

# Navigating AND–OR Graph Modifications to Debug Failing Proof Search

JUSTIN LUBIN, University of California, Berkeley, USA

MARLENA PREIGH, University of California, Berkeley, USA

MAX WILLSEY, University of California, Berkeley, USA

SARAH E. CHASINS, University of California, Berkeley, USA

Proof search powers our most advanced programming tools, from type systems, to search tactics for interactive theorem provers, to Datalog-backed program analyses. Although proof search tooling is powerful and now pervasive, *debugging* it is hard, even for experts. When proof search cannot prove the goal, the programmer’s best source of information is a massive AND–OR graph representing the tool’s internal state during the proof search process. The difficulty of understanding and debugging this vast trace of internal state locks programmers out of exactly the high-assurance automated reasoning tools we want them to adopt.

We propose a new formulation of proof search debugging, which: (i) views AND–OR graphs as a partial representations of the underlying proof system, (ii) treats debugging as a process of applying modifications to this proof system, and (iii) uses a debugging tool to solicit these modifications until the resulting proof system proves the original goal. This approach unifies decades of ad-hoc strategies in a single general-purpose framework and is applicable to the diverse range of programming tools that use proof search. Our framework can express existing “why-not” debugging strategies as well as new strategies, and we evaluate such strategies on 284 AND–OR graphs. We find that a strategy that enforces a property called **STRONG SOUNDNESS** reduces the number of decisions by  $1.4\times$ – $3.2\times$  compared to an unsound baseline, and a new property we call **STRONG COMPLETENESS MODULO OBSERVABILITY** enables pruning to further reduce decisions by  $1.0\times$ – $2.8\times$  for an overall reduction of  $2.0\times$ – $3.8\times$ .

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Debugging, Proof Search, Interactive Program Synthesis

## ACM Reference Format:

Justin Lubin, Marlena Preigh, Max Willsey, and Sarah E. Chasins. 2026. Navigating AND–OR Graph Modifications to Debug Failing Proof Search. *Proc. ACM Program. Lang.* 10, PLDI, Article 266 (June 2026), 25 pages. <https://doi.org/10.1145/3808344>

## 1 Introduction

When proof search fails, it is rarely clear what to do next. A tool can share that the goal was not derivable with the current proof system, but not *why*. This leads to widespread challenges around “why-not” debugging [11], which programmers face across many settings including debugging Datalog programs, Rust trait errors, and proof search tactics in their favorite interactive theorem provers. In a proof assistant, for example, programmers face “why-not” debugging when an automation tactic gets stuck on a known-true subgoal; in Datalog, they engage in “why-not” debugging when Datalog fails to answer a query that the programmer knows should be entailed by the rules.

---

Authors’ Contact Information: [Justin Lubin](mailto:justinlubin@berkeley.edu), University of California, Berkeley, Berkeley, USA, [justinlubin@berkeley.edu](mailto:justinlubin@berkeley.edu); [Marlena Preigh](mailto:mpreigh@berkeley.edu), University of California, Berkeley, Berkeley, USA, [mpreigh@berkeley.edu](mailto:mpreigh@berkeley.edu); [Max Willsey](mailto:mwillsey@berkeley.edu), University of California, Berkeley, Berkeley, USA, [mwillsey@berkeley.edu](mailto:mwillsey@berkeley.edu); [Sarah E. Chasins](mailto:schasins@cs.berkeley.edu), University of California, Berkeley, Berkeley, USA, [schasins@cs.berkeley.edu](mailto:schasins@cs.berkeley.edu).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/6-ART266

<https://doi.org/10.1145/3808344>

To date, our best strategies revolve around revealing traces of the proof search process, typically represented as AND–OR graphs. Before a failure, the proof search tool will have explored a vast space of possibilities specified by the underlying proof system of routes to a proof; an AND–OR representation of the proof search process represents all of these explored routes. When a goal node should be true but is not, somewhere in the AND–OR graph there must be a lemma that is too weak, a missing trait implementation, a guard that is too restrictive, an unexpected interaction between constraints that seemed reasonable in isolation, or some other relationship between nodes that diverges from the programmer’s expectation about how the proof system would behave. Unfortunately for the debugging process, the AND–OR graph includes not only the paths with those unanticipated behaviors, but also *every single route the proof search considered*. This representation exposes every intermediate proof step that the engine tried and discarded, without any information about which neighborhood of the search space is relevant to the programmer’s intended behavior.

A range of debugging approaches have offered support for exposing this AND–OR graph [17–19, 22, 41], and sometimes for navigating from a node to its children or parent [22]. Programmers are left to reverse-engineer the proof system from this internal trace, manually traversing the graph to seek out, for example, nodes that should have been true but weren’t and rules that should have fired but didn’t. For many decades, “why-not” debugging has remained an open problem—even as proof search has become a core component in modern programming tools—because of this combination of massive search traces, multi-node AND–OR interactions, and relying on programmer intuition alone for guidance on how to traverse search traces.

We build on the insight that the aim of debugging is to switch the goal node from false to true, and that the AND–OR graph (being itself a representation of the proof system) has much of the information necessary for reasoning about what changes will make the goal node true. Debugging should terminate when the programmer has accumulated a list of proof system *modifications* that (i) they are willing to implement and (ii) make the goal provable.

For example, consider one particular class of proof system modifications we may want to allow: labeling a given node in the AND–OR graph as implementable. For proof assistants, such a label would mean that we know we can manually prove the subgoal, or for Rust, that we are willing to directly implement the trait. With this perspective, a debugging strategy is a strategy for soliciting proof system modifications until the goal becomes provable—or, in this more specific framing, a strategy for soliciting node labels until the goal becomes provable.

For a given buggy proof system, there are a vast number of possible modifications that could make the goal node true. In this sense, the buggy proof system is an *underspecification* for a proof system modification. Unfortunately, we cannot accept just any solution to this underspecification—only solutions that are actually reasonable to implement in the program that triggered the proof search. *Thus, we must somehow solicit information to figure out what constitutes a “true” solution to this underspecification.* Accordingly, we formulate the “why-not” debugging problem as an instantiation of Programming by Navigation (PBN) [35], an interactive synthesis technique for iterative specification refinement. In our context, the PBN *step provider* offers a set of valid labels for nodes, each of which represents a particular modification to the underlying proof system. At each step, the PBN *step decider*—perhaps a programmer—selects one of those proof system modifications.

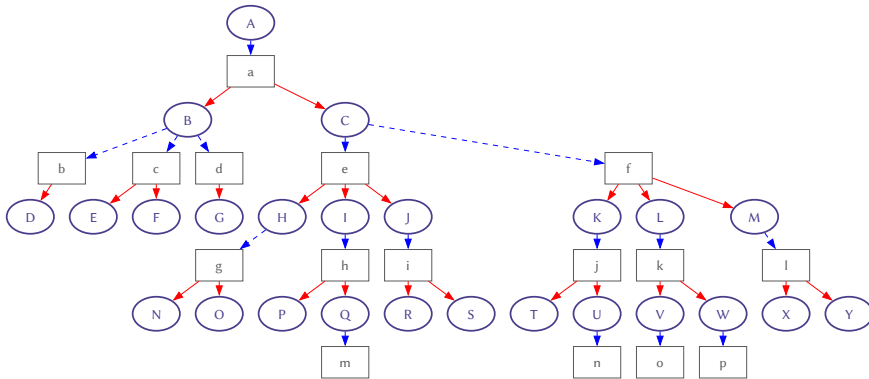
Using PBN, we define a variety of step providers for “why-not” debugging, including existing debugging approaches as well as debugging approaches that have not appeared in the literature so far. Specific to these new approaches is the ability to jump across different parts of the AND–OR graph based on what parts of the proof remain unfinished, given the step decider’s proof system modifications so far. We find that enforcing PBN’s STRONG SOUNDNESS property reduces the number of decisions required from the step decider, and that filtering for relevance via a novel property we call STRONG COMPLETENESS MODULO OBSERVABILITY reduces the number of decisions even more.

Overall, we make the following contributions:

- **A general framework for “why-not” debugging of proof systems.** We use AND-OR graphs to represent the proof system, and we use Programming By Navigation (PBN) to reach a set of proof system modifications that make the goal provable. A given debugging strategy is implemented as a PBN *step provider*. Our formulation unifies existing “why-not” debugging strategies and can also be used to express new ones.
- An extension to PBN, **STRONG COMPLETENESS MODULO OBSERVABILITY.** In contrast to STRONG COMPLETENESS, which enforces that all valid programs are reachable, STRONG COMPLETENESS MODULO OBSERVABILITY enforces that all *equivalence classes* of valid programs are reachable. If an equivalence class is definable such that the step decider is equally satisfied with all of its members, STRONG COMPLETENESS MODULO OBSERVABILITY allows step providers to prune irrelevant steps, reducing the number of decisions required from the step decider.
- An implementation of our framework and an empirical evaluation on 284 synthetic AND-OR graphs, investigating, among other comparisons, the number of decisions required from the step decider when the step provider presents random steps (no STRONG SOUNDNESS), these steps filtered such that only valid steps are shown (STRONG SOUNDNESS), and these steps further filtered such that (only) any valid equivalence class is reachable (STRONG SOUNDNESS + STRONG COMPLETENESS MODULO OBSERVABILITY).

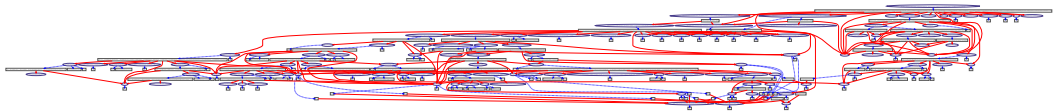
## 2 Overview

Suppose we seek to prove a statement *A* in our favorite proof assistant. We turn to one of its automation facilities, but, unfortunately, it cannot prove our goal. It does, however, emit the following AND-OR graph showing all the paths it tried:



Circled nodes are OR nodes, corresponding to proof goals (i.e, propositions), and boxed nodes are AND nodes, corresponding to inferences (those without children are vacuously true). Using this information, we may be able to visually inspect the AND-OR graph to figure out which subgoals we would like to manually prove in order to make the proof go through. For instance, say we know that we can manually prove all subgoals in the set  $\{D, T, X, Y\}$ ; this AND-OR graph has all the information we need in order to know that proving  $\{D, T, X, Y\}$  will make goal node *A* true.

Of course, we’re not always so lucky. Sometimes the AND-OR graph looks more like this:

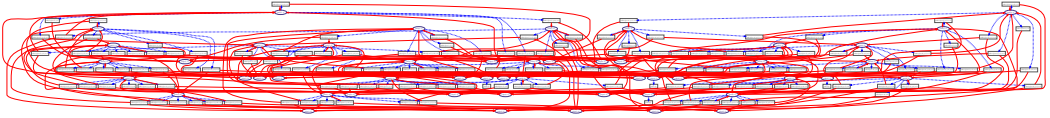


And this is not a phenomenon peculiar to proof automation. Indeed, this second example is from a realistic Rust trait error by Gray et al. [22] extracted using their Rust trait error debugger Argus; OR nodes are trait obligations on types, and AND nodes are implementation blocks for traits.

The problem gets even worse in Datalog, where all rules are quantified over a finite domain. This setting requires grounding all rules, resulting in AND–OR graphs that explode in size. The following sample Datalog program checks whether paths exist between nodes in a graph:

```
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
edge(1, 3). edge(2, 3). edge(4, 2). edge(4, 5).
```

It results in the following AND–OR graph when queried for `path(1, 5)`, where OR nodes are grounded facts and AND nodes are grounded rules:



If proof automation does not succeed, a trait is not implemented, or a fact is not derivable, we must debug these AND–OR graphs to find out what went wrong. It is no mystery why Caballero et al. [11] write that such missing answers are “the most complicated source of errors in logic programming.” Debugging a missing answer is very different from debugging a wrong answer; when we find a proof of a proposition that should be false, we have a representation of exactly what steps produced the proof. That is, in the wrong answer setting, we have a standard graph (an AND graph) representing the proof steps that were actually used to derive the goal—and *only* the proof steps that were actually used to derive the goal. This is a much smaller artifact than an AND–OR representation of all proof steps that *could* have been used, given a particular proof system.

