rules

Debugging

Miscellaneous Features

Applications, Shortcomings and Work In Progress

Search Algorithm

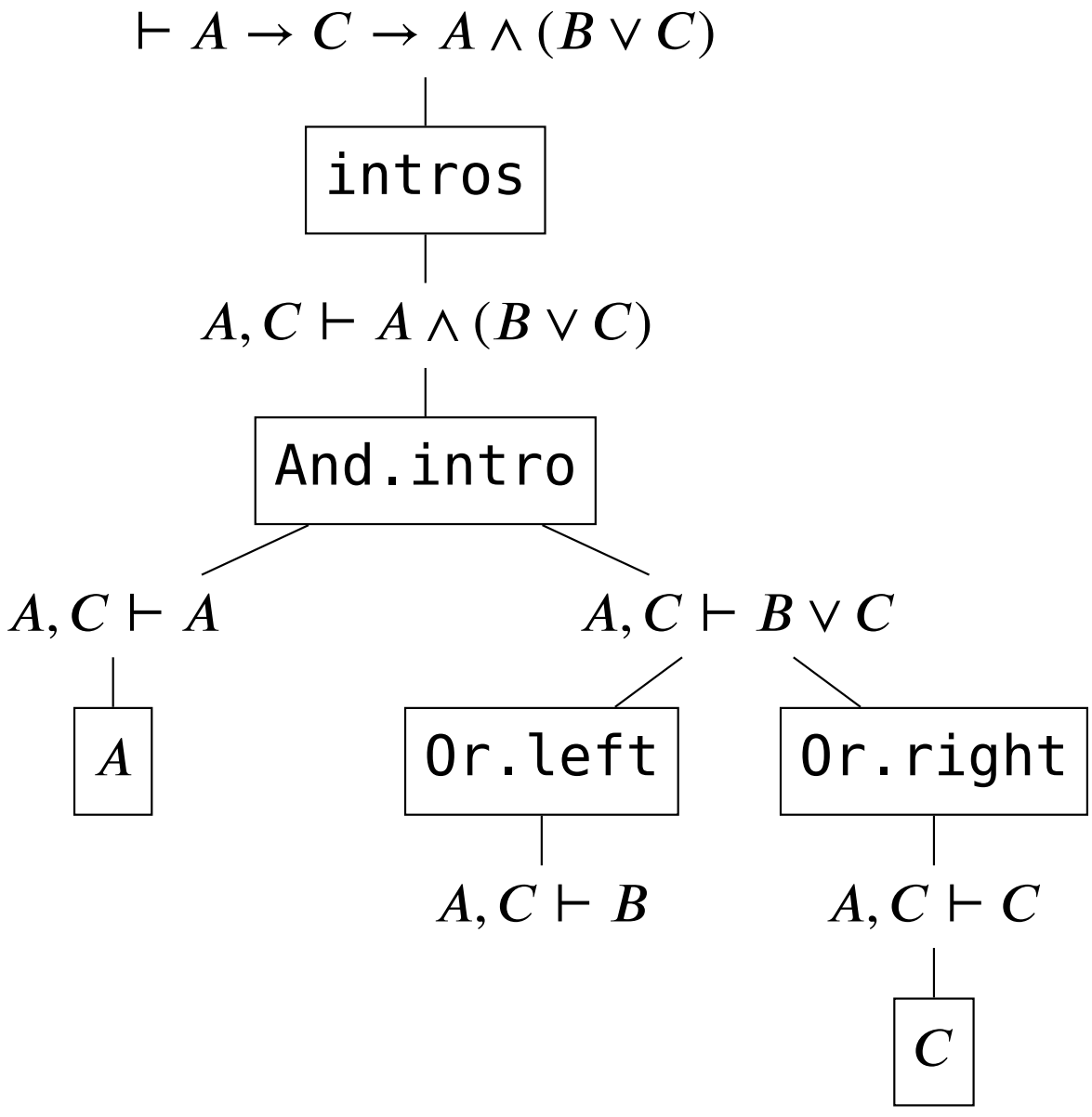
Rules

A *rule* is an arbitrary Lean tactic.

