SMTScope: Automated and Efficient Analysis of SMT Traces

Jonáš Fiala and Peter Müller

Department of Computer Science, ETH Zurich, Zurich, Switzerland
{jonas.fiala, peter.mueller}@inf.ethz.ch

Abstract. SMT solvers enable the automated verification of complex software, but they frequently also cause performance problems, proof brittleness, and spurious errors. Debugging such issues is challenging because it is difficult to understand and predict how the solver's complex algorithms and heuristics will interact with a given query.
We present SmtScope, a tool for the automatic analysis and visualisation of z3 executions. It helps detect and explain issues in SMT queries, such as indefinite chains of quantifier instantiations (matching loops), and proliferation of quantifier instantiations. Additionally, it provides a useful interface for other tools. Compared to the existing Axiom Profiler, SmtScope offers more effective analyses, novel visualisations of matching loops and proof search, as well as more efficient algorithms, which allow SmtScope to scale to large SMT traces. Our evaluation shows that SmtScope can identify problem behaviours in real-world traces, finding hundreds of unique issues in four state-of-the-art verifiers.

1 Introduction

SMT solvers are powerful and highly optimised tools, often able to handle complex queries efficiently. However, their performance is sometimes slow and unpredictable [20]. Identifying the root cause of these performance issues is very hard, as SMT solvers are essentially a black box for their applications.

The input to SMT solvers is a query consisting of a sequence of variable declarations and first-order logic constraints written as boolean assertions. The solver tries to either find a satisfying assignment to the variables or prove that the constraints are unsatisfiable. Especially in the important application domain of program verification, the constraints may contain quantifiers, enabling complex program semantics to be encoded as axioms. The complicated interactions of these axioms, as well as the sheer size of generated queries, make it infeasible to manually debug their performance. For example, the Dafny [12] verifier turns a 30-line tutorial program into a 1000-line query with almost 200 quantifiers.

Unfortunately, the quantifiers which are so critical for verification are also a leading cause of performance issues. Their solving is tricky to automate due to their unbounded nature, thus SMT solvers expose a customisable instantiation mechanism via *e-matching* [6]. E-matching requires quantifiers to be annotated with syntactic *patterns* containing all the quantified variables. When the SMT

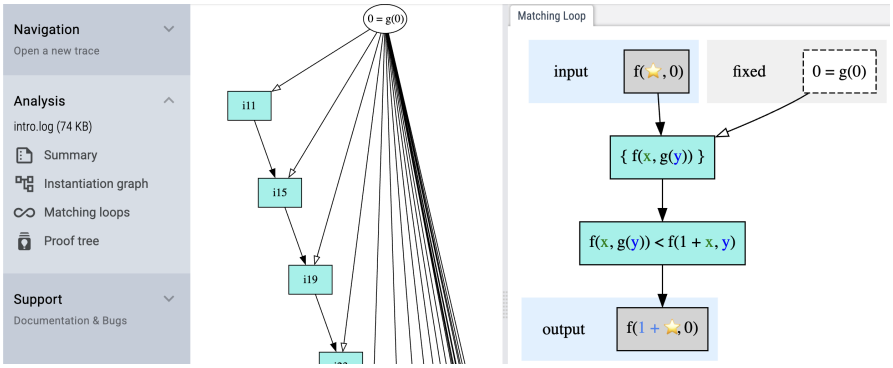


Fig. 1. A screenshot of SmtScope explaining a matching loop. The middle section shows the individual instantiations of the loop (turquoise nodes), how each subsequent one depends on a term produced by the prior (black arrows), and how each depends on `0 = g(0)` (white arrows and node). Our tool automatically finds the loop and generates the explanation graph on the right. It generalises one iteration of the loop: the pattern (top turquoise node) is matched by using the equality `0 = g(0)` to rewrite the ever-growing term `f(⭐, 0)`. The resulting instantiation produces a new term (bottom grey node). The ⭐ denotes an arbitrary term, with the blue 1+ indicating how it grows.

solver encounters a *ground term* (one with no quantified variables) matching the pattern, it unifies the two and uses the sub-terms that unified with the quantified variables to instantiate the quantifier. Importantly, this matching is not purely syntactic, instead being performed modulo equalities. Thus, understanding and predicting SMT quantifier instantiations is a formidable task [5].

Badly chosen patterns may lead to poor performance. As an extreme case, *matching loops* cause an excessive number of instantiations [15]. They occur when a term causes an instantiation which, in turn, produces a new term matching the same pattern, continuing the loop indefinitely and producing many new terms in the process. Figure 1 describes a non-syntactic matching loop. The quantifier `∀x:Int, y:Int :: {f(x, g(y))} f(x,g(y)) < f(1+x,y)`, with its e-matching pattern in the braces, cannot *syntactically* produce a new term matching the pattern. However, in our example, z3 proved `0 = g(0)` which sets off a loop starting with `f(n,0)` and continuing with `f(1+n,0)`, `f(2+n,0)`, ... An equality required for a matching loop may not be proved or even present in all queries, making such loops harder to detect, reproduce, and debug the caused performance problems.

Matching loops are not the only *problem behaviour* that significantly impacts performance. Quantifiers with patterns that can turn a few terms into many instantiations are another common issue. An example of this proliferation of quantifier instantiations is the following axiom expressing transitivity of subtyping:

```
∀x:Type, y:Type, z:Type :: {subtype(x, y), subtype(x, z)}
  subtype(x, y) ∧ subtype(y, z) ⇒ subtype(x, z)
```

Given n terms of the form `subtype(A, T`$_i$`)`, this quantifier will be instantiated n^2 times with `[x ↦ A, y ↦ T`$_i$`, z ↦ T`$_j$`]`. Additionally, these instantiations produce n^2 new terms; all combinations of `subtype(T`$_i$`, T`$_j$`)`, which in turn cause

a further n^3 instantiations. When these problem behaviours are combined, they can exponentially degrade performance.

Understanding SMT behaviour is not only necessary to debug performance bugs, but also to enable useful tool features. Recent work [9,21] analyses successful verification runs to optimise future queries. Both tools use a solver-provided unsatisfiability (UNSAT) core; a minimal subset of the assertions required to logically prove UNSAT. However, Gopinathan et al. [9] found that this core is often not sufficient to replay an SMT proof. They describe the missing key: axioms that are not logically required, but rather, produce terms required for instantiating axioms in the UNSAT core. Finding these *lurking axioms* requires understanding *how* the solver proved UNSAT. These three examples show that understanding and optimising solver behaviour is difficult and requires tool support that scales to the queries produced by modern verification tools.