In this paper, we explore: **How can we debug AND–OR graphs when the goal is unprovable?**

To explain our key insights, we will explore the extremely simple AND–OR graph in Figure 1. The goal is the node *A*, and there are two rules that could provide it. The first rule, *f*, depends on *B* and *C* and the second rule, *g*, depends on *C* and *D*. None of *B*, *C*, or *D* succeeded. Typically these AND–OR graphs are viewed as simply computational artifacts produced as a trace of a failed search procedure, with no formal semantics. To debug these graphs, then, we could traverse their topology until we understand where the search procedure diverged from our expectations.

For example, a *top-down strategy* would start at the goal node *A* and enable exploration of the rules that could have directly provided the node at hand (but failed to do so). In Figure 1, the exploration would start at the node *A*, then either the *f* rule or the *g* rule could be explored, resulting in exploring either *B* and *C* or *C* and *D*. A *bottom-up strategy* would start at the failing leaves and enable exploration of nodes that directly depend on the node at hand. In Figure 1, the exploration would start from choosing among *B*, *C*, and *D*; for *B* or *C*, the rule *f* could then be explored, and for *C* or *D*, the rule *g* could be explored, in either case ending up at the node *A*.

From this topological perspective, an AND–OR graph is just that: a graph. However, we argue for the perspective that hidden within each of these AND–OR graphs is a *proof system* that can be poked, prodded, and queried to report whether certain propositions hold or do not hold. This is most clear in the Datalog example above (where the AND–OR graph is literally derived from a proof system), but the intuition extends to all the AND–OR graphs we have mentioned. For example, an AND–OR graph from a failed proof automation search does not just capture a trace of the tool, but rather defines an (incomplete) subset of the possibilities of the underlying logic it is operating over. We can even modify these proof systems—e.g., by making new assumptions—and

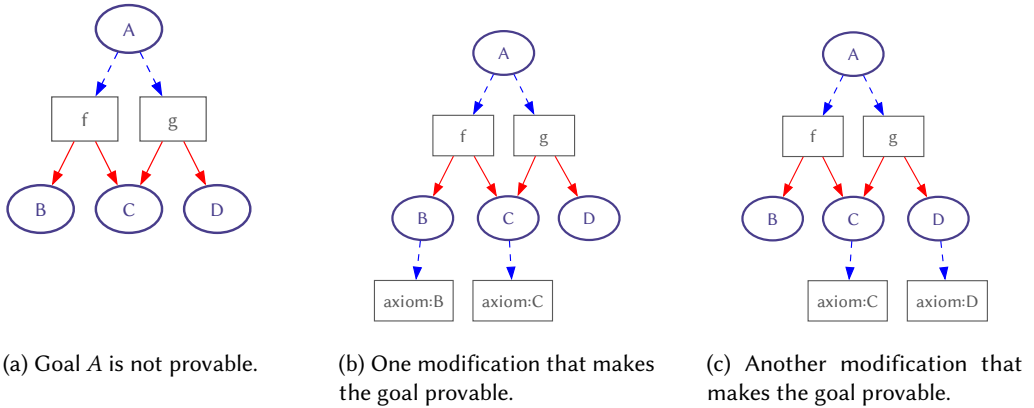


Fig. 1. **Three AND-OR graphs representing a simple proof system and plausible modifications.** Each graph represents a proof system with node  $A$  representing the proof goal. Circled nodes are OR nodes and boxed nodes are AND nodes. AND nodes with no outgoing edges are vacuously true and can represent axioms. OR nodes with no outgoing edges are unprovable. Thus, our initial AND-OR graph (a) represents a failed proof search.  $B$ ,  $C$ , and  $D$  are OR nodes without outgoing edges, so they are unprovable; thus,  $f$  and  $g$  are both unprovable, and  $A$  is unprovable as well. In (b), we see a modification to the initial proof system represented in (a). In the modified proof system shown in (b), we have provided some additional information—namely, that we are willing to implement changes that will make  $B$  and  $C$  provable. For example, if  $B$  and  $C$  represent subgoals in a proof search, our modification indicates that we are willing to manually prove  $B$  and  $C$ . Applying the “I will prove this” label to  $B$  and  $C$  has produced a transformed proof system, in which `axiom:B` enforces that  $B$  is true and `axiom:C` enforces that  $C$  is true. With these modifications,  $f$  is provable, and the goal node  $A$  is provable. The transformed proof system now successfully proves goal  $A$ . In (c), we see an alternative modification to the initial proof system represented in (a). Say we know that we cannot prove  $B$ , but we know we can prove  $C$  and  $D$ . Then marking  $C$  and  $D$  with the “I will prove this” label produces the transformed proof system represented in (c), in which  $g$  is provable, making goal node  $A$  provable.

ask if these modifications make our goal succeed. For example, to assist in debugging the AND-OR graph in Figure 1a, a system could provide a minimal set of assumptions that would make the goal provable; say,  $\{B, C\}$ , as shown in Figure 1b; alternatively,  $\{C, D\}$ , as shown in Figure 1c. (In the databases community, this is known as instance-based why-not provenance [25].)

*Key Insight 1: Navigating the Space of Proof System Transformations.* Although the strategies we’ve discussed so far may seem quite different, they are all useful primarily if they lead to an insight about some modifications to the proof system that will make the goal provable. However, this is an incomplete specification. Not just any modification will do. In particular, we note that:

- (1) There are many trivial modifications that will make the goal succeed, such as adding a rule that goes from any provable proposition  $Q$  in the AND-OR graph to the goal node  $G$ . In a proof assistant, such a rule would mean that we can derive  $G$  assuming  $Q$ , and in Rust, this would mean that we can make an `impl` block for  $G$  for types satisfying  $Q$ . Neither of these are true in general. We are thus interested not just in *any* modification to the proof system, but only the ones that *represent actions the user could actually take to fix it*. Our proof system modifications will delineate *the grammar of possible modifications* we allow.
- (2) In all the examples above, the modification of interest is to assume a set of propositions. But even with this restricted class of modifications, not just any set of assumptions will do. For example, in Figure 1, perhaps  $B$  is *supposed* to be false; in such a case, the suggestion to assume

$\{B, C\}$  from instance-based why-not provenance is unhelpful. Put another way, the failing proof system serves as an *underspecification* for the debugging problem, and a debugging system must allow navigation to a set of assumptions that satisfies that underspecification—but also additional constraints, which are not known to the debugging system.

- (3) Finally, some proof system modifications represent a change the programmer wants to implement, while some do not. Even with a set of equally possible changes, the programmer may have preferences about how they *want* to change the program that produced the AND-OR graph. Again, the AND-OR graph is only an underspecification, and another portion of the specification is implicit in the programmer’s preferences about their program.

To address these challenges, we present a framework in Section 3 to view debugging AND-OR graphs as interactively synthesizing a proof system modification that makes the goal node true. Viewed from the perspective of interactive program synthesis, this amounts to iterative specification refinement from the initial underspecification of the failing proof system.

In Section 4, we instantiate our framework to the assumption modifications we mention in point (2) above. Formally, we model proof system transformations as a partition of the propositions, where each proposition gets assigned a label that specifies how it should behave in the transformed proof system. These labels include F (should be false), ? (may be false or true), and T (should be true), as well as A, which denotes that the proposition can be *assumed*—for example, because the programmer is willing to implement it manually. We start by assuming the goal proposition is labeled with T!, a label we use for propositions that should be true *and* on the final derivation of the goal. (We also include a label A! that is analogous for assumed propositions.)

The top-down interaction above can be modeled by a system providing two steps: one step to set  $B$  and  $C$  to T/A!, and one step to set  $C$  and  $D$  to T/A!, where T/A! is a label that can later be refined to either T! or A!. Selecting between these two steps corresponds to traversing the topology of the underlying AND-OR graph to one of the two subgraphs in a top-down fashion; the explored propositions get the label T! because, for the explored path to actually result in the goal node being true, those propositions *must* be true or assumed (and they will be on the selected derivation of the goal). We flesh out this approach and how to describe the bottom-up and instance-based why-not provenance strategies using our formalism in Section 5.

This formalism can also express entirely new navigation strategies. For example, Figure 1’s  $C$  would be a bad proposition to display for labeling, as it must always be assumed. In Section 5, we capture this intuition about “informative” propositions using information theory to define MAXINFOGAIN, which provides steps that minimize the expected conditional entropy. Assuming equal prior probability for all labels, the proposition with the minimal expected conditional entropy will be the one

	$B$	$C$	$D$
F	1 (4)	0	1 (4)
?	1 (4)	0	1(4)
T	0	0	0
T!	0	0	0
A	2 (8)	3 (11)	2 (8)
A!	2 (6)	3 (11)	2 (6)
<i>Average</i>	1.5	3	1.5
$\mathbb{E}[\text{Cond. entropy}]$	0.151	0.477	0.151

with the smallest average number of semantically-distinct valid partition completions. For Figure 1, the table to the right shows the number of possible semantically-distinct label assignments to the remaining propositions when the proposition in a given column is set to the label in a given row (parenthesized numbers are the total number of label assignments). From the table, we see that  $B$  and  $D$  have minimal conditional entropy, and are thus informative propositions to label.

A step provider like MAXINFOGAIN can be composed with any other step provider so that, for example, navigation can proceed in a top-down or bottom-up fashion starting at a proposition provided by MAXINFOGAIN. Our formalism provides a unifying lens on existing strategies and enables a wide range of new strategies to explore for debugging AND-OR graphs.

*Key Insight 2: STRONG SOUNDNESS and STRONG COMPLETENESS MODULO OBSERVABILITY.* To this point we have focused on expressing many strategies: Different domains and applications will benefit from different navigation strategies. Without strong assumptions about how a user will navigate, it is difficult to say whether one strategy is “better” than another.

And yet, perhaps surprisingly, some strategies are *provably* better than others. Lubin et al. [35] define a notion of STRONG SOUNDNESS in the context of interactive synthesis that requires all provided specification refinements to lead to at least one valid solution. Applied to our setting, STRONG SOUNDNESS requires that navigation strategies should only provide labeling of partitions that could eventually lead to a valid partition. All else being equal, a step provider that does a post-hoc filter to eliminate any steps that would violate STRONG SOUNDNESS is better than the same strategy that does not, as it will require fewer decisions from the step provider. (Not only will there be fewer decisions, but all decisions will lead to valid solutions. If a user were to select a step that couldn’t lead to a valid solution, they would eventually be forced to *backtrack* to the incorrect step.)

In fact, we can go further than STRONG SOUNDNESS. Our approach provides the ability to navigate a space of partitions to arrive at a particular set of assumptions. Only **A** and **A!** labels actually change the semantics of a partition, but applying the other labels can substantially prune the search space. Naïvely requiring that all valid solutions be reachable using the provided steps (what Lubin et al. [35] call STRONG COMPLETENESS) means that many steps that redundantly lead to the same final outcome must be included in the provided set. As an example, say a goal node has two child AND nodes,  $f$  and  $g$ , with no overlapping descendants, and the step decider has indicated via the label **T!** that a premise of  $f$  must be on the derivation path. Then any labels on the descendants of  $g$  would represent different label sets (and thus must be reachable under STRONG COMPLETENESS), but cannot change anything about the derivation for the eventual, completed set of proof system modifications. Thus, we extend PBN with a new concept, STRONG COMPLETENESS MODULO OBSERVABILITY, which captures that step providers need only provide steps to cover the set of equivalence classes over a specified observability relation.

We can then implement a post-hoc filtering step, *relevancy pruning*, to eliminate steps that are valid but irrelevant—they change the final partition, but cannot change the *equivalence class* of the final partition. With our specific definition of equivalence, this ensures that the interactive refinement process can label any set of propositions as assumptions while substantially cutting down the number of decisions to be made. In Section 8, we empirically measure the extent to which STRONG SOUNDNESS and relevancy pruning reduce the number of decisions.

*System Walkthrough.* To perform our evaluation in Section 8, we implemented a Programming by Navigation synthesizer we call AO-NAV. To make the above key insights more concrete, we briefly give a concrete walkthrough of what it looks like to use AO-NAV to do Programming by Navigation on AND–OR graph modifications to debug a failing proof.

