Developing Tool Support for Understanding Quantifier Instantiations

Bachelor Thesis

Oskari Jyrkinen

March 2, 2024

Advisors: Prof. Peter Müller, Jonas Fiala, Dionisios Spiliopoulos

Department of Computer Science, ETH Zürich

Abstract

SMT solvers are commonly used in software verification to formally verify program properties. To express these program properties, undecidable theories and quantifiers are often required. SMT solvers resort to heuristic approaches such as e-matching to deal with quantified assertions. There are various ways in which quantified assertions can lead to poor performance and non-termination. In this project, tool support for visualizing and understanding SMT solver runs is developed. This project is based on an existing tool. The existing tool was redesigned from scratch to be more performant and improve user-friendliness. New features were designed to identify and analyze problems from a commonly encountered class of problems referred to as matching loops.

Contents

Chapter 1

Introduction

Program verification refers to a set of methods to formally verify software. Its strong correctness guarantees make it a preferable alternative to software testing in cases where software bugs are very costly. Program verifiers such as Viper [21], Dafny [17], and Boogie [4] encode program properties in terms of logical formulas and use SMT solvers to prove their validity. For formulas only involving propositional logic, checking satisfiability is decidable. However, propositional logic is not sufficiently expressive for many problems in program verification. Therefore, predicate logic is often needed, particularly formulas involving universal quantifiers. But since the satisfiability of SMT assertions with quantifiers is undecidable in general, SMT solvers such as Z3 [10] resort to heuristic approaches such as *e-matching* [9].

During e-matching, SMT solvers repeatedly instantiate quantified formulas by binding the quantified variables to some terms in order to gain new facts. We will refer to this as a *quantifier instantiation*. There are various ways in which quantifier instantiations can lead to poor performance. Understanding the root cause of these issues can help the user adjust the input problem passed as an input to the SMT solver and hence achieve better performance.

In this project, we developed the *Axiom Profiler 2.0* (AP2). It is a tool loosely based on the *Axiom Profiler* (AP1) [6] [22] for visualizing quantifier instantiations made via e-matching. AP2 takes as input a log file generated during a solver run of Z3 and generates a visual representation of the instantiations and how they depend on each other. It can help the user identify and understand potential culprits of poor performance. It is implemented as an OS agnostic web application. AP2 runs purely in a web browser and does not have a backend.

In chapter 2, we will cover the necessary theoretical concepts to understand this project. In chapter 3 we will cover in detail how AP2 was implemented.

In chapter 4, we will evaluate AP2 and compare it qualitatively and quantitatively to AP1.

Chapter 2

Background

2.1 SMT Solvers

The SMT problem is concerned with determining whether a logical formula is satisfiable. The simplest case in this context is checking if a formula in propositional logic is satisfiable. This is a decidable problem and SMT solvers use methods such as DPLL [8] to efficiently find a model. DPLL involves searching through models using backtracking until either a model is found or all models are found to be unsatisfiable. To deal with formulas from first-order logic that involve constraints between terms from some theory, SMT solvers can use for instance an extended version of DPLL, often denoted by DPLL(T) [13]. The first-order logic formula is abstracted into a formula in propositional logic on which a DPLL-like model search is perfomed. During this search, the SMT solver repeatedly generates candidate models and uses a theory solver to check if these candidate models are consistent. Some theories, such as quantifier-free linear arithmetic, are decidable and hence the SMT solver will terminate reporting the satisfiability of the input formula in such cases.

Program specifications often require quantifiers to express complex properties. For instance, the property that an array `arr` is sorted in ascending order could be expressed with the formula $\forall i, j. 0 \leq i < j < \texttt{len(arr)} \implies \texttt{arr}[i] \leq \texttt{arr}[j]$. In general, proving the satisfiability of formulas is undecidable and hence SMT solvers might not terminate or return that the formula is either unsatisfiable or `unknown`. In the context of program verificiation, we are interested in proving the validity of a formula P which encodes a program specification. Therefore, it suffices to prove that $\neg P$ is unsatisfiable. A commonly used approach used for dealing with quantifiers is *e-matching* which will be described next.

2.2 E-Matching

To understand in more detail how SMT solvers such as Z3 deal with quantified assertions (we will refer to these as *quantifiers*), we are first going to cover an example. Consider the example in figure 2.1. Lines 1 and 2 declare constants `a` and `b` of type `Int`. Lines 3 and 4 declare functions `f` and `g`. Both functions have a single argument of type `Int` and output type `Int`. On lines 6 and 7 two assertions are encoded. Line 8 encodes a quantifier with the quantified variable `x` of type `Int`. The expression `{g(f(x))}` is a *pattern* or *trigger* that is used during e-matching for pattern matching against a ground term with structure `g(f(t))` and bind the bound variable `x` to `t`. Triggers have to contain all the quantified variables and at least one non-constant function symbol. The SMT solver will use the trigger to find a ground term with a matching term structure in the e-graph such that it can instantiate the quantified formula and obtain a new fact.

```
1   (declare-const a Int)
2   (declare-const b Int)
3   (declare-fun f (Int) Int)
4   (declare-fun g (Int) Int)
5
6   (assert g(b) = 0)
7   (assert b = f(a))
8   (assert ∀x:Int {g(f(x))} g(f(x)) = 1)
9   (check-sat)
```

Figure 2.1: A simple SMT-encoding of a problem. We use pseudocode roughly based on SMT-LIB format with presentational liberties.

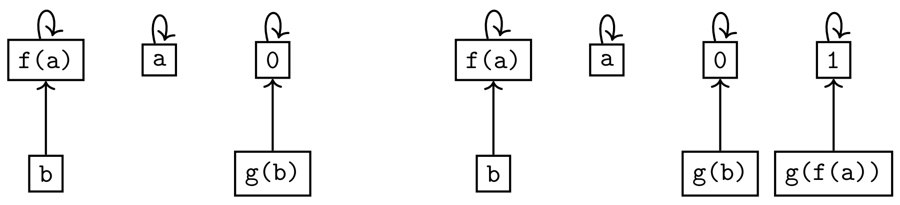
In such a case, Z3 will construct an *e-graph* to represent the equalities between the various *ground terms*, i.e., terms without any quantified variables. In this example, we can visualize the e-graph as shown in figure 2.2a. Each component of the e-graph represents an equivalence class. Nodes with self-loops represent roots of an equivalence class. Note that the terms `a` and `b` are ground terms as they are declared as constants on lines 1 and 2. For this reason, the terms `f(a)` and `g(b)` on lines 6 and 7 are also ground terms and hence added to the initial e-graph. The nodes in the e-graph are often called *e-nodes*. In figure 2.2 we represent an equality between two terms using an arrow pointing from the former to the latter node.

In our example, the trigger matches against the ground term `g(f(a))` which the SMT solver can obtain by rewriting the ground term `g(b)` with the equality `b=f(a)`. Therefore, the SMT solver can bind the quantified variable `x` to `a` and instantiate the quantifier, obtaining the new fact `g(f(a))=1`. This newly obtained equality among ground terms is then added to the e-graph as shown in figure 2.2b. At this point, the SMT solver has recorded in the e-graph that `g(b)=0` as well as `g(b)=g(f(a))=1` which is a contradiction. Therefore, the SMT solver will terminate this solver run returning `unsat`, meaning that the assertions in the input problem in figure 2.1 are not satisfiable.

In the previous instantiation, the SMT solver

- used the equality `b=f(a)`
- to rewrite the ground term `g(b)`
- into `g(f(a))` which matches against the trigger `{g(f(x))}`
- by binding the quantified variable `x` to `a`
- yielding the term `g(f(a))=1`.

We will refer to the term that is bound to the quantified variable in such an instantiation as *bound term* (here `a` is the bound term). The term, possibly rewritten using equalities, that the trigger was matched against we will refer to as *blame term*. In our example `g(b)` is the blame term. Any equalities that are used to rewrite the blame terms we will refer to as *equality explanations*. We will refer to the body of the quantified formula, with all the bound variables replaced by the corresponding bound terms, as the *resulting term* (here `g(f(a))=1`). We will refer to all new ground terms that are subterms of the resulting term as *yield terms*. In this example, the yield terms are `g(f(a)) =1`, `g(f(a))`, and `1`. Note that the subterms `f(a)`, and `a` are also ground terms but, since they were already present in the e-graph in figure 2.2a, they are not yield terms.



(a) Initial state of e-graph. **(b)** State of e-graph after instantiation.

Figure 2.2: Example of e-graph.

2.3 Instantiation Graph

Given a log generated by Z3, we can define the *instantiation graph* or *blame graph* $G = (V, E)$. The set of nodes V corresponds to all logged quantifier instantiations. The set of edges E is defined by the *blames* relation between instantiations. Instantiation i_2 blames instantiation i_1 either if any of the blamed terms of i_2 is a yield term of i_1 or if any of the equality explanations of i_2 involve an equality term that is a yield term of i_1. We will refer to the former kind of dependency as *blame-term dependency* and to the latter kind of dependency as *equality dependency*. Note that the log implicitly defines a total order $<$ on the quantifier instantiations, i.e., if i_1 is instantiated before i_2 then $i_1 < i_2$. If $i_1 < i_2$ then it is not possible that i_1 blames i_2 as i_2's yield terms do not exist at the time when the SMT solver instantiates i_1. Therefore, this total order also defines a topological order on the instantiations that is consistent with all edge directions, i.e., $(i_1, i_2) \in E \implies i_1 < i_2$. Therefore, we can conclude that the instantiation graph is a directed acyclic graph.

2.4 Matching Loops

The performance of an SMT solver run heavily depends on the triggers used for e-matching. These triggers are either chosen automatically [18] or the user chooses them by annotating the quantified assertions in the input problem with suitable triggers such as on line 8 in figure 2.1. If the triggers are too restrictive then the solver might fail to instantiate the quantifiers and hence not be able to prove unsatisfiability. On the other hand, if triggers are too permissive then the SMT solver might perform many unnecessary instantiations. A special case of this is if instantiating a quantifier Q with trigger T generates a new yield term that matches against the trigger T hence allowing Q to be instantiated again thus leading to a self-sustaining looping behaviour that can in some cases continue indefinitely. Such self-sustaining, repeated instantiations of the same quantifiers are called *matching loops* [6] [22] [18] [11]. Note that matching loops may in general involve more than a single quantifier. We will see various examples in this project, e.g, in section 3.3.

In theory, matching loops can continue indefinitely. In practice, however, SMT solvers eventually stop after some limit is reached. Matching loops can nevertheless have a major impact on the runtime of an SMT solver and hence tool support for identifying, understanding, and avoiding such matching loops is desirable.

2.5 Previous Work

To understand potential performance issues during a solver run of Z3, it is possible to generate logs which contain, among other things, detailed information about how quantifiers were instantiated during a solver run. To obtain such log files, one can pass the options `trace=true` and `proof=true` on the command line of Z3. Logs generated by Z3 can be very large and are hard to analyze manually. This issue has lead to various efforts to develop tool-support for analyzing these logs such as the Axiom Profiler (AP1) and the z3tracer library [5] which will be described next.

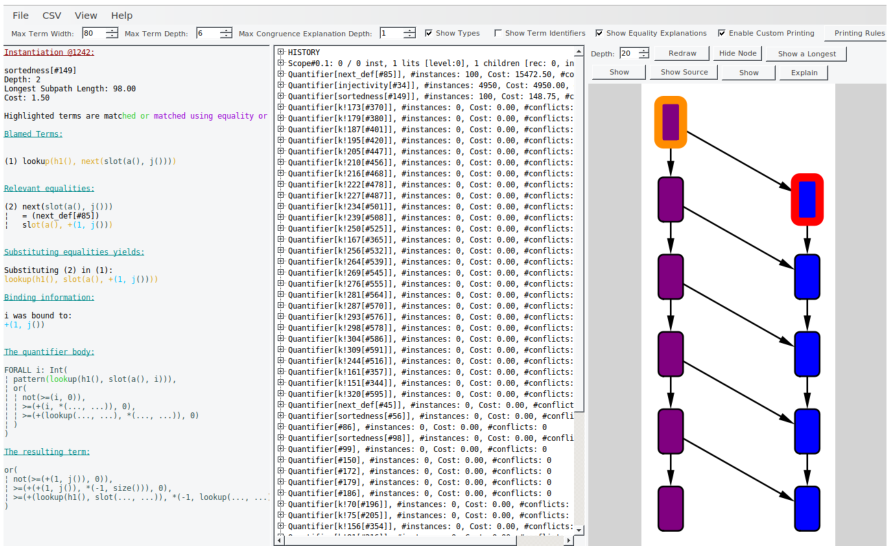
The z3tracer library takes as input a log generated during a solver run of Z3 and reconstructs which quantified formulas were instantiated and why. It allows users to generate various plots that show the most frequently instantiated quantifiers, conflicts and backtracking levels that are found during the DPLL-like search as described in section 2.1. However, it only supports logs generated by Z3 version 4.8.9 and does not provide an interactive graphical user interface.

AP1 takes as input a Z3 log file and generates a graphical representation of the instantiation graph (see figure 2.3). It offers functionality to select specific nodes and display the information associated with the corresponding instantiations, i.e., the blame terms, bound terms, equality explanations, yield terms, and resulting terms. This feature was reimplemented in AP2. Additionally, we implemented functionality in AP2 to select specific dependencies between instantiations which was not possible in AP1. Unlike in AP1, we implemented different styles for the nodes and edges in AP2 to visually convey more information about the dependencies and instantiations. A direct comparison can be found in section 4.1.

To identify troublesome instantiations, AP1 offers various filtering operations that can be applied to the instantiation graph. For instance, it offers a filter for showing the n instantiations with the most children or the n most expensive instantiations, where the cost is defined in terms of how many instantiations it caused directly or indirectly. One of the issues with applying filtering operations is that the resulting graph can contain many nodes or edges that lead to slow rendering or to unintelligible graphs. Therefore, AP1 displays a warning prompt in case the resulting graph applied after a filter contains many nodes or edges. In such a case the user can decide to proceed despite the warning or apply some filters to reduce the number of displayed nodes and edges.

One drawback of this approach in AP1 is that it is not transparent to the user which filters were applied in what order and undoing specific filters that were applied is not possible. Furthermore, AP1 does not represent indirect dependencies between instantiations, i.e., if there are dependencies (A, B) and (B, C) in the instantiation graph and the node B is filtered out then the filtered graph does not contain any indication that there is an indirect dependency between A and C. In this project, we implemented a reconnecting algorithm (see section 3.2.2) for displaying such indirect dependencies. Futhermore, we introduced a *filter chain* feature that displays all the filters that the user has applied and allows the user to remove any of them or reset it to the default filter chain (see lower left corner in figure 2.4).