The current state of the art in troubleshooting Z3 [16] issues with quantifiers is the Axiom Profiler [3]. It presents three views: a list of all instantiations, an instantiation dependency graph, and a textual explanation of an instantiation when selected. A variety of filtering operations help find problematic instantiations within the graph. Additionally, it can detect and explain some matching loops, although this is not fully automated as the user must manually identify suspicious chains to analyse. The tool focuses on quantifier instantiations only and does not support any other aspects of SMT execution, thus being ill-suited to precisely find, e.g., lurking axioms. Unfortunately, bad performance and frequent crashes make the tool notoriously cumbersome to use.

This work. We present SmtScope, a novel tool for automated and efficient analysis of Z3 executions. It analyses a Z3 trace and automatically detects and reports the aforementioned quantifier problem behaviours. It also understands other Z3 aspects, such as proof search, enabling it to precisely identify lurking axioms. Integration with existing verifiers is easy, enabling the fully-automatic analysis of a broad set of real-world queries, for instance, in continuous integration. Additionally, SmtScope allows for comprehensive inspection of said executions when problems are detected. It has already helped discover over 300 unique problems in verifier axiomatisations. Compared to prior work, SmtScope can identify more problem behaviours while doing so automatically and without any observed false positives. Our analyses asymptotically outperform those of the Axiom Profiler, are more stable, and support all modern versions of Z3 (v4.8.5+).

Our contributions are:

- We develop efficient algorithms for analysing Z3 trace files, including novel matching loop and multiplicative axiom detection.
- We identify a new class of equality-based matching loops and describe how to detect and eliminate them.
- We develop functionality to analyse and visualise the proof search, which can be used to precisely identify lurking axioms.
- We evaluate our tool on the state-of-the-art verifiers Dafny, F* [19], Verus [11], and Viper [17], and find over 300 unique problem behaviours.
- We release SmtScope as open-source, available and runnable online [7].

2 Analysing an SMT Trace

In this section, we present the main features of SmtScope. We describe how it helps pinpoint troublesome axioms and expensive lemmas and how it explains matching loops and multiplicative behaviour. Moreover, we show how our tool visualizes z3's proof search and how it finds lurking axioms.

We illustrate SmtScope on a simplified extract of a vector axiomatisation, shown in Figure 2. Our vector supports two operations: `pop`, which removes one element and returns the `empty` vector if the input was already `empty`, and `trim`, which pops N times. The `pop-trim` axiom equates a `pop` of a `trim` with the simpler expression of a one-larger `trim`. We use a custom uninterpreted `add` function to get around the fact that z3 does not allow the built-in `+` in patterns. The `add-lt` axiom gives an ordering to this `add` function. Though this example seems innocuous, it results in both a matching loop and multiplicative behaviour. `pop-trim` and `add-lt` are based on axioms from Viper and F* that exhibit the same problem behaviours—issues which went unnoticed for years, but have now been identified by SmtScope.

2.1 SMT Trace Summary

After opening a trace file, SmtScope presents a summary of each analysis. Figure 3 shows this for the example from Figure 2. This screen consists of three data components: aggregate (highlighted red), quantifier (blue), and proof (omitted in the figure). Each component helps the user identify potential problem areas.

The aggregate data component consists of the two leftmost boxes in Figure 3. The first lists all problem behaviours detected and the axioms involved. The second has the raw totals for a variety of metrics. These totals are useful when comparing traces to quickly see if a change was effective.

The quantifier component helps pinpoint troublesome axioms that should be changed to avoid unnecessary instantiations. It consists of three boxes and a summary graph. The boxes list all quantifiers sorted by: instantiation count,

```
(declare-sort Vec)                          ; vector sort
(declare-const empty Vec)                   ; empty vector
(declare-fun pop (Vec) Vec)                 ; pop one element
(declare-fun trim (Vec Int) Vec)            ; pop N times
(declare-fun add (Int Int) Int)             ; + (for patterns)
(assert pop(empty) = empty)                 ; pop-empty axiom
(assert trim(empty, 1) = empty)             ; trim-empty axiom
(assert ∀v:Vec, n:Int :: {pop(trim(v, n))}
  pop(trim(v, n)) = trim(v, add(1, n)))     ; pop-trim axiom
(assert ∀x:Int, y:Int, z:Int :: {add(x, y), add(x, z)}
  add(x, y) < add(x, z) = y < z)            ; add-lt axiom
```

Fig. 2. A simplified encoding of a vector. We use pseudocode based on the SMT-LIB format with presentational liberties.

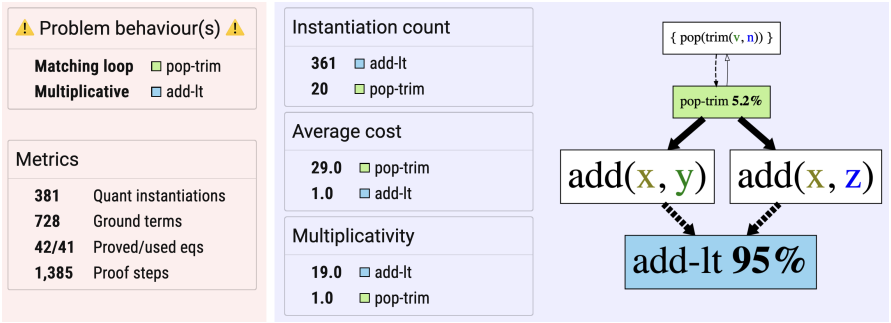


Fig. 3. SMTScope's summary screen for the example from Figure 2. The two leftmost boxes show aggregate data, the middle three and the graph show quantifier data. This toy example has no interesting proof data, thus we have omitted it. In the graph, coloured nodes represent the instantiations of a quantifier. White nodes and dashed arrows represent a matched pattern leading to an instantiation. Solid arrows represent term- or equality-dependencies, denoted by solid or hollow arrowheads respectively. Both nodes and edges are sized according to their percentage contribution to the total instantiation count. This percentage is also listed next to the quantifier name.

average cost, and multiplicativity. An instantiation's cost is based on the number of instantiations directly or transitively caused by this instantiation [3]; a quantifier's cost is the average over all of its instantiations. We say that a child instantiation directly depends on a parent one if the parent produced a term or equality used for matching the child's pattern. Multiplicativity represents a quantifier's ability to turn a few terms into many instantiations. It is calculated per quantifier as the number of unique instantiations divided by the number of terms matched for these instantiations. We call a quantifier *multiplicative* if this value exceeds a threshold (based on our experience, 5).