Suppose we wish to use the Aesop proof search tool [31] for the Lean proof assistant [36] to prove the following proposition about a big-step semantics we have defined:<sup>1</sup>

$$\dots \vdash (\text{ifThenElse } B \ S \ T, \sigma) \Downarrow \sigma' \leftrightarrow B(\sigma) \wedge (S, \sigma) \Downarrow \sigma' \vee \neg B(\sigma) \wedge (T, \sigma) \Downarrow \sigma'$$

When we run Aesop, it fails and emits an AND–OR graph with 57 OR nodes (Figure 5 in Appendix A). We can load this AND–OR graph into AO-NAV to construct a modification to it that will prove this goal. Our modifications will correspond to *manually proving subgoals (lemmas)* in Lean.

Upon loading our graph into AO-NAV using a step provider we call CUT (Section 5), we are asked select among the labels **?**, **T**, **A**, **T!**, and **A!** for the following proposition:

$$\dots \vdash (\text{ifThenElse } B \ S \ T, \sigma) \Downarrow \sigma' \rightarrow B(\sigma) \wedge (S, \sigma) \Downarrow \sigma' \vee \neg B(\sigma) \wedge (T, \sigma) \Downarrow \sigma'$$

<sup>1</sup>Appendix A contains the full Lean program, which is based on code by Inoue and Limperg in the Aesop codebase.

If we wish to manually prove this subgoal, we can label this proposition as an assumption (**A**); otherwise, we can indicate that the subgoal is true (**T**) or that we are unsure (?). If we wish to assume the proposition or indicate the proposition should be true, we can further *require* that proposition is used in the final proof (**A!** or **T!**). Notably, the **F** label is not an option because there is no way to modify the AND–OR graph to prove the goal while this proposition remains unprovable.

Because this proposition is quite similar to our goal (it’s just one direction of the if-and-only-if), we select the label ? to proceed to the next round. Now we are asked to label the following:

$$\dots, (\text{ifThenElse } B \ S \ T, \sigma) \Downarrow \sigma' \vdash B(\sigma) \rightarrow (S, \sigma) \Downarrow \sigma'$$

The selectable labels include ?, **T**, **A**, **T!**, and **A!** and **F**, indicating that this proposition can remain unprovable while the goal proposition is provable (i.e., Aesop has other ways of proving the goal). We realize that the proof is failing because we proved a similar inversion lemma for  $\neg B(\sigma)$ , but we are missing the case for  $B(\sigma)$ . Thus, we do want to prove this subgoal manually, so we label it as **A**. With this modification, Aesop is able to prove the final goal and the interaction is complete.

### 3 Navigation Beyond the Graph Topology

We begin with a set of standard preliminaries.

*Definition 3.1 (Preliminaries).* A **proof system**  $\mathcal{P}$  is a pair  $(\mathcal{A}, \mathcal{R})$ , where  $\mathcal{A}$  is a finite set of symbols called **atomic propositions** and  $\mathcal{R}$  is a finite set of Horn clauses over  $\mathcal{A}$  called **rules**.

Each rule is of the form  $\frac{P_1 \ \dots \ P_N}{C}$  (**RULE-NAME**), where each **premise**  $P_i$  and the **conclusion**  $C$  are atomic propositions in  $\mathcal{A}$  and “**RULE-NAME**” is a unique string that identifies the **name** of the rule. A **derivation** of  $Q \in \mathcal{A}$  is a proof tree over  $\mathcal{R}$  whose final conclusion is  $Q$ . If a derivation of  $Q$  exists, we write  $\mathcal{P} \vdash Q$ .

We can translate between AND–OR graphs and proof systems by identifying the OR nodes with propositions and the AND nodes with rules. For ease of notation and to emphasize the provability relation  $\mathcal{P} \vdash Q$ , we exclusively use proof systems as a formalization of AND–OR graphs.

Next, we can formalize what we mean by a proof system transformation (which we notate  $e$ , for *expression*). These proof system transformations will model the changes a user is permitted to make to a proof system to make the goal proposition true.

*Definition 3.2 (Proof system transformations).* We assume a notion of **expressions**  $e$  whose semantics  $\llbracket e \rrbracket$  are functions from proof systems to proof systems. For a proof system  $\mathcal{P}$  and **goal** proposition  $G$ , we also assume a notion of **validity** on expressions (written  $e$  **valid**) specifying which expressions are permissible to navigate to. We require that  $e$  **valid** implies  $\llbracket e \rrbracket \mathcal{P} \vdash G$ .

The requirement that  $e$  **valid** implies  $\llbracket e \rrbracket \mathcal{P} \vdash G$  ensures validity captures the idea of proof system transformations “fixing” a broken proof system. Indeed, our objective will be to arrive at such an  $e$  for a given  $\mathcal{P}$  and goal  $G$  with  $\mathcal{P} \not\vdash G$ . We can observe that such a proof system  $\mathcal{P}$  and goal  $G$  forms an *underspecification* for expressions  $e$  in the sense that  $e$  is satisfactory if  $e$  **valid**. We emphasize that not just any solution to this underspecification will do; only ones that are implementable in practice will be helpful. Thus, our perspective is to view this as an interactive synthesis problem in which this underspecification is to be iteratively refined.

To model this process, we use Programming by Navigation [35]. Briefly, in addition to a notion of expressions and validity (as above), Programming by Navigation requires a notion of *steps*  $\sigma$  and a relation  $e_1 \xrightarrow{\sigma} e_2$  that holds if  $\sigma$  transforms  $e_1$  to  $e_2$ . (We write  $e_1 \prec e_2$  if there exists a  $\sigma$  such that  $e_1 \xrightarrow{\sigma} e_2$ .) Specification refinement proceeds by iteratively updating a starting expression  $e_0$  with steps selected among a set of steps  $\Sigma$  provided by a step provider (a function from the

current working expression to a set of steps). At each round of synthesis, the step set  $\Sigma$  provided by the step provider must satisfy **STRONG SOUNDNESS** (all provided steps lead to at least one valid expression) and **STRONG COMPLETENESS** (all valid expressions reachable from the current expression are reachable using some provided step). The task of the synthesis designer is to design a step provider that satisfies these property. We refer the reader to Lubin et al. [35, §3] for additional background, although the level of detail provided here suffices for the present purposes.

## 4 The Space of Partitions

The question, then, is: What choice of expressions and steps should we use? One choice that we have found to be particularly expressive for the domains we are interested in is to let expressions be special types of partitions of the propositions of the proof system (which we call *assume-partitions*). Put another way, our expressions are functions from propositions to labels, where a subset of these labels indicate which propositions to axiomatize. We formalize this notion below.

*Definition 4.1.* A **partition** of a proof system  $\mathcal{P} = (\mathcal{A}, \mathcal{R})$  is a function from the propositions  $\mathcal{A}$  to a finite strict partial order of labels  $L$  with order  $<$  and distinguished bottom element  $\perp$ . An **assume-partition**  $e$  (which we will take as our notion of expressions) is a partition whose set of classes contains a designated set  $L_{\text{assume}} \subseteq L$  of **assume-labels**. The semantics of an assume-partition  $e$  is defined by  $\llbracket e \rrbracket (\mathcal{A}, \mathcal{R}) = (\mathcal{A}, \mathcal{R}')$ , where  $\mathcal{R}' = \mathcal{R} \cup \left\{ \frac{}{Q} \text{ (AXIOM}_Q\text{)} : Q \in e^{-1}[L_{\text{assume}}] \right\}$ . By  $[Q := \ell]e$ , we mean  $\lambda X. \ell$  if  $X = Q$  else  $e(X)$ . (We use the notation  $\llbracket e \rrbracket \mathcal{P}$  for the semantics of  $e$  applied  $\mathcal{P}$ , and the notation  $e(Q)$  for the label the proposition  $Q$  is assigned to in the partition  $e$ .)

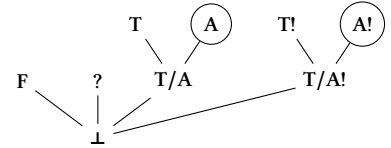
Next, we can define steps on these expressions. We found the following simple definition to be expressive enough for our purposes. Intuitively, these steps are either partition updates respecting the underlying strict partial order of labels or a sequence of such updates.

*Definition 4.2 (Steps).* A **step**  $\sigma$  is defined by  $\sigma ::= Q := \ell \mid \sigma_1 ; \sigma_2$  with the following semantics:

$$\frac{e(A) < \ell}{e \xrightarrow{Q := \ell} [Q := \ell]e} \text{ STEP/SET} \qquad \frac{e_1 \xrightarrow{\sigma_1} e_2 \quad e_2 \xrightarrow{\sigma_2} e_3}{e_1 \xrightarrow{\sigma_1 ; \sigma_2} e_3} \text{ STEP/SEQ}$$

### 4.1 Choosing a Label Set

We have one last question to answer for our choice of expressions: What label set should we choose for our partitions? Different applications and step providers will necessitate particular choices of label sets; we found the label set defined by the Hasse diagram to the right with assume-labels circled to be sufficient. The meanings of these labels are as follows:



- $\perp$  means the proposition is unlabeled.
- $F$  means the proposition should be false.  $?$  means the proposition may be false or true.  $T$  means the proposition should be true.  $T!$  means the proposition should be true and there exists a derivation of the goal that contains the proposition and all other  $T!$  and  $A!$  labels.
- $A$  and  $A!$  are the assume-labels, meaning the user is willing to assume those propositions as axioms, with the latter label additionally requiring there exists a derivation of the goal that contains the proposition and all other  $T!$  and  $A!$  labels, mirroring  $T!$ .
- $T/A$  and  $T/A!$  are intermediates that get refined to  $T$  or  $A$  (for  $T/A$ ) or  $T!$  or  $A!$  (for  $T/A!$ ).

*Remark 4.3.* The labels  $T!$  and  $A!$  capture the idea that the labeled proposition must be used in the proof of the goal. Although these labels complicate our theory and algorithms, we found them

to be powerful for both (i) expression of intent (they narrow the search space by effectively “locking in” on a certain proposition) and (ii) defining desired behavior of step providers, as in Section 6.

We take our starting expression  $e_0$  to be the partition with all propositions assigned to  $\perp$  (unseen), except the goal, which is assigned to  $T!$  (should be true, should not be assumed, and must be used).

Finally, we can define a notion of validity that satisfies the requirements of Definition 3.2.

*Definition 4.4 (Assume-partition Validity).* We say  $e$  **valid** for an assume-partition  $e$  over the above labels if the following conditions hold:

- (1)  $e(Q)$  is maximal for each  $Q \in \mathcal{A}$ .
- (2)  $\llbracket e \rrbracket \mathcal{P} \vdash Q$  for all  $Q \in e^{-1}[\{T, T!\}]$ .
- (3)  $\llbracket e \rrbracket \mathcal{P} \not\vdash Q$  for all  $Q \in e^{-1}[\{F\}]$ .
- (4) There exists a derivation of the goal  $G$  that contains all propositions in  $e^{-1}[\{T!, A!\}]$ .

## 4.2 Defining a Nonempty-Completion Oracle

Before defining step providers using our above choice of expressions and steps, we take one final detour. Lubin et al. [35] identified that an oracle that can determine whether the completion  $C(e) = \{e' \mid e \preceq e' \wedge e' \text{ valid}\}$  is nonempty was a crucial component of their step provider. We find this abstraction helpful in our setting as well and use it heavily for the remainder of the paper. We rely on compilation to SAT for our oracle. For an expression  $e$ , we define a boolean formula  $\varphi(e)$  such that  $\varphi(e)$  is satisfiable iff  $C(e)$  is nonempty. In addition to assigning truth values to propositions, the formula must also select a single “active” derivation tree that is required to contain all the  $T/A!$ ,  $T!$ , and  $A!$  nodes. We first define three variables per proposition  $Q \in \mathcal{A}$ :

- (1)  $\text{true}_Q$ , which is true iff  $Q$  should be true.
- (2)  $\text{assume}_Q$ , which is true iff the user is willing to assume (i.e., axiomatize)  $Q$ .
- (3)  $\text{active}_Q$ , which is true iff  $Q$  is on the active derivation tree (i.e., is “active”).