The left panel of AP2 contains all filters that the user can apply to the instantiation graph and below it, the filter chain is shown. In addition to the filtering operations available in AP1, we implemented some more filters that

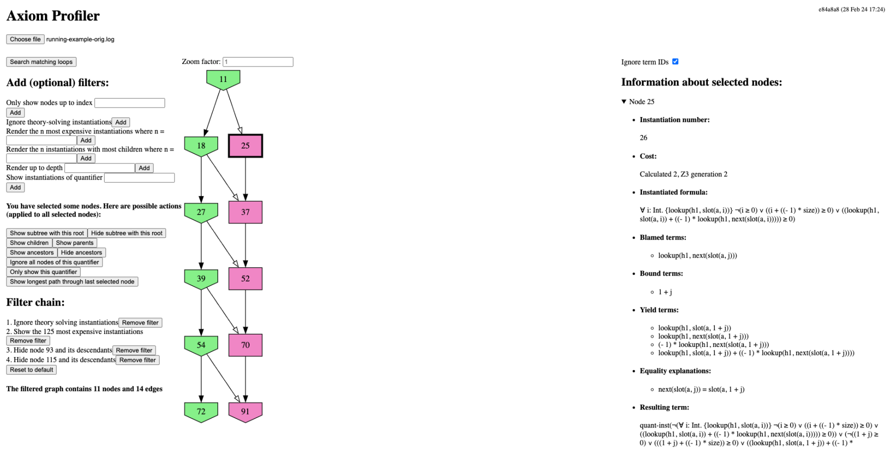


Max Term Width: 80 Max Term Depth: 6 Max Congruence Explanation Depth: 1 [✓ Show Types] Show Term Identifiers [✓ Show Equality Explanations] [✓ Enable Custom Printing] Depth: 20

```
Instantiation @1242:

sortedness[#149]
Depth: 2
Longest Subpath Length: 98.00
Cost: 1.50

Highlighted terms are matched or matched using equality or

Blamed Terms:


(1) lookup(h1(), next(slot(a(), j())))


Relevant equalities:

(2) next(slot(a(), j()))
¦   = (next_def[#85])
¦   slot(a(), +(1, j()))


Substituting equalities yields:

Substituting (2) in (1):
lookup(h1(), slot(a(), +(1, j())))

Binding information:

i was bound to:
+(1, j())


The quantifier body:

FORALL i: Int(
¦ pattern(lookup(h1(), slot(a(), i))),
¦ or(
¦ ¦ not(>=(i, 0)),
¦ ¦ >=(+(i, *(..., ...)), 0),
¦ ¦ >=(+(lookup(..., ...), *(..., ...)), 0)
¦ )
)

The resulting term:

or(
¦ not(>=(+(1, j()), 0)),
¦ >=(+(+(1, j()), *(-1, size())), 0),
¦ >=(+(lookup(h1(), slot(..., ...)), *(-1, lookup(..., ...
)
```

```
⊞ HISTORY
⊞ Scope#0.1: 0 / 0 inst, 1 lits [level:0], 1 children [rec: 0, in
⊞ Quantifier[next_def[#85]], #instances: 100, Cost: 15472.50, #co
⊞ Quantifier[injectivity[#34]], #instances: 4950, Cost: 4950.00,
⊞ Quantifier[sortedness[#149]], #instances: 100, Cost: 148.75, #c
⊞ Quantifier[k!173[#370]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!179[#380]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!187[#401]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!195[#420]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!205[#447]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!210[#456]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!216[#468]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!222[#478]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!227[#487]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!234[#501]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!239[#508]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!250[#525]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!167[#365]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!256[#532]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!264[#539]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!269[#545]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!276[#555]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!281[#564]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!287[#570]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!293[#576]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!298[#578]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!304[#586]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!309[#591]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!244[#516]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!161[#357]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!151[#344]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!320[#595]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[next_def[#45]], #instances: 0, Cost: 0.00, #conflict
⊞ Quantifier[sortedness[#56]], #instances: 0, Cost: 0.00, #confli
⊞ Quantifier[#86], #instances: 0, Cost: 0.00, #conflicts: 0
⊞ Quantifier[sortedness[#98]], #instances: 0, Cost: 0.00, #confli
⊞ Quantifier[#99], #instances: 0, Cost: 0.00, #conflicts: 0
⊞ Quantifier[#150], #instances: 0, Cost: 0.00, #conflicts: 0
⊞ Quantifier[#172], #instances: 0, Cost: 0.00, #conflicts: 0
⊞ Quantifier[#179], #instances: 0, Cost: 0.00, #conflicts: 0
⊞ Quantifier[#186], #instances: 0, Cost: 0.00, #conflicts: 0
⊞ Quantifier[k!70[#196]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!75[#205]], #instances: 0, Cost: 0.00, #conflicts:
⊞ Quantifier[k!156[#354]], #instances: 0, Cost: 0.00, #conflicts:
```

Figure 2.3: GUI of the Axiom Profiler (AP1).


Choose file
Search matching loops
Add
Show subtree with this root
Hide subtree with this root
Show children
Show parents
Show ancestors
Hide ancestors
Ignore all nodes of this quantifier
Only show this quantifier
Show longest path through last selected node
Remove filter
Reset to default
11
18
25
27
37
39
52
54
70
72
91

Axiom Profiler

e84a8a8 (28 Feb 24 17:24)

running-example-orig.log

Zoom factor: 1

Add (optional) filters:

Only show nodes up to index
Ignore theory-solving instantiations
Render the n most expensive instantiations where n =
Render the n instantiations with most children where n =
Render up to depth
Show instantiations of quantifier

You have selected some nodes. Here are possible actions (applied to all selected nodes):

Filter chain:

1. Ignore theory solving instantiations
2. Show the 125 most expensive instantiations
3. Hide node 93 and its descendants
4. Hide node 115 and its descendants

The filtered graph contains 11 nodes and 14 edges

Ignore term IDs ☑

Information about selected nodes:

▼ Node 25

- **Instantiation number:**

  26
- **Cost:**

  Calculated 2, Z3 generation 2
- **Instantiated formula:**

  ∀ i: Int. {lookup(h1, slot(a, i))} ¬(i ≥ 0) ∨ ((i + ((- 1) * size)) ≥ 0) ∨ ((lookup(h1, slot(a, i)) + ((- 1) * lookup(h1, next(slot(a, i))))) ≥ 0)
- **Blamed terms:**
  - lookup(h1, next(slot(a, j)))
- **Bound terms:**
  - 1 + j
- **Yield terms:**
  - lookup(h1, slot(a, 1 + j))
  - lookup(h1, next(slot(a, 1 + j)))
  - (- 1) * lookup(h1, next(slot(a, 1 + j)))
  - lookup(h1, slot(a, 1 + j)) + ((- 1) * lookup(h1, next(slot(a, 1 + j))))
- **Equality explanations:**
  - next(slot(a, j)) = slot(a, 1 + j)
- **Resulting term:**

  quant-inst(¬(∀ i: Int. {lookup(h1, slot(a, i))} ¬(i ≥ 0) ∨ ((i + ((- 1) * size)) ≥ 0) ∨ ((lookup(h1, slot(a, i)) + ((- 1) * lookup(h1, next(slot(a, i))))) ≥ 0)) ∨ (¬((1 + j) ≥ 0) ∨ (((1 + j) + ((- 1) * size)) ≥ 0) ∨ ((lookup(h1, slot(a, 1 + j)) + ((- 1) *

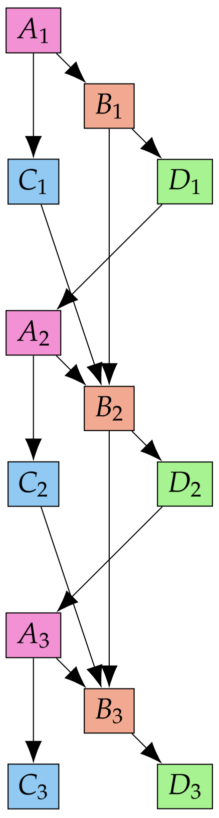
Figure 2.4: GUI of the Axiom Profiler 2.0 (AP2).

were deemed useful during the development of this tool. These filters will be discussed in more detail in section 3.2.3.

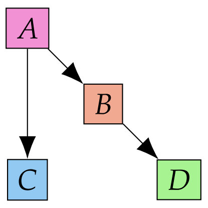
AP1 offers functionality to automatically select a path through the instantiation graph that represents a likely matching loop, find a repeating sequence of quantifier instantiations in that path and explain how this repeated sequence sustains itself.

This approach can only deal with matching loops that are restricted to a path and hence fails to explain matching loops with more complex structures. Consider the instantiation graph in figure 2.5a. There are multiple paths with repeated sequences such as $A_1B_1D_1A_2B_2D_2A_3B_3D_3$ which has the repeating sequence ABD or the path $B_1B_2B_3$ with the repeating sequence B. But the repeating pattern in this instantiation graph is not a sequence but rather the subgraph shown in figure 2.5b. Motivated by this observation, we developed an alternative approach in this project to analyze potential matching loops by extracting the repeating pattern in a potential matching loop and to represent the terms involved in such a repeating pattern using a *matching loop graph*. This method is described in detail in section 3.3.

The AP1 does not offer functionality to search for all potential matching loops in the instantiation graph. One of the key contributions in this project is the automatic matching loop search that makes use of the reconnecting algorithm. The basic idea is to project the instantiation graph onto each quantifier and find long, repeated instantiations of the same quantifier. This algorithm is discussed in more detail in section 3.3.1.



(a) Instantiation graph with repeating pattern.



(b) Repeating pattern in figure 2.5a.

Figure 2.5: Matching loop candidate with complex structure. Any single path does not explain the self-sustaining behaviour. Nodes that have the same color correspond to instantiations of the same quantifier and the same trigger used in the match.

We have found that AP1 does not support all versions of Z3 logs. According to the tool's documentation it should support versions at least 4.8.5 of Z3. However, we have found that for some logs generated by Z3 versions newer than 4.8.9, the tool either explicitly notifies the user that the version is not supported or simply crashes. Crashes were also found to occur in logs generated by Z3 version 4.8.9, even though the tool does not explicitly notify the user that the log version is not supported. Furthermore, AP1 does not handle logs of version 4.8.7 that contain line cases with log lines that span over multiple lines due to line breaks. In such cases, we have found that AP1 crashes after some time. The parser used by AP2 was developed to handle versions 4.8.5 until and including 4.12.6. AP2 can handle line breaks. Unknown line cases or line cases that are not relevant for the construction of the instantiation graph are skipped by AP2.

We have found that AP1 suffers from slow performance in processing logs. In section 4.2 we show the results of a quantitative comparison between processing speeds in AP1 and AP2 and have found that AP2 processes logs roughly ten times faster.

For this project, we used as a starting point the parser [20] and GUI [19] linked in the references. The tool linked in the references was built as a web application with a frontend that allows the user to select a log file. The selected log is sent to a server which parses the log and generates an SVG using Graphviz. The generated SVG was sent back to the client-side and rendered using a GUI built with the yew framework for Rust. The GUI did not, however, offer any functionality other than displaying the generated SVG graph. Furthermore, the parser was extended in this project to handle more line cases and versions of Z3 logs.

In this project, we decided to build the AP2 as a pure frontend application running in the browser by using a WebAssembly build of Graphviz [2]. Because our tool runs in the browser, it is OS agnostic. This is a convenient improvement over AP1. As AP1 is written in C#, using the tool on operating systems other than Windows either requires building the tool with Mono or Docker.

Chapter 3

Implementation

The goal of this project was to develop tool support for identifying and understanding performance issues due to quantifier instantiations in SMT solver runs. To this end, we developed a tool that takes as input a log file generated by Z3 and visualizes the quantifier instantiations and the dependencies between these instantiations such that the user can identify possible culprits of slow performance, such as matching loops. The construction and visualization of the instantiation graph are described in sections 3.1 and 3.2.

In general, the instantiation graph can be very large, and hence, displaying all nodes and dependencies would make it cumbersome to extract any meaningful information. Therefore, we implemented various filters that the user can apply to the instantiation graph to identify troublesome instantiations. These filters are described in section 3.2.3.

A common performance culprit in SMT solvers are matching loops, and therefore, we implemented features to search for potential matching loops and analyze them. This is described in section 3.3.

In section 3.4, we introduce an alternative way to represent equality dependencies between instantiations. In some cases, this alternative representation can significantly reduce the number of displayed edges and hence improve readability and rendering speed.

3.1 Constructing the Instantiation Graph

To construct the instantiation graph from the log files generated by Z3, we need to parse the log and generate a data structure that represents the dependencies between the instantiations and allows us to easily generate a graph representation. Therefore, we need to understand how these dependencies are represented in the log files. To this end, we are going to consider the example in figure 3.1.

```
1   (declare-sort Number)
2   (declare-const z Number)
3   (declare-fun inc (Number) Number)
4   (declare-fun f (Number) Number)
5
6   (assert ∀x:Number {f(x)} f(x) = f(inc(x))) ; Q1
7
8   (assert f(z) = z)
9   (check-sat)
```

Figure 3.1: A simple SMT-encoding of a problem. We use pseudocode roughly based on SMT-LIB format with presentational liberties (see appendix C for syntactically correct encoding).

```
[new-match] 0x1541b25a0 #29 #28 #30 ; #31
```

Figure 3.2: The `[new-match]` logs the possibility of an instantiation. In our example, #29 denotes the quantifier Q_1, #28 denotes the trigger `{f(x)}`, #30 denotes the bound variable `z`, and #31 denotes the term `f(z)` that was blamed for the match. See appendix A for a complete documentation.

```
[instance] 0x1541b25a0 ; 1
[attach-enode] #459 1
[attach-enode] #460 1
[attach-enode] #461 1
[end-of-instance]
```

Figure 3.3: The lines within `[instance]` and `[end-of-instance]` show the updates to the e-graph due to an instantiation. In our case #459 represents `inc(z)`, #460 represents `f(inc(z))`, and #461 represents `f(z)=f(inc(z))`.

Note that on line 8, we have the ground term `f(z)` since `z` is declared as a constant. As the trigger `{f(x)}` of Q_1 matches with this ground term, Q_1 can be instantiated with `x` bound to `z` to obtain the fact `f(z)=f(inc(z))`.
Through this instantiation, we have obtained the three new ground terms `f(z) = f(inc(z))`, `inc(z)`, and `f(inc(z))`. We will refer to this first instantiation of Q_1 as Q_1^0. As discussed in section 2.2, Z3 keeps track of ground terms and equalities between ground terms in an e-graph. As this instantiation generated new ground terms, they are added to the e-graph as e-nodes.

Such a quantifier instantiation is logged in two stages [26]: first, the possibility for an instantiation due to a matching pattern is logged using `[new-match]` as seen in figure 3.2 . After that, the updates to the e-graph are logged within lines starting at `[instance]` and ending with `[end-of-instance]` as seen in figure 3.3 .

Note that the newly obtained term `f(inc(z))` again matches with the pattern `{f(x)}` and hence Q_1 can again be instantiated but this time with `x` bound to `inc(z)`. This is represented in the log with a `[new-match]` as shown in figure 3.4. We will refer to this second instantiation of Q_1 as Q_1^1.

```
[new-match] 0x1541b2a38 #29 #28 #459 ; #460
[instance] 0x1541b2a38 ; 2
```

Figure 3.4: Second pattern match for Q_1 followed by the start of the corresponding instantiation. Note that the quantifier and trigger identifiers correspond to the ones in figure 3.2. The only difference is that now the bound term is #459 which represents `inc(z)` and that the blamed term is #460 which corresponds to `f(inc(z))`.

Representing Blame-Term Dependencies

Our goal is to represent such a *blame-term dependency* from the first to the second instantiation of Q_1 with one node for each instantiation and a directed edge from the node representing the first to the node representing the second instantiation as depicted in figure 3.5. To do this, our parser needs to record for each instantiation which terms caused the pattern match, i.e., the *blame terms*, and which instantiations created the blame terms. In our example, we would need to record that Q_1^1 was triggered by the term `#460` which in turn was created by Q_1^0.

Note that in general, not every `[new-match]` also has a corresponding `[instance]`-block. In other words, not every possibility for a quantifier instantiation also leads to an actual instantiation. Therefore, our parser constructs two distinct vectors: one for the matches and one for the actual instantiations, which are populated by elements representing the matches and instantiations as seen in figure 3.6.