The list sorted by instantiation count helps compare traces across changes and rule out irrelevant quantifiers. The average cost indicates how effective reducing a quantifier's instantiation count would be. Optimising the top quantifiers in this list is often the most fruitful. In our example, all 381 instantiations depend on the first instantiation of `pop-trim`, contributing to its high average cost.

However, the cost metric can miss problematic multiplicative quantifiers, such as `add-lt` in our example. As we will see in the next subsection, `pop-trim` produces ever larger terms of the form `add(1, add(1, ...))`. These terms match `add-lt`'s pattern pairwise, causing n^2 instantiations from only n terms. None of these instantiations has any children, and thus their cost is 1. Solving the problem of many `add-lt` instantiations by reducing the number of `add` terms (by fixing the `pop-trim` matching loop, which produces these terms) is not always possible. For instance, a user assertion (from the program under verification) could result in the following line being added to the query (with 19 nested `add`s):

```
(assert 20 = add(1, add(1, add(1, ...))))
```

In this case, fixing `pop-trim` would not help, as the query itself contains all the `add` terms. Instead, we must identify and fix the problematic `add-lt` itself. Our

novel multiplicativity analysis precisely identifies these problematic quantifiers. Concretely, for `add-lt` we calculate 19^2 instantiations/19 terms = 19.

Lastly, SmtScope renders a summary graph to visualise the most important quantifiers (those at the top of the three quantifier lists) and causal relations between them, as seen on the right of Figure 3. The graph provides a visual summary and explanation of the quantifier lists that is faster to comprehend and clearly shows the relative importance of the included quantifiers. For our running example, we see the cyclic dependency of `pop-trim` and its own pattern indicative of a matching loop. Below this, we see how a few `pop-trim` instantiations cause many more `add-lt` instantiations, indicative of high multiplicativity.

The proof component (omitted from Figure 3) provides a list of lemmas z3 proved by contradiction, sorted by proof cost, i.e., the number of proof steps required to reach the contradiction. Lemmas that are proved using axioms purely from a verifier's background theory are particularly interesting, as they indicate missing axioms. Asserting these lemmas directly (e.g., by adding an axiom) is sound w.r.t. the background theory and would potentially speed up the proof search, since z3 could use them directly rather than re-proving them each time.

2.2 Understanding Matching Loops

SmtScope can automatically find all matching loops in the instantiation graph. The *instantiation graph* [3] is a DAG where nodes represent instantiations and edges represent causality (producing a term or equality that a child depends on). A matching loop must be a sequence of one or more quantifiers that repeats down a path of instantiations. Analysing the full graph path-by-path is infeasible, since the number of paths can be exponential. We describe our matching loop detection algorithm, which solves this exponential blow-up, in the next section.

Once a matching loop is identified, SmtScope builds a *generalised iteration graph* to explain it. Figure 4 shows this graph for the `pop-trim` matching loop that the summary warned us about in Figure 3. This graph explains how a group of inputs results in a sequence of instantiations, and how these instantiations lead to outputs that set off the next iteration. In our example, the graph starts with the term (T) `pop(empty)` and the equalities (E) `pop(empty) = empty` and (A) `empty = trim(empty, ⋆)`. z3 rewrites (T) using (A) into the term `pop(trim(empty, ⋆))`. This instantiates `pop-trim` with `[v ↦ empty, n ↦ ⋆]`, yielding the equality (Y) `pop(trim(empty, ⋆)) = trim(empty, add(1, ⋆))`. Using the three equalities we now have, the graph shows that z3 derives the term (B) `empty` $=^{\texttt{E}}$ `pop(empty)` $=^{\texttt{A}}$ `pop(trim(empty, ⋆))` $=^{\texttt{Y}}$ `trim(empty, add(1, ⋆))`. This closes the loop as a new iteration can now start with (T), (E), and (B).

This matching loop is interesting because it is perpetuated by equalities only, while the term used for matching the pattern remains fixed. We are able to identify it as a matching loop due to SmtScope's advanced understanding of z3's equality reasoning, which is described in Subsection 2.3. To our knowledge, such equality-based matching loops have not been described previously, and prior to SmtScope, no tool was able to detect them. Matching loops of this form often happen when two axioms can be used to define the same term.

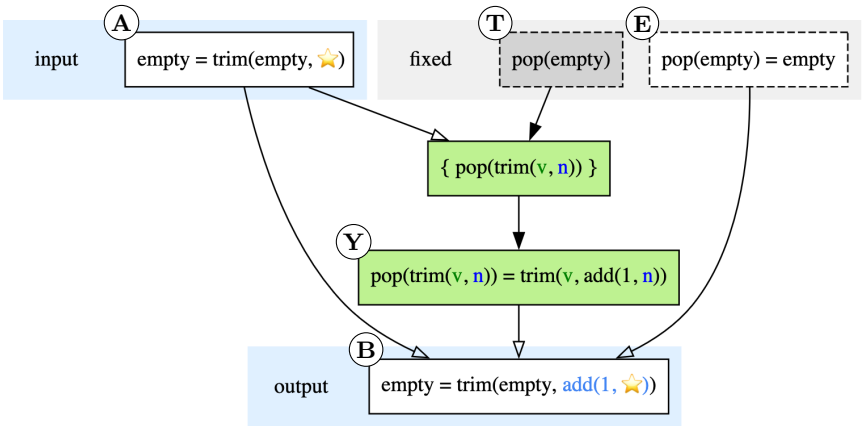


Fig. 4. The generalised iteration graph for the matching loop from Figure 2. The inputs are at the top and are categorised into those that stay fixed (grey background) and those that change (blue). The ones that change are also displayed as outputs at the bottom, closing the loop. The grey nodes are terms, white nodes are proved equalities, and coloured nodes are the pattern and body of the corresponding quantifier (`pop-trim`). Arrows leading to a pattern explain how it is matched, those leading to an equality explain which equalities are transitively combined to derive it. The ⭐ denotes the same arbitrary term in the input and output, with the blue text indicating how it grows.

They can be fixed by guarding one of the axioms with a (stronger) implication. Indeed, this also applies to our example; both `pop-empty` and `pop-trim` "define" the term `pop(empty)`. To fix this, we should rewrite the body of `pop-trim` to `trim(v, n)` $\neq$ `empty` $\Rightarrow$ `pop(trim(v, n)) = trim(v, add(1, n))`. This differs from other matching loops, which are often resolved by changing the pattern.

