A Aesop Overview Example

We used the following program for the Aesop overview example. It is based on the `BigStep.lean` file in the `AesopTest` directory of the Aesop Github repository (https://github.com/leanprover-community/aesop). This file was originally written by Asei Inoue and Jannis Limperg and released under the Apache 2.0 license (https://www.apache.org/licenses/LICENSE-2.0). We provide our adaptation here (also under the Apache 2.0 license).

```
/-
INCLUDES ADAPTATIONS FROM THE AUTHORS OF THIS PAPER.

ORIGINAL COPYRIGHT NOTICE:
Copyright (c) 2024 Asei Inoue. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Authors: Asei Inoue, Jannis Limperg
-/

import Aesop

set_option aesop.check.all true

abbrev Variable := String

def State := Variable → Nat

inductive Stmt : Type where
  | skip : Stmt
  | assign : Variable → (State → Nat) → Stmt
  | seq : Stmt → Stmt → Stmt
  | ifThenElse : (State → Prop) → Stmt → Stmt → Stmt
  | whileDo : (State → Prop) → Stmt → Stmt

infix:60 ";; " => Stmt.seq

export Stmt (skip assign seq ifThenElse whileDo)

set_option quotPrecheck false in
notation s:70 "[" x:70 "↦" n:70 "]" => (fun v ↦ if v = x then n else s v)

inductive BigStep : Stmt → State → State → Prop where
  | protected skip (s : State) :
    BigStep skip s s
  | protected assign (x : Variable) (a : State → Nat) (s : State) :
    BigStep (assign x a) s (s[x ↦ a s])
  | protected seq {S T : Stmt} {s t u : State} (hS : BigStep S s t) (hT : BigStep T t u) :
    BigStep (S;; T) s u
  | protected if_true {B : State → Prop} {s t : State} (hcond : B s) (S T : Stmt) (hbody : BigStep S s t) :
    BigStep (ifThenElse B S T) s t
  | protected if_false {B : State → Prop} {s t : State} (hcond : ¬ B s) (S T : Stmt) (hbody : BigStep T s t) :
    BigStep (ifThenElse B S T) s t
  | while_true {B S s t u} (hcond : B s) (hbody : BigStep S s t) (hrest : BigStep (whileDo B S) t u) :
    BigStep (whileDo B S) s u
  | while_false {B S s} (hcond : ¬ B s) :
    BigStep (whileDo B S) s s

notation:55 "(" S:55 "," s:55 ")" " ⇓ " t:55 => BigStep S s t

add_aesop_rules safe [BigStep.skip, BigStep.assign, BigStep.seq, BigStep.while_false]
add_aesop_rules 50% [apply BigStep.while_true]
add_aesop_rules safe [
  (by apply BigStep.if_true (hcond := by assumption) (hbody := by assumption)),
  (by apply BigStep.if_false (hcond := by assumption) (hbody := by assumption))
]
```

```
@[aesop unsafe]
theorem state_extensionality {S : Stmt} {s1 s2 t : State}
  (eq: ∀ v, s1 v = s2 v) (h : (S, s1) ⇓ t) : (S, s2) ⇓ t :=
  by
    have h : s1 = s2 := by funext; aesop
    aesop

namespace BigStep

@[aesop safe destruct]
theorem cases_if_of_false {B S T s t} (hcond : ¬ B s) : (ifThenElse B S T, s) ⇓ t → (T, s) ⇓ t := by
  intro h; cases h <;> aesop

@[aesop 30%]
theorem and_excluded {P Q R : Prop} (hQ : P → Q) (hR : ¬ P → R) : (P ∧ Q ∨ ¬ P ∧ R) := by
  by_cases h : P <;> aesop

set_option trace.aesop.tree true

theorem if_iff {B S T σ σ'} : (ifThenElse B S T, σ) ⇓ σ' ↔ (B σ ∧ (S, σ) ⇓ σ') ∨ (¬ B σ ∧ (T, σ) ⇓ σ') := by
  aesop

end BigStep
```

