ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Integration and Analysis of Alternative SMT Solvers for Software Verification

Master's Thesis

Frederik Rothenberger
frothenb@student.ethz.ch

Supervisors:
Dr. Alexander J. Summers
Prof. Dr. Peter Müller

Chair of Programming Methodology
ETH Zurich

March 7, 2016

Abstract

SMT solvers are commonly used in software verification. Software verification often requires undecidable theories, which are only unreliably solved by SMT solvers. In this thesis, two SMT solvers are compared in order to decide whether the reliability of software verifiers can be increased by opportunistically switching the underlying solver. The evaluation shows that, for the particular comparison made, this is not the case. Therefore another approach to alleviate the reliability problems is pursued: Improvement of tools to understand the problematic behaviour of SMT solvers. In particular, an existing tool to analyse the proving behaviour of the SMT solver, Z3, is improved and extended. The focus of the extension is on features designed to explain a class of problems which cause infinite runtimes, called matching loops.

Acknowledgements

I would like to express my gratitude to Prof. Müller for giving me the opportunity to work on this thesis. Many thanks also to Dr. Alex Summers, for his great support and thoughtful comments throughout the project. Furthermore, I want to thank Michał Moskal for his time, good suggestions and expertise. Finally, I want to thank all my friends and family, who supported and motivated me during all the years at ETH Zurich.

Contents

1 Introduction

A common approach in software verification is to encode a given program property as a boolean formulas, in order to verify that this property holds in the context of said program. These boolean formulas are then verified by a satisfiability modulo theory solver (SMT solver). Although some logical theories are decidable, many program properties require undecidable theories to be expressed. SMT solvers try to approach these undecidable problems using clever heuristics and exhaustive search (while risking infinite runtime).

The heuristics are usually hidden away under opaque abstraction layers. It is therefore difficult to predict the runtime based on the input problem, especially since small changes in the formulation can make huge differences in runtime. This leads to the unsatisfactory situation that tools relying on SMT solvers might become unpredictable. Even worse, it is sometimes unclear how to understand such problematic situations and avoid them, due to the lack of powerful debugging tools for SMT solvers. There are multiple SMT solver implementations available: Simplify [9], Z3 [7], CVC4 [3] and Yices [10] are only a small selection.

This project aims to improve the situation by considering two possibilities: either the SMT solvers' heuristics are sufficiently different that opportunistically switching the solver reliably resolves issues or the solvers struggle with the same difficulties. In the latter case, the only solution to more reliable verifiers, with respect to runtime, is by improving the tooling for understanding the problems on the level of SMT solvers. Then the issues can be resolved at the root by reformulating the SMT solver input.

In order to examine the first of these two possibilities, one alternative SMT solver implementation, CVC4, was integrated into the Viper tool chain [20]. The goal was to determine whether another SMT solver could improve the reliability and speed of the verifiers. The evaluation (Section 6) shows that the integration of CVC4 does not improve the Viper tool chain significantly.

Therefore the lack of debugging tools was addressed: the Z3 Axiom Profiler was improved and extended during this project. The Z3 Axiom Profiler is the main tool to look at and understand quantifiers instantiations of Z3. Additionally it also shows information about the proof search process. In this thesis, the improvements to accessibility and ease of use are explained as well as the feature set to deal with a special class of problems, namely matching loops (Section 2.6). In particular, this involves displaying more relevant information about quantifiers and their instantiations. This is important because the quantifiers are often carefully annotated with patterns, which control the context in which quantifiers are instantiated. Feedback on the effect of these patterns is crucial in order to improve runtimes and prevent matching loops. This feedback was previously very hard to get. Quantifier and their instantiations are explained in detail in Section 2.

Z3 is a popular SMT solver used in many projects, such as Boogie [2], Dafny [15], Ironclad apps [14] and others [1, 16]. All these projects use Z3 as a back end. Improving the Z3 Axiom Profiler therefore has important application beyond the scope of the Viper tool chain. We are not aware of any other analytical tools for SMT solvers, especially Z3. In particular for finding and explaining matching loops, there is a distinct lack of tools available.

1.1 Outline

Section 2 introduces the necessary concepts and outlines the problem in more detail. In particular, quantified assertions, E-matching and matching loops are discussed. Then, in Section 3 the original Z3 Axiom Profiler and the improvements made during this thesis are explained. This section focuses on features and their improvements to the tasks at hand, whereas Section 4, explores the technical implementation in detail. The Viper tool chain is introduced in Section 5 to give the context for the evaluation of CVC4 described in 6. Finally, the significance of our contributions is outlined in the conclusion in Section 7.

2 Background

In this section, selected fundamental concepts used for automated reasoning are explained with a focus on applications for SMT solvers. This section is heavily tailored to the SMT solvers Simplify, Z3 and CVC4, which have similar architecture and are relevant to this thesis.

These SMT solvers support many of different theories which can be switched on and off selectively. Supported theories include propositional logic, uninterpreted functions, integer and real arithmetic, bit vectors, arrays and quantifiers. This list is not exhaustive. This modular approach allows efficient solving of simple problems while still having the option to enable more sophisticated theories if needed. Consequently, enabling and disabling modules has a great impact on runtime. In order to combine multiple theories, the solvers have a common representation — the E-graph, described in Section 2.5 — which is modified by each of these theory modules. The main focus of this thesis is on the quantifier module, which deals with quantified assertions.

The process of proving some conjecture is called *proof search*. In this process the SMT solver tries to learn new facts and combine known ones to either find a model satisfying the conjecture or a conflict rejecting it. If the solver cannot prove a conjecture with its known facts, it might make *case splits*. When case splitting, the solver explores the conjecture separately for an assumption and its negation. By doing so, the solver gains an assumption in both branches, which might help to successfully prove a conjecture as a whole.

2.1 Terms

Terms are the building blocks of all facts known to an SMT solver. They are represented as finite trees of function applications. Even constants are represented as function applications of their respective data type, returning a constant value (e.g. 5 is represented as an application of `Int()` returning 5). Terms consist of three parts: a name, a unique identifier and a list of direct subterms. The size of a term is the number of nodes in the complete term tree. Terms with the same identifier are identical. Term equalities are equalities proven by the SMT solver. Which equalities can be proven is dependent on the enabled theories. This is visualized in Figure 1. Terms with the same name but different identifiers and identical sets of subterms are also equal. Terms are immutable: this means that every occurrence of a term with a certain id is the same.

The SMT solver learns terms from different sources. The set of ground terms is directly taken from the solver input file. From these terms, the solver generates new ones by using its modules such as the arithmetic theory module or the quantified assertions module.

2.2 Quantifiers

Central to this thesis is the concept of quantified assertions; quantifiers in short. There are two fundamental types of quantifiers in predicate logic supported by SMT solvers: the existential quantifier $\exists$ and the universal quantifier $\forall$. While SMT solvers allow the use of existentially quantified assertions, internally they are removed using skolemization [8]. This allows the solvers to deal with existential quantifiers the same way as with universal quantifiers, using an instantiation

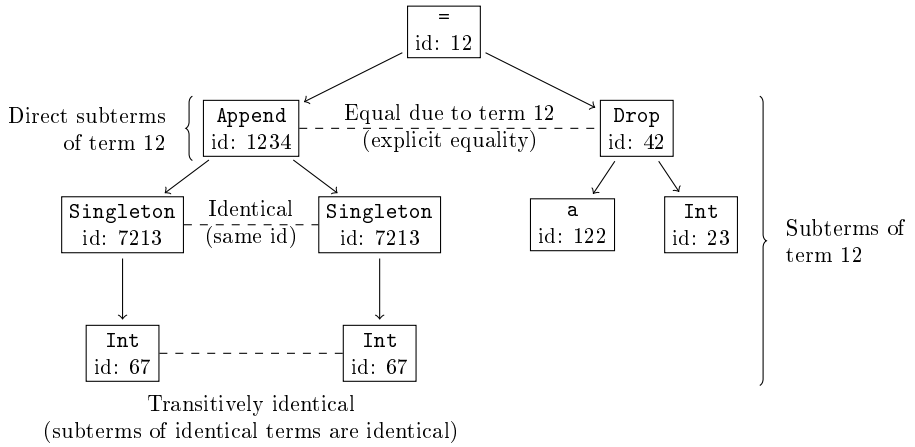


Figure 1: Tree representation of a term. The differences between identical and equal terms as well as direct subterms and subterms are indicated. The size of the term =(id 12) is 9.

based approach. For the remaining part of the thesis all quantifiers are regarded as universal quantifiers.

Quantifiers can be instantiated. Given a quantifier $\forall x : P(x)$ and some term a the resulting term $P(a)$ can be generated. The quantifier is an assumption and not the conjecture to prove. Quantifier instantiations are used abundantly by SMT solvers to learn new terms.

Quantified theories are usually hard to solve [5]. Part of the reason is that by introducing quantifiers, the number of possible terms involved in a proof becomes endless when using infinite domains. Infinite domains, such as integers, sequences or sets are very common in software verification. To elaborate, assume the quantifier $\forall x : P(x)$ for any integer x. The SMT solver could now learn all the terms $P(x)$ for any integer, which are infinitely many. Therefore SMT solvers need some way of deciding which quantifiers to instantiate for which terms.

A common approach to handle this problem is using heuristics to reduce the number of possible instantiations. More specifically, quantifiers may be annotated with one or multiple *patterns*. A pattern is a term describing term shape. The shape captures the concrete function applications as well as their hierarchy. Conceptually, the shape described by the pattern defines a context in which the instantiation of the quantifier is meaningful. The SMT solvers then use an approach called E-matching to decide whether or not a given term is within the context specified by the pattern. This process is also called *matching*. A pattern must contain all the bound variables. A concrete example of a pattern is shown in Example 1. A detailed explanation of how pattern matching works is given in Section 2.5. If done correctly, this is an effective measure to reduce the number of instantiations. It does, however, bring its own difficulties as described in Section 2.6.

Example 1 Example of a quantifier pattern.

A quantifier $\forall x : P(x)$ annotated with a pattern $req(x, y)$ will only be instantiated for terms with that structure (e.g. $req(foo(a), b)$). In particular, an isolated occurrence of a term that is not an argument to $req()$ will no longer trigger an instantiation.

2.3 Quantifier Instantiations

SMT solvers use quantifier instantiations to generate new terms. From the point of view of an SMT solver, a quantifier can be thought of as a function which, if applied to some input, returns new terms. This function application is called *instantiation*. The shape of possible inputs to this function is specified by the pattern, whereas the output is defined by the quantifier. The resulting new terms are called *yield terms*. Terms are new, if before the instantiation these terms did not exist. In particular, the terms used as input to the instantiation are not new and thus never contained in the yield terms. Terms that were bound to the free variable of the quantifier are referred to as *bound*. Terms that were not bound but nonetheless relevant for the instantiation because they matched the pattern are denoted as *blamed*. All bound terms are contained in the set of direct subterms of all blamed terms.

An instantiation *blames* another instantiation if any input terms of the former are contained within the yield terms of the latter. This causal relation between instantiations forms a directed acyclic graph, because an instantiation can blame multiple other instantiations and the blame relation is antisymmetric and irreflexive. This graph is called *blame graph*. An instantiation blames multiple other instantiations if its quantifier pattern was matched against multiple terms that were the result of different instantiations.

The *weight* of an instantiation represents its importance in the sense that instantiations with higher weight have more descendants in the blame graph. It is the number of elements in the set of instantiations that transitively blame the instantiation divided by the number of other instantiations that are also blamed by this set of instantiations.

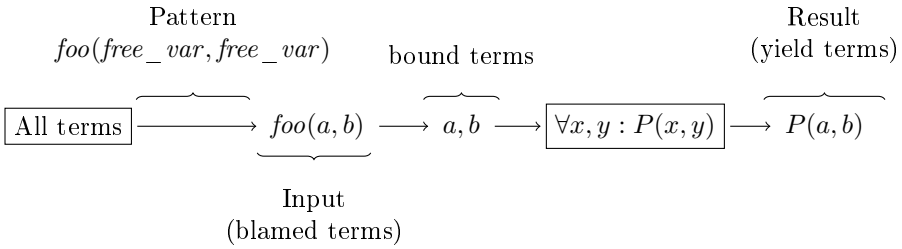


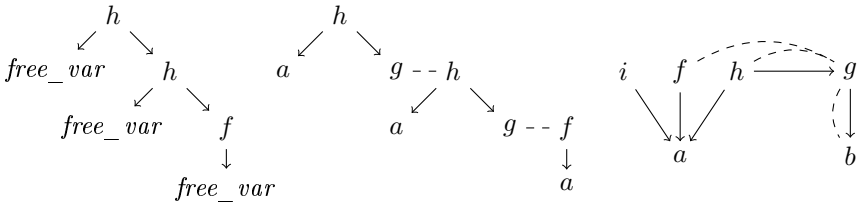
Figure 2: Schematic overview of a quantifier instantiation.

2.4 E-Graph

The E-graph is an efficient way of representing all terms known to an SMT solver. It is used as the common representation of the solver state for all modules to work on. Changes to the E-graph are reversible which enables interactive mode. Interactive mode allows a user to make assumptions, verify a conjecture using them and then discard them again. This feature is extensively used by Silicon, one of the verifiers of the Viper tool chain described in Section 5.4, to explore both sides of a possible split in the execution path of a program.

The E-graph is created by embedding all terms within a single graph. The tree structure of the terms is kept. Additional bidirectional edges are inserted for known equalities. The equivalence class of a term is found by visiting all terms reachable by only following the equality edges. This allows the solver to quickly find equal representations of a known term. Using the E-graph, infinite equivalence classes (e.g. nested function applications of the identity function) can be expressed in a finite representation. Because inequality cannot be explicitly expressed within the E-graph, SMT solvers usually use additional data structures to do so. Example 2 shows an E-graph and also E-matching explained in the next Section.

Example 2 Example of an E-graph and E-matching.



Pattern | Term matching the pattern, modulo equalities | E-Graph

On the right, an E-graph built from the terms $f(a) = g(b)$, $h(a, g(b)) = g(b)$, $g(b) = b$ and $i(a) \neq f(a)$ is shown. The solid arrows represent the term structure, whereas the dashed lines signify equality. By following the dashed lines, transitive equalities can be found (e.g. $h(a, b) = f(a)$). The inequality $i(a) \neq f(a)$ is only implicitly expressed by the absence of an equality edge. This format is particularly useful to express nested equalities:

$$b = g(b) = g(g(b)) = ... = g^n(b)$$

The pattern $h(\mathit{free\_var}, h(\mathit{free\_var}, f(\mathit{free\_var})))$ shown as a tree on the left can be successfully matched using the terms present in the E-graph, by using the appropriate equalities. A schematic overview of that process is shown in the middle. The concrete term $h(a, h(a, f(a)))$ is never explicitly generated. The bindings are as follows: to both h pattern terms, the corresponding $h(a, g(b))$ term is bound and the f pattern binds $f(a)$. These are reported as blamed terms. The term bound to *free_var* is a in all three occurrences.

2.5 E-Matching

E-matching is the process of matching a term to a pattern using the E-graph. The resulting matching is modulo equalities, in order to allow more matches and to prevent multiple instantiations of terms within the same equivalence class (which would also yield an equally equivalent result).

The E-matching proceeds in the following way: find a term in the E-graph that matches the root of the pattern. Then visit each subpattern and subterms. If they do not match, find a term that does within the same equivalence class. If there exists such a term, continue the matching process with the equal term instead of the subterm. The match fails only if there is no such equal term. A term matches a pattern if the name and the number of children are the same or the pattern is a free variable. A term that is matched against a free variable is bound and used to instantiate the yield terms. Terms matched against other patterns are reported as blamed terms.

Matching against specific patterns gives rise to the notion of *relative positions* of an instantiation. The notion of relative positions enables to refer to the set of terms having the same role in different instantiations. Valid relative positions occur in three categories: blamed, bound and yield terms. Relative positions are then further differentiated, depending on the category:

- Bound terms: for bound terms the relative position is defined by the free variable the term is bound to.
- Blamed terms: for blamed terms the relative position is defined by the pattern the term was matched to.
- Yield terms: for yield terms the relative position is defined by the term in the quantifier body it corresponds to.

2.6 Matching Loops

Choosing appropriate patterns is difficult: if the patterns are too restrictive, important instantiations might not get triggered thus leading to proof failure. If the patterns are too general there might be a loop: quantifier instantiations triggered by previous instantiations. This happens if an instantiation yields terms that match patterns of other quantifiers. These quantifiers are then instantiated using the yield terms. If a chain of such instantiations involves the same quantifiers in a repeating fashion, this behaviour is considered a loop. Such loops are called *matching loops*. This dilemma is outlined in Example 3. Matching loops can involve an arbitrary number of different quantifiers. The *size* of a matching loop is the number of instantiations within one loop iteration. The same quantifier might be instantiated multiple times in different roles within a single loop iterations.