And two variables per rule  $R \in \mathcal{R}$ :

- (1)  $\text{true}_R$ , which is true iff  $R$  should be true (i.e., should fire).
- (2)  $\text{active}_R$ , which is true iff  $R$  is on the active derivation tree (i.e., is “active”).

For each proposition  $Q \in \mathcal{A}$ , we define *semantic* formulas that capture the semantics of the underlying proof system (standard AND–OR graph semantics), *activity* formulas that constrain the active derivation tree, and *consistency* formulas that constrain propositions based on their partition labels in the expression  $e$ . For each rule  $R \in \mathcal{R}$ , we have only semantic and activity formulas, as we do not track partition labels for rules. Our final formula  $\varphi(e)$  is a conjunction of all five of these categories of formula. The details of how to construct this formula are available in Appendix B.

Interestingly, all formulas except the consistency formula are constant in a given run of synthesis, so, as we will see in Section 8,  $\varphi(e)$  can be solved efficiently using incremental SAT solving.

## 5 Navigating Partitions

The nonempty completion oracle defined above makes it easy to define many different step providers to explore the space of expressions, including both new and existing exploration strategies. Due to the use of the oracle as a final check before emitting any step, all step providers we define below trivially satisfy **STRONG SOUNDNESS** (all provided steps lead to at least one valid solution). As we describe the step providers, we will mention which ones also satisfy **STRONG COMPLETENESS** (all valid expressions reachable from the current expression are reachable using some provided step).

We also note which of these step providers are *one-shot* (i.e., provide a single step to a valid expression) and which support *exploration* of the space of expressions (i.e., yield alternative, intermediate steps). Of the ones that yield more than one step, we will also note which step providers are

---

**Algorithm 1** REMAINING Step Provider

```

1: procedure REMAINING( $e$ )
2:   for  $Q \in \mathcal{A}$  do
3:     for  $\ell > e(Q)$  do
4:        $\sigma \leftarrow Q := \ell$ 
5:       if SAT( $\varphi(\sigma e)$ ) then yield  $\sigma$ 
    
```

---



---

**Algorithm 2** RANDOM Step Provider

```

1: procedure RANDOM( $e$ )
2:    $Q \leftarrow$  random element of  $e^{-1}[\{\ell \in L : \ell \text{ non-maximal}\}]$ 
3:   for  $\ell > e(Q)$  do
4:      $\sigma \leftarrow Q := \ell$ 
5:     if SAT( $\varphi(\sigma e)$ ) then yield  $\sigma$ 
    
```

---

what we will call *topological*; that is, which step providers are restricted to traversing the topology of the underlying AND–OR graph. All existing ways of navigating AND–OR graphs are topological; our assume-partition perspective supports both non-topological and topological step providers.

*REMAINING (Non-topological Exploration).* We begin with the simplest step provider for our setting, which we call REMAINING and define in Algorithm 1. For each proposition, REMAINING checks each possible label assignment in the partition  $e$  to see if the resulting partition has a nonempty completion; if so, it yields that assignment. This step provider is a direct adaptation of Classical–Constructive Synthesis by Lubin et al. [35, §4].

REMAINING trivially satisfies STRONG COMPLETENESS, as it yields every allowable assignment. However, the number of allowable assignments can be quite large—on the order of  $|\mathcal{A}| \times |L|$ . Beyond toy examples, so many possibilities shown at once would likely be overwhelming.

Nonetheless, REMAINING can be useful as a foundation for downstream post-processing (e.g., filtering, sorting, combining) the steps before display. Indeed, the step providers RANDOM, MAXINFOGAIN, CUT, and BOTTOMUPINVERSION that we describe later in this section will always provide a subset of the steps provided by REMAINING and could be implemented as filters on its output.

*RANDOM (Non-topological Exploration).* The next step provider we will consider is RANDOM, which we define in Algorithm 2. In contrast to REMAINING, RANDOM picks a *single* proposition (at random), then, like REMAINING, tries each possible label assignment in the partition  $e$ , yielding partitions with nonempty completions. Compared to REMAINING, RANDOM has the benefit of yielding only order  $|L|$  steps rather than  $|\mathcal{A}| \times |L|$  while still maintaining STRONG COMPLETENESS. As a downside, however, some of the propositions provided by REMAINING could, intuitively, be much more “informative” than the one randomly chosen by RANDOM.

*MAXINFOGAIN (Non-topological Exploration).* To combine some of the benefits of REMAINING and RANDOM into a single step provider, we define MAXINFOGAIN in Algorithm 3. MAXINFOGAIN operates identically to RANDOM except that the propositions are not randomly chosen, but rather chosen to maximize the *information gain* (or equivalently, minimize the *conditional entropy*).

We define conditional entropy  $H$  following the standard information-theoretic definition [16, §2.2], but with the entropy of possible completions only calculated with respect to semantically distinct completed expressions (i.e., equivalence classes of the kernel of  $\llbracket \cdot \rrbracket$ ):

$$H(e \mid Q) := \mathbb{E}_{\ell > e(Q)} [H(e \mid Q = \ell)] \quad \text{where} \quad H(e \mid Q = \ell) := \mathbb{E}_{e' \in C((Q=\ell)e)/\text{Ker} \llbracket \cdot \rrbracket} [-\log p(e')].$$

The first equation states that the entropy of  $e \mid Q$  (i.e.,  $e$  after assigning  $Q$ ) is the expected entropy of the possible assignments to  $Q$ . The entropy of  $e \mid Q = \ell$  (i.e.,  $e$  after assigning  $Q$  to  $\ell$ ) is defined as the expected *surprise* ( $-\log(\text{probability})$ ) of the semantically-distinct completions  $e'$  of the assignment.

---

**Algorithm 3** MAXINFOGAIN Step Provider

```

1: procedure MAXINFOGAIN( $e$ )
2:    $Q^* \leftarrow \arg \min_{Q \in e^{-1}[\{\ell \in L : \ell \text{ non-maximal}\}]} H(e \mid Q)$ 
3:   for  $\ell > e(Q^*)$  do
4:      $\sigma \leftarrow Q := \ell$ 
5:     if SAT( $\varphi(\sigma e)$ ) then yield  $\sigma$ 
    
```

---

**Algorithm 4** TOPDOWNINVERSION Step Provider

---

```

1: procedure TOPDOWNINVERSION( $e$ )
2:   yield all from COMMIT( $e$ )
3:   for  $Q \in e^{-1}[\{\text{T/A!}, \text{T!}\}]$  do
4:     for  $R \in \mathcal{R}$  with conclusion  $Q$  do
5:        $P \leftarrow$  premises of  $R$  with label  $\perp$ 
6:       if  $P$  empty then continue
7:        $\sigma \leftarrow P_1 := \text{T/A!}; \dots; P_j P_j := \text{T/A!}$ 
8:       if  $e(Q) = \text{T/A!}$  then  $\sigma \leftarrow \sigma; Q := \text{T!}$ 
9:       if SAT( $\varphi(\sigma e)$ ) then yield  $\sigma$ 

```

---

**Algorithm 5** BOTTOMUPINVERSION Step Provider

---

```

1: procedure BOTTOMUPINVERSION( $e$ )
2:   yield all from LEAF( $e$ )
3:   for  $Q \in e^{-1}[\{\text{F}\}]$  do
4:     for  $R \in \mathcal{R}$  with premise  $Q$  do
5:        $Q' \leftarrow$  conclusion of  $R$ 
6:       for  $\ell > e(Q')$  do
7:          $\sigma \leftarrow Q' := \ell$ 
8:         if SAT( $\varphi(\sigma e)$ ) then yield  $\sigma$ 

```

---

This general formulation allows for arbitrary prior probability to be placed on various propositions (first equation) and labels (second equation). However, assuming a uniform prior on  $e'$  means that  $p(e') = \frac{1}{|C((Q:=\ell)e)/\text{Ker}[\cdot]|}$ , and the second equation above simplifies to:  $H(e | Q = \ell) = \log |C((Q := \ell)e)/\text{Ker}[\cdot]|$ . We can then observe that the argmin in Algorithm 3 simplifies to

$$\begin{aligned} \arg \min_Q H(e | Q) &= \arg \min_Q \mathbb{E}_{\ell > e(Q)} [\log |C((Q := \ell)e)/\text{Ker}[\cdot]|] \\ &= \arg \min_Q \mathbb{E}_{\ell > e(Q)} [\log \text{ProjectedModelCount}_{\text{assume}}(\varphi((Q := \ell)e))], \end{aligned}$$

where  $\text{ProjectedModelCount}_{\text{assume}}(\phi)$  is the *projected model count* [2] of the formula  $\phi$  on the *assume* variables. (This projected model count is the number of assignments to the *assume* variables that have extensions to full assignments that satisfy  $\phi$ .) Fortunately, there are many off-the-shelf projected model counters (also called  $\#$ SAT solvers) that can be slotted into Algorithm 3 with no modification. Additionally, if desired, we can also approximate this argmin by using *approximate* projected model counters or place weight on certain models to adjust the prior probabilities using projected *weighted* model counters, both of which are available in off-the-shelf solvers.

MAXINFOGAIN demonstrates the use of conditional entropy as a ranking function to choose next propositions to show instead of randomly selecting them like in RANDOM, but any ranking function will suffice. MAXINFOGAIN (or any ranking-based solution) satisfies STRONG COMPLETENESS.

*CUT (Non-topological exploration)*. For example, we can define a different ranking function  $r_{\text{cut}}$  for a step provider we call CUT that works identically to MAXINFOGAIN but lexicographically minimizes this new ranking function instead of conditional entropy. For a proposition  $Q$ , we define  $r_{\text{cut}}(Q) = (r_1, r_2)$ , where  $r_1$  is 0 if  $\llbracket (Q := \text{A})e \rrbracket \mathcal{P}$  proves the goal and +1 otherwise, and  $r_2$  is the proportion of labels assignable to  $Q$  with nonempty completions that are *not* assume-labels (lower is better for both components). The first component determines if  $Q$  is a “cut” point (i.e, assuming the node will make the goal provable), and the second component incentivizes displaying propositions that are more likely to result in completing navigation (assuming equal prior probability of labels).

*TOPDOWNINVERSION (Topological exploration)*. The rest of our providers demonstrate how to support existing debugging strategies. We start by defining TOPDOWNINVERSION in Algorithm 4, which depends on an auxiliary step provider COMMIT defined in Algorithm 8 in Appendix C and enables top-down exploration of the underlying graph topology.

Momentarily skipping over line 2 of Algorithm 4, TOPDOWNINVERSION works by looping over all propositions marked with T/A! or T!, which we think of as the *exploration frontier*. These propositions will be the ones that are at the brim of where the user has explored in the AND-OR graph. For any of the propositions in the exploration frontier, the user can select a rule that provides the proposition to *invert* and further explore the premises of the rule. These premises are marked with T/A! as they will need to be either true or assumed on the selected path; the selected

---

**Algorithm 6** MINIMALLEAFHEURISTIC Step Provider
 

---

```

1: procedure MINIMALLEAFHEURISTIC( $e$ )
2:    $\mathcal{L} \leftarrow \{Q \in \mathcal{A} : Q \text{ not a conclusion of any rule in } \mathcal{R} \wedge e(Q) = \perp\}$ 
3:    $e' \leftarrow e$  with every proposition not in  $\mathcal{L}$  set pessimistically
4:   if  $\neg\text{SAT}(\varphi(e'))$  then return  $\emptyset$ 
5:    $M \leftarrow$  model of  $\varphi(e')$  with minimal number of assumeQ set to true for  $Q \in \mathcal{L}$ 
6:    $\mathcal{L}' \leftarrow \langle Q \in \mathcal{L} : M \models \text{assume}_Q \rangle$  (in arbitrary order)
7:   if  $\mathcal{L}'$  empty then return  $\emptyset$ 
8:   return  $\mathcal{L}'_1 := \mathbf{A}; \dots; \mathcal{L}'_{|\mathcal{L}'|} := \mathbf{A}$ 
    
```

---

proposition is also assigned to **T!** (since it can no longer be assumed).<sup>2</sup> The call to the SAT solver on line 9 can be omitted if the only step provider being used is `TOPDOWNINVERSION`. However, it is necessary for the step provider to maintain `STRONG SOUNDNESS` when composed with other step providers; other step providers could, for example, designate certain propositions as being false (**F**), which should preclude top-down navigation to them. Returning to line 2, the `COMMIT` step provider simply allows the committing of **T/A!**-labeled propositions (which are created on line 7 of `TOPDOWNINVERSION`, for example) to either **T!** or **A!** (and likewise for **T/A** propositions).