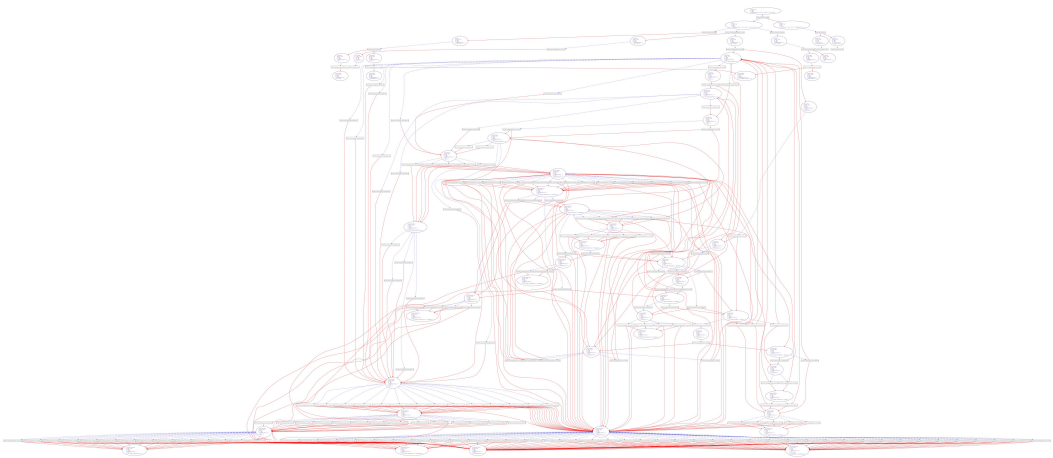

Fig. 5. **A high-level view of the AND–OR graph search trace of running Aesop on the big-step example.** The details are too small to see clearly because there are 57 OR nodes.

B SAT-Based Oracle to Check for Nonempty Completions

Semantic Formulas for Propositions. Propositions can either be assumed, in which case they are automatically true, or else they are subject to the standard interpretation of Horn clauses in a proof system: a proposition is true iff there exists a rule that fires with that proposition as a conclusion. This is captured by the following two formulas for each $Q \in \mathcal{A}$:

$$\mathsf{assume}_Q \implies \mathsf{true}_Q \qquad \text{and} \qquad \neg\mathsf{assume}_Q \implies \left(\mathsf{true}_Q \iff \bigvee_{R \text{ with conclusion } Q} \mathsf{true}_R\right)$$

Activity Formulas for Propositions. We start with two necessary conditions for a proposition $Q \in \mathcal{A}$ to be active. First, Q can be active only if it is actually true ($\mathsf{active}_Q \implies \mathsf{true}_Q$). Moreover, if Q is not the goal node, then it must have an active node that uses it:

$$\mathsf{active}_Q \implies \bigvee_{R \text{ with premise } Q} \mathsf{active}_R \qquad \text{(only for non-goal nodes)}$$

Only a single rule that provides a proposition can be active at a given time.

$$\sum_{R \text{ with conclusion } Q} \mathsf{active}_R \leq 1$$

This constraint uses a *cardinality constraint* to limit the number of literals set to true in a given set, which is often directly supported by off-the-shelf SAT solvers; alternatively, since the upper bound of the constraint is 1, it can easily be desugared into standard boolean SAT.

Finally, if Q is to be assumed, then none of the rules that provide Q as a conclusion can be active. Otherwise, Q will be active iff there exists such a rule that is active:

$$\mathsf{assume}_Q \implies \bigwedge_{R \text{ with conclusion } Q} \neg\mathsf{active}_R$$

$$\neg\mathsf{assume}_Q \implies \left(\mathsf{active}_Q \iff \bigvee_{R \text{ with conclusion } Q} \mathsf{active}_R\right)$$

Consistency Formulas for Propositions. For each proposition $Q \in \mathcal{A}$, we add a simple consistency formula that enforces the desired meanings of the label partitions:

$$\begin{cases} \top & \text{if } e(Q) = \bot, \\ \neg\mathsf{assume}_Q & \text{if } e(Q) = \mathbf{?}, \\ \neg\mathsf{true}_Q \wedge \neg\mathsf{assume}_Q \wedge \neg\mathsf{active}_Q & \text{if } e(Q) = \mathbf{F}, \\ \mathsf{true}_Q & \text{if } e(Q) = \mathbf{T/A}, \\ \mathsf{true}_Q \wedge \mathsf{active}_Q & \text{if } e(Q) = \mathbf{T/A!}, \\ \mathsf{true}_Q \wedge \neg\mathsf{assume}_Q & \text{if } e(Q) = \mathbf{T}, \\ \mathsf{true}_Q \wedge \neg\mathsf{assume}_Q \wedge \mathsf{active}_Q & \text{if } e(Q) = \mathbf{T!}, \\ \mathsf{true}_Q \wedge \mathsf{assume}_Q & \text{if } e(Q) = \mathbf{A}, \\ \mathsf{true}_Q \wedge \mathsf{assume}_Q \wedge \mathsf{active}_Q & \text{if } e(Q) = \mathbf{A!}. \end{cases}$$

Semantic and Activity Formulas for Rules. Because rules do not have partition labels or assume-semantics, the formulas for them are much simpler:

$$\mathsf{true}_R \iff \bigwedge_{Q \text{ is premise of } R} \mathsf{true}_Q \qquad \text{and} \qquad \mathsf{active}_R \implies \mathsf{true}_R \qquad \text{and} \qquad \mathsf{active}_R \implies \bigwedge_{Q \text{ is premise of } R} \mathsf{active}_Q$$

The first formula is standard AND semantics, the second captures that a rule can only be active if it is true, and the third captures that a rule can only be active if all its premises are active.