Finding and understanding matching loops is tricky because they often occur due to the combination of different quantifiers that come from different sources. For example, quantifiers generated by a verification tool might cause a loop in combination with quantifiers written in the user-provided precondition for a method in the input program.

SMT solvers have deployed various heuristics to detect and avoid matching loops during proof search. These heuristics range from explicitly detecting

loops, assigning instantiation limits by number or depth to quantifiers, to implementations of balancing mechanics, such that instantiations with less nesting are processed first [11, 24]. Despite these efforts, matching loops cannot be completely avoided using these heuristics. For some proofs, the classic signs of matching loops, deep nesting and repeating instantiations, are actually required. Example 13 which appears in Section 3.4.7 shows this in detail.

Example 3 Illustration of the matching loop problem.

Given is a quantified assertion relating the lengths of two sequences with the length of their concatenation:

$$\forall x, y : Length(Append(x, y)) = Length(x) + Length(y),$$

where x and y represent arbitrary sequences. This quantifier causes a matching loop when it is used without a pattern. No pattern means, that there are no restrictions imposed on x and y. The reason for the matching loop is that the quantifier generates a new term that triggers an instantiation, namely $Append(x, y)$. Given an instantiation where the bound variables are the sequences a and b generates the new sequence $Append(a, b)$ since it is required to express the term $Length(Append(a, b))$. This new sequence can be used to instantiate the quantifier again to produce the following result:

$$Length(Append(Append(a, b), b)) = Length(Append(a, b)) + Length(b)$$

Note that the sequence $Append(a, b)$ also causes a loop in conjunction with a since both a and $Append(a, b)$ are sequences which can be bound to either x or y. To fix this, the quantifier should be annotated with the pattern $Append(x, y)$. This pattern requires the concatenation of the sequences to already exist. Therefore the only new terms generated by an instantiation of the quantifier are the $Length(...)$ terms, which do not cause another instantiation. The pattern is also not too restrictive since it still allows to relate the length of any existing sequence concatenation to the lengths of its two subsequences.
An example of a too restrictive pattern would be $Append(x, Singleton(y))$, where $Singleton(y)$ represents a sequence with only one element, y. Using this pattern would restrict the instantiations to just the cases where an arbitrary sequence is concatenated with another one of length one. The SMT solver could therefore no longer prove that the concatenation of two empty sequences is indeed zero.

3 Z3 Axiom Profiler

The Z3 Axiom Profiler is a tool to troubleshoot problems with Z3. More specifically, it is concerned with quantifiers and their instantiations. The Z3 Axiom Profiler aims to help when troubleshooting problems with quantifier instantiations. This information is made available by parsing the trace logs of the Z3 SMT solver. The information in these logs is presented in a very condensed way by the original Z3 Axiom Profiler. The tool was developed by Microsoft Research as part of VCC [6]. VCC allows the user to annotate C code to verify its correctness. Like Carbon, one of the verifiers in the Viper tool chain described in Section 5.1, VCC uses Boogie [2] as a back end. Therefore, when using VCC some of the same problems concerning quantifier instantiation arise (e.g. matching loops in user provided annotations). The purpose of the Z3 Axiom Profiler is to facilitate understanding and resolving these issues. In particular, the tool emphasizes expensive parts of the proof search, such as the quantifiers that were most often instantiated. This is done by both listing the most expensive quantifiers first as well showing how the proof search progressed with a particular focus on the number of case splits.

The overall goal of this project was to improve the reliability of software verifiers with respect to the reliability issues propagated by the underlying SMT solvers. The comparison of Z3 to CVC4 (Section 6) shows that switching solvers does not help with that issue. Since the evaluation of the two SMT solvers was in favour of Z3, for the remaining time of the project the focus was on improving the tools available to understand and avoid issues with SMT solvers. This included improvements to the Z3 Axiom Profiler. The improvements were both focused on making the information already presented by the tool more accessible as well as implementing functionality that enhances and enriches the shown information, such as the use of colour to guide the user. Further, the functionality of the Z3 Axiom Profiler was extended to help understand and solver occurrences of the matching loop problem in particular. Multiple such features were implemented. Consequently, finding and understanding matching loops is easier and faster with the improved tool. Features of the original tool that were not improved were left in place.

3.1 Z3 Trace Log

The Z3 trace log is a verbose log file printed by Z3 during proof search. It is the input for the Z3 Axiom Profiler. The log file contains information about all terms, quantifiers and instantiations. Additionally, information about scopes, case splits and conflicts is recorded as well. An excerpt of a such a log file is given in Example 4.

Since recording every action of Z3 would be overwhelming, the log is only concerned with proof search and quantifier instantiation. Thus, any information about other solver modules such as the arithmetic module is completely missing. Equality information is only available if there are explicit terms in the Z3 input file or equality terms generated by quantifiers. Information about numeric constants is abstracted to function applications of their data types. Even for quantifier instantiations, the information is incomplete: the matching pattern and exact binding information is unknown.

The only information available for a quantifier instantiation is the terms

bound to the free variables as well as the blamed terms that matched the pattern. However, the information which blamed terms matched which part of the pattern and which bound term were bound to which free variables is missing. Because E-matching is modulo equalities, inferring this information from the blamed and bound terms is not trivial (see Section 4.1). Since modifications of Z3 to write more complete log files were outside of the scope of this thesis, binding information reconstruction was added to the Z3 Axiom Profiler instead.

Example 4 Example excerpt from a Z3 trace log.

```
...
[new-match] 000000F6A814A958 #1125 #836 ; #2553
[mk-app] #4197 = #2651 #8
[mk-app] #4198 or #1394 #2646 #4197
[mk-app] #4199 or #1637 #1394 #2646 #4197
[instance] 000000F6A814A910 ; 3
[attach-enode] #4197 3
[assign] #4197 justification
[end-of-instance]
[mk-app] #4200 or #1423 #2660 #2661
...
```

The `[new-match]` lines indicates a possibility to instantiate a quantifier. The actual instantiation happens later at the line starting with `[instance]`. Lines starting with `[mk-app]` define new terms.

3.2 Original Version

The original version of the Z3 Axiom Profiler was a side product of VCC. It is used to troubleshoot adverse SMT solver behaviour, such as long runtimes and non-termination. Figure 3 shows the main window of the original Z3 Axiom Profiler. The tool already had many views:

- Main panel: a tree view containing detailed information about quantifiers, instantiations, terms, scopes and conflicts. Nodes can be expanded further to show their relations to other nodes, such as all instantiations for a specific quantifier or all instantiations that were caused by another one.
- Info panel: text field with more information about the node selected in the main panel. Terms are written in prefix notation as shown in Example 5.
- Visualization of the proof strategy: visualization of the search tree, where every node represents a case split. Leaves represent cases that Z3 was able to prove unsatisfiable and backtracked.
- Instantiation bitmap: an automatically generated image, where each pixel represents an instantiation. The colour indicates which quantifier was instantiated. The pixels are ordered row wise chronologically.
- Graph showing the causal relations between quantifiers: visualization of a graph, where the nodes represent quantifiers, scaled by the number of instantiations. The edges represent how often an instantiation of one

quantifier cause an instantiation of the other one, scaled by the number of occurrences. An example of such a visualization is shown in Figure 4. This visualization was added later by Clément Pit–Claudel [22].

In this original version the visualizations are helpful to get an overview of the problem and finding the relevant quantifiers. But to fully understand a matching loop problem in detail, the main panel and the info panel proved to be more useful. The reason is, that in order to understand matching loops, the concrete details of a loop must be visible. Otherwise the relation between looping instantiations is unclear. The main panel and the info panel are the two only places in the original tool where concrete albeit limited information is accessible.

3.3 Issues

Despite the many different views on the data, understanding a sizeable matching loop problem remains difficult and tedious. The main reasons are that the relevant looping instantiations are hard to find and large terms are hard to read. Terms are printed in a format that requires manual effort to understand. Combined with the fact that any log file contains millions of terms, this issue

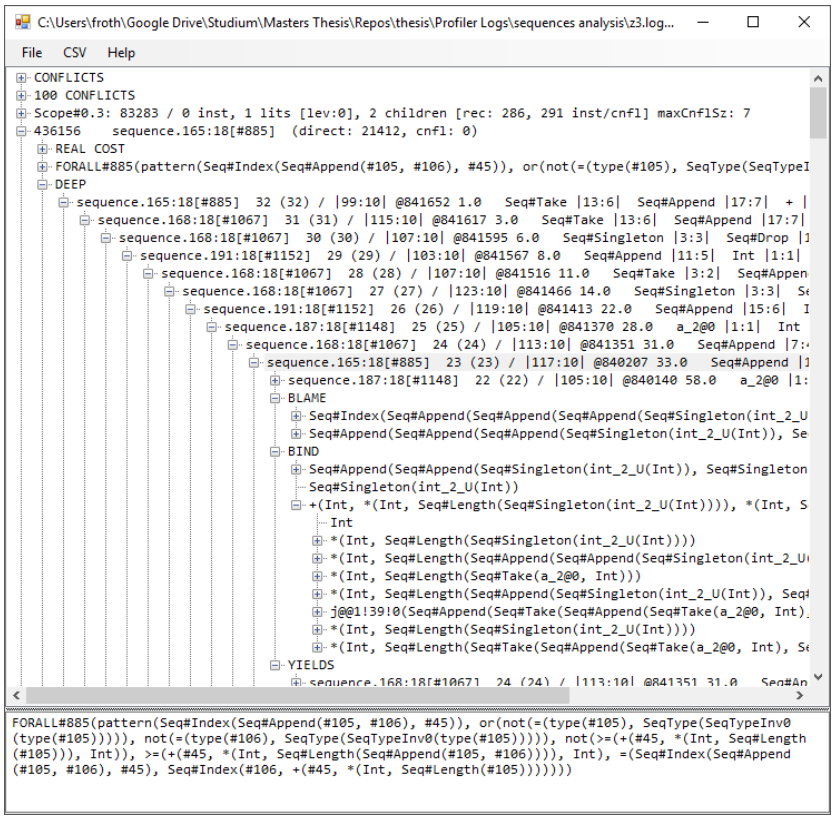


Figure 3: The main window of the original Z3 Axiom Profiler, showing the tree view and info panel in the original horizontal layout.

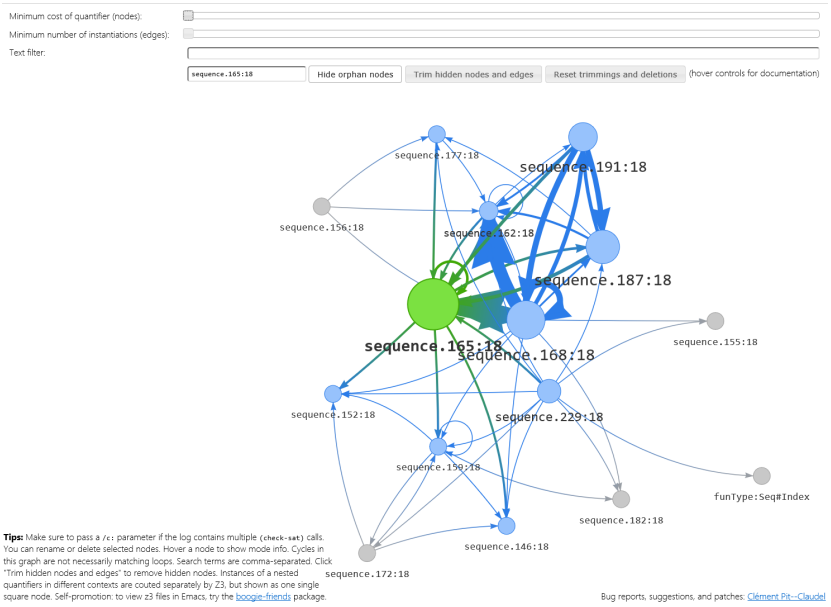


Figure 4: Visualization of the blame relations between quantifiers. This is a dynamic JavaScript visualization opened within a browser.

has a major negative impact on usability.

Relevant instantiations are hard to find because the tool shows all instantiations categorized by quantifier. This is useful to identify problems with single quantifiers but does not help when exploring issues that involve multiple quantifiers, as matching loops commonly do. Large terms are commonplace when dealing with matching loops, as the initial starting terms get nested deeper and deeper with each loop iteration.

It is difficult to find long paths in the blame graph formed by the blame relations of instantiations. This is relevant because matching loops tend to form long paths, due to the their repeating nature. The main panel shows all quantifiers and, as children of these top level nodes, the associated instantiations. This makes it cumbersome to follow a path because matching loops usually involve multiple different quantifiers, as visible in Figure 3. Additionally, some matching loops require setup instantiations that are not part of the loop but produce the necessary terms for the matching loop to start. These instantiations make it more difficult to find matching loops as then the repeating pattern starts

Example 5 Term printed in the original printing style.

```
Seq#Append(Seq#Take(Seq#Append(Seq#Take(a_2@0(), Int()), Seq#Append(Seq#
Singleton(int_2_U(Int()))), Seq#Drop(a_2@0(), Int()))), Int()), Seq#Append
(Seq#Singleton(int_2_U(Int())), Seq#Drop(Seq#Append(Seq#Take(a_2@0(), Int
()), Seq#Append(Seq#Singleton(int_2_U(Int())), Seq#Drop(a_2@0(), Int())))
, Int())))
```

This is a real example taken unaltered from the original Z3 Axiom Profiler.

deeper. The same is true for trailing instantiations when exploring, starting from the most deeply nested instantiations: paths containing looping behaviour might cause other instantiations that are not involved in the matching loop, but are matched to terms generated as part of the loop. This effect happens often, since SMT solvers try to instantiate quantifiers in a balanced way. Balancing quantifier instantiations is one heuristic to avoid the impact of matching loops; quantifiers get instantiated in the order of some measure of relevancy. In particular, relevancy usually decreases with deeper nesting.

To illustrate this point with realistic numbers: for a matching loop involving three quantifiers and starting after five setup instantiations, to even see three repetitions (to realize it is a loop), the tree view has to be expanded to a depth of 14 levels. This shows that the views were not designed with this use-case in mind.

As mentioned before, large terms printed to the info panel are hard to read due to the flat prefix notation. More specifically, the hierarchy of the term is hard to recognize, since the only structure comes from the separating characters (parenthesis and commas), which are used abundantly. The hierarchy is one of the key elements in pattern matching and thus necessary to understand a matching loop. Repeated subterms are also very hard to recognize. This is shown in Example 5 referred to previously, which contains multiple repeated subterms, such as `Seq#Take(a_2@0(), Int())`.

Lastly, the node labels in the main panel contain a lot of information in very little space. To do this, almost every bit of information is abbreviated, omitting helpful information, especially for instantiation nodes. This is shown in Example 6.

Example 6 Example of an abbreviated instantiation node description.

The following information is contained in the node name of instantiations as indicated by the help text:

```
<name of quantifier> <depth> (<weighted depth>) / |<size of bindings>|
@<line in log file> <cost> <head of binding 0> |<size of binding 0>| ...
```

For a concrete instantiation:

```
sequence.165.18[#885] 3 (3) / |26.:6| @2501 10.0 Seq#Append |11:5| Seq#
Singleton |3:3| Seq#Length...
```

Extracting the relevant information from this node name requires more mental effort compared to the new instantiation details shown later in Figures 9 and 10 and yields less information.

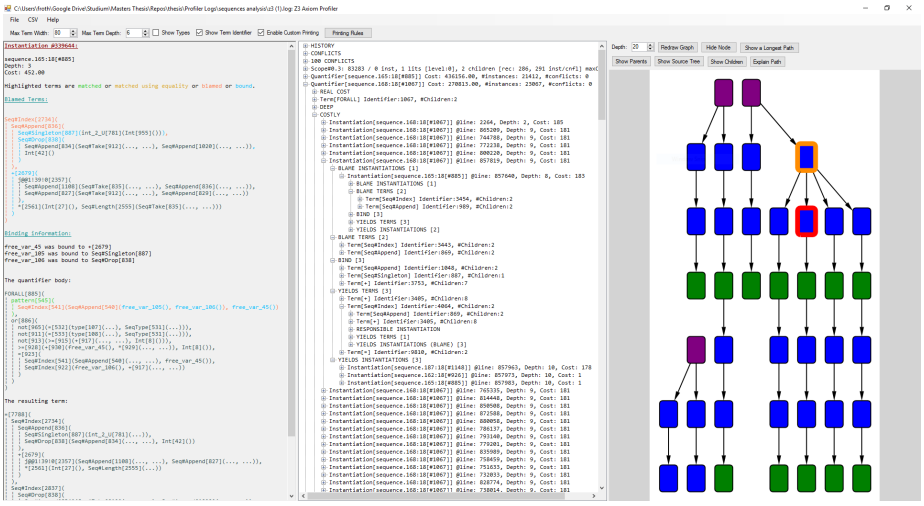


Figure 5: The main window of the redesigned Z3 Axiom Profiler. The newly introduce three column layout shows side by side the info panel, tree view and graph view (left to right). This allows to simultaneously see detailed information and the bigger context of any quantifier, instantiation or term.