By design, `TOPDOWNINVERSION` does *not* satisfy `STRONG COMPLETENESS` (for example, propositions cannot be set to **F**); it replicates the existing strategy for exploring the topology of the underlying AND-OR graph in a top-down fashion. It does satisfy the weaker property that, for any valid expression  $e$ , it can navigate to an expression with the same set of **A!** nodes.

*BOTTOMUPINVERSION (Topological exploration).* We define `BOTTOMUPINVERSION` similarly to `TOPDOWNINVERSION` in Algorithm 5. `BOTTOMUPINVERSION` starts by showing all possible label assignments for any leaf propositions using the auxiliary `LEAF` provider we define in Algorithm 9 in Appendix C. Then, for each proposition that should be *false*, `BOTTOMUPINVERSION` considers each rule that has that proposition as a premise, and offers all label assignments of the conclusion of that rule. This captures the idea that traversing an AND-OR graph bottom-up starting at the leaves is essentially communicating that the proposition at hand should be false, but that the “parent” proposition is of interest to explore. `BOTTOMUPINVERSION` does not satisfy `STRONG COMPLETENESS`.

*MINIMALLEAFHEURISTIC (One-shot).* The final step provider we present is `MINIMALLEAFHEURISTIC`, which we define in Algorithm 6. This step provider emulates instance-based why-not provenance [25], as we discuss in Section 10.4.1. It proceeds by first collecting all the unseen leaf propositions, then sets the labels for all other propositions “pessimistically” to a maximal label in a new expression  $e'$ . (We provide details on pessimistic label setting in Appendix C.1, but, briefly,  $\perp$ -labeled propositions get set to  $\perp$ , **T/A**-labeled propositions to **T**, and **T/A!**-labeled propositions to **T!**). If the completion of  $e'$  is empty, then `MINIMALLEAFHEURISTIC` simply returns  $\emptyset$ . Otherwise, `MINIMALLEAFHEURISTIC` computes a model of  $\varphi(e')$  with a minimal number of `assume` nodes set to true, which corresponds to a minimally-sized set of leaf assumptions that make the goal proposition true. If this set of leaves is non-empty (i.e., some leaf propositions are actually required to be assumed), `MINIMALLEAFHEURISTIC` returns a step that assigns all these propositions to **A**.

Line 5 (computing the minimal model) can be done using an off-the-shelf MaxSAT solver with the optimization objective to minimize  $\sum_Q \text{assume}_Q$ . This can also simply be done (as we do in our

---

<sup>2</sup>In theory, choosing one of these steps forecloses the other options, preventing backtracking. In practice, we also provide a meta “undo” option (for `TOPDOWNINVERSION` and all other step providers) that simply reverts to a previous expression if it is desirable to explore a different part of the expression space.

implementation) via iterative deepening using a cardinality constraint of the form  $\sum_Q \text{assume}_Q \leq k$  for  $k = 0, \dots, k_{\max}$ , where  $k_{\max}$  is the number of `assume` variables set to true on line 4.

By design, `MINIMALLEAFHEURISTIC` does not satisfy `STRONG COMPLETENESS`; it replicates an existing heuristic strategy for arriving at *some* minimal set of leaf assumptions.

*Remark 5.1 (Composing step providers).* All step providers we define above can be composed with one another by running both step providers and displaying both outputs (i.e., taking the union of their results). For example, `MAXINFOGAIN` can be composed with `BOTTOMUPINVERSION` so that if a user selects `F` for a proposition provided by `MAXINFOGAIN` in some arbitrary location in the underlying `AND-OR` graph topology, then `BOTTOMUPINVERSION` will allow them to go “up” the topology *starting at* that proposition. Any of the step providers that do not satisfy `STRONG COMPLETENESS` can be composed in this manner with any of the step providers that *do* satisfy `STRONG COMPLETENESS` as an “escape hatch” to ensure all valid expressions are reachable.

## 6 STRONG COMPLETENESS MODULO OBSERVABILITY

It may seem like the fitness of a strategy depends entirely on the domain and application it is employed for. Interestingly, though, some step providers may be *strictly better* than others. To see why, we consider the question: Do we really need to be able to navigate to *any* expression? For example, when navigating assume-partitions, we may only care that we are able to label arbitrary propositions with the  $L_{\text{assume}}$  labels; the other labels have no effect on the semantics, after all.

This line of reasoning motivates a property we call `STRONG COMPLETENESS MODULO OBSERVABILITY`. For a given expression  $e$ , `STRONG COMPLETENESS` requires a provided step set  $\Sigma$  to satisfy  $\bigcup_{\sigma \in \Sigma} C(\sigma e) \supseteq C(e) \setminus \{e\}$ , or that all valid expressions reachable from  $e$  (except possibly  $e$  itself) are reachable via some step in in the set  $\Sigma$ . The key idea for `STRONG COMPLETENESS MODULO OBSERVABILITY` is to quotient the space of possible expressions by an *observability* equivalence relation  $\approx$  on expressions. Letting  $[e]_{\approx} = \{e' : e' \approx e\}$  denote the equivalence class of an expression  $e$  and  $[E]_{\approx} = \{[e]_{\approx} : e \in E\}$  denote the sets of equivalence classes of a set of expressions  $E$ , `STRONG COMPLETENESS MODULO OBSERVABILITY` instead requires that

$$\bigcup_{\sigma \in \Sigma} [C(\sigma e)]_{\approx} \supseteq [C(e) \setminus \{e\}]_{\approx} \quad (\text{STRONG COMPLETENESS MODULO OBSERVABILITY}).$$

Crucially, relaxing our notion of completeness from `STRONG COMPLETENESS` to `STRONG COMPLETENESS MODULO OBSERVABILITY` puts less of a burden on step providers in the sense that they need not provide as many steps. If the observability relation is chosen in such a way that equivalent expressions are truly equivalent for the purposes of debugging, then satisfying `STRONG COMPLETENESS MODULO OBSERVABILITY` but not `STRONG COMPLETENESS` will cut down the number of steps that need to be considered to arrive at a valid expression.

*Choosing an Observability Relation.* In general, we would like to choose an observability relation that is as coarse as is acceptable for the application at hand. Coarser observability relations will enable step providers to provide fewer steps, thus leading to fewer decisions. For example, if we have a semantics  $\llbracket e \rrbracket$  for expressions  $e$ , we may only be interested in being able to navigate to a single representative expression among the equivalence class of all expressions with the same semantics. In other words, we would want  $\llbracket e_1 \rrbracket = \llbracket e_2 \rrbracket$  to imply  $e_1 \approx e_2$ , as in traditional observational equivalence. But we can also consider coarser notions of equality.

For our assume-partition setting, traditional observational equivalence would require that the set of propositions labeled with  $L_{\text{assume}}$  are exactly the same in both partitions, *even if those propositions are never used in any proof of the goal*. We can instead consider the following notion of observability

that is coarser than traditional observational equivalence:

$$e_1 \approx e_2 \quad \text{iff} \quad \mathbf{Assumes}(e_1; \mathcal{P} \vdash G) = \mathbf{Assumes}(e_2; \mathcal{P} \vdash G),$$

where

$$\mathbf{Assumes}(e; \mathcal{P} \vdash G) = \{\{Q \in e^{-1}[L_{\text{assume}}] : Q \text{ appears in } P\} : P \text{ a proof of } G \text{ in } \llbracket e \rrbracket \mathcal{P} \text{ respecting } e\}$$

is the set of sets of assume-labeled propositions in some proof of  $G$  in  $\llbracket e \rrbracket \mathcal{P}$  that respects the labels of  $e$  in the sense of Definition 4.4. (In database provenance terminology,  $\mathbf{Assumes}(e; \mathcal{P} \vdash G)$  corresponds to a particular type of why-provenance of  $G$  in  $\llbracket e \rrbracket \mathcal{P}$  over assumed nodes.)

Using this notion of observability, the equivalence class  $[e]_{\approx}$  of an expression  $e$  is the set of expressions that have the same sets of “active” assume propositions (assume propositions that could actually be used in a proof of the goal that respects the constraints from  $e$ ).

*Remark 6.1.* As a rule of thumb, traditional observational equivalence is a good starting point for an observability relation (and may be sufficient). To coarsen the relation, one can gradually note which steps are irrelevant to the problem at hand, identify what makes them irrelevant, and build up the observability relation to rule out these cases. If there is a worry that an observability relation may be too coarse, the “irrelevant” steps can be shown in a special section of the output (e.g., at the bottom and dimmed), so that the user need only consider them in rare circumstances.

*Leveraging STRONG COMPLETENESS MODULO OBSERVABILITY.* Now, given any step provider that satisfies STRONG COMPLETENESS, we can perform *relevancy pruning* on its output step set  $\Sigma$  to get a new step set that satisfies (only) STRONG COMPLETENESS MODULO OBSERVABILITY by checking each step to see if it could result in a different equivalence class. This filtering operation will result in a step set that is either the same as  $\Sigma$  or strictly smaller.

Depending on the observability relation, it may be challenging to perform this relevancy pruning. It turns out that, in our context of assume-partitions, we can leverage all the machinery we have already introduced to perform this pruning in Algorithm 7. The key insight is to realize that, given a proof system  $\mathcal{P}$  and expression  $e$ , if a proposition  $Q$  cannot be set to  $A!$  (assumed and force-used), then

---

**Algorithm 7** Relevancy pruning
 

---

```

1: procedure RELEVANCYPRUNE( $\mathcal{A}, \Sigma$ )
2:    $\Sigma' \leftarrow \emptyset$ 
3:   for  $\sigma \in \Sigma$  do
4:     if  $\sigma = Q := \ell$  for some label  $\ell \in L \setminus \{A!\}$ 
5:       and  $\neg \text{SAT}(\varphi((Q := A!)e))$  then continue
6:      $\Sigma' \leftarrow \Sigma' \cup \{\sigma\}$ 
7:   return  $\Sigma'$ 
    
```

---

there cannot be any proof of the goal using the assumption  $Q$  that respects the partition  $e$ . Thus, setting  $Q$  in  $e$  cannot change  $\mathbf{Assumes}(e; \mathcal{P} \vdash G)$ , so it is not necessary for the step set  $\Sigma$  to contain any assignment of  $Q$ . Therefore, all we need to do in order to perform relevancy pruning is to filter out steps whose proposition cannot be assigned to  $A!$ , which is checkable using our SAT oracle.

## 7 Implementation

We implemented Programming by Navigation for AND–OR graph modifications in a system we call AO-NAV in approximately 3,000 lines of Rust code. AO-NAV is generic over the class of modifications, and we instantiate it to the space of partitions we describe in Section 4. We also implement all step providers defined in Section 5. For our SAT oracle, we use the RustSAT API [26] to access the CaDiCaL SAT solver [5]. To do projected model counting for MAXINFOGAIN, we use Ganak2 [44].

In our repository, we define a standard format for AND–OR graphs we call the AND–OR JSON Graph Format that is a refinement of the JSON Graph Format (JGF) v2 [4]. We use Argus [22] to extract AND–OR graphs for Rust trait errors to this format and a small custom parser to extract AND–OR graphs from the exploration traces of the Aesop proof search tool [31] for the Lean proof assistant [36]. We also implemented a small tool to obtain AND–OR graphs from Datalog programs

using a subset of egglog syntax [51]; as AND–OR graphs are quantifier-free, we simply ground all Datalog rules using the literals found in the program. We hope a standardized format will encourage the programming languages community to build new AND–OR tools beyond AO-NAV and new AND–OR extractors beyond the ones we mention in this section.

## 8 Evaluation

We use AO-NAV to explore the following empirical research questions. **RQ1:** How much does STRONG SOUNDNESS reduce the number of decisions? **RQ2:** How much does relevancy pruning reduce the number of decisions? **RQ3:** What is the performance cost of STRONG SOUNDNESS and relevancy pruning? **RQ4:** Overall, how do our step providers perform on our benchmarks? **RQ5:** How do our step providers empirically scale? **RQ6:** How much does incremental SAT solving help?