Putting It All Together. For a given expression e, let $\varphi(e)$ be the conjunction of the above boolean formulas. Then $\varphi(e)$ is satisfiable iff $C(e)$ is nonempty, as desired. Because all formulas except the consistency formula are constant in a given run of interactive synthesis, $\varphi(e)$ can be solved efficiently using an incremental SAT solver.

C Additional Step Providers

Algorithm 8 COMMIT Step Provider

```
1: procedure COMMIT(e)
2:     for Q ∈ e^{-1}[{T/A, T/A!}] do
3:         for ℓ > e(Q) do
4:             σ ← Q := ℓ
5:             if SAT(φ(σe)) then yield σ
```

Algorithm 9 LEAF Step Provider

```
1: procedure LEAF(e)
2:     for Q ∈ 𝒜 not a conclusion of any rule in ℛ do
3:         for ℓ > e(Q) do
4:             σ ← Q := ℓ
5:             if SAT(φ(σe)) then yield σ
```

C.1 PessimisticFinalize (One-shot)

RANDOM and MAXINFOGAIN, both reduce the number of steps yielded at each round of interaction while maintaining STRONG COMPLETENESS. However, even once enough propositions have been assumed to make the goal hold, the interactive synthesis must continue until all remaining propositions are labeled. Thus, as a convenience, we also provide the PESSIMISTICFINALIZE step provider, which we define in Algorithm 10 below. When applicable, this step provider provides a single step that sets all $\bot$ propositions (unseen) to **?** (unknown), **T/A** propositions to **T** (not assumed), and **T/A!** propositions to **T!** (not assumed). All other labels remain untouched. This represents a worst-case scenario where no other propositions are assumed and no further information is provided. PESSIMISTICFINALIZE does not achieve STRONG COMPLETENESS, but rather serves as a convenience provider to be composed with other step providers.

Algorithm 10 PESSIMISTICFINALIZE Step Provider

```
1: procedure PESSIMISTICFINALIZE(e)
2:     𝒳 ← ⟨Q ∈ 𝒜 : e(Q) non-maximal⟩ (in arbitrary order)
3:     for i = 1, ..., |𝒳| do
4:         if e(𝒳_i) = ⊥ then 𝒴_i ← ?
5:         else if e(𝒳_i) = T/A then 𝒴_i ← T
6:         else if e(𝒳_i) = T/A! then 𝒴_i ← T!
7:     σ ← 𝒳_1 := 𝒴_1 ; ··· ; 𝒳_{|𝒳|} := 𝒴_{|𝒳|}
8:     if SAT(φ(σe)) then yield σ
```

D Additional Data and Graphs

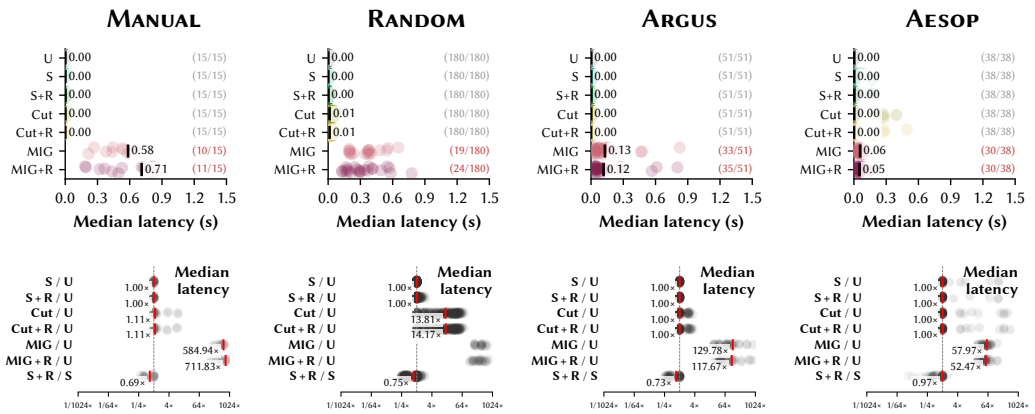

Fig. 6. **Median latency results for each step provider. Median latency** is the median per-round latency. Labeled bars appear at the median when enough data points exist to compute it.