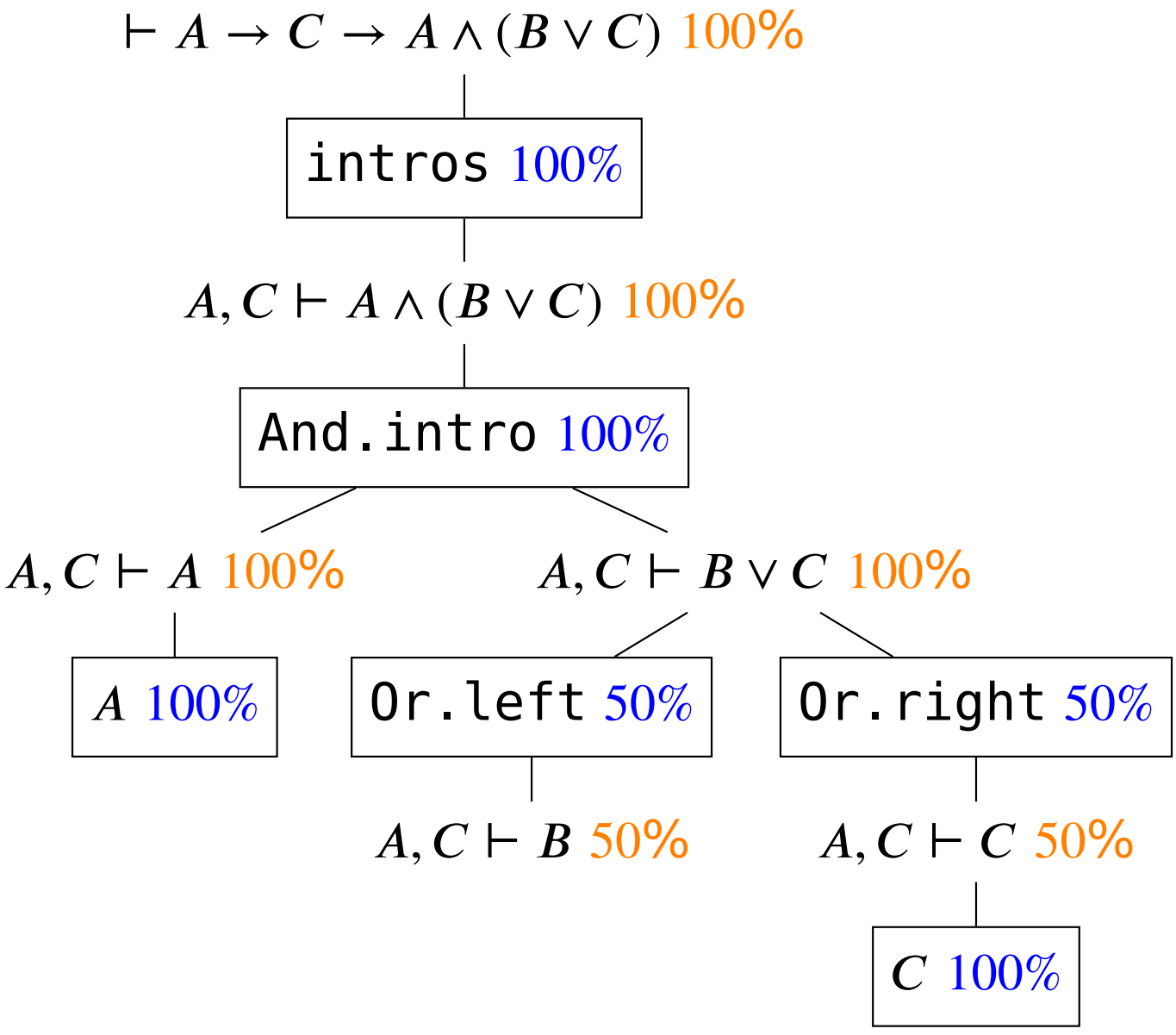
Aesop provides convenient syntax (*rule builders*) for creating rules from theorems.