When the parser reaches a `[new-match]`-line, it creates an element of type `Match` where it stores the blame terms' indices. This element is then pushed onto the `Insts::matches` vector such that we obtain a `MatchIdx` that we can map the `Fingerprint` to. If the parser reaches an `[instance]` block with the corresponding fingerprint, it can look up the `MatchIdx` corresponding to the

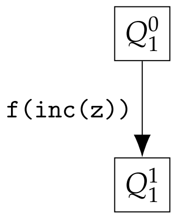

$$\begin{tikzcd}\boxed{Q_1^0} \arrow[d, "\texttt{f(inc(z))}"'] \\ \boxed{Q_1^1}\end{tikzcd}$$

Figure 3.5: Pictorial representation of the blame-term dependency between the first and the second instantiation of Q_1. Node Q_1^0 corresponds to the match in figure 3.2 and node Q_1^1 corresponds to the match in figure 3.4.

```
struct Insts {
  fingerprint_to_match: FxHashMap<Fingerprint, (MatchIdx, Option
    <InstIdx>)>,
  matches: TiVec<MatchIdx, Match>,
  insts: TiVec<InstIdx, Instantiation>,
}

struct Match {
  blamed: Box<[BlameKind]>,
  // other fields omitted
}

struct Instantiation {
  match_: MatchIdx,
  fingerprint: Fingerprint,
  // other fields omitted
}
```

Figure 3.6: Data structures used for recording information about instantiations and their corresponding matches.

`Fingerprint` in the `Insts::fingerprint_to_match` hash map and thus correctly populate its `Instantiation::match_` field.

This way, once we have parsed the log, we can iterate over `Insts::insts` and create nodes for each `Instantiation`. Since we have stored a reference to the match for each instantiation, which stores the blame terms' indices, we now only need some data structure to store for each blame term which instantiation created it such that we know which instantiations to blame.

As we saw in the example at the beginning of section 3.1, the `[attach-enode]` -lines within an `[instance]`-block are used to log which terms were attached to the e-graph due to this instantiation. Therefore, our parser has a data structure (see `ENode` in figure 3.7) representing the e-nodes where the term indices of the created terms are stored (see `ENode::owner`) , as well as the instantiation which created the e-node (see `ENode::created_by`). This way, we have all the information we need to correctly represent dependencies between instantiations A and B where B was triggered by a term generated by A.

Representing Equality Dependencies

Consider the SMT problem in figure 3.8. It is identical to the problem in figure 3.1 except that we have added a new quantified formula, Q_2, and changed the third assertion.

```
pub enum BlameKind {
  Term { term: ENodeIdx },
  // other fields omitted
}
pub struct EGraph {
  enodes: TiVec<ENodeIdx, ENode>,
  // other fields omitted
}
pub struct ENode {
  pub created_by: Option<InstIdx>,
  pub owner: TermIdx,
  // other fields omitted
}
```

Figure 3.7: Data structures used for recording information about e-nodes. Note that in figure 3.6, `Match::blamed` stores elements of type `BlameKind` where the `ENodeIdx` of the created term is stored.

```
(assert ∀x:Number {f(x)} f(x) = f(inc(x))) ; $Q_1$
(assert ∀x:Number {sum(f(x),x)}
    sum(f(x),inc(x)) = inc(sum(x,x))) ; $Q_2$

(assert sum(f(z),z) = z)
```

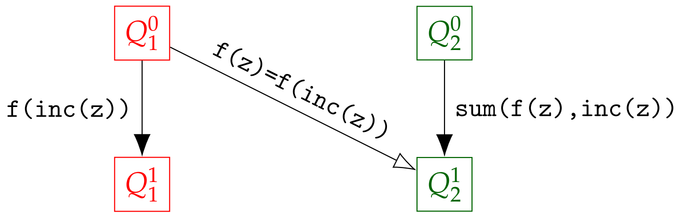
Figure 3.8: The SMT-encoding of the same problem as in figure 3.1 but extended with Q_2 and different third assertion (see appendix C for syntactically correct encoding).

As the ground term `sum(f(z),z)` matches with the trigger `{sum(f(x),x)}` by binding `x` to `z`, Z3 can instantiate Q_2 and obtain the fact `sum(f(z),inc(z))= inc(sum(z,z))`. By doing this, Z3 obtains the ground term `sum(f(z),inc(z))`. We will refer to this first instantiation of Q_2 as Q_2^0. At this point, there are no more ground terms that match with the trigger of Q_2. However, just as in section 3.1 Z3 can instantiate Q_1 and generate the equality term `f(z)=f(inc(z))`. Z3 can then use this fact to rewrite the ground term `sum(f(z),inc(z))` into `sum (f(inc(z)),inc(z))` which again matches Q_2's trigger by binding `x` to `inc(z)` and therefore Z3 can once again instantiate Q_2 to obtain the new fact `sum(f( inc(z)),inc(inc(z)))=inc(sum(inc(z),inc(z)))` which contains the ground term `sum(f(inc(z)),inc(inc(z)))`. Table 3.9a gives an overview of the instantiations and the corresponding blame and yield terms we have covered. We will refer to this second instantiation of Q_2 as Q_2^1.

Recall that in section 3.1, we had a blame-term dependency because the

Inst.	Matched term	Binding	Relevant yield terms
Q_2^0	`sum(f(z),z)`	`x=z`	`sum(f(z),inc(z))`
Q_1^0	`f(z)`	`x=z`	`f(z)=f(inc(z))`
Q_2^1	`sum(f(inc(z)),inc(z))`	`x=inc(z)`	`sum(f(inc(z)),inc(inc(z)))`
Q_1^1	`f(inc(z))`	`x=inc(z)`	`f(inc(z))=f(inc(inc(z)))`

(a) Q_2^i represents the i^{th} instantiation of Q_2 and Q_1^i represents the i^{th} instantiation of Q_1.



(b) Instantiation graph: The edges with filled arrowheads represent blame-term dependencies. The edges with empty arrowheads represent equality dependencies.

Figure 3.9: Summary of considered instantiations.

blame term of Q_1^0 directly matched against the trigger of Q_1. Here, however, we are using the equality generated by Q_1^0 to rewrite a ground term generated by Q_2^0 such that it matches with Q_2's trigger. Our goal is to also represent such equality dependencies in the instantiation graph as depicted in figure 3.9b.

Implementing Equality Edges

If we inspect the log generated by Z3, we will find that the `[new-match]` of the instantiation corresponding to Q_2^1 lists the equality `f(z)=f(inc(z))` as shown in figure 3.10.

```
[instance] 0x1541b25a0 ; 1
[attach-enode] #461 1
[end-of-instance]
⋮
[eq-expl] #31 lit #461 ; #460
[eq-expl] #460 root
[new-match] 0x1541b2a68 #38 #37 #459 ; #464 (#31 #460)
```

Figure 3.10: The `[new-match]` for instantiation Q_2^1. The term #31 represents `f(z)` and #460 represents `f(inc(z))`. The tuple `(#31 #460)` represents the equality between those two terms (see appendix A for more details).

As we can see in figure 3.10, the `[new-match]` for Q_2^1 is preceded by lines starting with `[eq-expl]`, which are used to log the equality explanations (see appendix A). In general, these equality explanations can be more complex, but Z3 guarantees that each blamed equality in a `[new-match]` is preceded by `[eq-expl]`-lines such that we can follow each equality until the root of the union-find data structure representing the equivalence class of these terms [26]. Therefore, we can design our parser such that it records these equality explanations in a way that we can follow these steps until we reach the root of the equivalence class. In this example, this would correspond to a graph as depicted in figure 3.11.

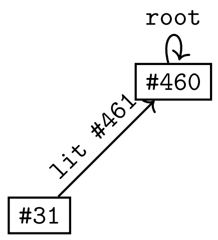


Figure 3.11: The parser's internal representation of the equivalence class with root #460 after parsing the `[eq-expl]`-lines in figure 3.10.

As discussed earlier, the parser stores for each created term logged by `[attach -enode]` which instantiation created it (see `ENode::created_by`). Therefore, when the parser encounters the `[new-match] ... ; (#31 #460)` in figure 3.10 it can follow the internal e-graph representation from both nodes until it reaches the root, identify the blamed equalities along the way and add those to the blame terms of the corresponding match (see `Match::blamed` in figure 3.6). In our example, the parser will find the path `[#31, #460]` from the node `#31` to the root of its equivalence class and the trivial path `[#460]` from the node `#460` to the root of its equivalence class and hence blame the equality term `[#461]` as it is on former path (see figure 3.11). During the construction of the instantiation graph, we hence have all the information we need to blame the appropriate instantiations.

Consider the example in figure 3.12. In general, whenever the parser explains an equality $a = c$, it will first find the path from the root of a to a (here `[`$f(i)$`,`b`,`a`]`) and the path from the root of c to c (here `[`$f(i)$`,`b`,`c`]`) and then drop the shared part of the path before concatenating them into a single path. (here it would only keep `[`a`]` and `[`c`]` and concatenate them into `[`a`,`c`]`). This way, the parser only blames the equalities eq_1 and eq_2 but not eq_3. The intuition behind this is that the equality $a = c$ is explained by applying transitivity to $a = b$ (eq_1) and $b = c$ (eq_2), and the equality $b = f(i)$ does not help explain this equality.

If some congruences are used to explain an equality, we recursively explain the pairwise equalities between the arguments. In the example of figure 3.13

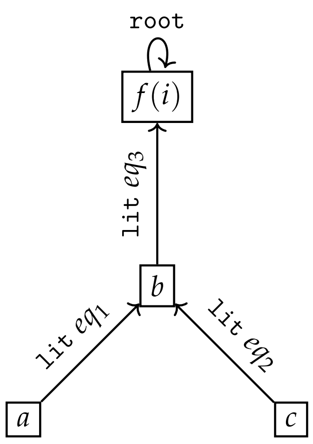
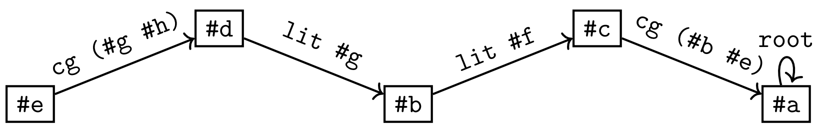


Figure 3.12: Example for an e-graph.

```
[eq-expl] #e cg (#g #h) ; #d
[eq-expl] #d lit #g ; #b
[eq-expl] #a root
[eq-expl] #b lit #f ; #c
[eq-expl] #c cg (#b #e) ; #a
[new-match] ... ; (#a #b)
```

(a) Example for equality explanations using congruence closure property.



(b) Partial e-graph corresponding to the equality explanations in figure 3.13a.

Figure 3.13: Example of equality explanations with cg.

when explaining the equality `[new-match] ... (#a #b)` the parser would find that the path from `#a` to `#b` involves a `lit`-equality due to equality term `#f` and hence blame the instantiation that created said equality term. But the path also involves a `cg` with pairwise argument-equality `(#b #e)` and hence also recursively explain said equality. In this example, the parser would find that the path from `#e` to `#b` involves a `lit`-equality due to equality term `#g` and hence also blame the instantiation that created that equality term. But since the path also contains a `cg`-equality, it would recursively explain the equality `(#g #h)`.

Note that in order to represent equality and blame-term dependencies in distinct ways, the `Match`-struct from figure 3.6 contains a boxed slice of elements of type `BlameKind` which is defined in figure 3.14 such that we can

```
pub enum BlameKind {
  Term { term: ENodeIdx },
  Equality { eq: ENodeIdx },
}
```

Figure 3.14: Definition of `BlameKind` such that we can use different styles for blame-term and equality dependencies in the instantiation graph.

make the style of the edges dependent on this field.

3.2 Visualizing the Instantiation Graph

After parsing the log and constructing the instantiation graph, our goal is to display the graph to the user in practical ways. Like AP1, we opt to use a *layered graph drawing* (also known as *Sugiyama-style graph drawing*) [23] for the graph layout as it is well-suited for hierarchical structures and because there are established libraries to generate such graph layouts such as Graphviz [12].

We use the `petgraph`-library [1] as the data structure for the instantiation graph as it provides various graph algorithms as well as methods to generate dot file format output from which Graphviz can generate a layered graph drawing. As we have a web application, we use a WebAssembly build of Graphviz called Viz.js [2] such that we can run Graphviz directly in the browser and generate SVG code from the dot file that the browser can display

In general, displaying the entire instantiation graph is not a viable option as it can contain arbitrarily many nodes and edges. Therefore, we implemented various filters that the user can apply to the graph to make it more intelligible. Whenever the user applies a filter, it is appended to what we call *filter chain*. The filter chain represents the concatenation of all applied filters. The user can remove individual filters from the filter chain or reset it to the default. These filtering features are described in section 3.2.3.

Whenever nodes are filtered out, we still want to reflect the indirect dependencies between instantiations to the user. More concretely, if there is a path from node A to B in the original instantiation graph, but the nodes along that path have been filtered out, then there should still be an indication of the indirect dependency between instantiations the represented by node A and B. Therefore, we implemented a *reconnecting algorithm* to reconnect a filtered graph in a way that reflects these indirect dependencies. This method is described in section 3.2.2, and the corresponding code is implemented in `InstGraph::retain_visible_nodes_and_reconnect`.

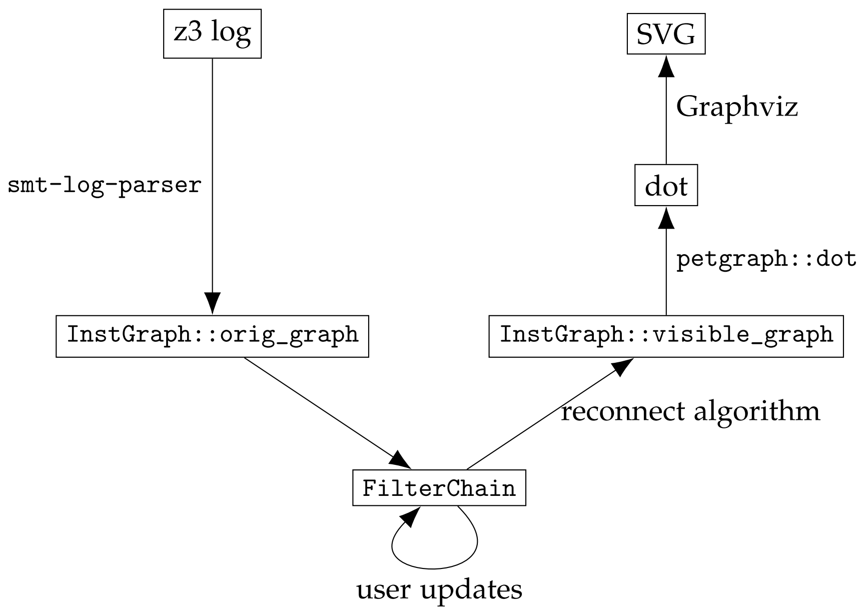


Figure 3.15: Toolchain for converting the log generated by Z3 into SVG format that the browser can display. The `smt-log-parser` library was developed in this project.

At any point in time, there are two instantiation graphs stored in memory; the *original instantiation graph* which contains all information obtained during parsing (`InstGraph::orig_graph`) and the currently *visible instantiation graph* that the user sees in the GUI which represents the filtered original instantiation graph. Each time the user updates the filter chain, we recompute the visible instantiation graph and compute a new SVG for the updated graph. Note that the Z3 log only needs to be parsed once. This toolchain is visualized in figure 3.15.