3.4 Improvements and Extensions

This section presents the improvements to the Z3 Axiom Profiler made during the project. These includes a complete redesign of the interface as well as the addition of many new features.

3.4.1 Accessibility Improvements

Several fundamental changes to the user interface were made in order to make the available information more accessible. In general, relevant information is now displayed right from the start and more easily readable. In particular, the addition of explanations with related information, such as blamed, bound and yield terms, convey the presented information clearer to users than before. This additional information is called context. The previous way of obtaining the same information was more tedious: the info panel only showed information concerning exactly the selected node in the tree view. More information concerning the reason for its existence and the effects on the proof search had to be collected by selecting other relevant nodes and combining the information manually. More information on context information is available in Section 3.4.4.

GUI Improvements The layout of the main window was reworked to display more information and to make the commonly used operations easily accessible. The main window now shows three vertical panels: the info panel, the tree view and the graph view. The info panel displays information about the selected element. It was inherited from the original Z3 Axiom Profiler and enhanced with more functionality such as pretty printing and context information. The tree view, also already present in the original tool, shows all elements of the trace log in an easily accessible way. It can be used to find expensive quantifiers,

specific instantiations or explore the relations between terms and instantiations. A history of previously selected instantiations is provided as well. Finally, the graph view was introduced as a powerful new view on the data and is described in detail in Section 3.4.3. The main window was split vertically because all three panels benefit more from additional height rather than width. A screenshot of the redesigned main window is shown in Figure 5.

Pretty Printing Terms are the foundation of almost all the elements displayed in the Z3 Axiom Profiler. The quantifier bodies are expressed as terms named `FORALL` with subterms describing the patterns and the assertion. Instantiations are collections of terms containing all blamed, bound and yield terms. Therefore almost every interaction with the tool breaks down to reading terms. Because of that, it is extremely important that terms are easy to read, compare and recognize. The original printing style made this very difficult as outlined in Section 3.3.

All terms displayed in the info panel are now pretty printed. By default, the prefix notation is kept but enhanced with indentation according to the term hierarchy. Hierarchy is a good way to convey the term structure which is important for E-matching. Emphasizing the term structure therefore helps to understand pattern matchings.

For better readability, line length and maximum printing depth can be customized. Toggles to disable printing of term unique ids and generic type information can be used to shorten the notation. The effect of these options is demonstrated in Example 7. The difference in clarity of the default prefix notation is apparent when comparing the term shown in Example 8 with the same term printed in the original notations, as shown in Example 5.

The printing behaviour can be further customized using rewriting rules described in Section 3.4.2. Printing rules can be used selectively and mix well with the default prefix notation. This allows to increase readability incrementally by adding rules one by one during the exploration of a trace log. By combining all these elements, even large terms can be reduced to a concise, familiar and easily readable form.

Example 7 Effects printing customization using the toggle options.

The default prefix notation can be customized with two toggle options: 'Show Term Identifiers' and 'Show Types'. The effects are demonstrated below.
Full term with identifiers in square brackets and type information in angle brackets:

```
append<ISeq~ISeq~ISeq>[115](
¦ es@1[109](),
¦ sgltn<Int~ISeq>[114](e@2[111]())
)
```

Term with only identifiers displayed:

```
append[115](es@1[109](), sgltn[114](e@2[111]()))
```

Term when both term identifiers and type information is disabled:

```
append(es@1(), sgltn(e@2()))
```

Example 8 Term printing using the default prefix notation with hierarchical structure.

```
Seq#Append(
¦ Seq#Take(
¦ ¦ Seq#Append(
¦ ¦ ¦ Seq#Take(a_2@0(), Int()),
¦ ¦ ¦ Seq#Append(
¦ ¦ ¦ ¦ Seq#Singleton(int_2_U(Int())),
¦ ¦ ¦ ¦ Seq#Drop(a_2@0(), Int())
¦ ¦ ¦ )
¦ ¦ ),
¦ ¦ Int()
¦ ),
¦ Seq#Append(
¦ ¦ Seq#Singleton(int_2_U(Int())),
¦ ¦ Seq#Drop(
¦ ¦ ¦ Seq#Append(
¦ ¦ ¦ ¦ Seq#Take(a_2@0(), Int()),
¦ ¦ ¦ ¦ Seq#Append(
¦ ¦ ¦ ¦ ¦ Seq#Singleton(int_2_U(Int())),
¦ ¦ ¦ ¦ ¦ Seq#Drop(a_2@0(), Int())
¦ ¦ ¦ ¦ )
¦ ¦ ¦ ),
¦ ¦ ¦ Int()
¦ ¦ )
¦ )
)
```

Repeated subterms such as `Seq#Take(a_2@0(), Int())` are clearly recognizable.

3.4.2 Rewriting Rules

Often terms get large due to deep nesting, syntax technicalities and long function names. Although pretty printing is a major step forward in readability, often notation could still be more concise if tailored to the specific situation. Additionally, users searching for matching loops are usually very familiar with the domain the terms are from (e.g. sequences). There is no reason why the notation of terms should reflect the internal names used by Z3, especially if using another notation would convey the presented information more clearly to the user.

To alleviate that problem rewriting rules were introduced. Rewriting rules allows a user to replace the prefix notation with arbitrary syntax, usually resulting in a more concise and natural notation. The printing style in Example 10 demonstrates this clearly. This example shows the same term as shown in Examples 5 and 8.

Rewriting rules can be either imported from an existing rule set[1] or created by using the rewriting rule editing dialogue shown in Figure 6. A rewriting rule consists of several elements:

- Match string: this string determines whether a printing rule gets used to

[1] Rewriting rules can be imported from / exported to CSV.

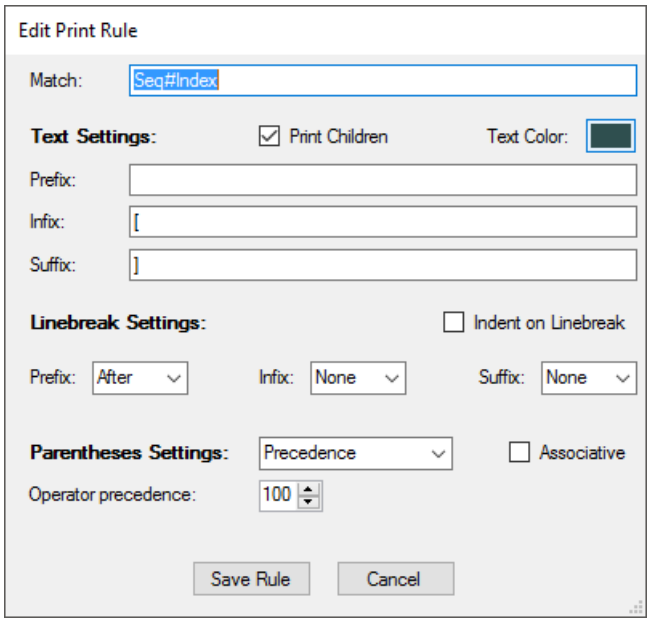


Figure 6: Rewriting rule editing dialogue. The rule shown here replaces applications of `Seq#Index(a, b)` with `a[b]`.

print a particular term. The match string is checked against the following properties, in descending order of binding strength: first the term id, then the term name together with the generic type, and finally just the term name. If no rule matches any of these properties, the term is printed in default prefix notation.

Having multiple levels of binding strength makes the feature much more flexible. This allows users to single out terms with specific identifiers while still having all other terms with the same name printed according to other custom rules; For example to name a '+' term with known value while still using the infix notation for all other '+' terms. This is also used internally for highlighting by defining temporary rules with different colours for these specific terms. The uses of term highlighting are discussed in Section 3.4.4.

- Prefix, infix and suffix: these strings get inserted before, between or after subterms, respectively. The colour option applies to all three strings. They can also be left blank. A blank printing rule can be used to 'hide' terms. This is useful for conversion terms needed to formally connect different domains but that do not carry information for the problem at hand, as shown in Example 9.

  Another sensible use case is the ability to reintroduce known constants. In the trace log there is no information about concrete numbers. Instead, function applications showing just the data type (e.g. `Int()`) are given. By using pretty printing, these function applications can be rewritten as actual number literals. Different applications of `Int()` can be distinguished by their identifier.

- Print children: toggles whether or not subterms should be printed. If this option is off, the term is printed as a constant string replacement and all subterms are ignored. This can be used to abbreviate known terms.
- Line break settings: print rules also control the behaviour regarding line breaks and indentation. Line breaks can be enforced, suppressed or added by line length. Indentation on line breaks can be switched on and off.
- Parentheses settings: by using these settings, the number of parentheses can be reduced to increase readability. These options allow to set operator precedence and associativity to flatten repeated applications of the same operator and exploit intuitive binding rules.

Example 9 Example of a blank matching rule.

Given a rule matching `int_2_U` with blank prefix, infix and suffix, suppressed parentheses and line breaks, and enabled subterm printing, then `Singleton(int_2_U(Int()))` will be printed as `Singleton(Int())`.

Example 10 Term printing using rewrite rules.

Example of a term printed using rewriting rules. This term is from the sequence domain. Naturally, the notation was changed to a more familiar one. The following four rules were used:

- `Seq#Append(a,b)` was replaced with `a ++ b`.
- `Seq#Take(a,n)` was replaced with `a[n..]`.
- `Seq#Drop(a,n)` was replaced with `a[..n]`.
- `Seq#Singleton(a)` was replaced with `{a}`.

The term is then printed as follows:

```
(a2[..1] ++ {-1} ++ a2[2..])[..1] ++ {22} ++
(a2[..1] ++ {-1} ++ a2[2..])[2..]
```

Associativity of operators and operator precedence was exploited to reduce the number of parentheses needed. The default prefix notation of this term is shown in Example 8.

3.4.3 Graph View

The right panel of the main window shows a reduced version of the blame graph given by the blame relations between instantiations. It is the only view which allows a user to easily see the relation between multiple instantiations of different quantifiers. It was implemented using the 'Microsoft Automatic Graph Layout' library [18].

The graph view provides extensive filter options, which are, in contrast to the tree view, not limited to instantiations of a single quantifier as shown in Figure 7. This view is useful to discover expensive instantiations and long paths, both of which are indicators for matching loops. Therefore it is an important tool to find matching loops.

By default, only the 40 most expensive instantiations are displayed. The instantiations are colour coded by quantifier. By using the 'Redraw Graph' button, the complete graph, up to the selected depth, can be redrawn. When clicked, an instantiation is marked as selected and its parents highlighted. The info panel then shows information about the selected instantiation. When an instantiation is selected, multiple operations are available:

- Hide: any instantiation can be hidden. All visible instantiations that are descendants of only the selected instantiation are hidden as well. This allows the user to reduce the amount of information shown and focus on more relevant instantiations.
- Show Parents: display all instantiations that the selected instantiation blames for its existence.
- Show Source Tree: transitively display all ancestors of the selected instantiation.
- Show Children: show all instantiations that blame the selected instantiation for their existence.
- Show a Longest Path: reveal a single, longest path through the complete graph, such that the path includes the selected instantiation. This feature is useful to reveal matching loops, as matching loops form long paths naturally.
- Explain Path: apply the path explanation feature to the most expensive, visible path through the selected instantiation. The path explanation feature is described in Section 3.4.5.

In order to avoid situations where a user accidentally displays an excessive number of nodes, a prompt is displayed whenever an operation would add more than 40 new nodes to the graph. This could easily happen, since it is not immediately apparent how many children a specific node has or how many nodes are within a specific depth of the graph. The prompt suggests to reduce the number of new nodes by filtering. When the user decides to filter, the filter dialogue shown in Figure 7 is displayed. Filtering options include depth, selecting specific quantifiers, cost and chronological order.

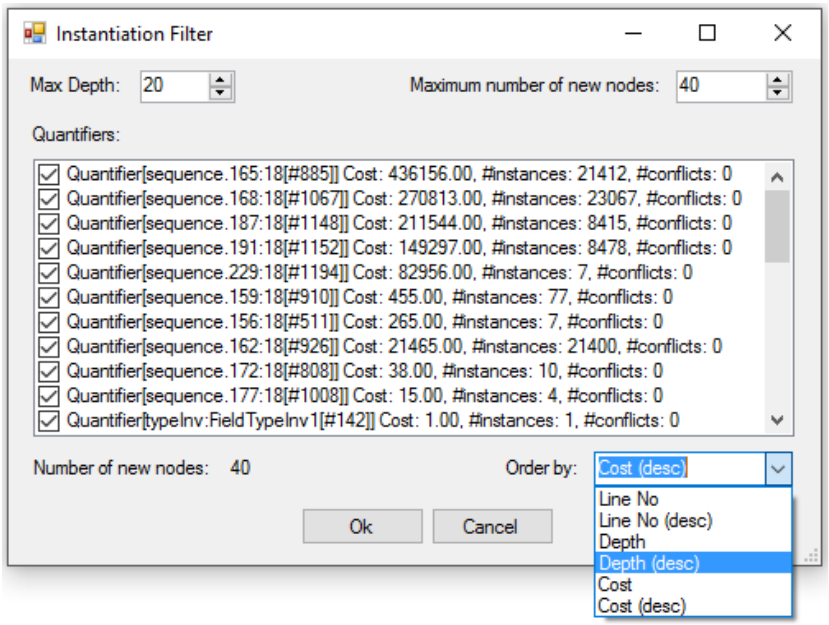


Figure 7: Filter dialogue for the graph view. This dialogue is shown whenever an action would cause more than 40 new nodes to be added to the graph.

3.4.4 Context Information

The redesigned info panel displays contextual information. Context is necessary to understand why instantiations happen and helps users to better judge their significance. This feature takes information from various sources and compiles it in a sensible way, conveying more information to the user with less cognitive overhead.

In the original Z3 Axiom Profiler this information was split across different nodes within the tree view. This caused two problems: first, the information had to be manually collected from all these nodes in order to understand which specific terms caused a quantifier to be instantiated and how these relate to other instantiations. This is a time-consuming and error prone task. Second, the lack of context also leads to information duplication by listing every blamed and bound term separately, despite them all being nested terms. This is illustrated in Example 11.

Context information is displayed for the most important elements in the Z3 Axiom Profiler, namely terms, quantifiers and instantiations. For quantifiers and terms, meta information such as the number of instantiations, the total weight or the number of direct subterms is displayed. A unique identifier for the selected element is also shown: the identifier for terms, the 'qid' for quantifiers and the trace log line number for instantiations. If the selected element is an instantiation, more relevant context displayed: in addition to just displaying the quantifier body, the blamed, bound and resulting terms are also displayed.

Binding reconstruction Binding reconstruction is a very important feature because it enables a much more sophisticated presentation of instantiations and creates the prerequisites for the other advanced features, namely path explanation (Section 3.4.5), matching loop detection (Section 3.4.6) and matching loop generalization (Section 3.4.7). It alleviates the issue of information missing from the trace log as described in Section 3.1. By matching the quantifier patterns to the reported blame terms, the missing information concerning quantifier instantiation is recovered. The implementation details of the binding reconstruction algorithm are described in Section 4.1.

This feature allows the Z3 Axiom Profiler to highlight the pattern that was matched for any quantifier instantiation. This not only indicates which pattern was used but also reinforces the meaning of the colour coding since the same colours are used for patterns and free variables as for bound and blamed terms. Pattern highlighting is helpful if there are multiple patterns, since the matched pattern is not always obvious, particularly when equalities were used in the match. Additionally, it also gives information about which term was matched to which subterm of the pattern as well as the context it was matched in. The context specifies which occurrence of a given term was actually matched.

In many cases, due to the hierarchical nature of patterns, blamed and bound terms are nested. Hence, by just displaying the largest term, all others are displayed as well. Knowing the exact bindings and the context allows the tool to exploit this and collapse nested blamed and bound terms instead of listing all terms individually, as shown in Figure 8.