Aesop always operates with a user-defined *rule set*.

Basic Tree Search



Best-First Tree Search



Safe Rules

- Run before unsafe rules
- If a safe rule succeeds on a goal G, no other rules are tried for G
- Integer penalty
- Treated as 100% success probability
- 🥳 Good for performance
- 😐 Users need to make sure that the rule really is safe

Examples

Safe rule: ∧-introduction

$$\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

Unsafe rule: left ∨-introduction

$$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

When Is A Rule Safe?

A rule R is *logically safe* if it preserves provability:

For each goal G, if G is provable and R, applied to G, generates subgoals $G_1, \ldots, G_n$, then $G_1, \ldots, G_n$ must still be provable.

A rule R is *relatively safe* if it preserves provability relative to a rule set S:

If a goal G is provable with rules from S and R, applied to G, generates subgoals $G_1, \ldots, G_n$, then $G_1, \ldots, G_n$ must still be provable with rules from S.

Normalisation Rules

- Run before safe rules
- Integer penalty
- Treated as 100% success probability
- May produce only one subgoal
- Run in a fixpoint loop, i.e. until no normalisation rule succeeds any more
- 🥳 Can establish invariants for other rules
- 😐 Typically run multiple times on every goal

Example

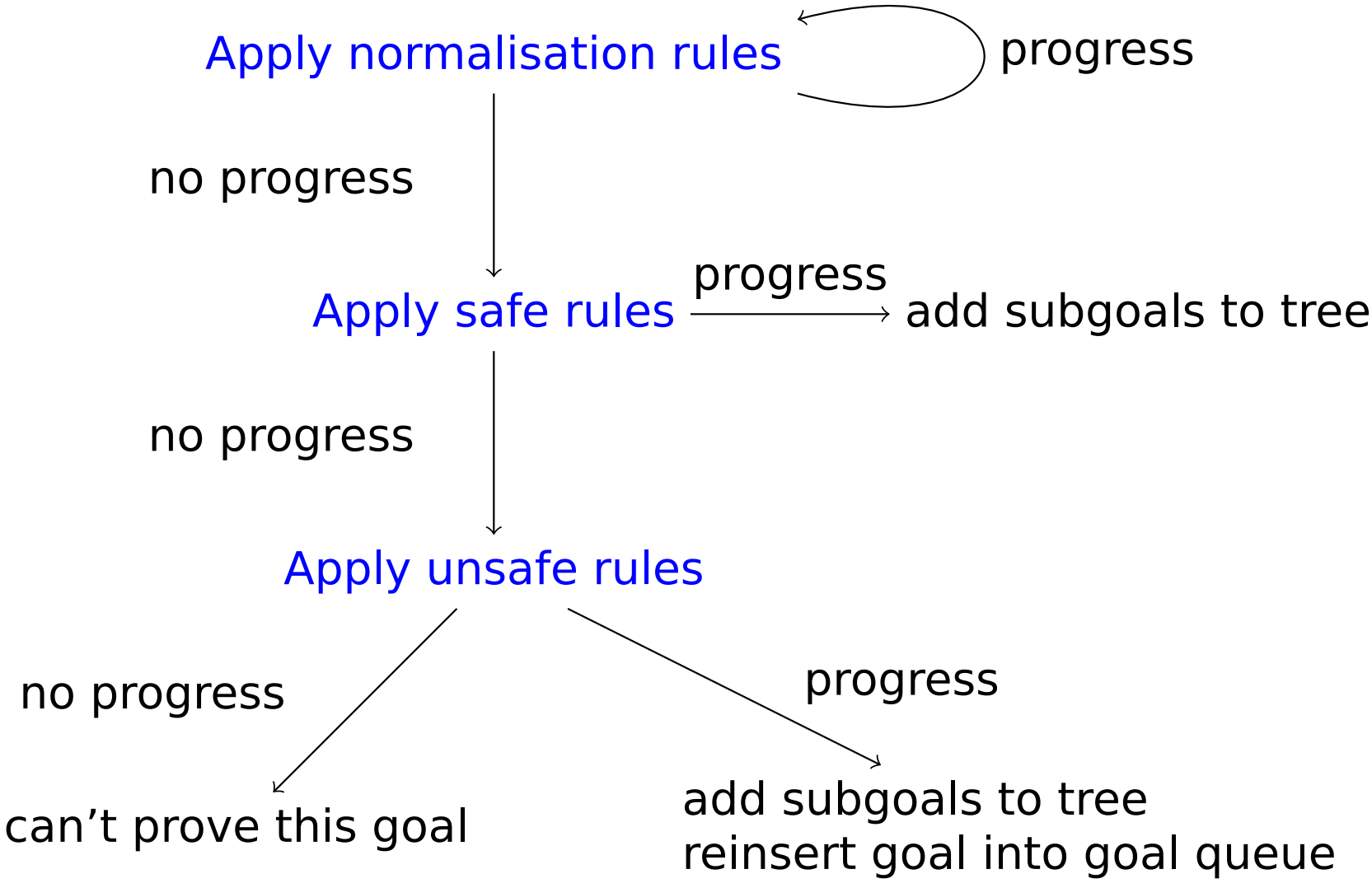
∧-elimination

$$\Gamma, h : A \wedge B \vdash T$$

$$|$$

$$\Gamma, h_1 : A, h_2 : B \vdash T$$

Summary: Aesop's Search Algorithm



Registering Rules

Registering Rules