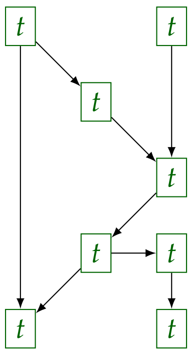
The filters do not directly change the graph structure of the original instantiation graph, i.e., the filters do not change the set of nodes or edges. Rather, the filters just modify the the `NodeData::visible: bool` field of the nodes. Only right before rendering, a new visible instantiation graph is constructed by removing all the nodes marked as invisible from the original instantiation graph and applying the reconnecting algorithm. This procedure is visualized in figure 3.17. As the reconnecting algorithm may add indirect edges, which do not represent blame-term or equality dependencies, we need different types for the edges of `InstGraph::orig_graph` and `InstGraph::visible_graph`. In the original graph, we only have blame-term or equality dependencies, and therefore we can just use `BlameKind` (see figure 3.14) whereas in the visible graph we distinguish between direct (`EdgeType::Direct`) and indirect (`EdgeType::Indirect`) dependencies where the former is a wrapper-struct for `BlameKind`.

```
  pub struct InstGraph {
  orig_graph: Graph<NodeData, BlameKind>,
  pub visible_graph: Graph<NodeData, EdgeType>,
  // other fields omitted
}

pub struct NodeData {
  pub is_theory_inst: bool,
  cost: f32,
  pub inst_idx: InstIdx,
  pub mkind: MatchKind,
  visible: bool,
  child_count: usize,
  parent_count: usize,
  pub orig_graph_idx: NodeIndex,
  pub min_depth: Option<usize>,
  max_depth: usize,
  topo_ord: usize,
  }

pub enum EdgeType {
  Direct {
      kind: BlameKind,
      orig_graph_idx: EdgeIndex,
  },
  Indirect,
}
```

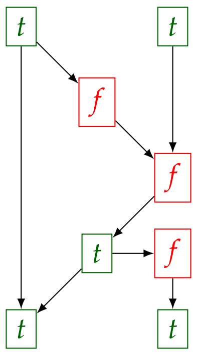
Figure 3.16: Definition of `InstGraph` used to represent the instantiation graph.

3.2.1 Displaying Node and Edge Information

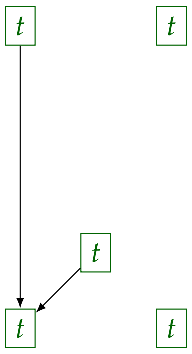
In addition to displaying the instantiation graph to the user, we also want to allow the user to select nodes and display additional information about the instantiation the selected node represents. The basic idea is to add event listeners to the HTML elements representing the nodes such that when they are clicked, the browser sends a message containing the index of the selected node. It can then use the instantiation-graph data structure `InstGraph` to look up the index of the instantiation corresponding to the clicked node (see `NodeData::inst_idx` in figure 3.16) and retrieve the information associated with the instantiation from the parser by indexing into `Insts::insts` with the instantiation index (see figure 3.6). An analogous approach can be used for



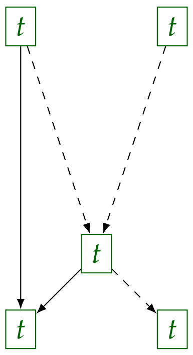
(a) Original instantiation graph before applying any filters. Note that t is used to indicate that the `NodeData::visible`-field is set to `true`.



(b) Original instantiation graph after applying some filters. Note that f is used to indicate that the `NodeData::visible` -field is set to `false`. Note how the filters do not remove any nodes or edges.



(c) Visible instantiation graph after storing in it the subgraph of original instantiation graph defined by the nodes marked as visible in figure 3.17b.



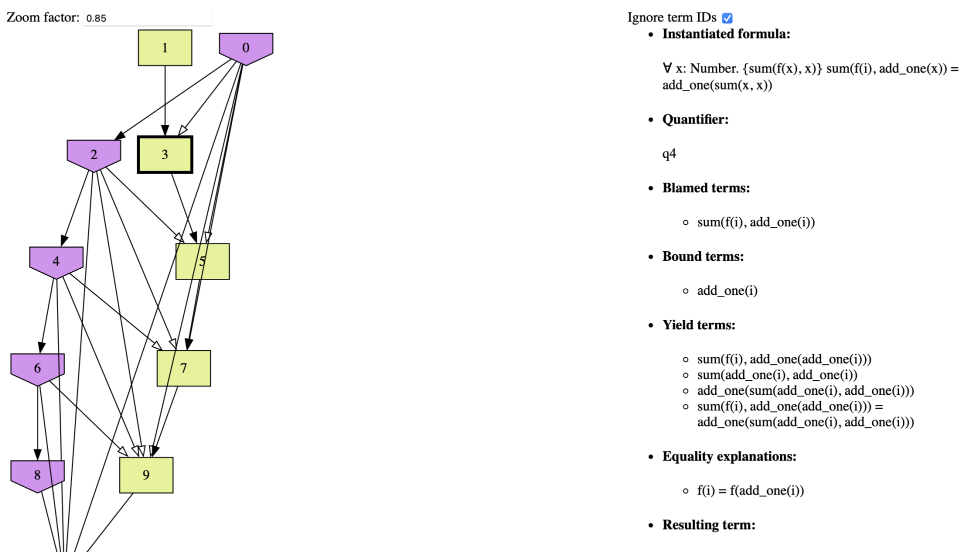
(d) Visible instantiation graph after applying the reconnecting algorithm. Note how indirect edges were added between nodes with indirect dependencies in the original graph of figure 3.17a.

Figure 3.17: Illustration of the procedure used to construct the visible instantiation graph for display after applying some filters and right before rendering. Note that the graph that gets rendered is the graph in figure 3.17d. In other words, the graph for which we create the dot-file in figure 3.15 is the visible instantiation graph after applying the reconnect algorithm.

the edges. An example of how this information is displayed in the tool can be found in figure 3.18.

Node Shapes

As there are filters for displaying the children of a node as well as for displaying the parents of a node, we want to indicate to the user whether or not these filters are applicable to a displayed node by using the node shapes illustrated in figure 3.20.



Zoom factor: 0.85

Ignore term IDs ☑

- **Instantiated formula:**

  ∀ x: Number. {sum(f(x), x)} sum(f(i), add_one(x)) = add_one(sum(x, x))

- **Quantifier:**

  q4

- **Blamed terms:**
  - sum(f(i), add_one(i))

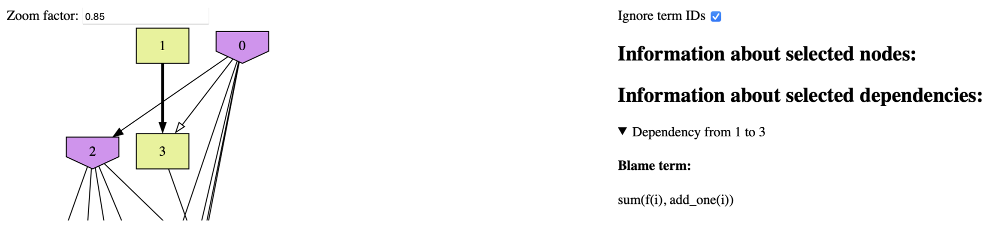
- **Bound terms:**
  - add_one(i)

- **Yield terms:**
  - sum(f(i), add_one(add_one(i)))
  - sum(add_one(i), add_one(i))
  - add_one(sum(add_one(i), add_one(i)))
  - sum(f(i), add_one(add_one(i))) = add_one(sum(add_one(i), add_one(i)))

- **Equality explanations:**
  - f(i) = f(add_one(i))

- **Resulting term:**

Figure 3.18: Information displayed about a selected node.



Zoom factor: 0.85

Ignore term IDs ☑

Information about selected nodes:

Information about selected dependencies:

▼ Dependency from 1 to 3

Blame term:

sum(f(i), add_one(i))

Figure 3.19: Information displayed about a selected edge.

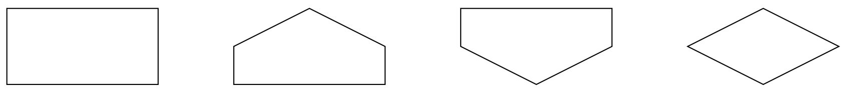


Figure 3.20: The shapes from left to right are used if all parents and children are displayed, some parents are not displayed, some children are not displayed, and some parents and some children are not displayed.

Node Coloring

We use different colors for different nodes based on the quantifier that they represent. If we have n different quantifiers, then a naive approach would be to divide the range of hues into $n-1$ intervals of equal length such that the hues are as distinct as possible and assign to the i^{th} quantifier where $0 \leq i < n$ the hue located at the left border of the i^{th} interval. If we have a dependency between instantiations of adjacent quantifiers and n is large, the distance between two successive hues that the adjacent quantifiers are mapped to may not be large enough to be visually distinguishable (see figure 3.21a). Therefore, we introduce a permutation on the indices such that any two adjacent quantifiers are mapped to more distinct hues.

If we have n quantifiers we need to find a permutation for the set $\mathbb{Z}_n := \{0, 1, ..., n-1\}$. A commonly used algorithm for generating a pseudo-randomized sequence $(X_k)_{k=0}^{n-1}$ of numbers is with a *mixed congruential generator* [16] which is defined by the recurrence relation

$$X_{k+1} = (aX_k + c) \bmod n$$

where $c \neq 0$, $0 < n$, $0 < a < n$, $0 \leq c < n$, $0 \leq X_0 < n$. Note that since all indices satisfy the last condition, we may choose any of them as the seed value X_0, but we will choose $X_0 = 0$ for simplicity. To obtain a permutation of $\mathbb{Z}_n$, we have to ensure that this sequence's period is n. According to the Hull-Dobell theorem [15], this occurs if and only if

- n and c are relatively prime
- $a - 1$ is divisible by all prime factors of n
- $a - 1$ is divisible by 4 if n is divisible by 4.

By choosing $a = 1$, we can satisfy the second and third conditions and reduce the sequence to

$$X_{k+1} = (X_k + c) \bmod n$$

which, by solving the recurrence and using $X_0 = 0$, can be simplified to

$$X_k = kc \bmod n.$$

This defines our permutation $p : \mathbb{Z}_n \to \mathbb{Z}_n$

$$p : k \mapsto kc \bmod n.$$

Recall that our goal was to construct a permutation such that two successive indices k, $k+1$ are mapped to values that have a sufficiently large difference $|p(k+1) - p(k)|$. By satisfying the conditions of the Hull-Dobell theorem, we ensure that the period of the sequence $(X_k)_{k=0}^{n-1}$ is n. This allows us to prove that p is indeed a permutation by showing that all elements of the sequence $(X_k)_{k=0}^{n-1}$ must be unique. Suppose there were two indices $i, j \in \mathbb{Z}_n$ such that $i < j$ and $X_i = X_j$ then the period of the sequence would be at most $j - i \leq n - 1$ as our sequence is obtained by repeatedly adding c modulo n and hence $\forall k \in \mathbb{Z}_n . X_{i+k} \equiv_n X_{j+k}$. This contradicts the fact that the period of the sequence $(X_k)_{k=0}^{n-1}$ is n.

Therefore, we are interested in finding a $0 \leq c < n$ satisfying $\gcd(n, c) = 1$ (to satisfy the first condition in the Hull-Dobell theorem) which also maximizes $|p(k+1) - p(k)|$. To find a c satisfying $\gcd(n, c) = 1$, we can choose a prime number that is not a divisor of n. Such a prime can always be found since any prime larger than $n/2$ is coprime to n.

In theory, the optimal value for maximizing the difference $|p(k+1) - p(k)|$ is $\lfloor n/2 \rfloor$ (see appendix B). Therefore, we are interested in finding a prime as close as possible to this theoretically optimal value. To find the prime closest to $\lfloor n/2 \rfloor$ we can use the prime number theorem [14] to approximate the number of primes smaller than $n/2$ by

$$\left\lfloor \frac{n/2}{\ln(n/2)} \right\rfloor$$

and skip this number of primes for choosing c and take the first such prime which is not a divisor of n. To avoid a division by 0 when $n = 2$, we can just take the first prime $a = 2$.

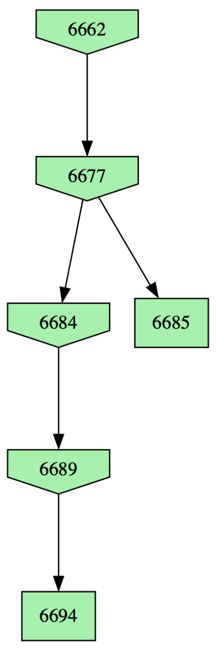
An example to illustrate the effect of our permutation can be found in figure 3.21.

3.2.2 Reconnecting Algorithm

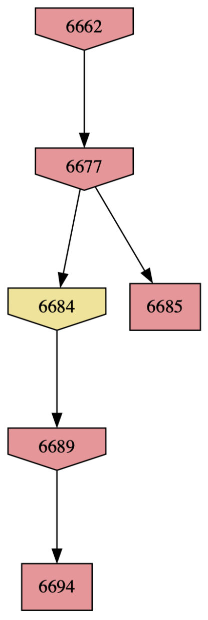
Given the instantiation graph $G = (V, E)$ and some subgraph $G' = (V', E')$ with $V' \subseteq V, E' \subseteq E$. See figure 3.22a and 3.22b for an example. Consider two nodes $u, v \in V'$ such that $(u, v) \in E^+ \setminus \mathrm{Id}_V$ (irreflexive transitive closure of the blames relation). If $(u, v) \in E'$, then the (direct) dependency between the instantiations represented by u and v is also represented in the subgraph G'. On the other hand, if $(u, v) \notin E'$, we would still like to somehow represent the indirect dependency between u and v as v is reachable from u via the blames relation.

For instance, nodes 1 and 5 in figure 3.22b indirectly depend on each other as there is a path from node 1 to 5 in the original instantiation graph G shown in figure 3.22a and therefore, it would be misleading not to represent this information to the user.

We opt to represent such indirect dependencies as dashed edges to distinguish them from the direct dependencies (see section 3.1), which are represented as non-dashed edges (see figure 3.22d). Note that only nodes in V' that have filtered out children with respect to G should have outgoing dashed edges; otherwise, all the dependencies are already represented. Likewise, only nodes in V' that have filtered out parents with respect to G should have incoming dashed edges. Therefore we define the sets $OUT = \{v \in V' |\, \deg^+_{G'}(v) < \deg^+_G(v)\}$ and $IN = \{v \in V' |\, \deg^-_{G'}(v) < \deg^-_G(v)\}$ where



(a) Colors without index permutation.



(b) Colors with index permutation.

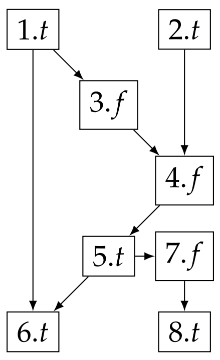
Figure 3.21: Note how on the left the colors of node 6684 and the other nodes are not visibly distinguishable as the corresponding quantifier indices are 763 and 765, and therefore the naive approach assigns very similar hues to both indices. On the right, we have permuted the indices as described in section 3.2.1.

$\deg_G^-(v)$ and $\deg_G^+(v)$ denote the in- and outdegree of node v with respect to graph G.