2.3 Understanding Other Instantiation Behaviours

As explained earlier, the summary screen provides indications for other quantifier behaviours, such as high cost and multiplicativity. To investigate those further, SmtScope offers an instantiation graph view, which displays a raw dependency graph of individual instantiations. Figure 5 shows SmtScope's instantiation graph for our running example. In this subsection, we explain how to use the graph to understand why the `add-lt` axiom is highly multiplicative. We first explain how the terms causing high multiplicativity are produced. Then, we see how the instantiation graph helps us understand the problematic behaviour of `add-lt`. Lastly, we propose a general solution for fixing a class of multiplicative axioms.

In Figure 5, the node $i10$ for an instantiation of `pop-trim` is selected, causing a detailed explanation to appear on the right. There, we see the matched pattern and which term and equalities were used. Importantly, we see all the *new* terms produced by this instantiation, which may be used for further instantiations. Additional relevant graph properties are also shown, such as the fact that $i10$ sets off a chain 39 nodes long—the matching loop explained earlier.

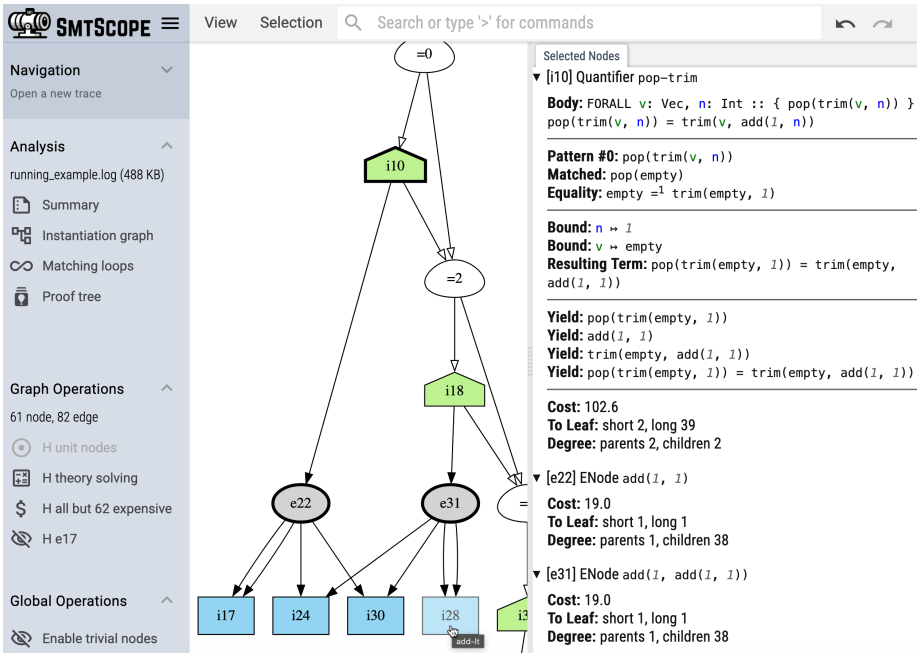


Fig. 5. A screenshot of SMTSCOPE's instantiation graph view for our running example. Instantiation nodes are coloured with the unique colour of their quantifier. Oval nodes represent terms (grey) and equalities (white). Instantiations with hidden parents or children indicate this with a pointy top or bottom, respectively.

Nodes for `add(1, 1)` ($e22$) and `add(1, add(1, 1))` ($e31$) are also selected. Under 'Degree' in the side panel, we see that each directly causes 38 instantiations. Since the overview indicated a high multiplicativity for `add-lt`, the user could select the instantiation nodes corresponding to this quantifier (the blue nodes) and observe that each `add(1, A)` term pairs with any other `add(1, B)` term to cause an instantiation, resulting in the 361 instantiations from only 19 terms. The dependency on the exact terms is visualised in our graph by explicitly representing terms as nodes (here, $e22$ and $e31$).

Our graphs also have explicit nodes for equalities, which are added by reconstructing the transitive equality reasoning that Z3 performs. As an ever-larger `empty = trim(empty, N)` (white nodes) is constructed in our example, only two nodes are blamed: the prior equality and the one obtained from `pop-trim` (green nodes) for the step `N` $\mapsto$ `add(1, N)`. Having an explicit representation of equalities results in a repeating pattern that can be detected visually and that our matching loop detection (Section 3) can generalise. Both the term and equality nodes can be disabled when users want to focus on instantiations only.

Since instantiation graphs of realistic examples are very large, SMTSCOPE offers various ways to filter out irrelevant nodes. This filtering is configurable by changing the entries under 'Graph Operations' in the left side panel. When an

intermediate node is hidden due to filtering, we add dashed edges to preserve transitive dependencies. Additional filters to pull out useful subgraphs are available under the 'View' and 'Selection' menus at the top. To help navigate the graph, one can use the search bar to find nodes by quantifier or function name.

In our evaluation, we found a common class of multiplicative quantifiers in verifier axiomatisations: ordering axioms. These encode properties such as transitivity of subtyping. We suggest replacing such axioms with Z3's built-in decision procedures for various types of ordering, including total and partial orders. The `add-lt` axiom from our example also expresses ordering. In this case, the most complete solution is to rely on Z3's native integer reasoning. We can do this by changing the quantifier to:

```
(assert ∀x:Int, y:Int :: {add(x, y)} add(x, y) = x + y)
```

2.4 Understanding the Proof Tree

SmtScope can explain other Z3 aspects, aside from e-matching. Understanding Z3's proof search and how it proved UNSAT is crucial for identifying missing axioms and finding a precise UNSAT core that includes lurking axioms. SmtScope parses and analyses the proof derivation tree that Z3 constructs while trying to prove UNSAT. Each node in the proof tree represents a proof step used to prove a boolean term, and incoming edges represent its prerequisites. The SMT solver reports UNSAT if it proves false. The resulting proof tree contains a `false` node; its ancestors correspond to the UNSAT core, but it does *not* contain lurking axioms. Consequently, the core logically implies false, but might not be sufficient to replay the proof in the SMT solver, as it might not contain all the terms needed to perform necessary quantifier instantiations.

```
(declare-fun non_empty (Vec) Bool)     ; is vector non-empty?
(declare-fun length (Vec) Int)         ; length of vector
(declare-fun append (Vec Vec) Vec)     ; append two vectors
(assert ∀x:Vec :: {non_empty(x)}
  non_empty(x) = 1 ≤ length(x))        ; non_empty-def axiom
(assert ∀x:Vec, y:Vec :: {length(x), length(y)} ; append ax.
  non_empty(append(x, y)) = non_empty(x) || non_empty(y))

(declare-const a Vec) (declare-const b Vec) ; inputs a and b
(assert non_empty(a) && non_empty(b)) ; precondition
(assert !non_empty(append(a, b)))     ; negated postcondition
(check-sat) (get-unsat-core)
```

Fig. 6. An extension of our vector encoding. The `non_empty-def` axiom defines the `non_empty` function, and the `append` axiom states that the result of appending two vectors is non-empty if and only if at least one of them is non-empty. The remainder simulates verification of a function taking two vectors `a` and `b` as input, with a precondition that both are non-empty and a postcondition that their append is also non-empty. The postcondition is negated, such that its validity is proved by an UNSAT result.