*Benchmark Creation.* We are not aware of an existing suite of AND–OR graphs whose goal nodes should be true but are not. Therefore, we created a publicly-available benchmark suite of 284 AND–OR graphs with underivable goals (see **Data Availability Statement**).

These graphs fall into four categories: MANUAL, RANDOM, ARGUS, and AESOP. We constructed 15 MANUAL graphs by hand, akin to the first AND–OR graph in Section 2. We automatically generated 180 RANDOM graphs ranging in size from 10 to 200 OR nodes, where the number of children for each node was random and the maximum number of children allowed per node ranged from 2–10. We generated 51 ARGUS examples by extracting Rust trait errors from Gray et al. [22]’s Rust programs using their Argus tool. These Rust programs include tasks Gray et al. [22] used for their user study that involve real Rust libraries such as the web framework Axum and the SQL query builder Diesel. We generated 38 AESOP examples by extracting proof search traces from all Lean programs in the Aesop test suite that Aesop failed to prove after searching at least one rule. Because many sub-nodes of the ARGUS and AESOP graphs were already provable, we pruned them using unit propagation first. Table 1 in Appendix D provides summary statistics for our benchmarks.

For each AND–OR graph, we also created a set of 3 *purely synthetic* partitions of the nodes that represent possible solutions (i.e., valid expressions) that make the goal proposition true by randomly selecting steps using the REMAINING step provider (which can navigate to any partition due to STRONG COMPLETENESS). We then removed assume-labeled propositions from the 3 solutions until no assumptions were redundant so that none of the step providers would need to steer to partitions with redundant assumptions (which may not be possible with relevancy pruning).

*Measures.* We used the ground-truth partitions (see above) as step deciders for the PBN process, automatically selecting (at each round) the first provided step consistent with the ground-truth partition until the partition proved the goal. We repeated this process for each ground-truth partition 3 times (all results are averaged over 3 replicates and 3 solutions). For each AND–OR graph, we recorded the **decision count** (the sum of the indexes of the selected step in whatever order the step provider yields the steps), the **round count** (number of rounds), the **duration** (total time taken), and the **median latency** (median time taken to return each round’s results).

*Step Providers.* To answer **RQ1–3**, we use a variant of the RANDOM step provider that selects propositions in alphabetical order (which we call ALPHABETIC) to ensure consistency across conditions and runs. (To avoid any spurious interactions with node names, we randomize all node names from all AND–OR graphs in our benchmark.) We consider three variants of ALPHABETIC: one in which the oracle check is turned off (no STRONG SOUNDNESS, abbreviated **U**), one in which the oracle check is turned on (STRONG SOUNDNESS as usual, abbreviated **S**), and one in which we also perform relevancy pruning afterward (abbreviated **S+R**). For **RQ4–5**, we additionally include

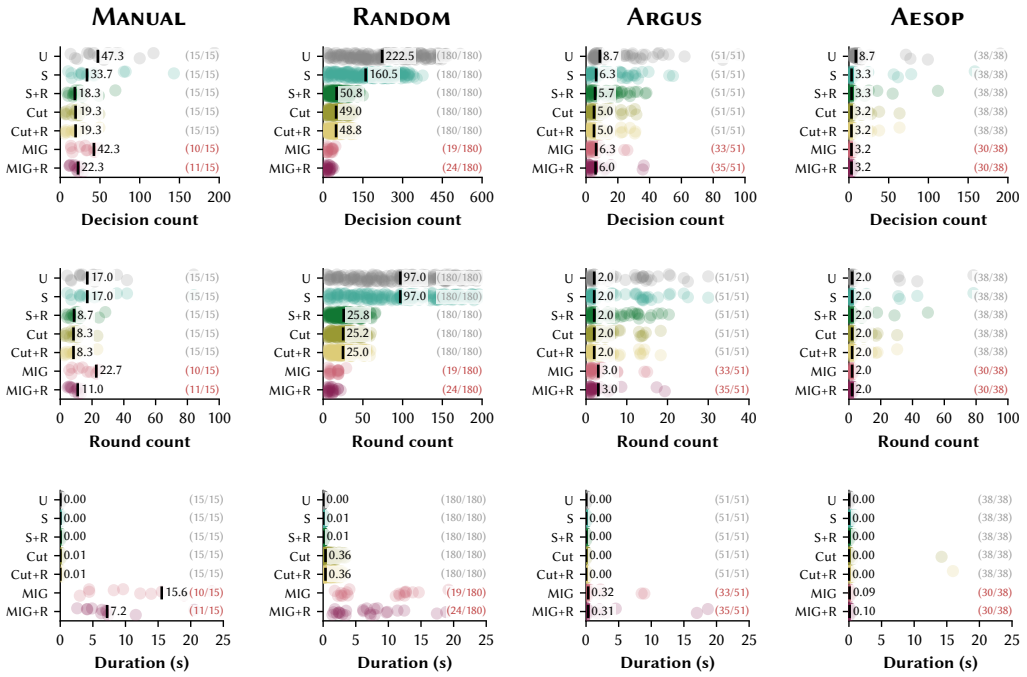


Fig. 2. Results for each step provider. Decision count is the sum of the index of each step at each round. Round count is the total number of rounds. Duration is the total duration. Labeled bars appear at the median when it can be computed. (Median latency results are in Figure 6 in Appendix D.)

MAXINFOGAIN (abbreviated **MIG**) and **Cut** (**Cut**), both with relevancy pruning (+**R** suffix) and without. For **RQ6**, we also ran all step providers without incremental SAT solving.

*Additional Information.* We ran the evaluation on a system running macOS Tahoe 26.31.1 (a) with an Apple M4 Pro CPU and 48 GB of RAM. All tasks ran on one core with a 30-second cutoff.

### 8.1 Results

Figure 2 summarizes each step provider’s performance across the four suites and three of the four metrics (median latency results are available in Figure 6 in Appendix D for brevity). Figure 3 also includes comparisons between each step provider and the unsound step provider (**U**), as well as a comparison between **ALPHABETIC** with (**S+R**) and without (**S**) relevancy pruning.

In the following sections, we report data as median (IQR).

**8.1.1 RQ1: How much does STRONG SOUNDNESS reduce the number of decisions?** Figure 3 shows that the introduction of **STRONG SOUNDNESS** resulted in  $0.71\times$  ( $0.64\times-0.75\times$ ) decisions for **MANUAL**,  $0.72\times$  ( $0.66\times-0.76\times$ ) decisions for **RANDOM**,  $0.61\times$  ( $0.33\times-0.76\times$ ) decisions for **ARGUS**, and  $0.31\times$  ( $0.29\times-0.73\times$ ) decisions for **AESOP** compared to the unsound condition.

**8.1.2 RQ2: How much does relevancy pruning reduce the number of decisions?** Figure 3 shows that, compared to **STRONG SOUNDNESS** alone, the further introduction of relevancy pruning resulted in  $0.61\times$  ( $0.54\times-0.73\times$ ) decisions for **MANUAL**,  $0.36\times$  ( $0.28\times-0.47\times$ ) decisions for **RANDOM**,  $0.96\times$  ( $0.82\times-1.00\times$ ) decisions for **ARGUS**, and  $1.00\times$  ( $1.00\times-1.00\times$ ) decisions for **AESOP**. Compared

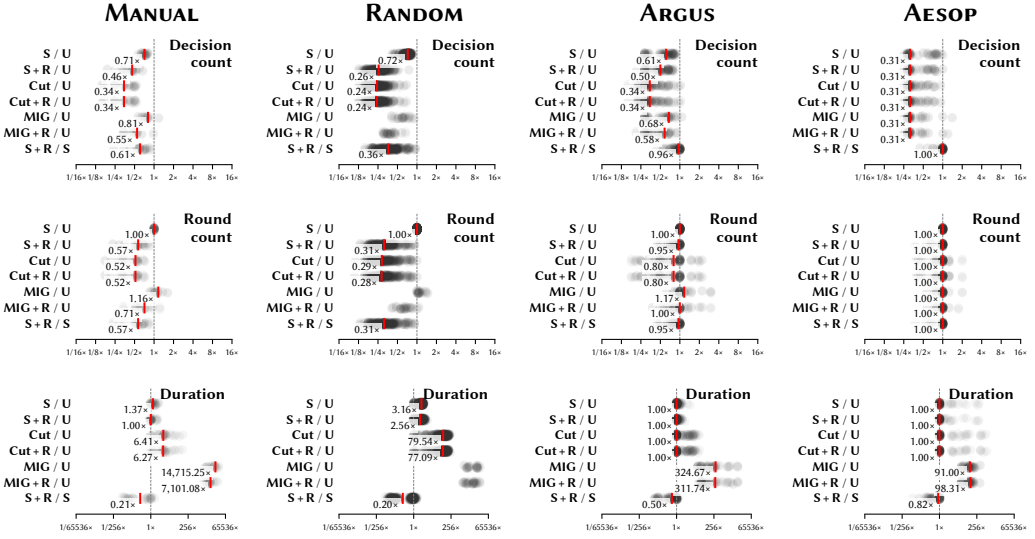


Fig. 3. **Comparative results for each step provider.** Each row in each plot compares against the unsound condition, except the last row, which ablates relevancy pruning. **Decision count** is the sum of the index of each step at each round. **Round count** is the total number of rounds. **Duration** is the total duration. Labeled bars appear at the median when it can be computed. (**Median latency** results are in Figure 6 in Appendix D.)

to the unsound condition, relevancy pruning resulted in  $0.46\times$  ( $0.34\times$ – $0.52\times$ ) decisions for MANUAL,  $0.26\times$  ( $0.21\times$ – $0.31\times$ ) decisions for RANDOM,  $0.50\times$  ( $0.33\times$ – $0.66\times$ ) decisions for ARGUS, and  $0.31\times$  ( $0.29\times$ – $0.59\times$ ) decisions for AESOP. Figure 7 in Appendix D shows that larger AND–OR graphs benefit more from relevancy pruning (Spearman’s  $\rho = -0.90$  between  $S+R/S$  and OR node count), suggesting that the varying effectiveness of relevancy pruning on the suites may be due to their varying composition of AND–OR graph sizes (but this correlation does not imply causation).

**8.1.3 RQ3: What is the performance cost of STRONG SOUNDNESS and relevancy pruning?** Figure 3 shows that the requirement of STRONG SOUNDNESS resulted in slowdowns, causing AO-NAV to have  $1.37\times$  ( $1.00\times$ – $1.89\times$ ) duration for MANUAL,  $3.16\times$  ( $2.44\times$ – $3.63\times$ ) duration for RANDOM,  $1.00\times$  ( $1.00\times$ – $1.22\times$ ) duration for ARGUS, and  $1.00\times$  ( $1.00\times$ – $1.10\times$ ) duration for AESOP. Compared to STRONG SOUNDNESS alone, adding relevancy pruning resulted in the same median latency (Figure 6 in Appendix D), but a less-severe slowdown— $1.00\times$  ( $1.00\times$ – $1.11\times$ ) duration for MANUAL,  $2.56\times$  ( $1.78\times$ – $3.45\times$ ) duration for RANDOM,  $1.00\times$  ( $1.00\times$ – $1.27\times$ ) duration for ARGUS, and  $1.00\times$  ( $1.00\times$ – $1.11\times$ ) duration for AESOP compared to the unsound condition—due to less overall work.

**8.1.4 RQ4: Overall, how do our step providers perform on our benchmarks?** Figure 3 shows that, for decision and round count metrics, STRONG SOUNDNESS with relevancy pruning matches the effectiveness of more sophisticated strategies (CUT and MAXINFOGAIN) while taking less time. Figure 6 in Appendix D shows that, aside from MAXINFOGAIN, all step providers had a median latency of  $<20$  ms (for all suites except RANDOM, MAXINFOGAIN had a median latency of  $<1$  s). Table 2 in Appendix D shows that the *maximum* latency among all step providers and all suites was  $<1.1$  s, except for MAXINFOGAIN for which it was not possible to compute a maximum (it did not finish all tasks within the timeout).