By only presenting the largest terms with colour coded subterms, the information conveyed is improved twofold: first, there is less text, leading to a clearer presentation and less distraction caused by duplicated information. Second, having multiple terms collapsed into one adds information, as the subterm relations between the collapsed terms is now explicit: The different blamed and bound terms are still clearly indicated using highlighting. The reconstructed binding information also contains the equalities that were used in order to per-

Example 11 Information duplication as a consequence of missing context.

Given the following pattern:

```
pattern(
¦ Seq#Index(
¦ ¦ Seq#Append(free_var_105(), free_var_106()),
¦ ¦ free_var_45()
¦ )
)
```

This pattern consists of six terms, two are matched to blamed terms and three are free variables which determine the bound terms. If this pattern is matched without using equality, then the bound terms are all nested within both blamed terms. Additionally, the term matched to `Seq#Append` is also contained in the other blamed term. The original Z3 Axiom Profiler reports both blamed terms and all three bound terms separately regardless. This means that the bound term `free_var_105` is reported three times. The only term reported only once is the `Seq#Index` term because it is the largest and therefore not contained in another blamed or bound term.

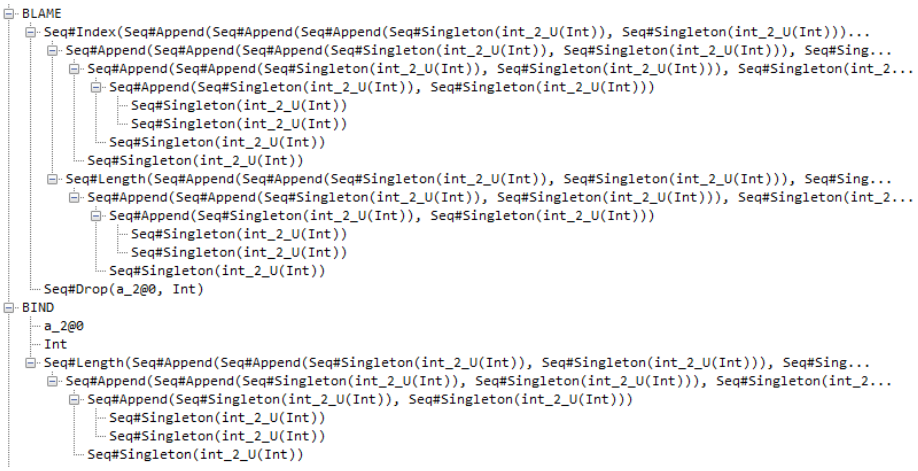


Figure 8: Bindings as reported by the original Z3 Axiom Profiler. The same instantiation is shown as in Figure 9 and 10. The bound terms are duplicated as they are also contained within the blamed terms. It is non-trivial to reconstruct the exact matchings: the bound terms have to be placed back into the context of the blamed terms and the pattern has to match the blamed terms. Since it does not, there must be an equality involved.

form the matching. As described in Section 2, E-matching matches against equivalence classes of terms in order to allow more matchings. Explicitly listing the equalities and highlighting the involved terms, makes it easier to understand such matchings that require equalities to work.

Figures 9 and 10 showcase all of the previously mentioned improvements in a concrete example. The figures show the detailed instantiation information that is printed to the info panel. Two rewrite rules, described in Section 3.4.2, were used: one to hide the `int_2_U` terms and another to shorten the application of `Seq#Singleton(x)` to `{x}`. The following improvements are apparent in the example when compared to the binding information shown in Figure 8:

- The default prefix notation printing style conveys hierarchy clearly and mixes well with rewriting rules.
- The depth cutoff for terms is useful to prune information that is too detailed. In the example the cutoff is set to depth 5.
- It is now immediately apparent how the bound terms relate to the blamed terms. Even more complicated relations such as equalities are clearly recognizable.
- The explicit binding information makes it easier to understand how the result of an instantiation relates to the inputs. This is also supported by printing the triplet (input, quantifier body, output) in a concise way.
- Terms that were matched only due to a term equality are marked in gold. To avoid confusion in matchings that involve multiple equalities, the explicit equalities are given as well.

- It is apparent how more vertical space is beneficial to the info panel.

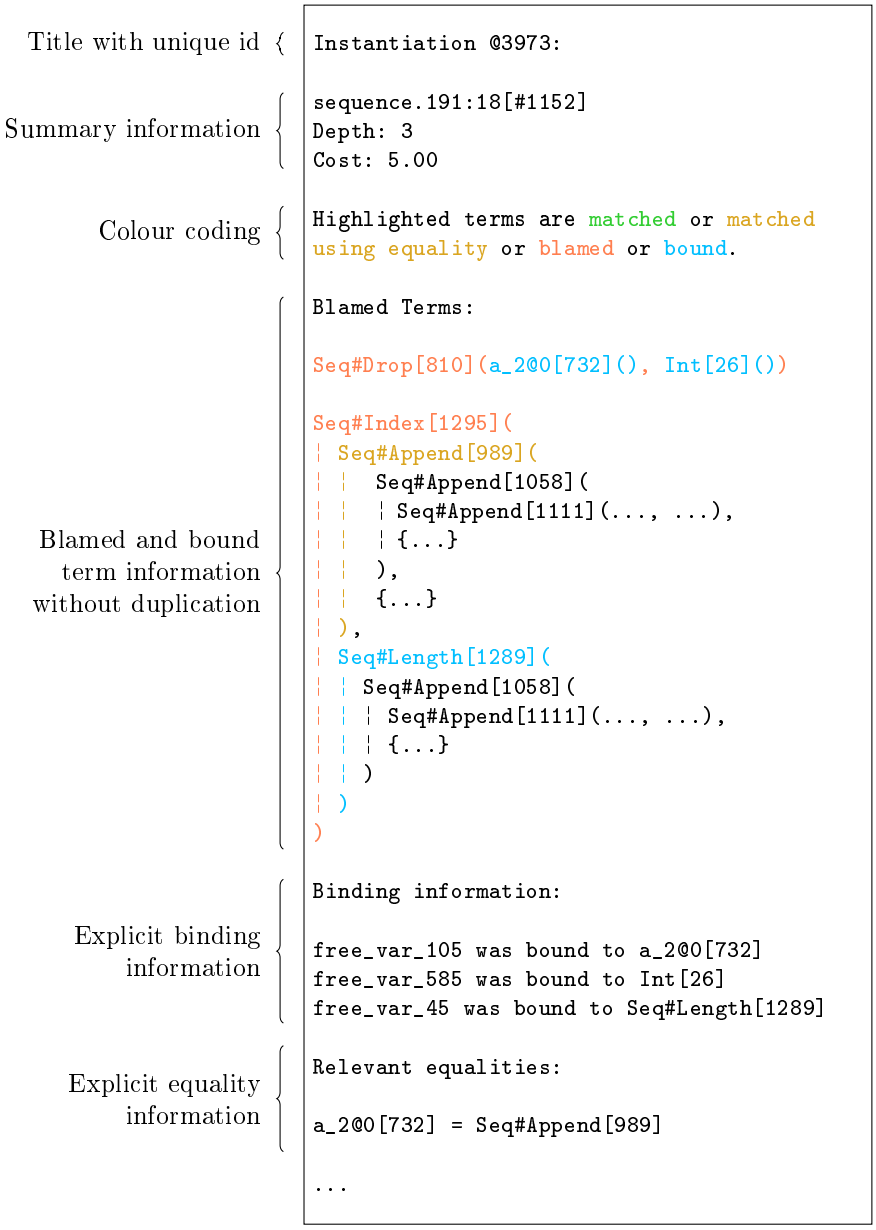


Figure 9: Detailed description of a specific instantiation as shown in the info panel, part 1 of 2. Part two is shown in Figure 10. Additional line breaks have been added for better readability.

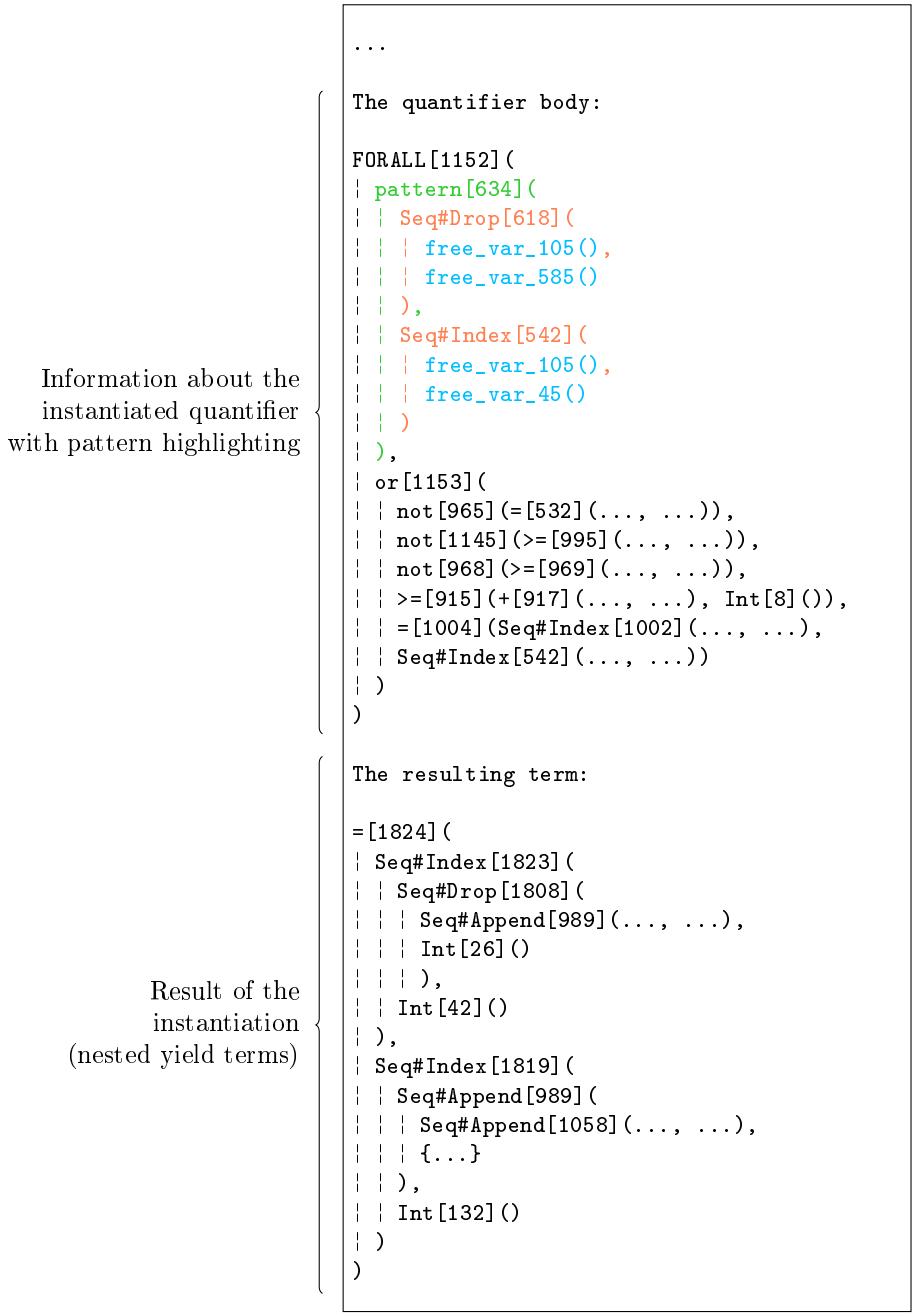


Figure 10: Detailed description of a specific instantiation as shown in the info panel, part 2 of 2. Part one is shown in Figure 9. Additional line breaks have been added for better readability.

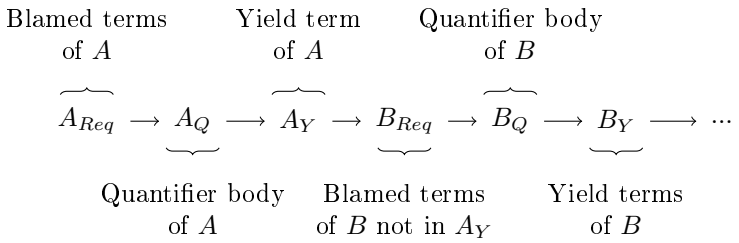


Figure 11: Schematic overview of the terms included in a path explanation. Only the beginning of a generic path with the instantiations $A \rightarrow B \rightarrow C \rightarrow ...$ is shown.

3.4.5 Path Explanation

The path explanation feature was added as a new feature during this project. It is a key component in understanding matching loops and other deeply nested chains of instantiations. This feature was specifically designed to display the relation between two instantiations and to emphasize how long paths are formed.

Given a path through the blame relation graph, the path explanation feature prints a detailed explanation to the info panel. The blame relation is formed by one instantiation blaming a term that was created by another instantiation. Consequently this happens whenever a term created by an instantiation is matched by a quantifier pattern.

The path explanation feature shows the blamed term explicitly within the resulting term of the blamed instantiation. An overview of the terms shown by the path explanation feature is shown in Figure 11. It is a useful tool to better understand the blame relations of instantiations. More specifically, the path explanation shows the blame relations between instantiations explicitly by highlighting the terms that cause the next instantiation within the result of the previous one.

The resulting path explanation is a chain of instantiation descriptions similar to those of a single instantiation. An excerpt of such an explanation, showing just two instantiations linked by the yield terms of the first one, is shown in Figure 12. The key difference is that the blamed and bound terms of a child instantiation are highlighted within the resulting yield terms of the parent instantiation. This adds the context to connect the two instantiations. Blamed terms that are not contained within the result of the preceding instantiation are listed separately. This is done to provide sufficient information so that every instantiation in the path can be explained. To keep the path explanation focused on the specific path some information is omitted: instantiation meta information, concrete bindings as well as explicit equality information is not shown. If this information is needed, it can be easily obtained by selecting the specific instantiation.

Quantifier body of the previous instantiation including pattern highlighting (the matched terms are not shown)

Result of the previous instantiation including highlighted blamed and bound terms of the next instantiation

Additional terms required for the next instantiation including highlighted blamed and bound terms

Quantifier body of the next instantiation including pattern highlighting

```
...
Application of sequence.168:18[#1067]

FORALL(
¦ pattern(
¦ ¦ Seq#Index(free_var_16(),free_var_45()),
¦ ¦ Seq#Append(free_var_15(),free_var_16())
¦ ),
¦ or(=(Seq#Index(...), Seq#Index(...)))
)

This instantiation yields:

=(
¦ Seq#Index(
¦ ¦ Seq#Append(
¦ ¦ ¦ Seq#Append(..., ...),
¦ ¦ ¦ Seq#Singleton(...)
¦ ¦ ),
¦ ¦ +(...)
¦ ),
¦ Seq#Index(
¦ ¦ Seq#Singleton(int_2_U(...)),
¦ ¦ +(...)
¦ )
)

Together with the following term(s):

Seq#Drop(a_2@0(),Int())

Application of sequence.187:18[#1148]

FORALL(
¦ pattern(
¦ ¦ Seq#Index(
¦ ¦ ¦ Seq#Drop(free_var_15(),free_var_55()),
¦ ¦ ¦ free_var_45()
¦ ¦ )
¦ ),
¦ or(
¦ ¦ >=(+(..., ..., ...), Int()),
¦ ¦ =(Seq#Index(...), Seq#Index(...))
¦ )
)

This instantiation yields:
...
```

Figure 12: Example of two instantiations where the first one caused the second one. They are shown in the style used within the path explanation feature. This example is heavily shortened to fit the page.

Example 12 Path with possible matching loops.

Given a concrete path containing multiple instantiations of the quantifiers A, B and C:

$$A \to B \to C \to B \to C \to B \to C \to B \to C \to A \to B \to A \to A \to A \to B$$

This path contains multiple repeated patterns, but only one is a possible matching loop: the instantiations $B \to C$ are immediately repeated four times. It is therefore reported as a possible matching loop. Other repeating patterns do not repeat immediately ($A \to B$) or less often than the most repeating candidate (A).

3.4.6 Matching Loop Detection

Finding loops in concrete paths is not a trivial task. It requires examining all instantiations of the path within the context provided by the others. For a given instantiation, the relevant context is comprised of the yield terms of the preceding instantiation and the blamed terms of the subsequent instantiation. Furthermore, instantiations of the same quantifier do not necessarily indicate a loop: if there are multiple patterns available, using different patterns is regarded as different behaviour and therefore not looping. To help users quickly identify possible matching loops, automated matching loop detection was implemented.

Whenever a path explanation is generated, the path is also examined for possible matching loops. If a potential matching loop is found, the path explanation is prepended with a short summary of the loop as well as a generalized version of the looping quantifier instantiations. The loop summary contains the loop length, the number of repetitions and the quantifier pattern. The loop generalization is described in more detail in Section 3.4.7. If multiple potential matching loops are found, the loop with the most repetitions is presented.