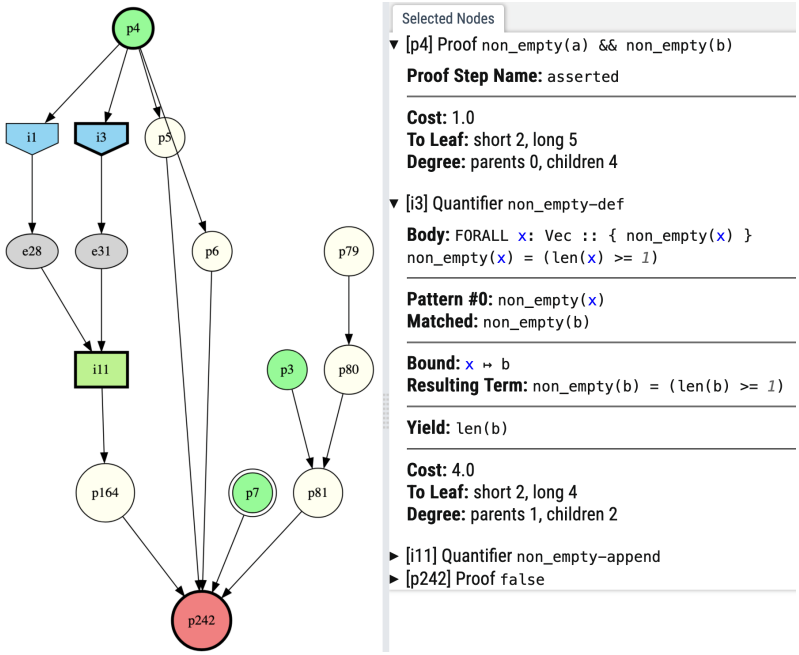


Fig. 7. A screenshot of SmtScope's proof view. The proof node $p164$ representing the instantiation of `append` is linked to the corresponding instantiation node $i11$ in the instantiation graph. The red $p242$ node represents a proof of `false` and the green nodes are the assertions from the query.

Gopinathan et al. [9] solve this problem by using our instantiation graph to extend z3's UNSAT core with the axioms that any core axiom transitively depends on to get instantiated. However, their approach overapproximates; some instantiations of a core axiom might not be required for the proof, but all their dependencies are included nonetheless. By combining our instantiation graph with the proof tree, we can find a precise extended UNSAT core.

To explain how, we adapt an example from Gopinathan et al. [9] in Figure 6. Here, `non_empty-def` is a lurking axiom: it is not logically required, but it produces the `length` terms required to instantiate `append`. Figure 7 shows the combined proof tree and instantiation graph view, which makes the dependency on `non_empty-def` explicit (instantiations $i1$ and $i3$). Walking over all ancestors of the `false` node, we are able to return the precise extended UNSAT core. It is also available via a non-GUI interface.

Our combined graph is also useful for understanding how, for a particular query, z3 proved UNSAT or why it failed to do so. Namely, the proof tree often includes lemmas that z3 proved by contradiction. By inspecting these derived lemmas, one can identify missing axioms that would provide the lemma outright, avoiding the need to prove them. As with the instantiation graph view, filtering operations are available to help focus on interesting parts of the graph.

3 Matching Loop Detection Algorithm

Our novel matching loop detection fundamentally improves on prior work [3] in two ways: (1) the false-positive rate is near-zero, and (2) it is asymptotically faster, enabling automatic analysis of full instantiation graphs for realistic examples. In this section, we explain the insights that enable these improvements and present how they are leveraged in our algorithm.

The key observation is that we need to consider instantiations only *pairwise*, without regard for the path connecting them. Compared to the algorithm of prior work, which is designed to analyse individual paths, our pairwise one asymptotically improves performance, because the number of paths is often exponential in the number of quantifier instantiations, whereas the number of pairs is quadratic. Our observation holds true even for matching loops where multiple quantifiers are instantiated in a single iteration of the loop; the repeating instantiations of each quantifier can be identified and generalised individually, and then later combined to build the generalised iteration graph for the entire matching loop. This pairwise analysis has worst-case quadratic time performance, though in practice it is often linear with the number of instantiations. Therefore, we can analyse full instantiation graphs of real-world queries with hundreds of thousands of instantiations, allowing us to give completeness guarantees.

Our matching loop algorithm is shown in Figure 8. It has two phases: first, a topological walk to perform the pairwise analysis, and second, a pass to collect the full matching loops and construct their generalised iteration graphs.

Phase 1: Identifying Candidate Pairs

We use three filters to identify instantiation pairs that are likely part of a matching loop. Any pair in a matching loop must satisfy these filters, while those not in a loop are very unlikely to do so. The first phase, for each instantiation `inst`, performs the following three steps (each one explains a filter).

Step 1: Root dependencies and ancestors, lines 4–6. We will use $I_0, I_1, I_2, \ldots$ to denote successive instantiations of `pop-trim` in our running example. The first instantiation I_0 depends on the term (T) `pop(empty)` and the equality (A) `empty = trim(empty, 1)` (see Figure 4, the loop starts with $[\star \mapsto \texttt{1}]$). Both are from the input query; we call these *root dependencies*. The next instantiation I_1 depends on the same term (T) and the equality (B), thus it transitively depends on (A), (Y), and (E). However, since (Y) is derived from a prior instantiation and is not from the input query, it is not a root dependency. Therefore, the root dependencies of I_1 are the same two as for I_0 plus the equality (E). All subsequent `pop-trim` instantiations depend on *the same three* root dependencies.