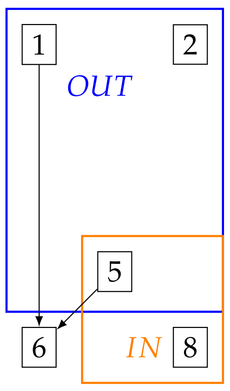
This explains the need for the `NodeData::parent_count: usize` and `NodeData::child_count: usize` fields in figure 3.16. By storing the number of children and parents of each node during the construction of the `InstGraph::orig_graph` we can easily check if the in- or outdegree of a node is smaller with respect to `InstGraph::visible_graph` than with respect to `InstGraph::orig_graph`.

Once we have computed the nodes with filtered-out children, OUT, and the nodes with filtered-out parents, IN, we need to decide for each pair $(u, v) \in OUT \times IN$ whether v is reachable from u in the original graph $G = (V, E)$. This is equivalent to deciding if (u, v) is in the irreflexive transitive closure of the blames relation, i.e., if $(u, v) \in E^+ \setminus \mathrm{Id}_V$.

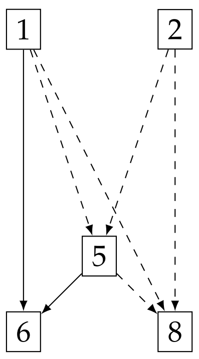
If we do this with the example in figure 3.22b we end up with a graph that looks like the one in figure 3.22c. Even though the reachability information is complete, it also contains a lot of redundant information that we prefer not to display. For instance, there is a dashed edge $(1, 8)$ as well as dashed edges $(1, 5)$ and $(5, 8)$. We compute the transitive reduction [3] to avoid such



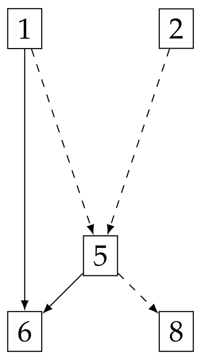
(a) Original instantiation graph $G = (V, E)$ after applying some filters that modify the `NodeData::visible`-fields of the nodes. The notation $n.t$ and $n.f$ should be understood as node n having `NodeData::visible` set to `true` and `false`, respectively.



(b) Subgraph $G' = (V', E')$ of G from figure 3.22a after removing the nodes marked as invisible. Notice that OUT contains those nodes that have filtered out child nodes, and IN contains those that have filtered out parent nodes.



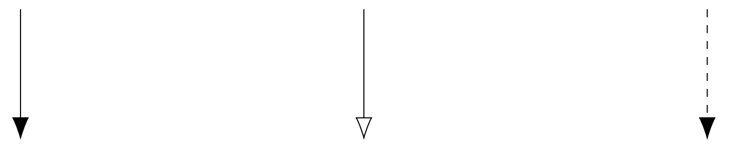
(c) Instantiation graph from figure 3.22b after recovering reachability information by adding edges in $(OUT \times IN) \cap (E^+ \setminus \mathrm{Id}_V)$.



(d) Instantiation graph from figure 3.22c after computing the transitive reduction. Note that the direct edge $(1, 6)$ is not removed as it represents a direct dependency.

Figure 3.22: Example to illustrate the reconnecting algorithm.

redundant edges and declutter the graph. Strictly speaking, the transitive reduction displayed in figure 3.22d should not contain the edge $(1, 6)$ as by removing it, we would obtain a graph with fewer edges, and there would still be a path from $(1, 6)$ (via node 5). We choose to keep these direct dependencies as; otherwise, it would indicate to the user that there is only an indirect dependency between nodes 1 and 6, which is false. The reconnected graph should represent all direct dependencies $(u, v) \in E$ as well as indirect dependencies $(u, v) \in E^+ \setminus \mathrm{Id}_V$. We use different edge styles to represent these different kinds of dependencies (see figure 3.23).



(a) Edge style for blame-term dependency **(b)** Edge style for equality dependency **(c)** Edge style for indirect dependency

Figure 3.23: Summary of edge styles.

Implementation Details

Note that computing the transitive closure is an expensive operation in general, and storing this information can also be highly memory inefficient if not done carefully. For memory efficiency, we opt to store the transitive closure information for each node in highly optimized Roaring bitmaps [7] which use a binary encoding to store for each node which nodes are reachable from it.

For runtime efficiency, we exploit the fact that our instantiation graph is a directed acyclic graph by traversing the instantiation graph in reverse topological order. The key idea is that for a node without any children, the only reachable node is itself, and therefore, we can just set the corresponding bit in the bitmap. If a node u has children, then we know that all the nodes that can be reached from each child v can also be reached from u. Therefore, we can do a bitwise OR of all the bitsets of the children of node u and set the bit of node u to obtain the reachable nodes of u and hence the correct bitmap. As we traverse the graph in reverse topological order, by the time we reach node u, all the bitsets of u's children have been computed.

By storing the bitmaps in the topological order of the nodes and storing in each node of the instantiation graph the associated topological order (see `NodeData::topo_ord` in figure 3.16), we can efficiently look up whether node v is reachable from node u by indexing into the vector of bitmaps (see `InstGraph::tr_closure` in figure 3.24) using the topological order of u and looking up whether the bit stored for node v is set.

3.2.3 Filtering Operations

As discussed at the beginning of section 3.1, the instantiation graph can consist of many nodes and edges, so displaying the graph in its entirety is not a viable option. Therefore, we will implement various filters that the user can apply to the graph.

The basic idea is that the GUI provides a set of filters that the user can concatenate. We will call the concatenation of the applied filters a *filter chain*.

```
    pub struct InstGraph {
      // omitted fields
      tr_closure: Vec<RoaringBitmap>,
      // omitted fields
  }
```

Figure 3.24: The `InstGraph` data structure also has a field for storing the bitmaps encoding the reachability information of each node. The bitmaps are ordered by the topological order of the nodes such that we can index into this vector using the topological order stored in each node.

As discussed in the beginning of section 3.2, each node of the instantiation graph has a field `NodeData::visible: bool` and each time the user applies a filter, the filter does not directly alter the structure of the instantiation graph but rather updates this field such that right before rendering the graph, we can apply the reconnecting algorithm described in section 3.2.2.

The advantage of applying the reconnecting algorithm right before rendering as opposed to each time a filter is applied is that if the user wishes to remove any arbitrary filter F_i from a chain of filters `[`$F_1, \ldots, F_i, \ldots, F_n$ `]` we can just update the filter chain and reapply the updated chain of filters updating the `NodeData::visible`-field.

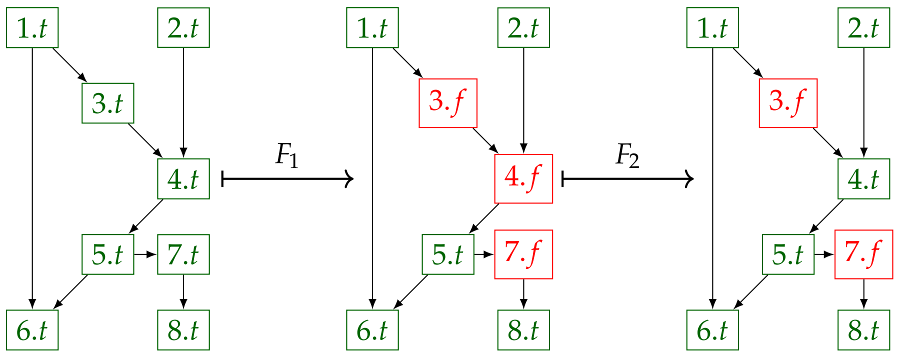


Figure 3.25: Note how filter F_1 hides nodes 3, 4 and 7 by marking them as invisible whereas filter F_2 adds nodes to the graph by marking node 4 as visible.

Note that the filtering operators do not necessarily remove nodes, as the name might suggest. For instance, we will implement a filter for displaying the parent nodes of a selected node. In a nutshell, this filter will just set the `NodeData::visible`-field of the selected node's parents to `true` such that the next time the graph is rendered, the parents are displayed. See figure 3.25 for an illustration.

Furthermore, we have empirically found that if the graph contains many edges, it can take a long time for Graphviz to generate the SVG output. Therefore, we also implement a warning prompt that is displayed to the user after applying a filter F_n but before rendering the filtered graph, which warns the user that based on the number of nodes and edges in the filtered graph $G' = F_n \circ F_{n-1} \circ \cdots \circ F_1 G$ it might take a long time to render the graph. The user then has the option to

- render the filtered graph despite the warning,
- apply the filter without rendering the graph (such that the user can apply more filters until the node and edge count become more reasonable),
- or undo the applied filter, returning to the previous filter chain `[` $F_1, \ldots, F_{n-1}$ `]`.

Figure 3.26 shows the GUI of the filter chain and the filters the user can append to the filter chain. Note that the user can remove individual filters from the filter chain and reset the whole filter chain to a sensible default, which filters out all theory-solving instantiations and only shows the 125 most expensive instantiations. These filters will be explained in more detail in the remainder of this section.

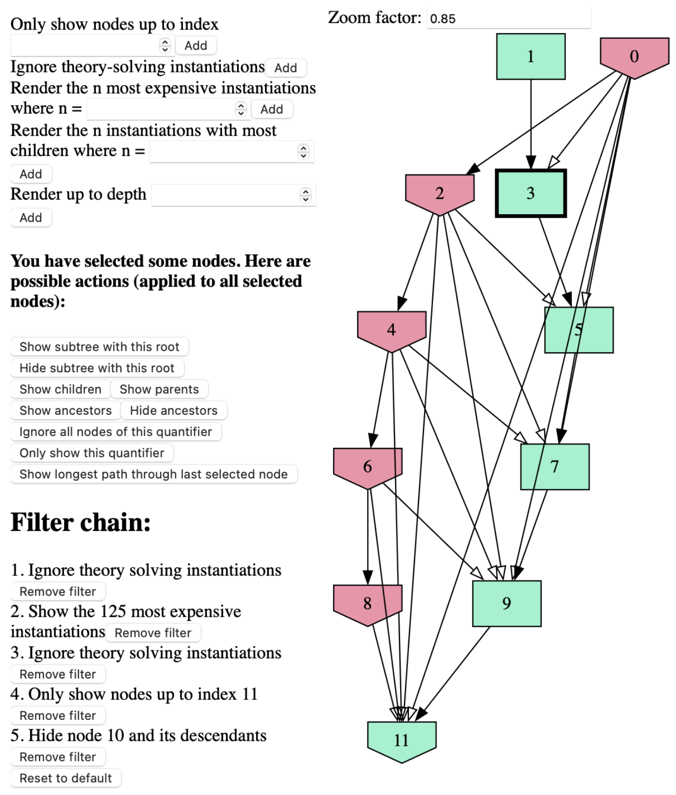
Showing a longest path through a selected instantiation

For this filter, we want to show the longest path through a user-selected node u. The key idea for this filter is that if we know the maximal depth of each node with respect to the entire graph, then we can compute the maximal depths with respect to the subgraph rooted at the selected node u and backtrack the longest path from the node that is furthest away from u in said subgraph until we reach a root of the entire instantiation graph.

Therefore, we compute during the construction of the instantiation graph the maximal depth of each node with respect to the roots of the instantiation graph and store it in each node (see `NodeData::max_depth: usize` in figure 3.16). The root nodes get assigned a maximal depth of 0. A non-root node gets assigned the maximal depth of its parents plus one. We traverse the graph in topological order to ensure all the parent nodes' maximal depths have been computed once we reach a node.

When the user applies this filter to a selected node u, we make a temporary copy of the graph rooted at node u and compute the maximal depths analogously to how we compute them during the construction with the difference that only node u is the root, which gets assigned maximal depth 0.

All the nodes that are visited during the backtracking are marked as visible.


Only show nodes up to index

Ignore theory-solving instantiations

Render the n most expensive instantiations where n =

Render the n instantiations with most children where n =

Render up to depth

Zoom factor: 0.85

You have selected some nodes. Here are possible actions (applied to all selected nodes):

Filter chain:

1. Ignore theory solving instantiations
2. Show the 125 most expensive instantiations
3. Ignore theory solving instantiations
4. Only show nodes up to index 11
5. Hide node 10 and its descendants

Add
Show subtree with this root
Hide subtree with this root
Show children
Show parents
Show ancestors
Hide ancestors
Ignore all nodes of this quantifier
Only show this quantifier
Show longest path through last selected node
Remove filter
Reset to default
0
1
2
3
4
5
6
7
8
9
11

Figure 3.26: GUI of the axiom profiler. The left panel shows the filters that can be applied and the filter chain, which lists all the filters that have been applied so far in the order in which they were applied.

Hiding theory-solving instantiations

Instantiations involved in theory-solving are often not of particular interest in identifying troublesome instantiations such as matching loops and hence we implement this filter just as in the AP1.

To implement this filter, each node of the instantiation graph has a field `NodeData::is_theory_inst: bool` (see figure 3.16) which is populated during the construction of the instantiation graph. When the user applies this filter, all the nodes that have this field set to `true` are marked as invisible.

Showing expensive and high-branching instantiations

Recall that each node stores in its `NodeData::child_count: usize` field (see figure 3.16) how many children it has in the original instantiation graph. Therefore, we can just initially sort the node indices of the original instantiation graph in descending order of the child count and store this sorted vector of indices in the `InstGraph::branching_ranked_node_indices` field of the instantiation graph (see figure 3.27). When the user wants to show the n

```
  pub struct InstGraph {
  // other fields omitted
  cost_ranked_node_indices: Vec<NodeIndex>,
  branching_ranked_node_indices: Vec<NodeIndex>,
  // other fields omitted
}
```

Figure 3.27: Definition of `InstGraph` used to represent the instantiation graph.

instantiations with the most children (where n is a user-defined parameter) a function is called which iterates over this vector and marks the first n nodes as visible and the remaining nodes as invisible.

An analogous idea can be used to show the n most expensive instantiations. We define the cost of an instantiation u to be 1 if it does not cause any other instantiations, i.e., if $\deg^+(u) = 0$. Otherwise, we recursively define the cost as follows (as in [22]). Let $u \in V$ be a node in the instantiation graph. We define

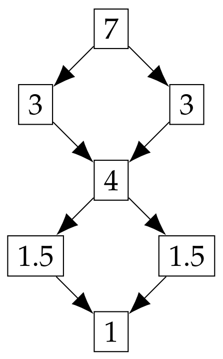
$$\text{cost}(u) = \begin{cases} 1 & \text{if } \deg^+(u) = 0, \\ 1 + \sum\limits_{(u,v)\in E} \frac{1}{\deg^-(v)}\text{cost}(v) & \text{else.} \end{cases}$$

Note that the cost of a node v is evenly distributed onto its parent nodes by only adding the fraction $\frac{1}{\deg^-(v)}$ of $\text{cost}(v)$ to each parent. Figure 3.28 illustrates why this is a sensible design choice as opposed to adding the whole cost of node v to all its parents.

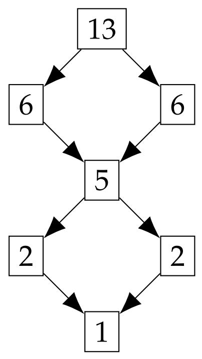
The cost of each instantiation is initialized to 1. Once the parser has processed the entire log file; we traverse the instantiations in reverse order (which coincides with their reverse topological order) and update the costs. When processing node v with cost $\text{cost}(v)$ we read how many instantiations it blames, $\deg^-(v)$, and increment the cost of each parent by $\text{cost}(v)/\deg^-(v)$.