### 8.1.5 RQ5: How do our step providers empirically scale?

Figure 4 shows that the ALPHABETIC step providers scale the best, then the CUT providers, then the MAXINFOGAIN providers. STRONG SOUNDNESS and relevancy pruning make ALPHABETIC slower on larger AND-OR graphs, as they both rely on SAT solver calls while the unsound condition does not. CUT without relevancy pruning already has to determine which nodes can be set to **A!**, so the overhead of relevancy pruning is negligible. For MAXINFOGAIN, model counting dominates the runtime, so relevancy pruning provides substantial improvements.

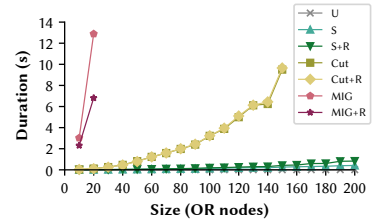


Fig. 4. Step provider execution time (30-second cutoff) vs. AND-OR graph size. Each point is the average of all AND-OR graphs of a given size in our RANDOM suite.

8.1.6 RQ6: How much does incremental SAT solving help? Figure 8 in Appendix D shows that incremental SAT solving results in large speedups for the step providers for which SAT solving is the bottleneck. Incremental SAT solving causes ALPHABETIC with STRONG SOUNDNESS to have  $0.06\times$  ( $0.05\times$ – $0.08\times$ ) duration and  $0.33\times$  ( $0.31\times$ – $0.43\times$ ) median latency, ALPHABETIC with STRONG SOUNDNESS and relevancy pruning to have  $0.04\times$  ( $0.03\times$ – $0.08\times$ ) duration and  $0.18\times$  ( $0.08\times$ – $0.33\times$ ) median latency, CUT to have  $0.06\times$  ( $0.04\times$ – $0.13\times$ ) duration and  $0.06\times$  ( $0.04\times$ – $0.18\times$ ) median latency, and CUT with relevancy pruning to have  $0.06\times$  ( $0.04\times$ – $0.13\times$ ) duration and  $0.06\times$  ( $0.05\times$ – $0.19\times$ ) median latency. The unsound condition does not use SAT solving (resulting in no speedup) and MAXINFOGAIN did not complete enough benchmark entries to compute a median.

## 8.2 Threats to Validity

Our decision counts are likely biased in favor of the unsound condition, as in our evaluation setup we assume the step decider never chooses steps that could force an invalid partition. This may threaten external validity, as it is likely that real users of a debugging tool will not always be able to pick a step that is guaranteed to lead to a valid solution in the absence of STRONG SOUNDNESS. We also created our own benchmark suite, which may threaten our internal validity. We mitigated this risk by including randomly-generated AND-OR graphs and AND-OR graphs from real Rust (Gray et al. [22]’s repository) and Lean (Limperg and From [31]’s repository) programs. Furthermore, although we include AND-OR graphs from real Rust and Lean programs, our solution partitions are randomly generated, which may threaten external validity. Lastly, our evaluation does not measure the time it takes a step decider to make a decision among the provided steps; the durations reported are purely timings of the step provider. This may threaten construct validity, as, for a real debugging tool, we are more interested in the overall time it takes a user to fix a bug. In particular, our metrics (decision count, round count, duration, latency) do not capture qualitative differences in the order in which questions are asked of the user, which could lead to differences in user performance that may make one step provider more suitable for a domain than another.

## 9 Limitations

Our approach aims to answer the question “*Why isn’t this proposition true?*” in a proof system. There are many related questions that we may wish to ask of particular proof systems, such as “*How do I need to change my function’s type signature to make it compile?*” or “*Why is the proof system proving this proposition that should not be true?*” that we do not directly support. For example, for the former question, in the context of Rust trait errors, the goal proposition in the trait proof system does not hold *because it should not hold*. There is no amount of implementing traits or anything else that the user could do to the proof system itself that would reasonably make the proof go through, because the goal is simply the wrong goal. Therefore, while our assume-partition

approach may indirectly help a user realize what they need to do to fix function signature, it will not directly support navigating to a fix. An exciting future direction would be to extend our notion of expressions to allow modifying the nodes of the proof system itself (e.g., wrapping the goal in another type that does or should implement the desired trait).

## 10 Related Work

A central dichotomy in the literature on debugging logic programs is whether to work with (i) proof *trees* of derived facts (AND graphs) or (ii) proof *systems* capturing the entire search space (AND-OR graphs). In the early 1990s, Senay and Lazzeri [41] pointed out:

Even though a proof tree gives a brief account of the solution steps performed during the computation, it only does so for the goals that have been solved in the process. However, it presents neither the goals that have been tried but unsolved nor the alternative clauses that could have been used to find an alternative solution.

In this paper, we tackle the problem of answering why a proposition does *not* hold. Although AND-OR graphs are harder to reason about than simple AND graphs, AND graphs struggle to answer why a particular proposition does *not* hold. For instance, Pacak and Erdweg [37] argue for why provenance-based debugging (i.e., debugging AND graphs) is not always enough to debug Datalog, notably that when a fact does not hold “there is no derivation tree that provenance-based debugging could provide to the user.” Further, we are interested in *non-topological* debugging; that is, debugging beyond the TOPDOWNINVERSION and BOTTOMUPINVERSION strategies that are directly tied to the topology of the underlying AND-OR graph.

### 10.1 Topological Debugging of AND-OR graphs

AND-OR graphs have been of great interest to the logic programming community due to the fact that they naturally capture the execution model of logic programming languages such as Prolog. As early as the 1980s, researchers have explored how best to produce and display AND-OR graphs corresponding to logic programs, starting with the seminal work on the DEWLAP system [17]. As AND-OR graphs of logic programs can be large and complex, the central thrust of research on these program visualization devices have been on techniques that abstract the underlying graph [18, 19], sort the entries of the graph [22], or otherwise result in visually compressed display of the information [22, 41]. Existing work either displays the entirety of the AND-OR graph [17–19, 41] (sometimes with sophisticated zooming strategies [19]), or allows iteratively traversing the graph using top-down or bottom-up strategies, analogous to our TOPDOWNINVERSION and BOTTOMUPINVERSION step providers [22]. In any case, these approaches are committed to displaying or traversing the topology of the AND-OR graph. In other words, the space they enable navigating over is the space of nodes and edges in the AND-OR graph.

In summary, existing techniques to debug AND-OR graphs use either TOPDOWNINVERSION or BOTTOMUPINVERSION, or display the entire graph. We demonstrate strategies to debug AND-OR graphs beyond these techniques and are excited for future work to explore a wide design space of proof system transformations that enable new kinds of debugging strategies.

### 10.2 Non-topological Debugging of AND graphs

In parallel with the techniques described in the previous subsection, there has been an explosion of literature on *algorithmic debugging*, starting with Shapiro [42]’s seminal thesis in 1983. These techniques debug individual programs or proofs (corresponding to single AND graphs) and are thus not immediately applicable to debugging AND-OR graphs. Nonetheless, algorithmic debugging researchers have explored an extensive number of non-topological debugging strategies.

Algorithmic debugging systems work in two phases. First, the system captures a *computational tree* (akin to an AND graph) of the execution of a program that summarizes the inputs and outputs

of the procedures called during execution of the program (via source-to-source program transformations or an instrumented interpreter or compiler). Second, the system queries the user about the validity of various input-output pairs for the called procedures (for example, if a call to function  $f$  on the input 3 returned output 4, the system may ask if  $f(3) = 4$  is desired behavior for  $f$ ).

This technique has been explored in a variety of programming paradigms, as surveyed by Caballero et al. [11]. For example, several works have provided algorithmic debugging of Hindley-Milner type conflicts [15, 45], for which a type conflict provides two full AND graphs of derived types for each conflicting expression. (Rust trait errors do not have AND graphs, because the type simply does not implement the trait.) The most closely related setting to ours is one considered by Caballero et al. [8] (later formalized by Caballero et al. [9]). They present a system to algorithmically debug Datalog programs using a data structure they introduce called the *computation graph*, a graph whose nodes represent query results and whose edges represent dependence. If a fact is missing from a the goal query (corresponding to our goal proposition being false), the user can mark the node as “invalid” as part of algorithmic debugging; however, they cannot explore the branches that *would have* produced this fact, a general limitation of approaches based on AND graphs. Later work on debugging SQL views took a similar approach [10].

Algorithmic debugging has also been explored with many user-querying strategies, as surveyed by Silva [43]. For example, Shapiro [42]’s thesis identified two intuitive strategies: single-stepping (post-order traversal of the computation graph starting at the leaves), and divide-and-query, which repeatedly divides the computation tree in two (reminiscent of binary search) to identify faulty nodes, in a strategy akin to the later-coined delta debugging [48]. Our MAXINFOGAIN step provider can be thought of as a analogue of divide-and-query generalized along two axes: (i) it works for AND-OR graphs, and (ii), it works with more labels than “valid” and “invalid.”

These strategies enable highly non-topological debugging techniques and can provide an important source of inspiration for debugging AND-OR graphs; we hypothesize that many of the proposed strategies can be adapted beyond the context of program executions to a setting such as constraint systems. However, algorithmic debugging techniques thus far operate over the equivalent of AND graphs in their domains; that is, no existing algorithmic debugging technique works for general AND-OR graphs. For example, Arora et al. [1] present an interface EXPLAIN to view provenance (AND graph) of queries, but write that “In general, a weakness of EXPLAIN is that it does not provide direct support for understanding why some facts are ‘missing’, i.e., why some facts that a user expects to be generated are not.” Such information is only possible when inspecting AND-OR graphs. Additionally, these approaches are typically used to find a *single* faulty node rather than, for example, a set of missing nodes, reflecting their origins as strategies for debugging programs rather than proof systems. (Such systems can, of course, be used iteratively to find multiple faults, but in the context of debugging a proof system, one may not want to commit to a particular missing proposition without seeing what other sets of propositions could work.)

### 10.3 Debugging Constraint Systems by Debugging Their Solvers

Of the difficulty in debugging constraint systems, Hempel [24] has noted that:

The issue may be that, although concise, constraint-based representations obscure the answer to the question, “When X occurs, what happens next and in what order?” With ordinary, straight-line code, one can answer that question by walking the code. In contrast, a trigger-action constraint system is almost necessarily concurrent, and constraint solvers don’t necessarily explain how they intend to satisfy the constraints.

One solution to this problem is thus to sidestep debugging the constraint system itself, and instead debug the *solver* of the constraint system, which is typically a normal program. Dating back to the highly influential Byrd box model of Prolog [7], a variety of strategies have been proposed for tracing constraint solvers. The dominant such strategy is textual logging (as found, for example,

in SWI Prolog [20]), but stepwise debugger-like interfaces have been proposed as well, including for Datalog [37, 38]. These techniques provide a complementary route to debugging constraint systems with a unique tradeoff: Debugging can now proceed using all the niceties of a traditional sequential debugger, but the program being debugged (i.e., the solver) is quite far abstracted from the proof system at hand, with its own idiosyncrasies and semantics.

## 10.4 Other Relevant Topics

**10.4.1 Datalog Provenance.** In the context of Datalog, AND graphs can be created from a variety of techniques that fall under the umbrella of *provenance*, which is unified under the Green et al. [23]’s seminal description of semiring provenance. Most closely related to our notion of AND-OR partitions are the notions of *how-provenance* (the set of derivation trees of a proposition, which Köhler et al. [28], for example, efficiently compute and apply to Datalog debugging), *why-provenance* [6] (minimal sets of facts that establish the truth of a proposition) and *knowledge base support* [3, 13] (minimal sets of rules that establish the truth of a proposition).

As a form of AND-based debugging, traditional provenance-based approaches inherit the limitation that they cannot ask “why-not” questions; indeed, the popular Datalog engine Soufflé provides a separate *explainnegation* from their standard *explain* primitive for computing provenance, noting that, while their system does support explaining negations, “In this case, proof annotations are not helpful, since they cannot describe non-existent tuples” [52]. This *explainnegation* primitive is implemented based on efficient techniques by Lee et al. [29] and provides a user interface analogous to our `TOPDOWNINVERSION` strategy instantiated for Datalog.