We can generalise this observation to obtain our first filter: in a matching loop, the set of (transitive) root dependencies cannot change because (1) by definition, the set of root dependencies cannot shrink from parent to child, and (2) it cannot grow in each iteration of a matching loop since the number of root dependencies is bounded by the size of the input query.

We store the set of root dependencies for each instantiation as the union of the direct root dependencies and the root dependencies of all parents. Similarly, we compute the set of ancestors (`get_ancestors`) by taking the union of the `ancestors` of all parents and adding the parents themselves. Next, we iterate over these ancestors to perform the pairwise analysis.

Step 2: Pairwise instantiation signature check, lines 7–10 and 15. In our example, `pop-trim`'s pattern **`{pop(trim(v, n))}`** successively matches the terms I_0: `pop(trim(empty, 1))`, I_1: `pop(trim(empty, add(1, 1)))`, ... Thus, the quantified variables are successively instantiated with I_0: `[v` $\mapsto$ `empty, n` $\mapsto$ `1]`, I_1: `[v` $\mapsto$ `empty, n` $\mapsto$ `add(1, 1)]`, I_2: `[v` $\mapsto$ `empty, n` $\mapsto$ `add(1, add(1, 1))]`, and so on. We define the size of a term as the node count of its AST, we then define the *size of an instantiation* as the vector of sizes of each substituted term; in our example, this is I_0: `[1, 1]`, I_1: `[1, 3]`, I_2: `[1, 5]`, and so on. The key

```
1  function find_matching_loops(g: InstGraph)
2      # Phase 1: Pairwise analysis
3      for inst in topological_sort(g):
4          inst.root_deps = direct_root_deps(inst) ∪
5                           parent_root_deps(inst)
6          for anc in get_ancestors(inst):
7              sig_compat = quant(inst) == quant(anc) &&
8                           inst.root_deps == anc.root_deps &&
9                           inst_size(inst) >= inst_size(anc)
10             if sig_compat:
11                 gen = try_generalise(anc, inst)
12                 inst.potential_loop = (anc, gen)
13                 (inst.leaf, anc.leaf) = (true, false)
14             else:
15                 inst.ancestors = inst.ancestors ∪ {anc}
16     # Phase 2: Build generalised iteration graphs
17     for inst in leaf_nodes(g):
18         (prev, gen) = inst.potential_loop
19         if gen.success:
20             others = inst.ancestors \ prev.ancestors
21             fixed = dependencies_of({inst} ∪ others) \
22                     produced_by(others ∪ {prev})
23             graph = build_graph(g, inst, others)
24             matching_loops += (inst, graph)
25         else:
26             suspicious += inst
27     return (matching_loops, suspicious)
```

Fig. 8. Our matching loop detection algorithm. It takes as input the instantiation graph and returns two lists: the identified matching loops (including the generalised iteration graph of each), and the suspicious instantiation chains that we failed to generalise. The $\geq$ on line 9 is shorthand for elementwise comparison of the two vectors.

observation is that in a matching loop, this size cannot decrease between successive instantiations of the same quantifier[1]—our second filter. If it were to decrease, we could use this as a termination measure due to the lower bound to prove the absence of a matching loop.

We call the set of root dependencies and the size vector the *instantiation signature* of a quantifier instantiation. The signatures of an ancestor and descendant instantiation are *compatible* if the instantiated quantifier is the same, the set of root dependencies is equal, and the size vector of the descendant instantiation is elementwise greater than or equal to that of the ancestor.

The first two filters are strong enough to rule out the majority of non-matching loop instantiation pairs: Filter 1 (root dependencies) avoids a class of false positives that prior work suffers from, such as an axiom for lookups in program heaps leading to long chains of instantiations of the same axiom but for different versions of the heap (each version is a new root dependency). Filter 2 (instantiation size) is complementary, avoiding false positives for quantifiers that recursively deconstruct a large term into smaller ones, such as one defining the length of a list with the pattern `{len(cons(x, xs))}`.

Lines 7–9 in Figure 8 check whether `inst`'s instantiation signature is compatible with that of each ancestor `anc`. If they are compatible, we consider the pair as a potential matching loop and proceed to the next step. Otherwise, we save `anc` for comparison with `inst`'s descendants (for loops with multiple quantifiers).

Step 3: Generalised repetition, lines 11–13. At this point, we have identified a pair of instantiations `(anc, inst)` that is likely part of a matching loop. In our example, (I_1, I_2) is such a pair (remember that I_0 is not compatible with I_1, since the root dependencies differ). The final key observation is that the term and equality dependencies of such pairs must generalise—our third filter. That is, they either remain the same (fixed inputs) or grow in a systematic way enabling the matching loop to continue indefinitely. The generalisation is done using syntactic anti-unification (from prior work [3,18]), slightly modified to help find the most general result. Applied to our example, where I_1 depends on the equality `empty = trim(empty, add(1, 1))`, while I_2 depends on the equality `empty = trim(empty, add(1, add(1, 1)))`, our anti-unification yields `empty = trim(empty, ⋆)`. The ⋆ represents a fresh variable; one that can be substituted to get either input. Next, we try to identify the iteration step. We take the subexpression that the variable replaced in `anc`, `add(1, 1)`, and try to find it in the corresponding subexpression it replaced in `inst`, `add(1, add(1, 1))`. In our example, we successfully identify the iteration step as `[⋆ ↦ add(1, ⋆)]`.

We save the reference to `anc` along with the generalisation: we will use these in Phase 2 of the algorithm to build a generalised iteration graph. Generalisation may fail when the subexpression cannot be found. This was rare during our evaluation; when we inspected such cases, we confirmed that none of them were matching loops. However, due to issues in how Z3 logs equality dependencies, it may be possible that an actual matching loop fails to generalise. Therefore, we also keep these failing cases and report them as `suspicious` in the next phase.

[1] All terms may remain equal-size due to arithmetic simplification, such as `3+1` to `4`.

Phase 2: Constructing Generalised Iteration Graphs

The pairs identified in Phase 1 form chains of matching loop instantiations. In our example, the pairs `(`I_1`,`I_2`)`, `(`I_2`,`I_3`)`, ... form the chain `[`I_1`,`I_2`,`I_3`,...]`. Phase 2 combines these *instantiation chains* with other interleaved chains to build generalised iteration graphs, possibly involving multiple quantifiers. For our example, this is trivial as there is only one quantifier involved: the generalised iteration graph in Figure 4 follows directly from the generalisation we just found.