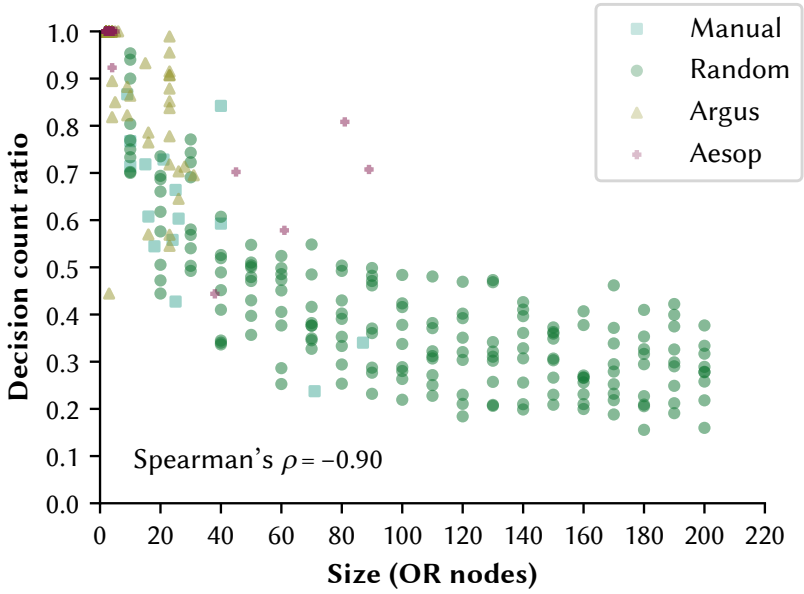

Fig. 7. **The fraction of decisions needed *with* relevancy pruning compared to *without* relevancy pruning plotted against AND–OR graph size.** Larger AND–OR graphs benefit more from relevancy pruning across all suites.

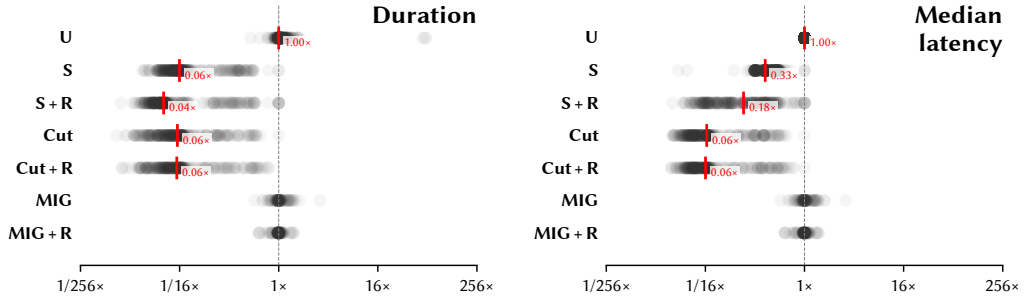

Fig. 8. **Incremental SAT solving ablation (all suites). Duration** is the total duration. **Median latency** is the median per-round latency. Labeled bars appear at the median when it can be computed.

Table 1. **Benchmark summary statistics.** Each entry describes the median and IQR. **Depth** is the maximum graph distance from the goal to any other node in the same connected component. **ORs** and **ANDs** are the OR and AND node count. **Consumers** is the median number of parents for OR nodes, **Providers** is the median number of children for OR nodes, and **Premises** is the median number of children for AND nodes.

Suite	Depth	ORs	ANDs	Consumers	Providers	Premises
Manual (15 entries)	8 (6–12)	24 (16–40)	12 (9–32)	1 (1–1)	0 (0–1)	2 (1–2)
Random (180 entries)	8 (6–12)	105 (60–150)	30 (16–48)	1 (1–1)	0 (0–0)	3 (2–4)
Argus (51 entries)	4 (2–4)	5 (3–23)	3 (2–18)	1 (0.5–1)	0 (0–0.5)	1 (1–4)
Aesop (38 entries)	2 (2–4)	3 (3–4)	1 (1–5)	1 (1–1)	0.5 (0–1)	1 (1–2)

Table 2. **Maximum latencies.** Each entry is the maximum latency of the step provider on the given suite.

	Manual	Random	Argus	Aesop
U	<1 ms	<1 ms	<1 ms	<1 ms
S	<1 ms	1 ms	4 ms	224 ms
S+R	<1 ms	6 ms	<1 ms	201 ms
Cut	16 ms	92 ms	4 ms	907 ms
Cut+R	15 ms	83 ms	4 ms	**1084 ms**