Globally

```
@[aesop unsafe 100%]
theorem And.intro : A → B → A ∧ B
```

Locally

```
aesop (add 100% And.intro)
```

Safe rules

```
@[aesop safe 10]
theorem And.intro : A → B → A ∧ B
```

Rule Builders

A *rule builder* turns a declaration into an Aesop rule.

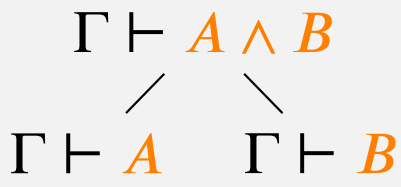
In the examples so far, we have implicitly used a default builder.

Aesop currently provides 7 rule builders.

Apply Builder

```
@[aesop safe apply 10]
theorem And.intro : A → B → A ∧ B
```

Builds a rule that runs **apply** And.**intro**.



Constructors Builder

```
@[aesop 50% constructors]
inductive Or (A B : Prop) where
  | left  : A → Or A B
  | right : B → Or A B
```

Equivalent to one **apply** rule for each constructor.

Cases Builder

```
@[aesop safe cases]
inductive Or (A B : Prop) where
  | left  : A → Or A B
  | right : B → Or A B
```

Builds a rule that runs **`cases`** on any hypothesis of type `Or A B`.

Forward Builder

```
@[aesop safe forward]
theorem pos_of_min_pos : ∀ {x y : ℕ},
  0 < min x y →
  0 < x ∧ 0 < y
```

$$\Gamma,\ x\ y : \mathbb{N},\ h : 0 < \min\ x\ y \vdash T$$

|

$$\Gamma,\ x\ y : \mathbb{N},\ h : 0 < \min\ x\ y,\ h_1 : 0 < x \wedge 0 < y \vdash T$$

Destruct Builder

```
@[aesop safe destruct]
theorem pos_of_min_pos : ∀ {x y : ℕ},
  0 < min x y →
  0 < x ∧ 0 < y
```

$$\Gamma,\ x\ y : \mathbb{N},\ h : 0 < \min\ x\ y \vdash T$$

$$\Gamma,\ x\ y : \mathbb{N},\ h : 0 < x \wedge 0 < y \vdash T$$

Simp Builder

Aesop runs **`simp_all`** as a built-in normalisation rule with penalty 0.