Showing children and parents of a selected node

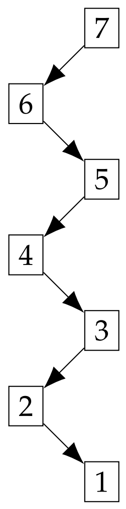
If the user wants to show the parents of a selected node, then a function is called, which marks the parent nodes of the selected nodes as visible. The filter for showing the children of a selected node works analogously.



(a) Instantiation costs with distributing costs



(b) Instantiation costs without distributing costs



(c) Instantiation costs in a long chain

Figure 3.28: Examples of costs associated with nodes. We want to associate the same cost to the root node in figures 3.28a and 3.28c as they are both responsible for seven instantiations in total (including themselves). This example illustrates why distributing the cost evenly among the parents is a sensible choice.

Showing and hiding ancestors or descendants of a selected node

If the user wants to show the descendants of a selected node, a function is called, which does a DFS traversal of the original graph starting from the selected node and marks the visited nodes as visible or invisible depending on whether the user wants to show or hide the descendants. The filters for the descendants work analogously, with the only difference that we do a reverse DFS traversal.

Ignoring or showing only specific quantifiers

Each node has a field `mkind: MatchKind` (see figure 3.16), which stores information that is logged when the parser encounters a `[new-match]`-line. For instance, the index of the instantiated quantifier is stored, and therefore, we can easily look up the quantifier that corresponds to a node in the instantiation graph. When the user selects a node, buttons appear, which allows the user to either show all nodes corresponding to the quantifier of the selected node or to hide all nodes except the ones corresponding to said quantifier. Depending on which one the user clicks, a function is called, which retrieves the quantifier index of the selected node and marks all nodes with the same quantifier index as visible and all others as invisible for the former button and vice versa for the latter button.

3.3 Matching Loop Analysis

In section 3.1, we introduced an example of an SMT-encoding for a problem involving quantified formulas (see figure 3.8). We noticed that Q_1 can be instantiated as the ground term `f(z)` matches against the pattern `{f(x)}` by binding `x` to `z`. One of the newly obtained terms from this instantiation is `f(inc(z))`, which again matches with the pattern `{f(x)}` and hence Q_1 can again be instantiated, but this time with `x` bound to `inc(z)`. This instantiation generates the new ground term `f(inc`^{2}`(z))` and therefore Q_1 can be instantiated yet again. We have a scenario where the instantiations of Q_1 yield terms that directly match with Q_1's pattern, hence causing a self-sustaining looping behavior that can continue indefinitely. Such self-sustaining, repeated instantiations of the same quantifiers are called matching loops as described in section 2.4.

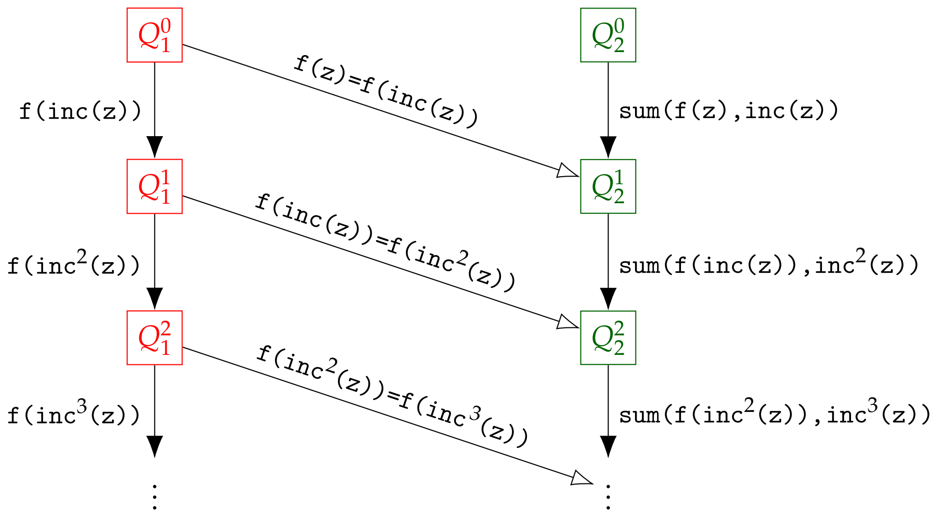
In section 3.1, we analyzed how the first few instantiations of Q_1 and Q_2 happen. These instantiations are summarized in figure 3.29. Note that Z3 can use the equality `f(inc(z))=f(inc`^{2}`(z))` (generated by Q_1^1) to rewrite the ground term `sum(f(inc(z)),inc`^{2}`(z))` (generated by Q_2^1) into `sum(f(inc`^{2}`(z)),inc`^{2}`(z))`. This term again matches against Q_2's trigger `{sum(f(x),x)}` by binding `x` to `inc`^{2}`(z)` hence allowing Q_2 to be instantiated yet again (see Q_2^2 in figure 3.29). Here, we have a scenario where the equalities generated by the matching loop involving only Q_1 allow Z3 to repeatedly rewrite the yield terms generated by the instantiations of Q_2 such that Q_2 can also be instantiated indefinitely.

We can argue that in this example, we have two matching loops. One which only involves quantifier Q_1 and the other, which involves both Q_1 and Q_2. We can see in figure 3.29 that the instantiations of Q_1 blame and yield the terms `f(inc(z))`, `f(inc`^{2}`(z))`, `f(inc`^{3}`(z))`, ... which have the common term structure `f(inc(_))` (note that we use `f(inc`^{2}`(z))` as an abbreviated notation for `f(inc(inc(z)))`). A graphical way to represent this matching loop is shown in figure 3.30 where we have a node for `f(inc(_))` and a self-edge labelled Q_1 indicating that Q_1's pattern directly matches with this term structure and hence can be directly instantiated, yielding a term with the same term structure, hence allowing repeated instantiations of Q_1.

The matching loop involving both Q_1 and Q_2 is more complex as they involve equalities to rewrite terms such that they match with Q_2's pattern. In figure 3.29b we can see that all the instantiations of Q_2 have blamed terms `sum(f(z),inc(z))`, `sum(f(inc(z)),inc`^{2}`(z))`, `sum(f(inc`^{2}`(z)),inc`^{3}`(z))`, ... which have the common term structure `sum(f(_),inc(_))`. But as discussed previously, Z3 uses equalities with common term structure `f(_)=f(inc(_))` to rewrite them into a term which matches against Q_2's pattern `{sum(f(x),x)}`, i.e., into terms with common term structure `{sum(f(_),_)}`. This matching loop is represented in figure 3.30 by having a node for the blamed term `sum(f(_),`

Inst.	Matched term	Binding	Relevant yield terms
Q_2^0	`sum(f(z),z)`	`x=z`	`sum(f(z),inc(z))`
Q_1^0	`f(z)`	`x=z`	`f(z)=f(inc(z))`
Q_2^1	`sum(f(inc(z)),inc(z))`	`x=inc(z)`	`sum(f(inc(z)),inc`^{2}`(z))`
Q_1^1	`f(inc(z))`	`x=inc(z)`	`f(inc(z))=f(inc`^{2}`(z))`
Q_2^2	`sum(f(inc`^{2}`(z)),inc`^{2}`(z))`	`x=inc`^{2}`(z)`	`sum(f(inc`^{2}`(z)),inc`^{3}`(z))`
Q_1^2	`f(inc`^{2}`(z))`	`x=inc`^{2}`(z)`	`f(inc`^{2}`(z))=f(inc`^{3}`(z))`

(a) Q_2^i represents the i^{th} instantiation of Q_2 and Q_1^i represents the i^{th} instantiation of Q_1.



(b) Instantiation graph: The edges with filled arrowheads represent blame-term dependencies. The edges with empty arrowheads represent equality dependencies.

Figure 3.29: Summary of considered instantiations.

`inc(_))` with edges to the equality and then to the pattern indicating that the blamed term is rewritten using the equality into a term that matches against Q_2's pattern. Furthermore, we have an edge labeled with Q_2 indicating that the rewritten term can again be instantiated to yield a term with structure `sum(f(_),inc(_))` hence closing the loop.

We call this kind of graphical representation of the terms involved in a matching loop a *matching loop graph*. It is an attempt to explain how the different terms are involved in a matching loop.

AP1 offered a feature to automatically select a path through the instantiation graph that, due to its length represents a likely matching loop and to abstract the instantiations on such a path to a string. Such a string representing the path is then analyzed by searching for a repeated substring to find the matching loop.

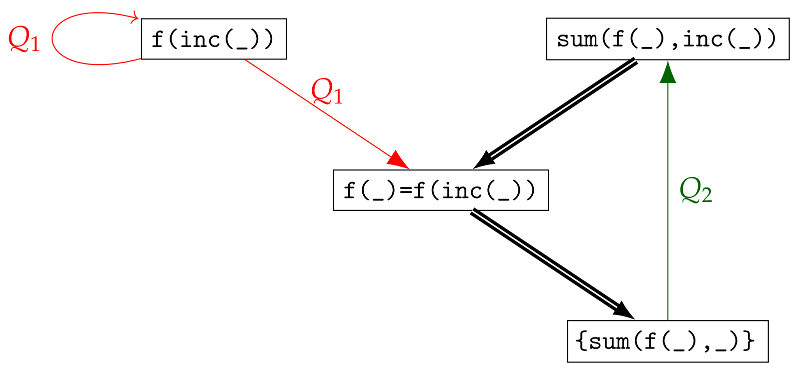


Figure 3.30: Example for a matching loop graph corresponding to the matching loop in figure 3.29b.

We have found that this is a somewhat restrictive approach to finding matching loops. It would be somewhat misleading to call the long path of instantiations of Q_2 in the previous example a matching loop as the repeated instantiations can only be sustained with the equalities generated by the instantiations of Q_1.

Our approach will be to implement an automated search for matching loop candidates by looking for long chains of repeated instantiations of the same quantifier and then automatically generate a matching loop graph such that the user can analyze these matching loop candidates to determine whether they indeed constitute a matching loop or not. The automated search is described in section 3.3.1 and the algorithm for generating the matching loop graph is described in section 3.3.2.

3.3.1 Matching Loop Search

Given an instantiation graph, we aim to find subgraphs that constitute potential matching loops. The key observation we will use to this end is that in a matching loop, we have instantiations of a quantifier that directly or indirectly cause an instantiation of the same quantifier. This leads to long chains of instantiations of the same quantifier. We have seen an example of a matching loop in figure 3.29.

Our approach works as follows:

1. For each quantifier Q_i
    - Filter out all nodes of the instantiation graph except those corresponding to an instantiation of Q_i. We will refer to this subgraph as $G[Q_i]$.
    - Compute the longest distances of all nodes with respect to the roots in $G[Q_i]$, i.e. the maximum depths.

   - Find the nodes without any children (as these represent end-nodes of potential matching loops) with maximum depth at least 2 (as we want to rule out short matching loops with fewer than three instantiations of Q_i). We will refer to this set of nodes as END_i.

2. Mark all nodes of the original instantiation graph as invisible.
3. For all nodes $v \in \bigcup_i \text{END}_i$
   - Mark all ancestors of v as visible.
4. Retain all visible nodes and reconnect the graph (see section 3.2.2). We will refer to this subgraph as *matching loop subgraph*.
5. Find the nodes without any children with respect to the matching loop subgraph. We will refer to this set of nodes as *matching loop end nodes*.
6. Compute the longest depths of the nodes with respect to the matching loop subgraph and sort the matching loop end nodes with respect to this maximum depth in descending order.

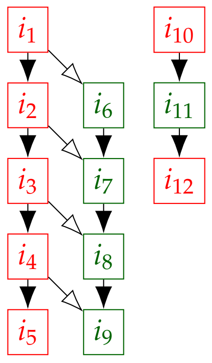
Figure 3.31 gives an example to illustrate this algorithm.

Note that in this approach, we are marking *all* the ancestors of the matching loop end nodes as visible. Therefore, the found matching loop candidates may contain some ”setup nodes” before the actual matching loop starts. We leave it up to the user to detect the repeating pattern and filter out any setup nodes using the filters described in section 3.2.3.

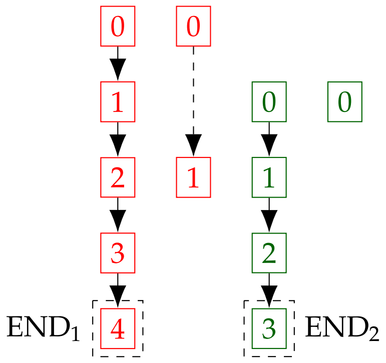
Implementation Details

The GUI of the axiom profiler has a feature called ”Search matching loops” (see figure 3.32a), which will execute the steps outlined above to find the potential matching loops. After that, the GUI allows the user to either click through all the found matching loops or display the entire matching loop subgraph by clicking on ”Show all matching loops” (see figure 3.32b).

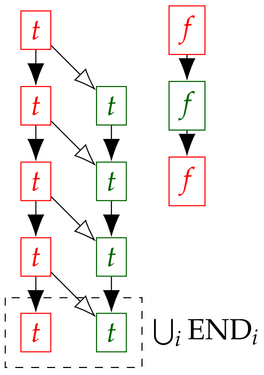
The data structure for the instantiation graph `InstGraph` has a field `matching_loop_end_nodes` such that when the user wants to display the n^{th} longest matching loop, a function can be called which marks all nodes as invisible and then does a reverse DFS traversal starting from the node stored at the $(n-1)^{\text{th}}$ index of `InstGraph::matching_loop_end_nodes`. This works because, during the matching loop search, we sort the nodes in `matching_loop_end_nodes` in descending order of the maximum depth with respect to the matching loop subgraph (see steps 4-6 in section 3.3.1). After displaying the entire matching loop subgraph using ”Show all matching loops”, the user might want to analyze a specific matching loop candidate ending in some node. To this end, we have implemented a feature that allows the user to select a specific end node and then click ”Analyze matching loop



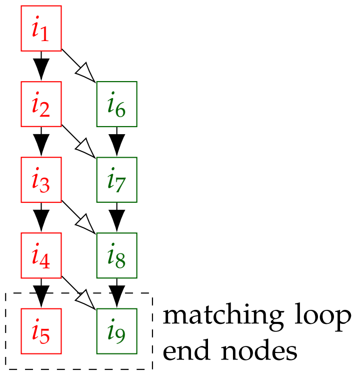
(a) Original instantiation graph G. We will refer to the red nodes as instantiations of Q_1 and to the green nodes as instantiations of Q_2.



(b) $G[Q_1]$ on the left and $G[Q_2]$ on the right. The nodes are labeled with their maximum depth with respect to the roots. Note that $\text{END}_1 = \{i_5\}$ and $\text{END}_2 = \{i_9\}$ as both i_5 and i_9 have no children and have maximum depth at least 2.



(c) Instantiation graph from figure 3.31a after marking the ancestors of the nodes in END_1 and END_2 as visible. Note that t means the node is marked as visible, and f means the node is marked as invisible.



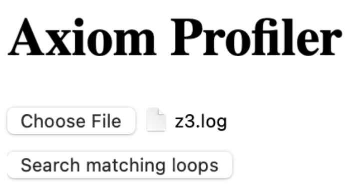
(d) Matching loop subgraph after filtering out the nodes marked as invisible in figure 3.31c.

Figure 3.31: Example to illustrate the matching loop search algorithm.