For this feature, immediately repeating patterns of quantifier instantiations are considered loops. Immediately repeating means, that there must be no other instantiations between two repetitions of the repeating pattern. This is illustrated in Example 12. Since every instantiation in a concrete path is different from every other one, the instantiations have to be categorized. Instantiations in the same category are regarded as a repetition. The repeating patterns are found by only considering the category of instantiations within the path. Instantiations that agree on the quantifier, the matched pattern and the number of equalities belong to the same category. In order to be flagged as a loop, a minimum of three repetitions is required. This threshold was set to reduce the number of false positives. The implementation details of the matching loop detection are explained in Section 4.2.

In some cases matching loops might be reported erroneously or inaccurately. One cause of false positives is that the loop detection cannot distinguish between repetitions that can continue forever and ones that cannot. An example of a repetition that cannot continue forever is the unpacking of a large term: quantifiers with patterns that require nested terms might match repeatedly on terms with deeper nesting than required by the patterns. These repetitions cannot continue infinitely, because they are limited by the nesting levels of the initial term. This is shown in Example 13. On the other hand, the reported match-

ing loop might be inaccurate, because the instantiation categories are not tight enough to always distinguish instantiations of the same quantifier with different roles. In such cases, the reported loop is shorter than the actual matching loop because one actual repetition gets erroneously reported as two. Therefore it is important to manually check and comprehend the reported matching loops. Example 14 illustrates this point.

3.4.7 Matching Loop Generalization

Understanding matching loops is difficult. Even if the concrete path explanation is available, it is still challenging to recognize the features that allow the loop to continue infinitely. More concretely, to understand a matching loop, one must know which terms exactly are generated by an iteration of the loop and how these terms lead to another iteration. While the matching loop generalization does not present all that information directly, it is still a good support feature to better convey specifics of a matching loop.

The matching loop generalization feature generates summaries of the potential matching loops found by the path explanation feature. The summary produced contains all information that is common across all loop iterations. Therefore the matching loop generalization allows a user to quickly distinguish between the changing behaviour and the static behaviour in a suspected matching loop. Here, static behaviour refers to terms and term structures that do not change across loop iterations. This static behaviour is detected by comparing all loop iterations with each other. The comparison is made separately for each *loop position*. The loop position is defined as the number of instantiations that preceded the current one within this iteration. This means that the sets of instantiations of a quantifier at certain loop positions are compared separately from those at different positions. The result of this process is a loop iteration of instantiations, which contains only static behaviour. All the changing term structures are abstracted in the comparison. This process is called *generalization*. The implementation details of the loop generalization algorithm are explained in Section 4.3.

The generated description of the loop is therefore valid for all its iterations. This description is presented similarly to the content produced by the path explanation feature. The quantifier bodies and terms are printed in the same alternating fashion, as shown in Figure 13. The highlighting of blamed and bound terms is analogous as well. There are, however, important differences:

Example 13 Non-looping repeating instantiation.

Given a quantifier $\forall x : f(x)$ annotated with the pattern $f(g(free\_var))$. If there is a ground term given in the solver input of this shape:

$$f(g(g(g(g(g(g(g(g(g(g(a)))))))))))$$

then the quantifier above can be instantiated 12 times unpacking the nesting levels one by one. The repeating pattern is therefore repeated often enough to be reported as looping behaviour. The last instantiation matches $f(g(a))$ and yields $f(a)$ which no longer matches the pattern and thus stops the repeating behaviour.

Example 14 False positives due to different usages of instantiations of the same quantifier.

Given a repeating instantiations of the quantifiers A, B and C:

$$A \to B \to C_1 \to A \to B \to C_2$$

This might be reported as two repetitions of the loop $A \to B \to C$ since the algorithm cannot always distinguish different use cases of the same quantifier (C in this example). An example of this would be a quantifier that relates the length of two sequences with their concatenation: At one position it might be used to calculate the length of the concatenated sequence whereas at another it might generate the terms necessary to calculate the length of just one of the concatenated sequences.

- The terms between quantifier bodies are generalized across iterations. These generalized terms are the result of generalizing the set of terms at the same relative position. A set of terms is generalized by reducing it to a single term with common term structure. Differences are reduced to subterms named `generalization_#n`, where `#n` is an identifier to distinguish the different generalizations. If the same set of terms is reduced multiple times within the same iteration, it is given the same identifier. The implementation details of the generalization algorithm are outlined in Section 4.3.
- The first and the last term in the description are the same, but from different loop iterations. These terms define the common shape of all terms in that relative position. In particular, since these terms are from different loop iterations, they are never the same in any concrete loop. The first term is repeated in order to give context to the first quantifier instantiation. To indicate that the first term belongs to the previous loop iteration, the generalizations are printed with a dash: `generalization_#n`$'$
- The generalized subterms are marked in purple. This emphasizes the changing parts within the loop.
- The colour coding of the matched terms changes meaning: instead of indicating which exact term was blamed or bound, it is now a representation of the term structure. The exact binding can be retrieved by consulting the detailed instantiation explanation of the instantiation in question.

Due to term generalization, there are no details in the description concerning specific iterations. But this is also a strength of this presentation: all terms

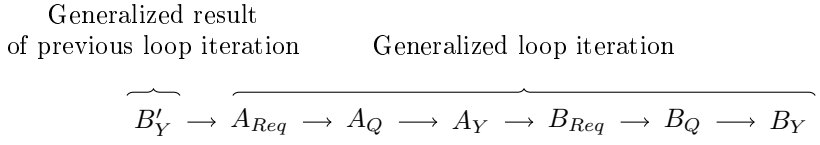


Figure 13: Schematic overview of the terms included in a generalized matching loop explanation. A generic loop with the instantiation pattern $A \to B$ is shown.

that were not reduced to `generalization_#n` are guaranteed to appear in each iteration. This means that in order to understand the matching loop, special attention should be given to the generalized parts.

For most generalized instantiations, at least one bound variable is either directly bound to a `generalization_#n` term or a term that contains such a generalized term. This is a consequence of looping behaviour. Each iteration must produce new terms which trigger the next loop iteration. Therefore the terms bound to the free variables must also change. In rare cases, the matching loop generalization does not produce generalized terms. This happens when the term structure is indeed static across all iterations. In such cases, the change that triggers the looping behaviour is hidden behind the abstraction of concrete number literals (e.g. `Int()`). Blamed terms are never entirely replaced by generalized replacements since the blamed terms are part of the repeating behaviour and are used to match the same pattern across all iterations. Consequently, all these terms agree in structure with the parts that matched the pattern.

An example of a generalized loop description is shown in Figure 14. The loop consists of only one quantifier instantiation, therefore the input and the yield term have the same structure. Note that the generalized terms of the input and the yield term are not the same. This is indicated by the dashed notation. The reason is, that the input term is from one iteration earlier than the yield term. The generalizations within the input or the yield term, however, do represent the same set of terms.

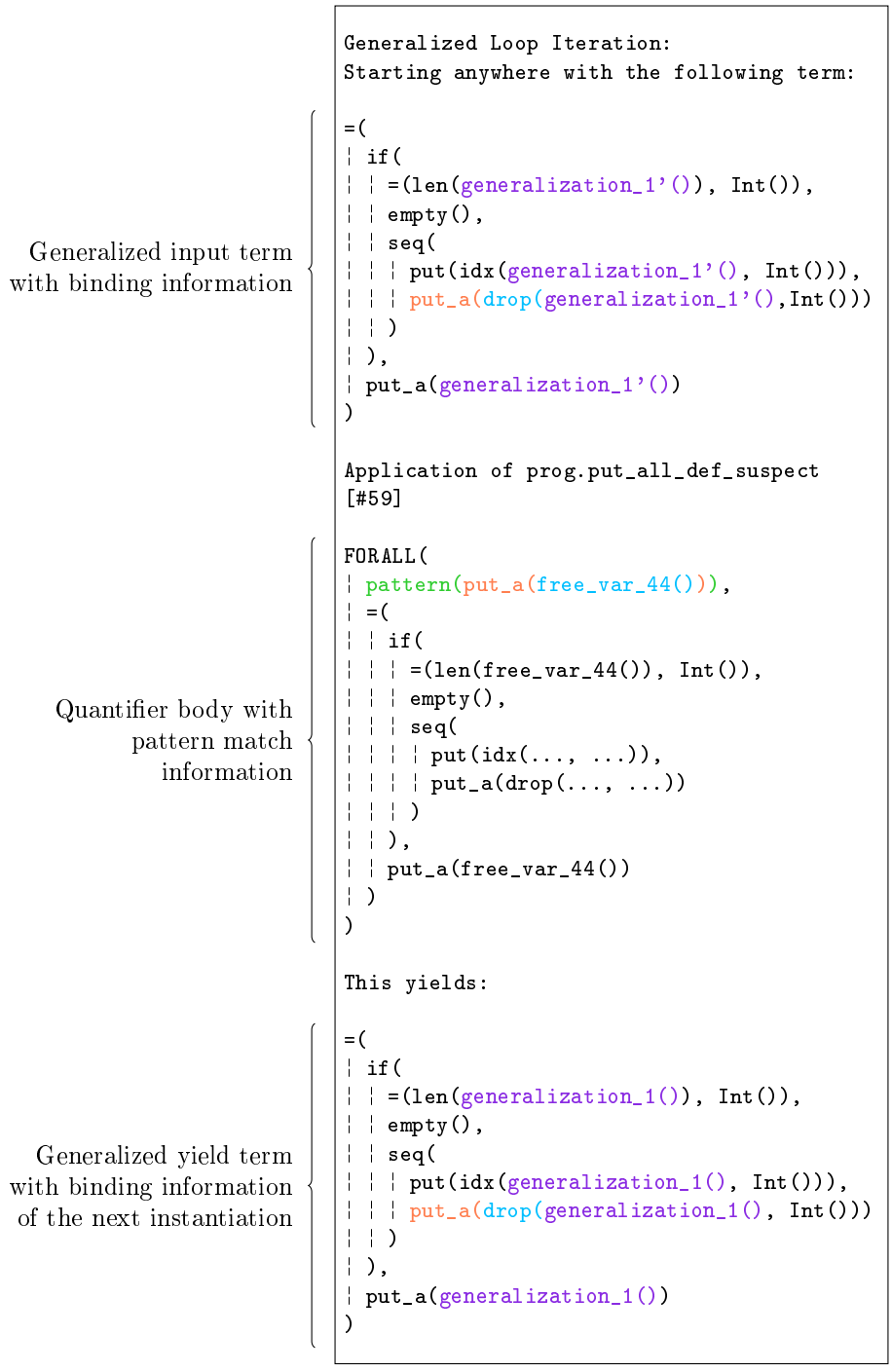


Figure 14: Generalized loop description of a loop of size 1. Note that the generalized input and yield terms do have the same term structure, but the differ in the concrete loop since the generalizations are from different iterations. This is a necessary requirement for a loop of size one, otherwise the instantiations would either not be looping or the loop would be longer.

4 Algorithms

This section outlines all algorithms and implementation details for the more complicated, new features of the Z3 Axiom Profiler.

4.1 Binding Reconstruction

As explained in Section 3.1, the binding information in the trace log is incomplete. More concretely: the log specifies which terms were blamed and which terms were bound. However, which pattern was matched, how the blame and bound terms relate to each other and which bound term was bound to which free variable within the pattern are unknown. Therefore, the algorithm described in Section 4.1.2 tries to reconstruct this binding information by over-approximating the E-matching. If this over-approximation yields multiple results, the algorithm tries to disambiguate by validating the matchings. Validating a matching means checking all equalities that are required for this matching. If the matching algorithm only returns one possible matching, the validation can be skipped, as there is nothing to disambiguate.

As explained in Section 2.5, pattern matching is not always straightforward. Thus the pattern matching reconstruction algorithm must be able to handle ambiguous cases as well as occurrences of implicit equalities. The algorithm relies on the following invariants learned by inspection of Z3 trace logs:

1. Each reported blamed term is matched exactly once against a pattern.
2. For each term in a pattern, except for free variables, there is at least one blamed term whose operator matches the pattern exactly.
3. Every free variable must be bound.
4. All terms bound to the same pattern or free variable are equal.

The algorithm then proceeds in two phases: the matching phase and the validation phase. In the matching phase, all possible pattern matchings that satisfy the invariants above are collected. The validation phase then tries to reduce the number of matchings by validating their assumptions if the matching phase reported multiple possible matchings. As explained in Section 4.1.4, calculating the binding information is very expensive. Therefore bindings are calculated lazily and at most once. The result is cached to be immediately accessible on later use.

The high level idea behind binding reconstruction is to traverse the patterns and try all possible matchings of blamed and bound terms to the pattern. During the traversal, information about the matchings, called *match state*, is carried along. The most important such information is, which blamed and bound terms are matched against which parts of a pattern. This information is stored in the *binding dictionary*. The exact data a match state is comprised of is described in Section 4.1.2. Whenever a pattern is visited, all possibilities to match this pattern with regard to the all the match states that are carried along are generated. This list of possible matchings replaces the previous list of match information. The list becoming empty signifies that it was not possible to match the current pattern given the previous match states. Since this list contains all possibilities at all times, this is an error state which should never occur, since Z3 managed to complete the match given the patterns and input terms.

4.1.1 Important Terminology and Concepts

This section outlines important concepts and terminology used in the description of the algorithm.

Binding Eviction The process of overwriting a binding in the binding dictionary is called *eviction*. Whenever a term is evicted, an equality is recorded. The eviction indicates that there exists another term that is different from the subterm of the parent blame term, with both being matched to the same pattern. Therefore, these terms must be equal for this match to be valid. This fact is checked in the validation phase (Section 2) if necessary. The newly bound term evicts the previously bound term. Note that binding the same term multiple times to the same pattern is not regarded as eviction. Therefore, eviction is an indication for either an equality or an invalid matching.

Pending Matchings The concept of pending matchings is important for efficiency: it allows matching the complete pattern within a single traversal, by postponing the matching of all the subterms to the subterms of the pattern. These matchings are checked when the appropriate subterm of the pattern is visited. Checking all matchings eagerly would result in repeated pattern traversals since checking transitively all the subterms results in a complete term traversal.

Pending matchings are subterms of terms that were already matched. Since the children of both patterns and terms are known, the patterns of the pending matchings are already known. The implications of applying the match, however, are not. Pending matchings might introduce new equalities by evicting existing bindings. When handling a pending matching, the subpatterns and subterms, if any, are added to the pending matching candidates. If a term is still bound after handling all pending matchings for the current pattern, then all pending matching candidates corresponding to this term are added to the pending matchings dictionary to be definitely handled later.

Example 15 Example of a match context.

Given the term `Seq#Drop[838]` which is matched within the following root term:

```
Seq#Index[2734](
¦ Seq#Append[836](Seq#Singleton[887](...), Seq#Drop[838](..., ...)),
¦ +[2679](j@@1!39!0[2357](..., ...), *[2561](..., ...))
)
```

Then the context of this match is the following path:
(`Seq#Index[2734]` → `Seq#Append[836]`)

Match Context Keeping track of the match context is important to distinguish between multiple occurrences of the same term within a larger term. To do this, paths as shown in Example 15 are used. A match context is added together with a pending match candidate. Only during handling of the parent term and the path to the pending match term is known. The context of a subterm is formed by taking all paths of the parent term context and appending the parent term to each of these paths. If there are multiple locations the parent

term could be matched in, there must also be the same multiple location for all of it subterms as well.

4.1.2 Matching Phase

The matching phase is concerned with collecting all possible matchings and is applied to each pattern in turn. If there are multiple patterns, the results of each pattern are aggregated in a collection of possible matchings, since the matching phase is an over-approximation of the E-matching. The validation phase (Section 2) following the matching phase is concerned with making the over-approximation precise. The most important procedures of the algorithm are given as pseudo code in Algorithm 1.

Pattern Traversal The TRAVERSE-PATTERN procedure of the matching algorithm traverses the input pattern, visiting each subpattern. While traversing the pattern, the matching algorithm has to keep track of the list of all possible matchings, subsequently denoted *matching-list*. This list contains matching state objects described above, representing the matchings. It is initialized with one possible matching: the empty matching. This matching represents the initial state before any blamed term has been matched to a pattern. When visiting a subpattern, the procedure MATCH-PATTERN is executed. The *matching-list* is updated to reflect the result.

The procedure MATCH-PATTERN is responsible for generating the list of all possible matchings given a list of candidate matchings and a pattern. In order to traverse the pattern only once, subpatterns and the corresponding subterms are matched later. The information about the pending matchings is stored in the match state. If the pattern represents a free variable, every candidate matches since every term matches a free variable. Therefore the resulting list of matchings has the same number of elements as the candidate list. Matching a free variable might add equalities, if there are different terms in the pending matchings of that free variable. This happens if there are multiple structurally different, nested blamed terms matched to the parent terms of the free variable. These differences will then cause binding evictions once the pending matchings are processed.