This **`simp_all`** call uses the default **`simp`** set plus an Aesop-specific **`simp`** set.

The **`simp`** builder adds an equation or proposition to this Aesop-specific set.

Tactic Builder

```
aesop (add safe (by norm_num; done))
```

```
add_aesop_rules safe (by norm_num; done)
```

Requirement

If the tactic does not change the goal, it should fail.

Built-In Rules

Logic: ∧

∧-introduction (safe, penalty 101)

$$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A \quad \Gamma \vdash B}$$

∧-elimination (norm, penalty 0)

$$\dfrac{\Gamma,\ h : A \wedge B \vdash T}{\Gamma,\ h_1 : A,\ h_2 : B \vdash T}$$

Similar for `Prod`, `PProd`, `MProd`

Logic: ∨

∨-introduction (unsafe, success probability 50%)

$$\frac{\Gamma \vdash A \vee B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \vee B}{\Gamma \vdash B}$$

∨-elimination (safe, penalty 100)

$$\frac{\Gamma,\ h : A \vee B \vdash T}{\Gamma,\ h : A \vdash T \qquad \Gamma,\ h : B \vdash T}$$

Similar for `Sum`, `PSum`

Logic: ∀ and →

∀-introduction (norm, penalty -100)

Run the `intros` tactic

∀-elimination (unsafe, success probability 75%)

$$\frac{\Gamma,\ h : \forall (x_1 : A_1) \ldots (x_n : A_n),\ B \vdash B}{\Gamma,\ h \vdash A_1 \quad \ldots \quad \Gamma,\ h \vdash A_n}$$

Logic: ∃

∃-introduction (unsafe, success probability 30%)

$$\frac{\Gamma \vdash \exists x,\ P\ x}{\Gamma \vdash P\ ?x}$$

∃-elimination (norm, penalty 0)

$$\frac{\Gamma,\ h : \exists x : A,\ P\ x \vdash T}{\Gamma,\ x : A,\ h : P\ x \vdash T}$$

Similar for `Subtype`, `Sigma`, `PSigma`

Logic: $\leftrightarrow$

$\leftrightarrow$-introduction (safe, penalty 100)

$$\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash B \to A}{\Gamma \vdash A \leftrightarrow B}$$

$\leftrightarrow$ hypotheses

A hypothesis of type $A \leftrightarrow B$ is treated as the equation $A = B$ by the simplifier and the substitution rule.

Logic: $\top$

$\top$-introduction[1] (safe, penalty 0)

$$\frac{}{\Gamma \vdash \top}$$

✓

Similar for `Unit`, `PUnit`.

[1] $\top$ = `True`

Logic: $\bot$

The simplifier already solves goals with an assumption $h : \bot$.[2]

We add destruct rules to conclude $\bot$ from `Empty` and `PEmpty`.

[2] $\bot$ = `False`

Logic: ¬

¬-introduction

$$\frac{\Gamma \vdash \neg A}{\Gamma,\ h : A \vdash \bot}$$

Negated hypotheses

Given a hypothesis of type $\neg A$, the simplifier replaces A with $\bot$ everywhere in the goal.

Logic: Simplifier

The simplifier performs limited logical reasoning. If A and B are propositions:

- With assumption $h : A$: rewrite $A = \top$
- With assumption $h : \neg A$: rewrite $A = \bot$
- $(\top \wedge \bot) = \bot$
- $(\top \wedge \top) = \top$
- $(\top \to A) = A$
- etc.