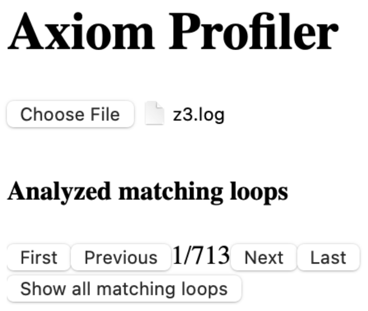
with end node being the last selected node" to only display the potential matching loop that ends in the selected node and generate the matching loop graph associated with it.

3.3.2 Construction of Matching Loop Graphs

This section will cover how to construct the matching loop graph given a subgraph of the instantiation graph that represents a potential matching loop. First, we show how to automatically generate the matching loop graph in figure 3.30 from the matching loop in 3.29b. The key idea here is to generalize the instantiations and the dependencies between them. Instantiations of the



(a) GUI for searching matching loops in the instantiation graph.



(b) GUI for clicking through the found potential matching loops one by one and for showing all matching loops at once (displayed after clicking on "Search matching loops").

Figure 3.32: GUI for searching and displaying potential matching loops.

```
pub struct InstGraph {
  // other fields omitted
  matching_loop_end_nodes: Vec<NodeIndex>,
  // other fields omitted
}
```

Figure 3.33: Data structures used for recording information about instantiations and their corresponding matches.

same quantifier that use the same trigger are reduced into a single *abstract instantiation*. Dependencies (A, B) and (C, D) are reduced if and only if A and C are reduced to the same abstract instantiation and B and D are reduced to the same abstract instantiation, and both have the same dependency type, i.e. either both are blame-term dependencies, or both are equality dependencies. This way, we obtain an intermediate representation, which we will call *abstract instantiation graph*. The abstract instantiation graph of the instantiation graph in 3.29b is depicted in figure 3.34.

From this intermediate representation, we can construct the matching loop graph. The idea is to iterate over all abstract instantiations and generate nodes from the abstract blame and yield terms. In the case of abstract instantiation Q_1 in figure 3.34 the only abstract blame term is `f(inc(_))` (as it is the term associated with the only incoming blame dependency), and the abstract yield terms are `f(inc(_))` and `f(_)=f(inc(_))` as they correspond to the terms associated with outgoing edges. After processing the abstract instantiation Q_1 we obtain the matching loop graph in figure 3.35a.

Processing abstract instantiation Q_2 in figure 3.34 is different because there is an incoming equality dependency indicating that some equalities are involved

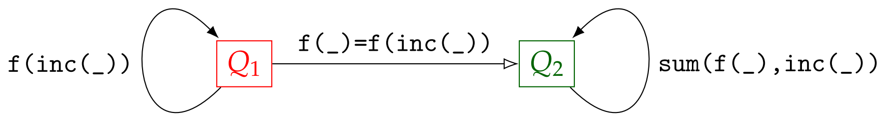
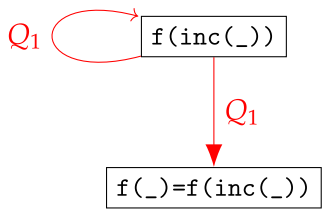
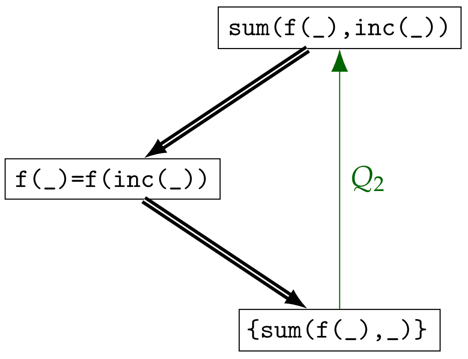


Figure 3.34: Abstract instantiation graph constructed from the instantiation graph in figure 3.29b. Note how the dependency types are still preserved. The dependencies with filled arrowheads represent blame-term dependencies and those with empty arrowheads represent equality dependencies, just as in the normal instantiation graph.

in rewriting the blame terms into a term that matches with Q_2's trigger, allowing it to be instantiated. Therefore, in case an abstract instantiation has incoming equality edges, we create a node for the abstract pattern `{sum (f(_),_)}` and add edges from the blame terms to the incoming equalities and from the incoming equalities to the abstract pattern. The nodes and edges created during the processing of this abstract instantiation are shown in figure 3.35b. After processing the abstract instantiation graph, we obtain a matching loop graph like the one depicted in figure 3.30.



(a) Created nodes and edges after processing Q_1 of the abstract instantiation graph in figure 3.34.



(b) Created nodes and edges after processing abstract instantiation Q_2 of the abstract instantiation graph in figure 3.34.

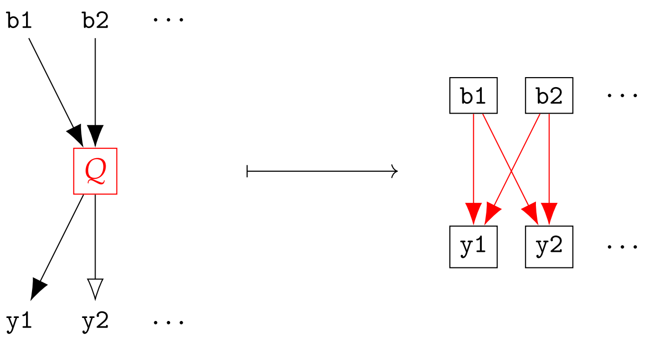
Figure 3.35: Intermediate states of matching loop graph during processing of abstract instantiation graph. Combining these gives us the graph in 3.30.

The general principle for computing matching loop graphs from an instantiation graph is as follows:

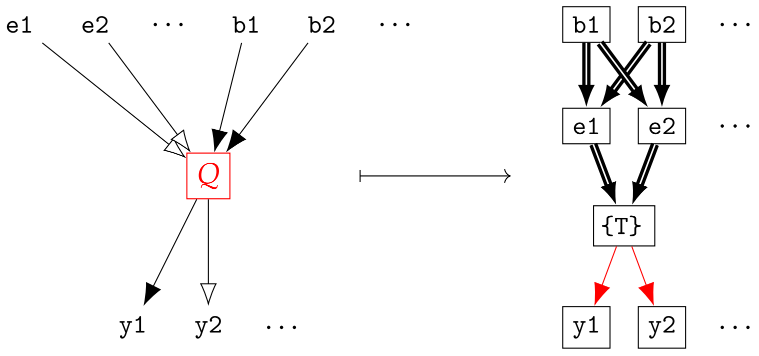
1. Given an instantiation graph (representing a potential matching loop), compute the abstract instantiation graph by reducing the instantiations and dependencies as follows:

- Two instantiations Q_1 and Q_2 are reduced to the same abstract instantiation if and only if they are instantiations of the same quantifier and use the same trigger.
- Two dependencies (A, B) and (C, D) are reduced to the same abstract dependency if and only if A and C are reduced to the same abstract instantiation and B and D are reduced to the same instantiation, and both have the same dependency type (blame-term or equality dependency).

2. Given the abstract instantiation graph from the previous step, compute the matching loop graph by processing each abstract instantiation as illustrated in figure 3.36.



(a) Illustration of how an abstract instantiation Q *without* incoming abstract equality dependencies is processed to generate the matching loop graph.

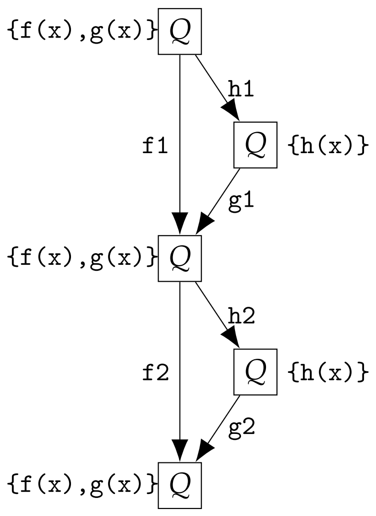


(b) Illustration of how an abstract instantiation Q *with* incoming abstract equality dependencies e1, e2, ... is processed to generate the matching loop graph.

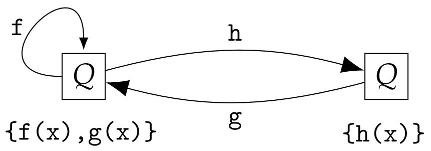
Figure 3.36: Illustration of how abstract instantiations are processed in the construction of the matching loop graph. The left graphs represent a node in the abstract instantiation graph, and the right graphs represent the nodes and edges generated when processing said abstract instantiation.

Quantifiers with multiple triggers

Note that the reason an abstract instantiation is defined by the quantifier *and* the used pattern is because quantifiers can have multiple, alternative triggers. For instance, suppose we have a matching loop as shown in figure 3.37a where quantifier Q is repeatedly instantiated but with two alternative triggers `{f(x),g(x)}` and `{h(x)}`. If we chose to reduce all these instantiations into a single abstract instantiation we would generalize all the blame terms `fi`, `gi`, and `hi` into the generalized term `_` as it is the common term structure of `f(_)`, `g(_)`, and `h(_)`. This example shows that defining abstract instantiations only via the instantiated quantifier does not make sense and that we should also define it via the used trigger. Figure 3.37b illustrates how the abstract instantiation graph looks if we define the abstract instantiation in this way.



(a) Example of instantiation graph with repeated instantiations of Q that use different triggers.



(b) Abstract instantiation graph corresponding to the instantiation graph in figure 3.37a. Note that `f`, `g`, `h` are obtained by generalizing `f1` with `f2`, `g1` with `g2`, and `h1` with `h2`, respectively.

Figure 3.37: Example illustrating why abstract instantiations are defined per quantifier *and* trigger as opposed to just per quantifier.

Limitations

Recall that during the construction of the abstract instantiation graph, dependencies (A, B) and (C, D) are reduced if and only if A and C are reduced to the same abstract instantiation and B and D are reduced to the same abstract instantiation, and both have the same dependency type, i.e. either both are blame-term dependencies, or both are equality dependencies.

One case where this is problematic is if we have *multiple dependencies of the same kind* between two abstract instantiations A and B. With our proposed algorithm, these dependencies would be reduced to the same abstract dependency, which does not make sense as they might have completely different term structures.

In this sense, this approach can only correctly handle a subset of all matching loops, namely those where there is either at most one blame-term dependency between any two abstract instantiations or at most one equality dependency between any two abstract instantiations, but it might be possible to deal with this case as well to obtain a complete solution.

Term Generalization

During the construction of the abstract instantiation graph, we need a way to generalize two terms t_1 and t_2 to extract the common term structure. We use a similar approach as in AP1. We use a basic recursive function, traversing the abstract syntax tree starting from the roots of both terms and constructing the generalized term. In case two nodes do not have the same meaning or kind, we replace the node with a *generalized primitive term*, which represents a 'wild card' (see _ in figure 3.39).

```
fn generalize(t1: TermIdx, t2: TermIdx) -> TermIdx {
    if t1 == t2 {
        t1
    } else if t1 is a generalized primitive term {
        t1
    } else if t2 is a generalized primitive term {
        t2
    } else t1 and t2 have same meaning {
        generalize the children of t1 and t2
        create a synthetic term with these generalized
        children as its children
        return this synthetic term
    } else {
        create generalized primitive term and return it
    }
}
```

Figure 3.38: Pseudocode for generalizing two terms to extract the common term structure.

3.4 Improving Equality Dependency Representation

In section 3.1, we saw how we can represent equality dependencies between instantiations. We will now consider a slightly modified version of the SMT

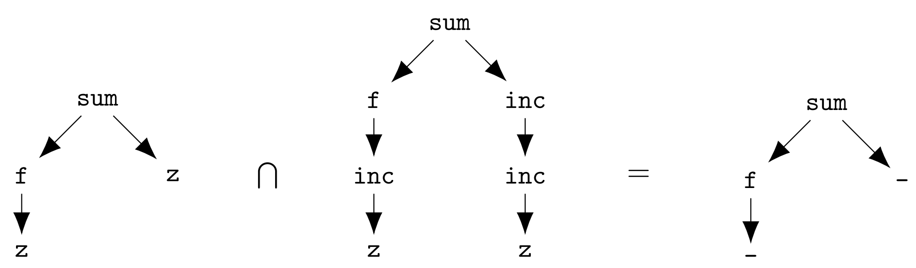


Figure 3.39: Exampe illustrating the effect of generalizing term sum(f(z),z) with sum(f(inc(z)), inc(inc(z)))) yielding the synthetic term sum(f(_),_).

problem in figure 3.8 where we replace the equation sum(f(x),inc(x))=inc(sum(x,x)) in quantifier Q_2 by sum(f(z),inc(x))=inc(sum(x,x)) such that we get the encoding shown in figure 3.40.

```
(assert ∀x:Number {f(x)} f(x) = f(inc(x))) ; Q1
(assert ∀x:Number {sum(f(x),x)}
     sum(f(z),inc(x)) = inc(sum(x,x))) ; Q2

(assert sum(f(z),z) = z)
```

Figure 3.40: The SMT-encoding of the same problem as in figure 3.8 but we have changed the left-hand side of the equation in Q_2 (see appendix C for syntactically correct encoding).

If we use the method described in this project to construct the instantiation graph, we will obtain the instantiation graph depicted in figure 3.41.

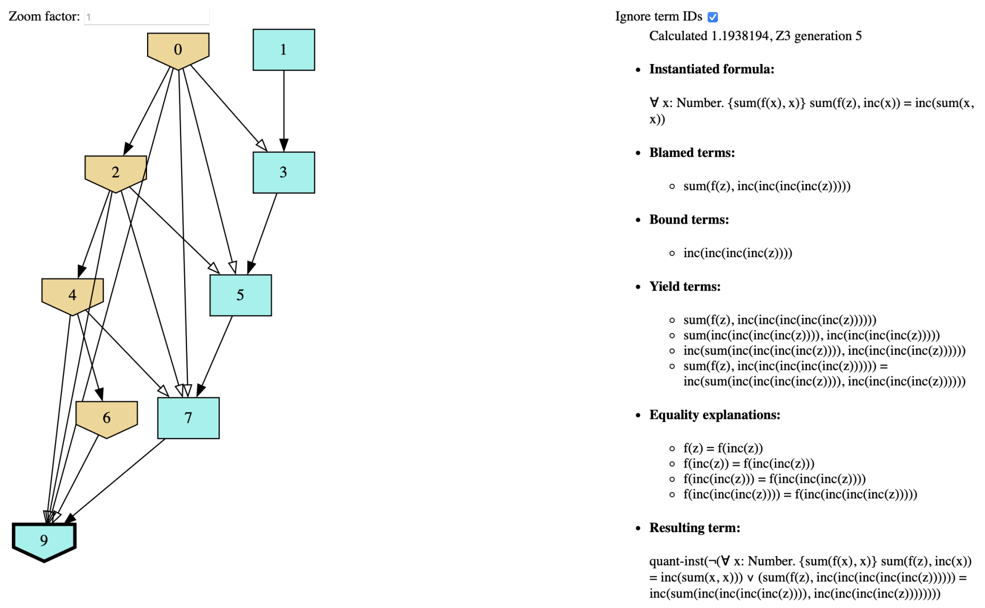
Observe that node 9 (which is an instantiation of Q_2) blames the term sum(f(z),inc^4(z)) and the equalities

- f(z)=f(inc(z)) created by node 0,
- f(inc(z))=f(inc^2(z)) created by node 2,
- f(inc^2(z))=f(inc^3(z)) created by node 4,
- and f(inc^3(z))=f(inc^4(z)) created by node 6.

This is because Q_2's pattern is {sum(f(x),x)} and therefore these equalities are required to rewrite the blame term sum(f(z),inc^4(z)) into sum(f(inc^4(z)),inc^4(z)) which matches against the pattern by binding x to inc^4(z), allowing Q_2 to be instantiated.