For any other pattern, the normal matching procedure is followed. For each remaining unused blamed term matching the pattern, the state is cloned. The required conditions for a match are equality of the term and pattern name and the number of children. More details about E-matching are explained in Section 2.5. The cloning is required to still have the original match state in case there are multiple matchings possible with the remaining blamed terms. Using the cloned match state, the blamed term is added to the pending matchings in the last position. This ensures that this term gets bound to the current pattern and is not evicted.

Algorithm 1 Main procedures of the binding reconstruction algorithm.

```
TRAVERSE-PATTERN(pattern)
1  result = EMPTY-LIST()
2  ADD(result, EMPTY-MATCHING())
3  // The order of traversal does not matter,
4  // as long as parents are visited before their children.
5  foreach subpattern in pattern:
6      result = MATCH-PATTERN(SUBPATTERN, RESULT)
7  return result

MATCH-PATTERN(pattern, candidates : list of matchings)
 1  result = EMPTY-LIST()
 2  foreach matching in candidates:
 3      if IS-FREE-VARIABLE(pattern):
 4          HANDLE-PENDING-MATCHINGS(matching)
 5          ADD(result, matching)
 6      else
 7          foreach term in matching.unmatched-blamed-terms:
 8              if MATCHINGS(pattern, term):
 9                  // Make a copy to keep the current state intact for the
10                  // following iterations.
11                  copy = CLONE(matching)
12                  ADD-TO-PENDING-MATCHINGS(copy, term)
13                  HANDLE-PENDING-MATCHINGS(copy)
14                  COLLECT-MATCH-CANDIDATES(copy)
15                  ADD(result, copy)
16  return result

HANDLE-PENDING-MATCHINGS(matching)
1  foreach pattern, term in matching.pending-matchings:
2      if MATCHES(pattern, term):
3          // ADD-TO-BINDING also deals with binding evictions.
4          ADD-TO-BINDING(matching, pattern, term)
5          foreach subpattern, subterm in (pattern, term):
6              ADD-MATCHING-CANDIDATE(matching, subpattern,
                                      subterm)
7              ADD-MATCHING-CONTEXT(matching, term, subterm)
8      else
9          ADD-EQUALITY-REQUIREMENT(matching, pattern, term)
```

Match State To keep track of the different match possibilities, stateful objects are used which track the following information:

- The binding dictionary, which maps patterns to blamed and bound terms. This dictionary is the central element of the matching object. A matching is only valid and complete if every blamed and bound term is mapped exactly once.
- The list of unmatched blamed terms to keep track of invariant 1. This list contains all blamed terms for the empty matching and none for a valid and complete matching.
- A dictionary of patterns mapping to a list of pending terms. This dictionary keeps track of the information that has to be checked later and enables to do the complete pattern matching with a single traversal of the pattern. Whenever a term is matched to a pattern, all the subterms, if any, will have to be matched to the corresponding subpatterns. Until these subpatterns are visited, the subterms are listed as pending matchings.
- A dictionary of terms mapping to a list of pending match candidates, where a match candidate is a pair of a pattern and a term. This dictionary is required in order to avoid unnecessarily matching (and possibly binding) subterms of terms that were evicted. More concretely, it is a temporary holding space for pending matchings while a MATCH-PATTERN call is in progress. Before and after calling MATCH-PATTERN this dictionary must be empty. This is required since only after having completely matched a pattern is it clear which pending matchings still need checking. Whenever a term is evicted from the binding dictionary, its pending match candidates are removed as well. Possible reasons for binding eviction are explained below.
- A dictionary of patterns mapping to a list of terms. This dictionary maintains a list of terms that are equal to the term that is mapped to the same pattern in the binding dictionary. This means, whenever a term is evicted, it is added to the list of terms corresponding to the pattern it was bound to previously, before the eviction. The reason is that the term that was evicted must be equal to the term that is bound to the corresponding pattern. Since evictions might happen multiple times for the same pattern, this storage format avoids storing the same equality multiple times. During validation these equalities are checked for validity in order to make the over-approximation of the matching phase more precise. Equality checking is explained in more detail in Section 2.
- Information about the context of a match. The context of a match is defined as a path of terms, from the root term where the term was matched in, to the matched term, as shown in Example 15. Since there might be multiple matchings in different contexts, the context information is recorded as a list of such paths. This information is required to be able to decide which occurrence of a given term within a bigger term was actually matched. It is mainly used for term highlighting in the detailed description of instantiations displayed in the info panel.

4.1.3 Validation Phase

Matching validation is used to disambiguate multiple possible matchings by discarding invalid ones. As mentioned previously, the matching phase is an over-approximation of the E-matching. Therefore, if there is only one matching found by the matching phase, the over-approximation is already precise and the validations phase is skipped. All equalities of that single matching candidate are considered valid, backed by the assumption that Z3 is sound and the fact that the quantifier was instantiated successfully. In case of multiple candidates being available, they are checked in turn in ascending order of the number of equalities that must be validated. Candidates with zero equalities are trivially valid. The first candidate that can be successfully validated is used and the rest discarded. This is sound since only one matching can be valid at a time. Additionally, validating matchings is expensive due to the procedure DIRECT-EQUALITY, therefore it is important that unnecessary validations are skipped. This procedure takes two terms and searches the space of all terms that have exactly those two terms as subterms. The search is successful if an equality (=) term is found. This search space can be very large, therefore matching validation is only used cautiously.

Because of missing information, in particular the E-graph, the tool is not able to prove inequality, only equal. If none of the matchings can be validated successfully, the binding reconstruction is considered ambiguous. This might happen due to equalities missing in the trace log. In such cases, the most likely matching is used and a warning is displayed. The most likely matching is the one with the fewest equality requirements.

4.1.4 Runtime Complexity

The runtime complexities for the matching and the validation phase are explained below.

Matching Phase The binding reconstruction has to consider all patterns of a quantifier, each of which with an arbitrary number of subpatterns. Relevant for the runtime are two measures: the total number of all subpatterns, denoted as n and the number of blamed terms denoted as k. For each of these n pattern the procedure MATCH-PATTERN has to be executed. In the worst case, every blamed term matches every pattern. This is, however, very unlikely. In that case, there are $\frac{n!}{(n-k)!}$ ways to match the k blamed terms to the n patterns.

A single execution of the procedure HANDLE-PENDING-MATCHINGS takes $O(ns)$ time, where s the maximum number of subterms for any given term. For real world applications s is usually small (e.g. less than 5). The outer loop of HANDLE-PENDING-MATCHINGS is executed at most n times. This is because the number of pending matchings for a specific pattern is limited by the maximum height of a pattern, which is smaller or equal to n. By definition, the number of inner loops is limited by s. All other operations of HANDLE-PENDING-MATCHINGS take constant time.

The procedure COLLECT-MATCH-CANDIDATES collects every pending match candidate and adds it to the pending match dictionary. Its time complexity is $O(ns)$, as there are at most s new pending match candidates for each of the

Algorithm 2 Matching validation procedures.

```
VALIDATE(matching)
1  foreach pattern, eq-terms in matching.equalities:
2      eq-base-term = matching.bindings[pattern]
3      foreach term in eq-terms:
4          if not RECURSIVE-EQ-LOOKUP(eq-base-term, term):
5              return false
6  return true

RECURSIVE-EQ-LOOKUP(term1, term2)
 1  if term1.id == term2.id:
 2      return true
 3  if DIRECT-EQUALITY(term1, term2):
 4      return true
 5  if term1.name ≠ term2.name ||
       term1.type ≠ term2.type ||
       term1.subterms.count ≠ term2.subterms.count:
 6      return false
 7  foreach subterm1, subterm2 in (term1, term2):
 8      if not RECURSIVE-EQ-LOOKUP(subterm1, subterm2):
 9          return false
10  return true
```

at most n pending matchings that had to be handled just before the call to COLLECT-MATCH-CANDIDATES.

The inner loop of the procedure MATCH-PATTERN is executed once for every blamed term for every possible way of matching the blamed terms to the patterns, $\frac{n!}{(n-(k-1))!}$ times in total. One loop iterations takes $O(ns)$ time due to the calls to HANDLE-PENDING-MATCHINGS and COLLECT-MATCH-CANDIDATES. The other procedure calls within the loop are constant time operations. Therefore, the time complexity of the matching phase is $O(ns\frac{n!}{(n-(k-1))!})$. In practice the factorial time complexity is not a problem. The reason is that all involved variables are usually small. The variables are small, because the k is bound by the size of the pattern. Patterns are small as they try to be as general as possible (without causing loops). Additionally, assuming that every blame term matches every patterns is a very coarse over-approximation.

Validation Phase Validating the equalities is very expensive as it involves reverse lookups of terms to their parents. This is expensive because terms are reused abundantly, thus the number of parents to search is very large. Additionally, the bound and blamed terms can have arbitrary sizes. This is especially an issue for matching loops as the deeply nested instantiations produce large terms.

In the worst case, every blamed term is matched via equality, thus leading to the same number of blamed terms and equalities, denoted as k. For each of the k equalities, two terms have to be compared using the procedure REVERSE-

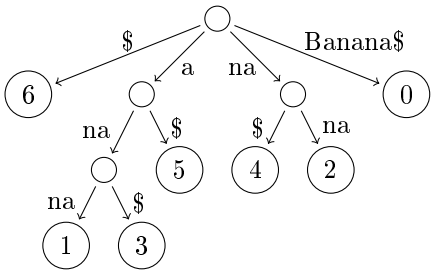


Figure 15: Suffix tree for the word 'Banana$'. The '$' character represents the end character, which ensures that no suffix is a prefix of another suffix. Edges are labelled with the prefix they represent. Leafs are numbered according to the start index of the suffix they correspond to.

EQ-LOOKUP. This procedures traverses both terms in parallel trying to find equal terms or enough differences that descending further is not necessary. In the worst case, every subterm of the smaller term has to be looked at and consequently causes a reverse lookup. The worst case cost of a reverse lookup is equal to the maximum number of different terms that have the same direct subterm, denoted t. This number is usually large. Let the size of the number of subterms of the smaller term be s and the biggest of all smaller terms in any equality validation be $s_{\max}$. Then the algorithmic time complexity of the full validation phase is $O(kts_{\max})$. In most cases, however, the equality lookup stops after few term comparisons on either equality or unknown. Also, in most cases, the validation phase is completely skipped because only one matching candidate is found during the matching phase.

If the validation process is considered too slow at some point, it could be easily improved. The equalities could be hashed during trace log parsing making the equality lookup a constant time operation. This would reduce the procedure REVERSE-EQ-LOOKUP to a simple term traversal of the smaller term with constant time operations on each subterm. The new runtime for REVERSE-EQ-LOOKUP would therefore be $O(s)$, which is optimal given the problem constraints. The total runtime of the validation phase would reduce to $O(ks_{\max})$.

4.2 Matching Loop Detection

Matching loop detection was implemented by reducing the problem to a repeated substring problem. The instantiation categories described in Section 3.4.6 are mapped to characters of a string. In order to produce viable matching loop candidates, the substrings have to be non-overlapping and immediate repetitions. The threshold k for the minimum number of repetitions to count as a matching loop was set to 3. This threshold is high enough to suppress most false positives.

4.2.1 Substring Problem

Substrings can be found efficiently by using suffix trees. An example of a suffix tree is shown in Figure 15. A suffix tree is a compressed trie containing all suffixes of a given string. Consequently, all inner nodes of a suffix tree represent suffixes with common prefixes, which are repeated substrings. The number of

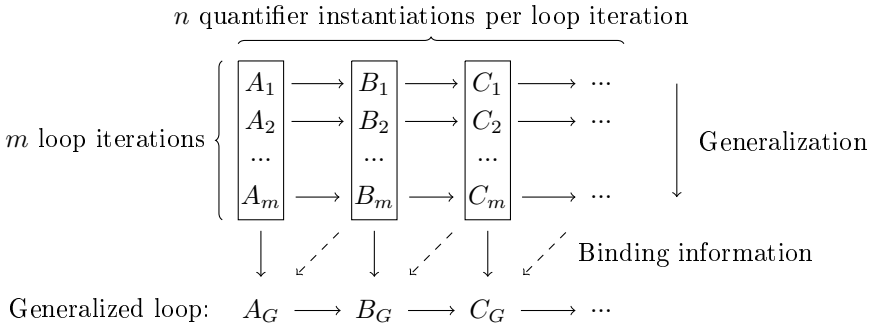


Figure 16: Schematic illustration of matching loop generalization. The boxes represent the different sets of instantiations at the same loop position. Binding information of the next instantiations is used to relate subsequent generalized instantiations.

repetitions is equal to the number of leaf descendants of that inner node. The repeated substring is represented by the path from the root to the inner node. The algorithm proceeds by traversing the inner nodes breadth first, collecting candidates that represent at least k immediate, non-overlapping repetitions. In order to facilitate this check, each inner node has references to all its leaf descendants. Additionally, the repeating substring is also stored directly at the inner nodes.

Immediately repeating, non-overlapping substrings are also called tandem repeats and are often encountered in bioinformatics. There exist more sophisticated algorithms for finding these [13]. However, the current implementation is considered sufficient for the use cases described above.

4.2.2 Runtime Complexity

Suffix trees can be built in $O(n)$ using Ukkonen's algorithm [27]. To find only immediate, non-overlapping substrings, at most $O(n)$ internal nodes of the suffix tree must be checked for these properties. In the worst case, this requires $O(n^2)$ checks, since many leaves might be checked multiple times for different substring lengths. This raises the total time complexity to $O(n^2)$. For matching loops that are contained in reasonably sized trace logs ($\leq$200 MB), the algorithm complexity is not a problem.

4.3 Matching Loop Generalization

The matching loop generalization is an algorithm to reduce all iterations of a matching loop to a single common representation. This is achieved by extracting the similarities of all instantiations at the same loop position. The algorithm is illustrated in Figure 16.

More formally, the generalization algorithm is concerned with the following problem: given a matching loop, extract the similarities between iterations and prune differences. The matching loop is described by a sequence of instantiations together with the length of one loop iteration. The algorithm has three steps:

1. Distribute all instantiations into sets such that each set represents all instantiations at the same loop position. Each set will be of the same size and have as many instantiations as there are loop iterations.

2. For each of the sets of instantiations, extract the sets of terms that need to be generalized. This is done by building a set of terms for each relative position that needs generalizing. For a given relative position, the new set of terms is formed by taking the term at that relative position for each instantiation in the instantiation set. The following relative positions need generalizing: the yield terms and any blamed term consistently not contained within the yield terms of the previous instantiation since the generalization of these terms are shown in the generalized loop explanation.

3. Generalize each set of terms generated by point two using the algorithm described in Section 4.3.1.

To generalize the sets of yield terms, additional bookkeeping is required to incorporate the binding information of subsequent instantiations.

4.3.1 Term Generalization

Terms are generalized by extracting the common hierarchical structure. A high level overview of how the term generalization algorithm operates is shown in Figure 17.

The term generalization algorithm is given a set of terms. This set of terms are traversed breadth first in parallel, visiting all sets of subterms. Each set of terms visited is generalized separately. The subterms of a set of terms are only visited if the terms *agree* in structure at the current level. A set of terms agrees in structure if and only if all terms have the same name and the same number of subterms. The structure of the subterms is not considered at this point. The generalization of an agreeing set of terms is a term reduced to the agreeing factors: the name and number of subterms. If the terms do not agree, the generalization is a term named `generalization_#n` with no subterms. The suffix `#n` is an identifier to keep different generalized terms apart. Additional bookkeeping allows the algorithm to identify repeating generalizations of the same terms such that the generalized terms are named consistently.

4.3.2 Runtime Complexity

The generalization of n terms with a minimum size of m takes $O(mn)$ time. Checking whether a term agrees with another takes constant time, therefore checking a set of n terms takes $O(n)$ time. The number of sets that need to be checked is determined by the smallest term, since all larger terms get pruned to the smaller size, thus m sets of terms have to be checked.

To generalize a matching loop of size n with m repetitions, at least n sets of yield terms have to be generalized. In many cases, there are additional blamed terms that have to be generalized, the number of such terms denoted as k. This results in a total of $n + k$ sets of terms of size m. The minimum term size is denoted as s. Taking all this into account, the matching loop generalization algorithm takes $O((n + k)ms)$ time.