Logic: Completeness

In practice, these rules solve most ‘purely logical’ goals.

However, they are incomplete for first-order and even propositional logic.

Equality

Simplifier (norm, penalty 0)

Run the **`simp_all`** tactic as described previously

Reflexivity (safe, penalty 0)

Run the `rfl` tactic

Substitution (norm, penalty -50)

Run the `subst` tactic on any hypothesis of type `x = t` or `t = x` where x is a variable.
This substitutes `t` for x everywhere in the goal and removes the now-redundant hypothesis.

Case Splitting

Split target (safe, penalty 100)

Runs the **`split`** tactic.
This tactic looks for **`if-then-else`** and **`match`** expressions in the target and performs case splits on their discriminees.

Split hypotheses (safe, penalty 1000)

Ditto, but we look for case splits in hypotheses.

Extensionality

Extensionality (unsafe, success probability 80%)

Run the ext tactic.
This exhaustively applies extensionality lemmas to an equational goal. E.g.:

$$\Gamma,\ p\,q\ :\ A \times B \vdash p = q$$
$$\Gamma,\ p\,q\ :\ A \times B \vdash p.1 = q.1 \wedge p.2 = q.2$$

Debugging

Leftover Goals

When Aesop fails to solve a goal, it reports the goals that remain after safe rules have been exhaustively applied.

This helps to check whether the safe rules do what you think they should.

Proof Generation

```
theorem last_cons {a : α} {l : List α} (h : l ≠ nil) :
    last (a :: l) (cons_ne_nil a l) = last l h := by
  aesop? (add 1% cases List)
```

aesop? generates a proof script:

```
intro h
unhygienic aesop_cases l
· simp_all only [last, ne_eq]
  split
· simp_all only [last]
```

One click replaces **aesop**? with the generated proof.

Tracing

```
set_option trace.aesop true in
aesop
```

- Lists the rules that Aesop (tried to) run and the resulting goals
- For other trace options see autocompletion for `trace.aesop.`

Miscellaneous Features

Custom Rule Sets

```
-- RuleSet.lean

-- Must be in a different file for technical reasons
declare_aesop_rule_sets [Foo]
```

```
import RuleSet

@[aesop 100% (rule_sets [Foo])]
theorem foo
```

Used for domain-specific automation, e.g. `continuity`, `measurability`, ...

Metavariables

Aesop supports rules that generate metavariables:

```
example {a b c d : Nat} (h₁ : a < b)
    (h₂ : a < c) (h₃ : c < d) : a < d := by
  apply Nat.lt_trans
  -- ⊢ a < ?x
  -- ⊢ ?x < d

example {a b c d : Nat} (h₁ : a < b)
    (h₂ : a < c) (h₃ : c < d) : a < d := by
  aesop (add 1% Nat.lt_trans)
```

- 🥳 Aesop's search algorithm is conjectured complete even in the presence of metavariables
- 😐 This is very expensive

Applications, Shortcomings and Work In Progress

Applications

- General-purpose automation
- Domain-specific solvers: `continuity`, `measurability`, `aesop_cat`, ...
- Domain-specific ‘goal preprocessors’ with **`aesop`**?
- Tree search backend for LLMs: Peiyang Song, Kaiyu Yang, and Anima Anandkumar. “Towards Large Language Models as Copilots for Theorem Proving in Lean”. In: *Workshop on Mathematical Reasoning and AI*. 2023. URL: https://mathai2023.github.io/papers/4.pdf

Shortcomings and Work In Progress

- The built-in logical rules do not deal well with all-quantified hypotheses and sometimes negation
- The default rule set is currently missing many rules
- Repeated calls to **`simp`** during normalisation are bad for performance
- Tactic scripts generated by **`aesop`**? are not particularly idiomatic (WIP)
- **`aesop`**? only works if Aesop solves the goal (WIP)
- Nontrivial sets of forward rules are very slow (WIP)