A separate but related line of work to address the “why-not” question is known in the databases literature as *why-not provenance*. In this body of work, there are two main categories of answer to the “why-not” question [30]: query-based answers and instance-based answers. Query-based why-not provenance, first proposed by Chapman and Jagadish [12], is the set of components of a query that filter out a datapoint from the results; such answers are only applicable in the context of a data point going through a set of operations. Instance-based why-not provenance, first proposed by Huang et al. [25], is a set of ground facts that, if true, would make the goal proposition true; such sets correspond to the results provided by our `MINIMALLEAFHEURISTIC` step provider.

**10.4.2 Type Error Localization.** One application of our AND-OR navigation is Rust trait error localization, which is related to the body of work on identifying source locations to blame for type errors in programs. As we discuss in Section 10.2, one line of this work draws explicitly on algorithmic debugging (exploring AND trees) from constraint-based type information [15, 45]. However, many approaches diverge from algorithmic debugging, from Wand [47] and Johnson and Walz [27]’s classic works on identifying possible and likely (respectively) error source locations, to more modern works using Bayesian reasoning [49, 50], machine learning [40], and domain-specific ranking strategies [22]. Following Wand [47], some approaches find non-heuristic conflicting sets of locations [46], which can, for example, be localized by clever construction of the underlying type system [53] or explored with a graphical user interface [21]. Other heuristic approaches include searching for minimal program edits to get type-correct program to serve as explanation [14, 39], a topic explored in a later chapter in Shapiro [42]’s thesis on algorithmic debugging.

Such approaches only make sense when there is an underlying program to modify. In contrast, our approach is agnostic to the source of the AND-OR graph. Accordingly, a benefit of our approach is that it works in settings where there is no underlying program at all, such as proof search, but a drawback is that it may miss out on information available in the source program, such as how costly certain edits may be to make [22]. Incorporating such domain-specific strategies as step providers in our framework is an exciting direction for future work.

## Data Availability Statement

AO-NAV and the benchmarks we created for this paper are open source and freely available at <https://github.com/justinlubin/aonav>. We also provide a Zenodo archive of AO-NAV's source code (including our benchmarks) [32] and a Zenodo archive of a Docker image for our evaluation [33, 34].

## Acknowledgments

We are grateful to our anonymous PLDI reviewers for their feedback. We are also grateful to Ishan Gaur for sharing helpful intuition about information-theoretic entropy. We would also like to thank the developers of Argus and Aesop programming tools for their excellent contributions to the Rust and Lean communities. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 175281 and is supported in part by NSF grants FW-HTF 2129008 and CA-HDR 2033558, as well as by gifts from Google, G-Research, Adobe, and Microsoft. Chasins is a Chan Zuckerberg Biohub Investigator.

## References

- [1] Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. 1993. Explaining Program Execution in Deductive Systems. In *International Conference on Deductive and Object-Oriented Databases (DOOD)*. doi:10.1007/3-540-57530-8\_7
- [2] Rehan Abdul Aziz, Geoffrey Chu, Christian Muise, and Peter Stuckey. 2015. #SAT: Projected Model Counting. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. doi:10.1007/978-3-319-24318-4\_10
- [3] Franz Baader, Rafael Peñaloza, and Boontawee Sontisriravaporn. 2007. Pinpointing in the Description Logic  $\mathcal{EL}^+$ . In *German Conference on Artificial Intelligence (KI)*. doi:10.1007/978-3-540-74565-5\_7
- [4] Anthony Bargnesi, Anselmo DiFabio, William Hayes, Georgiy Shibaev (@RangerMauve on Github), Cristophe Benz, Hugh Pyle, @erik470 on Github, and Travis Giggy. 2021. JSON Graph Format (JGF). <https://web.archive.org/web/2025110001441/https://jsongraphformat.info/>. Accessed: 2025-11-09.
- [5] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. 2024. CaDiCaL 2.0. In *International Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-031-65627-9\_7
- [6] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *International Conference on Database Theory (ICDT)*. doi:10.1007/3-540-44503-X\_20
- [7] Lawrence Byrd. 1980. Understanding the Control Flow of Prolog Programs. In *Logic Programing Workshop*.
- [8] R. Caballero, Y. Garcia-Ruiz, and F. Sáenz-Pérez. 2008. A New Proposal for Debugging Datalog Programs. In *International Workshop on Functional and (Constraint) Logic Programming (WFLP)*. doi:10.1016/j.entcs.2008.06.035
- [9] R. Caballero, Y. Garcia-Ruiz, and F. Sáenz-Pérez. 2008. A Theoretical Framework for the Declarative Debugging of Datalog Programs. In *International Workshop on Semantics in Data and Knowledge Bases (SDKB)*. doi:10.1007/978-3-540-88594-8\_8
- [10] Rafael Caballero, Yolanda García-Ruiz, and Fernando Sáenz-Pérez. 2012. Declarative Debugging of Wrong and Missing Answers for SQL Views. In *International Symposium on Functional and Logic Programming (FLOPS)*. doi:10.1007/978-3-642-29822-6\_9
- [11] Rafael Caballero, Adrián Riesco, and Josep Silva. 2017. A Survey of Algorithmic Debugging. In *ACM Computing Surveys (CSUR)*. doi:10.1145/3106740
- [12] Adriane Chapman and H. V. Jagadish. 2009. Why Not?. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. doi:10.1145/1559845.1559901
- [13] Akira Charoensit, David Carral, Pierre Bisquert, Lucas Rouquette, and Federico Ulliana. 2024. Rule-Aware Datalog Fact Explanation Using Group-SAT Solver. In *RuleML+RR*.
- [14] Sheng Chen and Martin Erwig. 2014. Counter-Factual Typing for Debugging Type Errors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2535838.2535863
- [15] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. doi:10.1145/507635.507659
- [16] Thomas M. Cover and Joy A. Thomas. 1991. *Elements of Information Theory*. John Wiley & Sons, Inc. doi:10.1002/047174882X
- [17] Alan D. Dewar and John G. Cleary. 1986. Graphical Display of Complex Information within a Prolog Debugger. In *International Journal of Man-Machine Studies*. doi:10.1016/S0020-7373(86)80020-7
- [18] Mireille Ducassé. 1999. Opium: An Extendable Trace Analyzer for Prolog. In *The Journal of Logic Programming*. doi:10.1016/S0743-1066(98)10036-5

- [19] Marc Eisenstadt and Mike Brayshaw. 1988. The Transparent PROLOG Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. In *The Journal of Logic Programming*. doi:10.1016/0743-1066(88)90001-5
- [20] Thom Fruehwirth, Jan Wielemaker, and Leslie De Koninck. 2012. SWI Prolog Reference Manual 6.2.2. <https://web.archive.org/web/20251009164929/https://eu.swi-prolog.org/pldoc/man?section=debugoverview>. Accessed 2025-11-08.
- [21] Shuai Fu, Tim Dwyer, Peter J. Stuckey, Jackson Wain, and Jesse Linossier. 2023. ChameleonIDE: Untangling Type Errors Through Interactive Visualization and Exploration. In *International Conference on Program Comprehension (ICPC)*. doi:10.1109/ICPC58990.2023.00029
- [22] Gavin Gray, Will Crichton, and Shriram Krishnamurthi. 2025. An Interactive Debugger for Rust Trait Errors. In *Proc. ACM Program. Lang., Issue PLDI*. doi:10.1145/3729302
- [23] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*. doi:10.1145/1265530.1265535
- [24] Brian Hempel. 2019. Compendium of OS and PL working hypotheses and questions. <https://web.archive.org/web/20191102080532/https://people.cs.uchicago.edu/~brianhempel/compendium.html>. Accessed: 2025-11-08.
- [25] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the Provenance of Non-Answers to Queries over Extracted Data. In *Proceedings of the VLDB Endowment*. doi:10.14778/1453856.1453936
- [26] Christoph Jabs. 2025. RustSAT: A Library For SAT Solving in Rust. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. doi:10.4230/LIPIcs.SAT.2025.15
- [27] Gregory F. Johnson and Janet A. Walz. 1986. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/512644.512649
- [28] Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis. 2012. Declarative Datalog Debugging for Mere Mortals. In *International Datalog 2.0 Workshop*. doi:10.1007/978-3-642-32925-8\_12
- [29] Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2018. Provenance Summaries for Answers and Non-Answers. In *Proceedings of the VLDB Endowment*. doi:10.14778/3229863.3236233
- [30] Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2019. PUG: A Framework and Practical Implementation for Why and Why-Not Provenance. In *The VLDB Journal*. doi:10.1007/s00778-018-0518-5
- [31] Jannis Limperg and Asta Halkjær From. 2023. Aesop: White-Box Best-First Proof Search for Lean. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. doi:10.1145/3573105.3575671
- [32] Justin Lubin, Marlena Preigh, Max Willsey, and Sarah E. Chasins. 2026. AO-Nav Source Code. Zenodo. doi:10.5281/zenodo.19007039
- [33] Justin Lubin, Marlena Preigh, Max Willsey, and Sarah E. Chasins. 2026. Docker Image for the Evaluation of “Navigating AND-OR Graph Modifications to Debug Failing Proof Search” (Artifact Evaluation Version). Zenodo. doi:10.5281/zenodo.19665844
- [34] Justin Lubin, Marlena Preigh, Max Willsey, and Sarah E. Chasins. 2026. Docker Image for the Evaluation of “Navigating AND-OR Graph Modifications to Debug Failing Proof Search” (Latest Version). Zenodo. doi:10.5281/zenodo.19006570
- [35] Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. In *Proc. ACM Program. Lang., Issue PLDI*. doi:10.1145/3729264
- [36] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *International Conference on Automated Deduction (CADE)*. doi:10.1007/978-3-030-79876-5\_37
- [37] André Pacak and Sebastian Erdweg. 2023. Interactive Debugging of Datalog Programs. In *Proc. ACM Program. Lang., Issue OOPSLA2*. doi:10.1145/3622824
- [38] Francesco Russo and Mirko Sancassani. 1992. A Declarative Debugging Environment for DATALOG. In *Russian Conference on Logic Programming*. doi:10.1007/3-540-55460-2\_32
- [39] Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3386005
- [40] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame: Localizing Novice Type Errors with Data-Driven Diagnosis. In *Proc. ACM Program. Lang., Issue OOPSLA*. doi:10.1145/3138818
- [41] H. Senay and S. G. Lazzeri. 1991. Graphical Representation of Logic Programs and Their Behaviour. In *IEEE Workshop on Visual Languages*. doi:10.1109/WVL.1991.238854
- [42] Ehud Y. Shapiro. 1983. *Algorithmic Program Debugging*. The MIT Press. doi:10.7551/mitpress/1192.001.0001
- [43] Josep Silva. 2011. A Survey on Algorithmic Debugging Strategies. In *Advances in Engineering Software*. doi:10.1016/j.advengsoft.2011.05.024
- [44] Mate Soos and Kuldeep S. Meel. 2025. Engineering an Efficient Probabilistic Exact Model Counter. In *International Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-031-98682-6\_5

- [45] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2003. Interactive Type Debugging in Haskell. In *ACM SIGPLAN Workshop on Haskell*. doi:10.1145/871895.871903
- [46] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2004. Improving Type Error Diagnosis. In *ACM SIGPLAN Workshop on Haskell*. doi:10.1145/1017472.1017486
- [47] Mitchell Wand. 1986. Finding the Source of Type Errors. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/512644.512648
- [48] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *European Software Engineering Conference (ESEC/FSE)*. doi:10.1007/3-540-48166-4\_16
- [49] Danfeng Zhang and Andrew C. Myers. 2014. Toward General Diagnosis of Static Errors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2535838.2535870
- [50] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2015. Diagnosing Type Errors with Class. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2737924.2738009
- [51] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. In *Proc. ACM Program. Lang., Issue PLDI*. doi:10.1145/3591239
- [52] David Zhao, Pavle Subotić, and Bernhard Scholz. 2020. Debugging Large-scale Datalog: A Scalable Provenance Evaluation Strategy. In *ACM Trans. Program. Lang. Syst.* doi:10.1145/3379446
- [53] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. In *Proc. ACM Program. Lang., Issue POPL*. doi:10.1145/3632910

Received 2025-11-14; accepted 2026-04-03