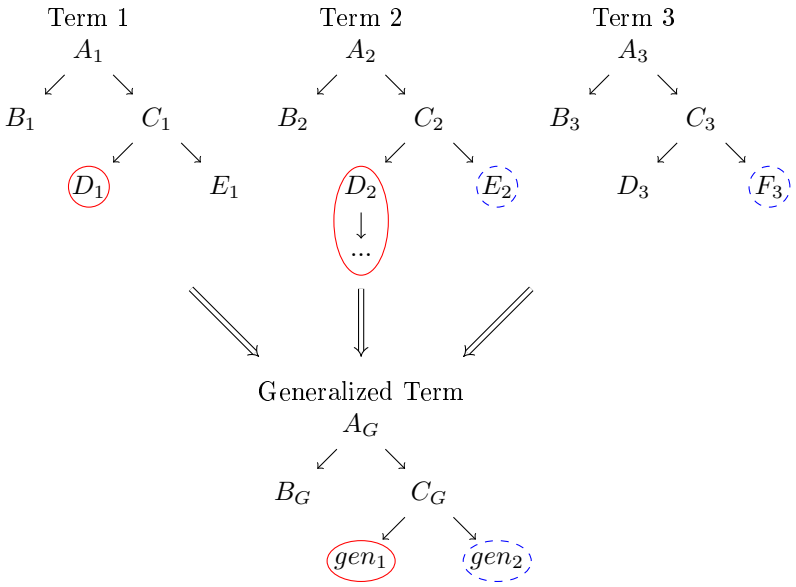


Figure 17: Schematic illustration of term generalization. The nodes represent arbitrary operations, where equal letters signify equal operations. The subscripts distinguish multiple different instances of the same operation. The marked differences are replaced by generalized substitutes. This approach can be used to generalize an arbitrary number of terms.

5 Background for SMT Solver Comparison

In this section the Viper Tool Chain is introduced. It was used to evaluate whether the reliability of software verification tools could be increased by making alternative solvers available. The default solver for the Viper tool chain is Z3. The alternative solver used for this purpose was CVC4.

5.1 Viper Tool Chain

The Viper tool chain is a software verification tool suite. It was developed by the Chair of Programming Methodology at ETH Zurich. Programs are verified by translating them into Silver, the Viper intermediate language, described in Section 5.2. Using an intermediate language has advantages for both the developers of the Viper tool chain and its users. The Viper developers benefit from their own, flexible and clearly defined interface to their verifiers. They do not have to define operational semantics and are free to extend Silver by adding new features at any time. Users only need to be concerned about finding a mapping to Silver, which expresses the properties of their program that need verification. This mapping or translation is usually automated. Translators are available for Java[12], Scala[21], Chalice[17] and OpenCL[26]. A translator for Rust[19] is currently in development.

A similar approach is used for the interface to a back end SMT solver. The SMT-LIB standard [4] defines a common language that many SMT solvers accept as input. This decreases the dependency of a tool chain on one specific solver. Originally, the Viper tool chain was developed using Z3, whereas CVC4 was integrated during this project. More details about the flexibility of this interface are provided in Section 5.5.1. The Viper tool chain contains two different, independent verifiers: Carbon (Section 5.3) in conjunction with Boogie and Silicon (Section 5.4). A complete overview of the Viper tool chain is shown in Figure 18.

The reason for why matching loops might occur when using the Viper tool chain is that many quantified assertions are required to encode a program in SMT-LIB. More specifically, the quantifiers come from four different sources:

1. The user can use quantifiers to formulate pre- and postconditions. This is a tradeoff chosen in favour of ease of use and expressiveness, as well as conciseness of notation. As a cost, the underlying SMT solver has to do more work for a successful proof.

2. Background theories are encoded with quantifiers. Background theories include axiomatizations of sequences, sets and multisets.

3. Native features, such as quantified permissions, require quantifiers for their encoding. Quantified permissions allow users to express access permissions using quantifiers.

4. Carbon requires quantifiers to encode reasoning about the heap. Since Carbon uses Boogie as a back end, which does not provide a heap abstraction, this has to be encoded in quantified assertions as well.

As a consequence, when confronted with a matching loop, it is not obvious which source introduced it. Therefore, strong tools to deal with matching loops,

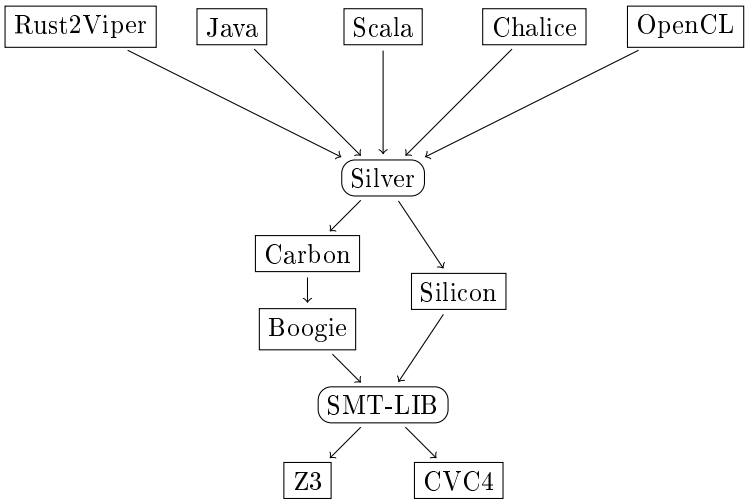


Figure 18: Overview of the Viper tool chain. Rectangles with rounded corners are intermediate languages, whereas sharp corners represent tools. Arrows indicate the dependencies between the different tools and languages. CVC4 support was added as part of this project. Rust2Viper is currently in development.

as well as problems related to quantifier instantiations in general are essential. It is important to note that long runtimes and excessive numbers of quantifier instantiations do not automatically imply a matching loop. The cause might also be an exponentially growing number of terms due to a non-looping, but nonetheless unfavourable encoding of quantified assertions.

5.2 Silver

Silver is the intermediate language and the core of the Viper tool chain. It defines the input format for both verifiers Carbon and Silicon. In order to be able to express and verify the semantics of a programming language with operational semantics, Silver supports a variety of techniques to abstract the details of the source language. These include uninterpreted functions, quantified assertions, custom domains, access permissions for heap locations and fields, as well as pre- and postconditions of methods and functions. Methods and functions are distinguished by purity: functions are pure and allow no state changes. Hence neither changes to the heap nor writes to fields are allowed within functions.

Access permissions are a central feature of Silver. There are three levels of access: no access, read access and write access. Silver supports encodings that do not allow shared write access. Thus any successfully verified Silver program using such encodings is guaranteed to be free of errors due to shared write permissions. Clever encoding enables to verify that after write permission was shared and returned, the write permission is indeed no longer shared and writes are safe again.

Silver also features axiomatizations of sets, multisets and sequences. Specifications of such language constructs usually need many quantified assertions, which require E-matching to verify. This is why the quantified predicates have to be designed carefully and annotated with hints about their instantiation to

help the SMT solver. If these annotations are made incorrectly they might lead to infinite runtimes as explained in Section 2.6.

5.3 Carbon

Carbon is one of the verifiers of the Viper tool chain. It verifies programs by translating them to the BoogiePL intermediate language, the input for the static verifier Boogie. Boogie does not provide a built in abstraction of the heap, as Silver does. Therefore, the heap has to be encoded manually when translating to BoogiePL, which requires additional quantified assertions compared to Silicon.

Boogie uses verification condition generation (VCG) to produce a formula in first order logic. It produces one single query for each method, containing all verification conditions for every single possible path through the method. Naturally, this leads to larger queries than produced by verifiers using symbolic execution, such as Silicon. This has performance implications (as shown in Section 6.3). The formula is then verified using an SMT solver such as Z3 or CVC4. If a query is proven unsatisfiable, the method or function is safe. This means that the postconditions hold for all inputs that satisfy the preconditions.

5.4 Silicon

Silicon is the second verifier of the Viper tool chain. Unlike Carbon, Silicon interacts directly and interactively with the underlying SMT solver.

This verifier is based on symbolic execution, which means it uses the SMT solver to check entailment between statements. The queries are in the context of one particular path as the symbolic execution engine handles case splits, e.g. for different branches of an `if` statement. This approach leads to much smaller queries compared to Boogie and has a positive effect on performance (Section 6.3.2).

5.4.1 Quantified Permissions

At the time of the evaluation of CVC4 (Section 6), the quantified permission feature was still in development and not yet integrated into Silicon. The former development version of Silicon with support for quantified permissions is therefore referenced to as Silicon QP. The feature has since been fully integrated.

Quantified permission allows a user to express not only pre- and postconditions with quantifiers, but also access permissions. Allowing quantified permissions often shortens the formulation of appropriate access permission as explained in [20].

5.5 CVC4

CVC4 is an open source SMT solver. It is a joint project between the New York University and the University of Iowa. Contributors from many other institutions (e.g. Google) are also taking part in the development.

The feature set includes many theories such as arrays, inductive data types, tuples, etc. as well as first-order quantifications of these theories. Strings and non-linear arithmetic are only partially supported and still in development.

5.5.1 CVC4 Integration into Viper

The integration of CVC4 into the Viper tool chain required some rewriting of the existing tools. Fortunately, the SMT-LIB standard facilitated the integration. Standard interactions with Z3, such as function definitions and assertions, were already implemented according to the SMT-LIB specifications. Notable exceptions were solver configuration and function overloading outlined below.

The SMT-LIB standard, as previously mentioned, is an international effort to facilitate SMT research and solver development. It specifies a common input language which all solvers should adhere to. CVC4 currently only supports SMT-LIBv2, the most recent version being v2.5.

5.5.2 Modifications in Carbon

Integration into Carbon was mainly problem-free as Boogie already featured experimental support for CVC4. The integration was limited to updating the experimental support for the current version of CVC4 and enabling all the required logic theories.

Some tests remained incompatible: Boogie, and consequently also Carbon, requires models for all proofs. A proof model contains information about which facts were responsible for the proof. Since the support for non-linear arithmetic is not yet complete in CVC4, models are unavailable if these logics are enabled. The models are used by Boogie to name the specific traces that could not be verified. This behaviour cannot be switched off.

Since models will work in combination with non-linear arithmetic once the implementation is complete, implementing support for solvers that do not provide models by modifying Boogie was discarded. In particular, since this is not the biggest problem of CVC4 in conjunction with Boogie (Section 6.3). More informative error reporting was implemented instead.

5.5.3 Modifications in Silicon

In order to integrate CVC4 as a back end into Silicon, the solver interactions had to be made more abstract. This included adding a general interface and treating the actual solver as an implementation detail. Consequently all solver interactions had to be changed to strictly adhere to the SMT-LIBv2 standard.

The following modifications to the solver interactions were necessary:

- Replacement with SMT-LIB compliant syntax: 'implies' to '=>', 'iff' to 'not xor', stricter bracketing, etc.
- Remove function overloading: while Z3 supports function overloading on generic types, CVC4 does not. Therefore, all occurrences of functions with generic arguments were renamed by appending the generic type to the function name. This is illustrated in Example 16.
- Implement support for different preambles depending on the solver used.
- Rewriting the static preambles into valid SMT-LIBv2 files.

This list shows that Z3 does implement a more general syntax than that required by SMT-LIB. As with Carbon, one major incompatibility remained: Silicon uses global variable declarations, which means that variable declarations

must remain valid even after the scope in which the variable has been defined was “popped" interactively. Since the symbolic execution engine of Silicon does case-splitting by itself, this feature is used quite extensively. Removing the need for global variable declarations would require rewriting the symbolic execution engine. This was not considered a good investment of effort thus remained unsolved. The integration into Silicon QP was analogous as the quantified permission feature is not connected to solver interactions. In practice, the global variable declaration issue rendered about 16 % for Silicon and 37 % for Silicon QP incompatible as indicated by Table 2.

Example 16 Replacement of function overloading on generic types.

Given two functions `foo(Seq<int> a)` and `foo(Seq<Ref> a)` which only differ on the generic type of the argument `a`. In order to distinguish these functions in absence of overloading on generic types, the functions are renamed to `foo_int()` and `foo_Ref()`, respectively.

6 CVC4 Evaluation

In this evaluation, CVC4 was compared to Z3 when used as a back end in the Viper tool chain. The evaluation was conducted to decide whether or not to include CVC4 in the Viper tool chain permanently. If significant differences had been revealed, including both solvers in the Viper tool chain and implementing a solver selection heuristic would have been a good way to reduce the time needed for the complete verification process. As the results show, there is not a significant number of test cases where CVC4 is faster than Z3.

Three tools, namely Carbon, Silicon and Silicon QP, were evaluated with the two SMT solvers Z3 and CVC4. Tests were always run completely with the original Silver file as opposed to using an intermediate step such as Boogie files or SMT files. The evaluation was conducted on the existing Silver test suite containing 288 tests. For Silicon QP an additional 148 tests were available. The program versions used for the evaluation are listed in Table 1. The evaluation was done in two parts: the correctness (Section 6.2) and the performance (Section 6.3) were evaluated.

6.1 Viper Runner

In order to compare the two solvers, data had to be collected. To facilitate the data collection, a highly configurable Python [25] benchmark framework, the Viper Runner, was developed.

A normal test framework was insufficient for this task, as these are not designed to handle programs that start other programs. Therefore, they do not terminate process hierarchies correctly when a test runs into a timeout. As shown previously in the Viper overview in Figure 18, the Viper tool chain is built upon the concept of process hierarchies. Terminating just the parent process leaves the resource-consuming SMT solver running while terminating the verification tool, which in turn then compromises the data collected for subsequent tests.

The Viper Runner is suited to run complete test suites as well as benchmarks. It allows a user to capture the output of all the tools, including the intermediate steps such as Boogie and SMT files, separates the standard output from the standard input and records return values. Discarding all output to reduce the IO overhead is also possible. Additionally, the Viper Runner is able to run tests repeatedly to provide a measure for the runtime variance for each individual

Tool	**Version**
Boogie	2.3.0.61016 (fork of 713e870, 28. Aug)
Carbon	1.0 Snapshot (fork of 309dcfec1da7, 25. Nov)
CVC4	1.5-prerelease (12. Dec)
Silicon	0.1 Snapshot (fork of 9ad15e481f6a, 27. Nov)
Silicon QP	0.1 Snapshot (fork of de59b1d78cfa, 27. Nov)
Silver	Snapshot (0d33695bd459, 27.Nov)
Z3	4.4.2 Snapshot (216c1b2, 2. Dec)

Table 1: Tool versions used for the evaluation of CVC4.

	Carbon	Silicon	Silicon QP
Pass	174	216	236
Ignore	10	11	18
Incompatible	22	47	160
Fail	82	14	22
Total	**288**	**288**	**436**

Table 2: Correctness evaluation results for all Viper verifiers, Carbon, Silicon and Silicon QP, using CVC4.

	Carbon	Silicon	Silicon QP
Timeout	63	4	4
Imprecise	10	7	13
More precise	2	1	3
Mixed[2]	0	1	1
Unsound	7	1	1
Total	**82**	**14**	**22**

Table 3: Detailed listing of test failure reasons for the Viper verifiers using CVC4.

test.

The Viper Runner was also extended with a plotter module. This module generates scatter plots comparing the runtimes for different run configurations and calculates difference metrics to facilitate the comparison. The Viper Runner was used for the correctness as well as the performance evaluation.

6.2 Correctness

In order to evaluate the correctness of the results obtained using CVC4, all tests were run using both solvers and validated by comparing the output. The results given by Z3 were used as a baseline. Test cases for which the output given by CVC4 was different than the Z3 output were evaluated individually and manually. Since Z3 was the baseline, ignored tests (for various reasons, mostly incompleteness of the verification tools) were also ignored for CVC4. An overview of the correctness evaluation for all tools is shown in Table 2. A more precise listing of the causes of test failures is presented in Table 3.

6.2.1 Carbon

The correctness test results for Carbon with CVC4 are shown in Table 2. Out of 193 tests that did not time out only 17 (9 %) were invalid; with 2 verification results being more precise than Z3 (Table 3). Notable is the large number of timeouts: 63 tests failed because the runtime exceeded the allowed runtime of 10 minutes.

[2]Tests may contain multiple properties to verify. In these tests CVC4 verified some assertions more precise and some less precise than Z3.

The large queries typical for Boogie encodings of Silver programs tend to be difficult to solve for CVC4. The timeout limit of 10 minutes was determined arbitrarily to keep benchmark times reasonable (one complete run with 5 repetitions takes about 24 hours). To put this into perspective: the complete test suite with all 288 tests completes in about 13 minutes using Z3.