To better illustrate Phase 2, we use a new example matching loop with two quantifiers: `q` and `r`, with instantiations Q_0, Q_1, ... and R_0, R_1, ..., respectively. An instantiation of `q` produces a term matching the pattern of `r`, which in turn produces a term matching the pattern of `q`, resulting in a matching loop. The instantiation graph contains the following chain of instantiations: $Q_0 \to R_0 \to Q_1 \to R_1 \to \ldots$. For this example, Phase 1 identifies two independent chains: `[`Q_0`,`Q_1`, ...]` and `[`R_0`,`R_1`, ...]`. Phase 2 combines these as follows.

Step 1: Identify suspicious chains, lines 17–19 and 26. We iterate over all instantiations marked as a leaf on line 13. Following the ancestor references identifies a chain, and we mark those as suspicious for which generalisation from Step 3 in Phase 1 failed. Otherwise, we proceed to the next step.

Step 2: Identify interleaved chains, lines 20–22. Let R_n (`inst`) and R_{n-1} (`prev`) be the last two instantiations of the second chain from the example above. This chain alone is not sufficient to explain the matching loop; each iteration requires one instantiation of `r` but also one of `q`. To identify such required interleaved instantiations, we take the set difference of the ancestors of R_n (`{`Q_0`,`R_0`, ...,`Q_{n-1}`,`Q_n`}`) and those of R_{n-1} (`{`Q_0`,`R_0`, ...,`Q_{n-1}`}`). This yields `{`Q_n`}`—all instantiations between R_{n-1} and R_n. The resulting instantiations (stored in `others`) come from an identified chain themselves; these are the other chains required to explain the matching loop. The matching loop may also require fixed dependencies to perpetuate (the nodes with a grey background in the generalised instantiation graph). To identify those, we take the term and equality dependencies of `inst` and `others` that are not produced by any instantiations in `others` or `prev`. Our current example has no fixed dependencies: each new iteration depends entirely on the term produced by the previous iteration.

Step 3: Generalised iteration graph, line 23. Now that we have identified all the instantiations involved in one iteration of the matching loop (`inst` and `others`) and we have their associated generalisations and fixed dependencies, we can build the generalised iteration graph. For each of these instantiations, we create nodes for the quantifier pattern and body in the generalised iteration graph (coloured nodes). We connect these nodes as the instantiations themselves are connected, but replace the term and equality nodes with their generalised forms (unless they are fixed). The result is a generalised iteration graph, which explains the full matching loop (involving both `r` and `q` in the example).

The algorithm returns all identified suspicious chains (as the leaf node, from which the chain can be reconstructed) and matching loops (as the leaf node and the generalised iteration graph).

4 Implementation and Evaluation

SmtScope operates on the trace file created by z3 when run with the trace flag. All of SmtScope's algorithms and analyses are equally applicable to any other SMT solver that uses e-matching (e.g., CVC5 [1]). If these SMT solvers were extended with the same tracing functionality, our tool would work with them as well. Our tool is written in Rust and consists of two crates: a parsing and analysis library (including a command-line interface) and a web-based visualisation frontend. The former allows one to incorporate our analyses into other tools. The latter is compiled to WASM and runs in a static webpage entirely in the user's browser. Such a web-app approach improves portability and ease of use, as the tool works with any OS and requires no installation.

We designed an evaluation to answer the following questions:

(Q1) Can SmtScope identify and explain problem behaviours such as matching loops, is the analysis accurate, and are these problem behaviours important?
(Q2) Is SmtScope fast and stable enough to scale to large real-world examples?

We assembled a test suite of 29,305 z3 v4.8.7 trace files generated while running the test suites of Dafny, F*, Verus, and Viper, as well as running z3 on the Smt-Comp 2024 [4] benchmarks (all logics). With the latter, we test that SmtScope does not crash, even when using a wide variety of SMT-LIB features.

Q1: Problem Behaviours. To identify problem behaviours, we ran SmtScope in the command-line mode on each of the trace files. This mode reports the number of matching loops, suspicious chains, and multiplicative behaviours detected, and prints the names of the axioms involved. So that we avoid counting repeated occurrences of the same issue, we recorded the set of names of the axioms involved in each issue. We then counted the number of unique issues for each problem behaviour. The results are summarised in Table 1. Clearly, SmtScope is able to automatically detect a significant number of problem behaviours.

To answer the second part of our question, we chose to focus on Viper-SE . In the visualisation frontend, we inspected the trace files containing the 15 longest (by iteration count) identified matching loops, all 5 suspicious chains and the 15 most multiplicative quantifiers. Using the matching loop view, we confirmed that all 15 matching loops were identified correctly. The generalised iteration graph helped us quickly understand the loops, and we have successfully implemented a few fixes to the Viper axioms. Of the 5 suspicious chains, none were matching loops. This indicates that our analysis is accurate and does not misclassify matching loops as mere suspicious chains. We were able to quickly understand all 15 multiplicative behaviours using the instantiation graph view. Interestingly, we noticed that some axioms had non-symmetric patterns such as `{f(x), g(y)}` but were still multiplicative due to a large number of `f` and `g` terms.

We then ran the Axiom Profiler on all trace files. It supports a very limited automated analysis that searches the longest instantiation chains for matching loops. Only 31 matching loops were reported (compared to 130 with SmtScope), and manual inspection showed that 11 of these were false positives.

For the last part of the question, we compared the proportion of quantifiers we identified as problematic (not including the suspicious ones) to the proportion of total instantiations they caused. We found that only 382 out of 159,031 (0.24 %) quantifiers across our entire test suite were problematic (multiple quantifiers may be involved in a single matching loop). However, these 0.24 % of quantifiers caused over 12.2 million out of 47.9 million total instantiations (25.5 %). This does not even include other instantiations resulting from the new terms or equalities produced by these 25.5 %. Clearly, the two problem behaviours we describe and tackle in this work cause a substantial fraction of all quantifier instantiations, and thus, are likely to degrade the performance of the SMT solver.

Q2: Scalability. To evaluate the stability and performance of SmtScope and to compare it to the Axiom Profiler, we timed both while analysing the 29,305 trace files from above. We modified the Axiom Profiler to log timing information and added a one-second timeout for parsing each line (when handling some trace lines, it would freeze). We found that SmtScope handily outperforms the Axiom Profiler by multiple orders of magnitude, even though it performs a more advanced and complete analysis. For a 200 MB trace file, SmtScope takes around 1 s while the Axiom Profiler takes over 40 s. SmtScope's runtime was on average 30 % that of the z3 execution which generated the trace file. This means that our tool can be run alongside existing verifier CI pipelines with little overhead.