The remaining 32 tests are also accounted for: 10 were set to ignore by the unit test framework using Z3, and were consequently also ignored for CVC4, in particular those involving many different quantifiers. The remaining 22 tests are incompatible with Carbon with CVC4 due to the lack of models when non-linear arithmetic is enabled. There were also 6 unsound test results. More information about unsound test results is available in Section 6.4. In summary: CVC4 is not yet well suited as a back end to Carbon and Boogie. The large queries generated due to VCG are solved unreliably by CVC4.

6.2.2 Silicon

In general, CVC4 works well with Silicon: 94 % of the tests passed with only 4 timeouts as listed in Tables 2 and 3. The timeouts were comparatively short (5 minutes) to the ones used for Carbon. This is justified by the shorter runtime of Silicon in general. A complete test suite using Z3 takes about 2 minutes and 30 seconds.

For Silicon 11 tests were ignored by default and 49 tests were incompatible with CVC4 as Silicon relies on global variable declarations. There was also one unsound test result.

6.2.3 Silicon QP

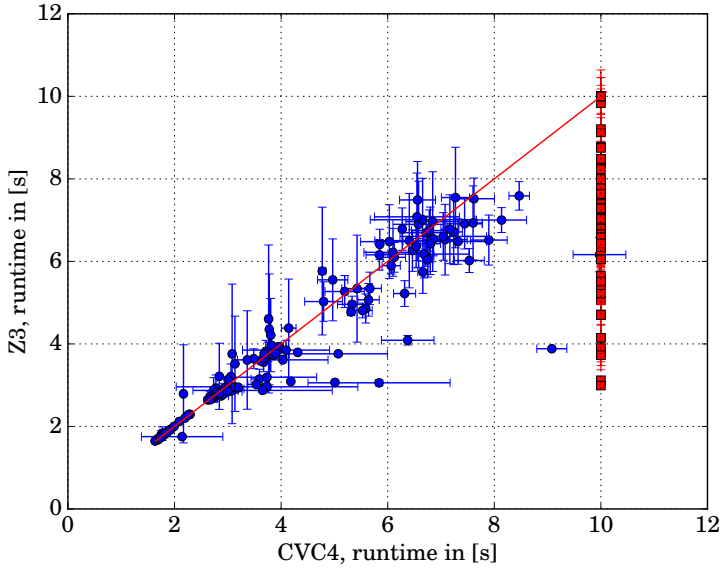
The evaluation of Silicon QP is consistent with the findings for the standard version of Silicon. The percentage of passed tests dropped slightly to 86 %.

The number of timeouts remained low. Most of the additional test cases making use of quantified permissions were incompatible due to the known issue with global variable declarations. Two new unsound test results are revealed as well. These findings are not surprising as most additional tests (113 out of 148) were incompatible with CVC4. Therefore, the test set for Silicon QP was very similar to the test set of Silicon.

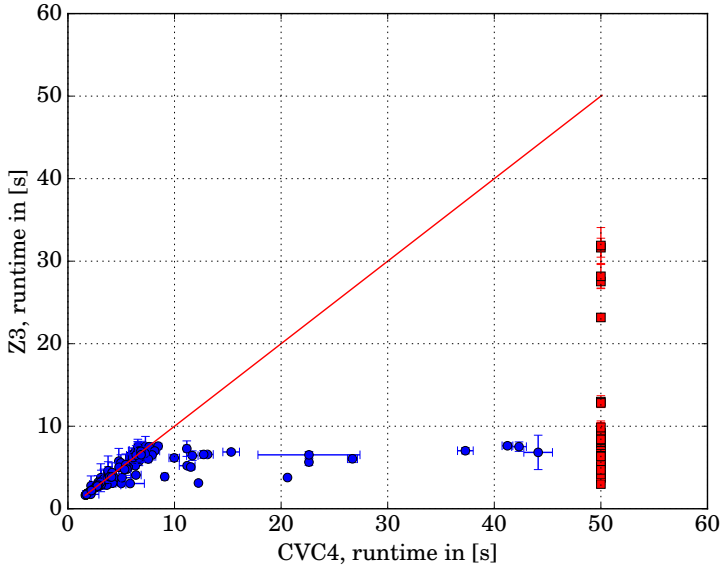
6.3 Performance

Performance was evaluated in a similar way as correctness using the Viper Runner. All available tests were run in all combinations of tools (Carbon, Silicon, Silicon QP) and available SMT solvers (CVC4, Z3). Every test was repeated 5 times to get a measure of consistency. Output of any kind was discarded to reduce the IO overhead. Only timing information and the return code was collected.

To evaluate the performance, failing tests, for reasons other than timeout, were disregarded as they do not yield a fair comparison regarding performance between the two solvers. Tests whose results indicates an error in verification are considered failing. Timeouts were included in the evaluation since these are the most extreme runtimes possible and therefore vitally important for a performance evaluation.



(a)



(b)

Figure 19: Runtime comparison of Carbon using CVC4 and Z3 with cutoffs of 10 seconds (19a) and 50 seconds (19b). The graphs show 237 tests that passed, or failed with a timeout. Due to the large differences in runtime, cutoffs of 10 and 50 seconds were used. Error bars indicate the standard deviation. Data points exceeding the cutoffs were adjusted to the cutoff value and are shown as red squares. 77 and 66 data points were adjusted, 7 in both axis and only measurements of CVC4, respectively.

For tests, where the first run took significantly longer than the subsequent ones, only 4 measurements were used while disregarding this first run. This was done to reduce the influence of the long start-up time of the Java Virtual Machine (JVM), which is the runtime environment for the Viper verifiers, and Scala in general. If the previous test used all the available RAM, some parts of the JVM get paged out, leading to an increase in runtime of $1 - 2$ seconds for the next test.

All benchmarks were conducted on a Windows 7 Enterprise machine with 16 GB main memory and an Intel Core i5-4570 CPU clocked at 3.2 GHz.

6.3.1 Carbon

The performance results for Carbon using CVC4 are shown in Figure 19a. Most tests (160, 92 % of the passing tests) completed within 10 seconds. The runtimes for these tests were very close to the results obtained using Z3. The mean difference favours Z3 by 0.20 seconds with a standard deviation of 0.70 seconds.

When factoring in the slower running tests CVC4 is less competitive. As shown in Figure 19b there were a lot of test cases for which Z3 finished within 10 seconds whereas CVC4 took more than that, often exceeding even the 50 second cutoff.

An additional test with a much higher timeout, 6 hours, was conducted with only the 63 tests that previously ran into a timeout (Table 3). Out of these 63, 19 (30 %) finished successfully given more time. 8 of these finished in less than 10 minutes, the previous timeout. This reveals some unreliability in runtime of CVC4 for inputs with many quantifiers. 2 test did finish but yielded a wrong result, one being imprecise and the other unsound. 42 tests ran into the timeout of 6 hours pointing to infinite or at least impractical runtimes.

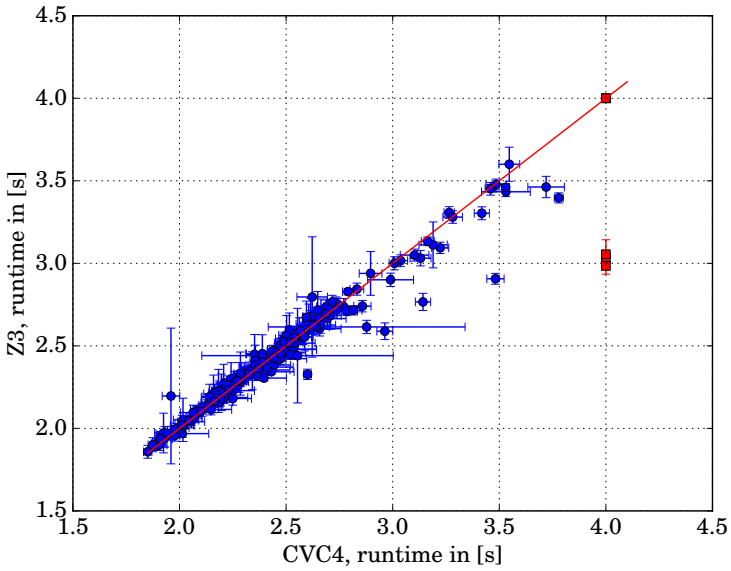
6.3.2 Silicon

The Silicon benchmark results in Figure 20a are more competitive compared to the results of the Carbon benchmark. There were almost no timeouts and for the vast majority of the tests Z3 and CVC4 performed on par. Only 6 tests took longer than 4 seconds. The comparison of the tests that completed within 4 seconds is slightly in favour of Z3, which ran tests on average 0.01 seconds faster. The standard deviation is 0.08 seconds.

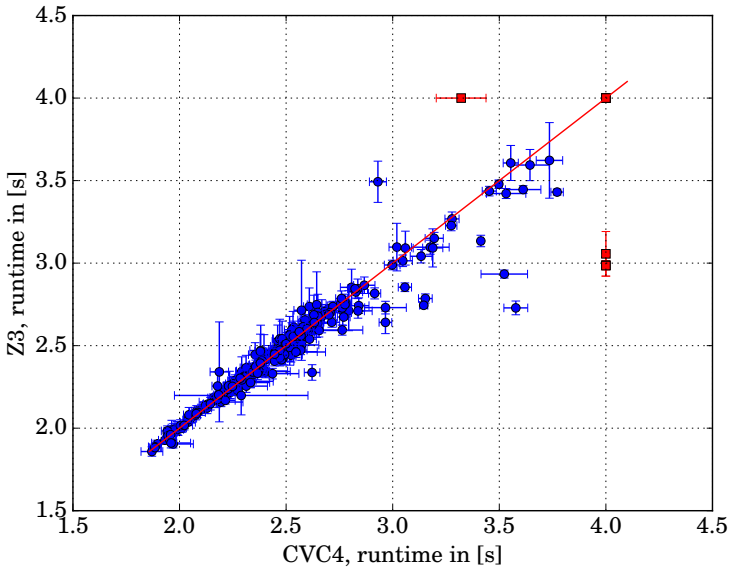
The timeout analysis of the 4 test cases that ran into the timeout of 5 minutes revealed a consistent result: 3 of the 4 tests also do not finish given 6 hours of runtime, whereas the 4th crashes after running for more than 3 hours.

6.3.3 Silicon QP

As expected, the Silicon QP benchmark (Figure 20b) shows very similar results to the Silicon benchmark. The number of timeouts (4) stayed the same while performance of the passing tests deteriorated slightly: the average runtime advantage of Z3 increased to 0.2 seconds with a standard deviation of 0.10 seconds. The subsequent tests with a timeout of 6 hours showed exactly the same results as for Silicon: 3 timeouts and one crash for the exact same test cases.



(a)



(b)

Figure 20: Runtime comparison of CVC4 and Z3 with the verifiers Silicon (20a) and Silicon QP (20b). A cutoff of 4 seconds was used. Data points with runtimes exceeding 4 seconds were adjusted to the 4 second mark and are shown as red squares. Error bars indicate the standard deviation. Figure 20a and 20b contain 220 (6 adjusted, 3 in both axis) and 242 (7 adjusted, 3 in both axis) data points respectively.

6.4 Unsound Results

As indicated in Table 3, some test results were considered unsound. This happened when CVC4 unexpectedly managed to prove a method safe, which should not be. This might indicate a bug in either the Viper tool chain or CVC4. It is important to note that CVC4 was used with default settings except for the theories. All theories were enabled, even the not yet complete non-linear arithmetic support. The CVC4 developers have been informed about this issue and offered to make this behaviour optional [23]. At the time of writing, investigations in that matter were not yet complete.

In any case, a significant point is the fact that CVC4 is not strict with quantifier patterns and instantiates quantifiers even if patterns do not match. This might have an effect on runtime as it might be the primary reason for the large number of timeouts observed with Carbon. Encodings by Boogie tend to contain more quantifiers because of the lack of a native abstraction of the heap.

6.5 Evaluation Summary

Considering both evaluation categories (correctness and performance) CVC4 is not yet on the level of Z3 for usage in the Viper tool chain. It is important to recognize that the test set was biased in favour of Z3 since all the tests in the Silver test suite are known to work with Z3.

Nonetheless, CVC4 has a tendency to not terminate on big queries involving a lot of quantifiers. For small queries CVC4 and Z3 are equally fast, with Z3 being a bit more consistent. The imprecisions, leading to unnecessary verification failures, are detrimental to the assessment of CVC4 as well. Over all, CVC4 is considered a solid solver but does not have any competitive advantage over Z3 for the usage in the Viper tool chain. Therefore the remaining time of this project was invested in improving the analytic tools for Z3 (Section 3).

7 Conclusion

During this project Z3 and CVC4 were compared as back ends to the Viper tool chain. The comparison showed that a permanent integration of CVC4 would not add much value, as both solvers struggle with similar problems; CVC4 more often than Z3.

As a consequence, the main tool to deal with these issues when working with Z3, the Z3 Axiom Profiler, was improved. It was completely redesigned and extended with more functionality. Hence the Z3 Axiom Profiler is now more accessible and easier to understand and use. In multiple examples, the process of finding a matching loop was reduced to loading the trace file and clicking two buttons: one to reveal a longest path and another to automatically detect the loop and generate a generalized explanation of the loop and a specific explanation of the path. Compared to the previous experience with manually expanding deeply nested nodes in search of repeating instantiations, this is a vast improvement as it is faster and more accessible.

In order to help the user understand the matching loop, the addition of binding reconstructions is a special highlight. This feature shifts the burden of tracing terms and determining their role within instantiations from the user to the tool. The term highlighting and the concise notation that comes with rewriting rules greatly enhances the clarity of the information printed to the info panel.

Taken together, the improvements to revealing and presenting matching loops make the Z3 Axiom Profiler a better tool, decreasing the time and effort required to solve problems with quantifier instantiations. This could be verified with multiple real world examples that were previously only very hard or even impossible to understand. Considering the improvements and features of the new Z3 Axiom Profiler, we are confident that the tool is of use for many developers working with Z3, even beyond the scope of the Viper tool chain.

7.1 Further Work

Despite the implemented improvements, the Z3 Axiom Profiler could still benefit from additional features. More specifically, the tool is currently able to show a generalized view of a matching loop, highlighting the important changing terms. It would be very useful, if these highlights were also available in the concrete path explanation in order to guide the user to the most relevant information.

As an additional step, presenting a concrete loop iteration starting from the generalized state could be implemented. This would make the effects of an additional loop iteration on the general state very clear and approachable. The implementation of such a feature would most likely use symbolic execution to simulate E-matching. Consequently, the implementation of this feature would require a major development effort.

Finding matching loops efficiently still requires manual effort, experience and, to some extent, luck. This could be improved by employing further heuristics to guide the user to instantiations that look suspicious or might reveal interesting information. If this feature turned out to be effective, automated search, guided by these heuristics, could implemented. Having automated matching loop search would decrease the knowledge required to solve a matching loop problem significantly.

List of Figures

List of Tables

List of Algorithms

List of Examples

References

[1] Aws Albarghouthi, Arie Gurfinkel, Yi Li, Sagar Chaki, and Marsha Chechik. UFO: Verification with interpolants and abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 637–640. Springer, 2013.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2005.

[3] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[5] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. *Handbook of satisfiability*, 185:825–885, 2009.

[6] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 429–430. IEEE, 2009.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] David Deharbe, Pascal Fontaine, and Bruno Woltzenlogel Paleo. Quantifier inference rules for SMT proofs. In *First International Workshop on Proof eXchange for Theorem Proving-PxTP 2011*, 2011.

[9] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[10] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2), 2006.

[11] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *Automated Deduction–CADE-21*, pages 167–182. Springer, 2007.

[12] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000.

[13] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004.

[14] Chris Hawblitzel, Jon Howell, Jacob R Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, 2014.

[15] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[16] K Rustan M Leino and Rosemary Monahan. Automatic verification of textbook programs. *Manuscript KRML*, 175:13, 2007.

[17] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[18] Tim Dwyer Lev Nachmanson, Sergey Pupyrev and Ted Hart. Microsoft Automatic Graph Layout. `http://research.microsoft.com/en-us/projects/msagl/default.aspx`. Accessed: 2016-03-03.

[19] Nicholas D Matsakis and Felix S Klock II. The Rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[20] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, pages 41–62. Springer, 2016.

[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, École polytechnique fédérale de Lausanne, 2004.

[22] Clément Pit-Claudel. Quantifier blame realation visualization for the Z3 Axiom Profiler. Private communication, 2015.

[23] Andrew Reynolds. CVC4 bug report. `http://church.cims.nyu.edu/bugzilla3/show_bug.cgi?id=713`. Accessed: 2016-03-07.

[24] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. Finding conflicting instances of quantified formulas in SMT. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 195–202. FMCAD Inc, 2014.

[25] Michel F Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[26] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(1-3):66–73, 2010.

[27] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.