Regarding stability, we observed that for 25 % of the trace files the Axiom Profiler either failed to parse the whole file or reported an error during analysis. That is despite using a version of z3 which it officially supports. In contrast, SmtScope successfully parses and analyses all trace files in the benchmark suite and supports all modern versions of z3 (v4.8.5+).

Table 1. Number of problem behaviours detected by SmtScope in each category. We split the Viper files based on the verification backend: symbolic execution (Viper-SE) and verification condition generation (Viper-VCG). A multiplicativity of 5 or more is categorised as a problem behaviour, and between 3 and 5 as suspicious.

Category	Trace files	Matching loops (+sus.)	Multiplicative (+sus.)
Dafny	13153	45 (+6)	38 (+33)
Viper-SE	2724	51 (+5)	37 (+60)
Viper-VCG	4774	17 (+1)	98 (+194)
F*	608	8 (+1)	6 (+6)
Verus	2581	3 (+0)	0 (+5)
Smt-Comp	5465	6 (+5)	32 (+14)
Total	**29305**	**130** (+18)	**211** (+312)

5 Related Work and Conclusion

The Axiom Profiler [3] is the most closely related tool; they developed the idea of an instantiation graph, and our summary graphs (in Figure 3) are an extended version of similar graphs implemented by Pit-Claudel for the Axiom Profiler. As discussed in Section 1, SmtScope supersedes it in several ways. First, we developed new algorithms to find and explain matching loops and multiplicative quantifiers more accurately and efficiently. Second, our tool is designed from the ground up to scale to large and complex real-world traces, enabling automated testing in verifier CI pipelines. Third, we analyse the proof tree to both provide a precise UNSAT core that includes lurking axioms and to visualise and explain the proof search, something that prior work does not do.

z3tracer [2] allows one to parse z3's trace files and compute high-level per-quantifier metrics, comparable to some of those on our summary screen. However, it reconstructs only a basic picture of e-matching, thus none of the analyses that our tool offers can be implemented on top of it. Further, z3tracer does not handle any aspect of proof search.

The challenge of selecting patterns that are neither overly-strict nor overly-permissive has been widely studied [15,13,5], with approaches presented that avoid some simple matching loops [14]. In addition to adjusting patterns, the solutions we propose (correctly conditioning equalities, replacing quantifiers with decision procedures) complement these.

Several techniques [9,20,21] optimise SMT-based verifiers by understanding and analysing successful proof runs to either reduce instability or to minimise future queries for faster proof replay. They can directly benefit from SmtScope.

In summary, SmtScope can automatically, efficiently, and reliably identify many problem behaviours in real-world z3 traces. Once identified, it effectively explains these behaviours by providing a breadth of visualisations. Additionally, our novel proof tree visualisation shines a light on this important but previously opaque aspect of SMT execution, enabling features such as finding precise UNSAT cores that include lurking axioms.

Acknowledgments. We thank Oskari Jyrkinen for his significant help in developing the initial version of the tool during his bachelor's thesis [10], Alexander J. Summers for helpful discussions, and the anonymous reviewers for their valuable feedback.

Data Availability. The two Rust crates comprising SmtScope, along with instructions to build and run it, are available online [7]. The tool, along with the SMT-LIB query files required to generate the trace files used in the evaluation, and sufficient instructions to reproduce our results are available as an artifact [8].

References

1. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: CVC5: A versatile and industrial-strength SMT solver. In: TACAS (1). Lecture Notes in Computer Science, vol. 13243, pp. 415–442. Springer (2022)
2. Baudet, M.: z3tracer (October 2021), `https://github.com/facebookarchive/smt2utils/tree/main/z3tracer`, GitHub
3. Becker, N., Müller, P., Summers, A.J.: The axiom profiler: Understanding and debugging SMT quantifier instantiations. In: TACAS (1). Lecture Notes in Computer Science, vol. 11427, pp. 99–116. Springer (2019)
4. Bromberger, M., Bobot, F., Jonáš, M.: SMT-COMP 2024 (July 2024), `https://smt-comp.github.io/2024/`
5. Bugariu, A., Ter-Gabrielyan, A., Müller, P.: Identifying overly restrictive matching patterns in SMT-based program verifiers. In: FM. Lecture Notes in Computer Science, vol. 13047, pp. 273–291. Springer (2021)
6. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
7. Fiala, J., Jyrkinen, O., et al.: SMTScope (October 2025), `https://github.com/viperproject/smt-scope`, GitHub
8. Fiala, J., Müller, P.: Artifact for "SMTScope: Automated and efficient analysis of smt traces" (2025). `https://doi.org/10.5281/zenodo.17491144`, `https://doi.org/10.5281/zenodo.17491144`
9. Gopinathan, K., Spiliopoulos, D., Goyal, V., Müller, P., Püschel, M., Sergey, I.: Accelerating automated program verifiers by automatic proof localization. In: International Conference on Computer Aided Verification. pp. 153–174. Springer (2025)
10. Jyrkinen, O.: Developing Tool Support for Understanding Quantifier Instantiations. Bachelor's thesis, ETH Zurich (2024)
11. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proceedings of the ACM on Programming Languages **7**(OOPSLA1), 286–315 (2023)
12. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR (Dakar). Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
13. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Proceedings of the 2009 ACM symposium on Applied Computing. pp. 615–622 (2009)
14. Leino, K.R.M., Pit-Claudel, C.: Trigger selection strategies to stabilize program verifiers. In: International Conference on Computer Aided Verification. pp. 361–381. Springer (2016)
15. Moskal, M.: Programming with triggers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. pp. 20–29 (2009)
16. de Moura, L.M., Bjørner, N.S.: Z3: an efficient SMT solver. In: TACAS. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
17. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: International conference on verification, model checking, and abstract interpretation. pp. 41–62. Springer (2015)

18. Plotkin, G.D.: A note on inductive generalization. Machine intelligence **5**(1), 153–163 (1970)
19. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., et al.: Dependent types and multi-monadic effects in F★. In: Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270 (2016)
20. Zhou, Y., Bosamiya, J., Takashima, Y., Li, J., Heule, M., Parno, B.: Mariposa: Measuring SMT instability in automated program verification. In: 2023 Formal Methods in Computer-Aided Design (FMCAD). pp. 178–188. IEEE (2023)
21. Zhou, Y., Shah, A., Lin, Z., Heule, M., Parno, B.: Cazamariposas: Automated instability debugging in SMT-based program verification. In: Conference on Automated Deduction. vol. 122 (2025)