However, if we inspect the corresponding Z3 log file, we will see that the [new-match]-line corresponding to instantiation 9 only blames a single equality (#33 #524) which corresponds to f(z)=f(inc^4(z)) (see figure 3.42).

The reason why node 9 blames so many equalities in the instantiation graph but only a single equality in the log file can be understood by studying the



Ignore term IDs ✓

Calculated 1.1938194, Z3 generation 5

- **Instantiated formula:**

  ∀ x: Number. {sum(f(x), x)} sum(f(z), inc(x)) = inc(sum(x, x))

- **Blamed terms:**
  - sum(f(z), inc(inc(inc(inc(z)))))
- **Bound terms:**
  - inc(inc(inc(inc(z))))
- **Yield terms:**
  - sum(f(z), inc(inc(inc(inc(inc(z))))))
  - sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z)))))
  - inc(sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z))))))
  - sum(f(z), inc(inc(inc(inc(inc(z)))))) = inc(sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z))))))
- **Equality explanations:**
  - f(z) = f(inc(z))
  - f(inc(z)) = f(inc(inc(z)))
  - f(inc(inc(z))) = f(inc(inc(inc(z))))
  - f(inc(inc(inc(z)))) = f(inc(inc(inc(inc(z)))))
- **Resulting term:**

  quant-inst(¬(∀ x: Number. {sum(f(x), x)} sum(f(z), inc(x)) = inc(sum(x, x))) ∨ (sum(f(z), inc(inc(inc(inc(inc(z)))))) = inc(sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z))))))))

Figure 3.41: Instantiation graph corresponding to the log generated by Z3 when running the problem in 3.40. The right panel shows the information associated with the selected node 9.

```
[new-match] 0x134243fd0 #40 #39 #534 ; #539 (#33 #524)
```

Figure 3.42: The `[new-match]`-line corresponding to node 9 in figure 3.41.

state of the e-graph when the parser processes the `[new-match]`-line of the instantiation corresponding to node 9. The state of the e-graph is depicted in figure 3.43.

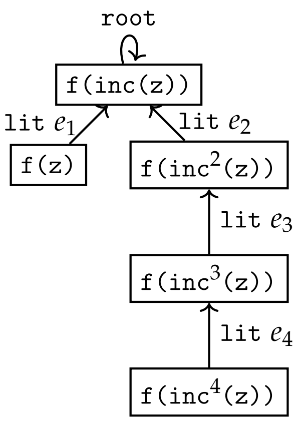


Figure 3.43: The parser's internal representation of the equivalence class with root `f(inc(z))` when parsing the `[new-match]` in figure 3.42.

As discussed in section 3.1, the parser will find the path from the node corresponding to `f(z)` to the node corresponding to `f(inc`^{4}`(z))` in the e-graph and blame all the instantiations that created the equality terms on that path, i.e., here e_1, e_2 e_3, and e_4 which were created by instantiations 0, 2, 4, and 6, respectively.

Ideally, we want node 9 in the instantiation graph to only have a single incoming equality edge as this reflects the fact that in the Z3 log the corresponding `[new-match]`-line only blames the equality `f(z)=f(inc`^{4}`(z))`. One approach is to add *equality nodes* into the instantiation graph, which represent equalities that are blamed or created by instantiations like in figure 3.44.

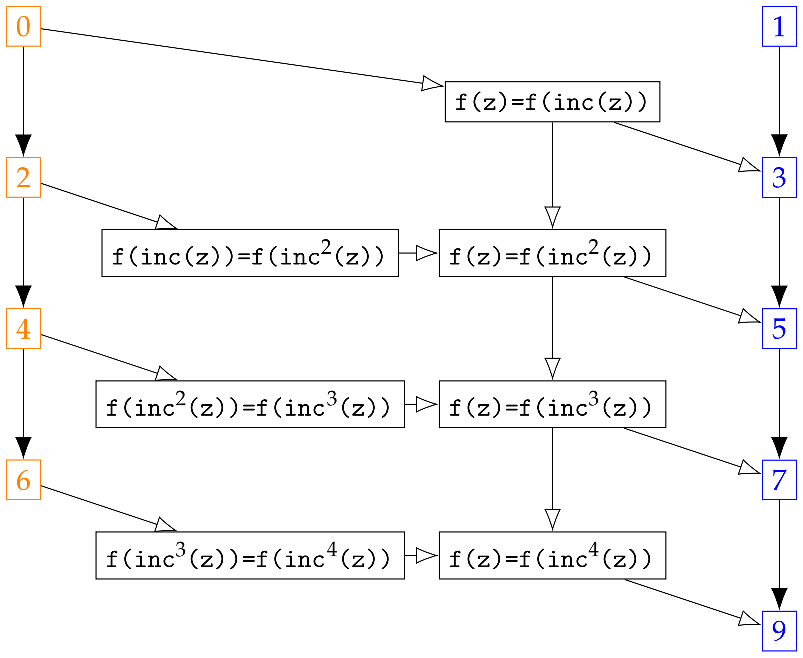


Figure 3.44: Instantiation graph corresponding to the one in figure 3.41 but augmented with equality nodes.

Implementing Equality Nodes

The simplest equality dependency between two instantiations A and B is if an equality generated by A is directly used to rewrite the blame term of B. This is the case for instantiations 0 and 3 in figure 3.44. A simple solution to deal with such equality dependencies is to store for each instantiation which equalities it yields and which equalities it blames, as shown in figure 3.45. When generating the instantiation graph, we can create nodes for the yield and blame equalities, adding edges to and from the instantiations, respectively. For instantiation 3 in figure 3.44, the `[new-match]` and the relevant `[eq-expl]`-lines are depicted in figure 3.46.

```
pub struct Match {
  // other fields omitted
  pub blamed_eqs: Vec<NodeEquality>,
}

pub struct Instantiation {
  // other fields omitted
  pub yields_equalities: Vec<(ENodeIdx, ENodeIdx)>,
}
```

Figure 3.45: Data structures used for recording information about blamed and generated equalities.

```
[eq-expl] #31 lit #461 ; #460
[eq-expl] #460 root
[new-match] 0x1541b2a68 #38 #37 #459 ; #464 (#31 #460)
```

Figure 3.46: The `[new-match]` and corresponding `[eq-expl]`-lines of instantiation 3 in figure 3.44. The tuple `(#31 #460)` represents the equality `f(z)=f(inc(z))`.

When the parser processes `[eq-expl] #31 lit #461 ; #460` it knows that the equality between term `#31` and `#460` is at some point involved in an equality explanation and hence stores this equality in the `yields_equalities`-field of the `Instantiation` corresponding to the instantiation that created the equality term `#461` (in our example instantiation 0). Recall that the parser has a data structure which stores for each e-node, which instantiation created it (see figure 3.7). When the parser processes the `[new-match]`-line in figure 3.46 it can just store in the `blamed_eqs`-field of the `Match` representing the match of instantiation 3 that it blames the equality between term `#31` and `#460`. During the construction of the instantiation graph, when processing instantiation 0, we can add an edge from 0 to the equality node representing `#31 = #460`. When we process instantiation 3, we can add an edge from said equality node to node 3. If we proceed as described, we will end up with an instantiation graph as in figure 3.47.

Synthetic Equalities

Consider again instantiation 9 in figure 3.47 that we already covered at the beginning of section 3.4. Note that when the parser processes the `[new-match]` of instantiation 9, it has already processed the `[new-match]` of instantiation 7 which blames the equality `f(z)=f(inc`^{3}`(z))` and hence this equality has already been explained, and it would be redundant to explain it again when explaining `f(z)=f(inc`^{4}`(z))`. Therefore, one idea is to keep track of *synthetic equalities* that represent equalities that have already been explained before.

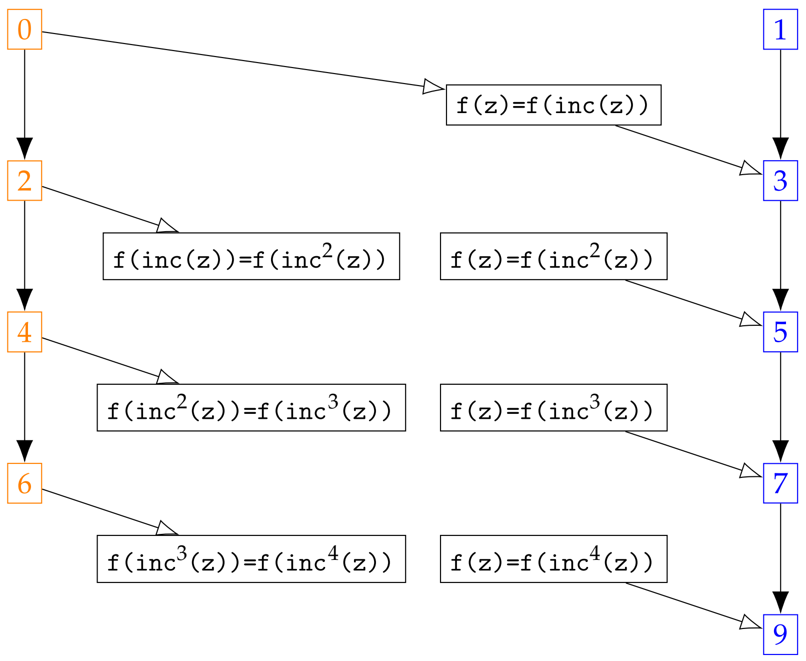


Figure 3.47: Instantiation graph corresponding to the one in figure 3.41 but augmented with equality nodes. Note how there is still missing information compared to the complete instantiation graph with equality nodes in figure 3.44.

This way, when we explain an equality, these can be taken into account to avoid blaming redundant instantiations.

More concretely, the parser will keep track of synthetic equalities in an *equality graph* which is similar to the e-graph but where the edges are undirected. The synthetic equalities are added to this equality graph after the parser has processed a `[new-match]` which blames equalities. When explaining an equality such as `f(z)=f(inc`^{4}`(z))` the parser will, just as before, find the path from `f(z)` to `f(inc`^{4}`(z))` in the e-graph and add the edges along that path to the equality graph. When the parser processes the `[new-match]` of instantiation 9 (see figure 3.42), the parser would construct the equality graph shown in figure 3.48. Note that for the equality `f(z)=f(inc(z))` there is already a synthetic equality in the equality graph, and therefore, we do not add another edge between the two nodes. Then, the parser can find the shortest path `f(z)` to `f(inc`^{4}`(z))` in the equality graph using any standard algorithm (in our case Dijkstra) and store the blamed equalities on that path. In our example, the shortest path would only involve the synthetic equality `f(z)=f(inc`^{3}`(z))` and the equality `f(inc`^{3}`(z))=f(inc`^{4}`(z))` due to the equality term e_4.

As the structure of the e-graph might change during the solver run, the non-synthetic edges that were added to the equality graph during an equality explanation are removed after processing a [new-match]-line such that the equality graph always reflects the current structure of the e-graph. This way, the synthetic equality edges added to the equality graph do not need to be removed. Each time an equality $a = b$ is explained, the parser will first add the path from a to b *in the e-graph* to the equality graph and then find the shortest path from a to b *in the equality graph*. Therefore, the shortest path from a to b in the equality graph will consist of edges that are also present in the e-graph or of synthetic edges.

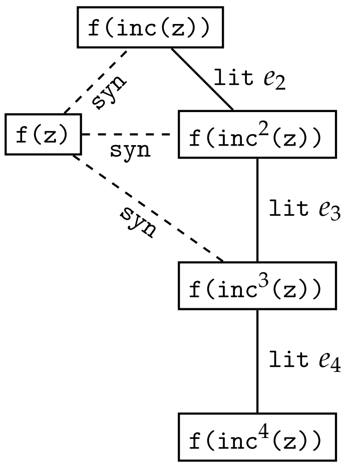


Figure 3.48: Equality graph corresponding to the e-graph in figure 3.43 but with synthetic equalities representing equalities that have been previously explained. Note that these synthetic equalities correspond to the equalities explained when processing the [new-match] of instantiations 3, 5, and 7 in figure 3.47.

Data Structure for Equality Explanations

The shortest path from `f(z)` to `f(inc`^{4}`(z))` only involved `lit` equalities and synthetic equalities `syn`. But as we saw in section 3.1, we can also have equalities explained by congruences.

Consider the equality `a = d` in figure 3.50a. The shortest path from `a` to `d` in the equaltiy graph in figure 3.50b is a direct edge of type `cg (b1 b2) (c1 c2)`. Whenever the shortest path contains such a `cg`-equality, we will recursively explain the equalities between the arguments and construct an *equality explanation tree* which is a recursively defined data structure shown in figure 3.49.

Figure 3.50 shows an example of how the equality between `a` and `d` is converted into an equality explanation tree. Since not all possibilities for quantifier instantiations logged with [new-match] are actually instantiated, we have a separate data structure in the parser that stores all these equality explanations created while processing the [new-match]-lines in the log. To

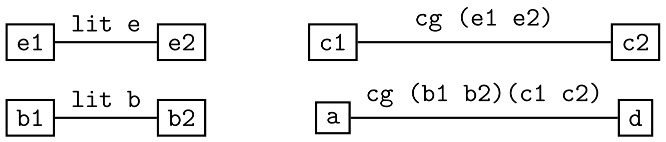
```
pub struct LeafEquality(pub ENodeIdx, pub ENodeIdx);

pub enum NodeEquality {
  Leaf(LeafEquality),
  Node(ENodeIdx, ENodeIdx, Vec<NodeEquality>),
}
```

Figure 3.49: Data structure used for representing equality explanations. Equality explanations of type cg are mapped to `NodeEquality::Node`, where the inner vector of equalities explains the equalities between the arguments. All other equalities are mapped to `NodeEquality::Leaf`.

make sure all the relevant equalities can be explained, we recursively process all these `NodeEquality` creating equality nodes for them in the instantiation graph before adding nodes for the instantiations that blame these equality nodes (see figure 3.50).

```
[eq-expl] e1 lit e ; e2
[eq-expl] c1 cg (e1 e2) ; c2
[eq-expl] b1 lit b ; b2
[eq-expl] a cg (b1 b2) (c1 c2) ; d
[new-match] ... ; (a d)
```

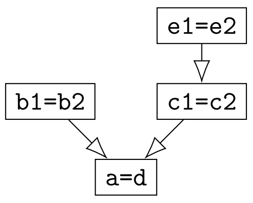
(a) Example of equality explanation involving cg.



(b) State of the equality graph when processing the `[new-match]` in figure 3.50a.

```
Node(a, d, [
  Leaf(LeafEquality(b1, b2)),
  Node(c1, c2, [
    LeafEquality(e1, e2)
  ]),
])
```

(c) Computed `NodeEquality` after processing the equality `a = d` in the `[new-match]` of figure 3.50a.



(d) Visual representation of the equality explanation tree in figure 3.50c.

Figure 3.50: Example illustrating how equality explanation trees are constructed from equality explanations in the Z3 log.

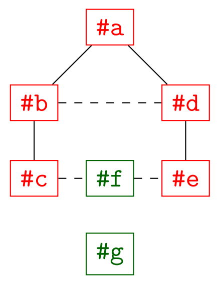


Figure 3.51: Example of an equality graph after adding the path from `#c` to `#e` from the e-graph. Nodes of the same color indicate that they belong to the same equivalence class as defined by the root of the equivalence class.

Dealing with Invalidated Synthetic Equalities

Note that due to the updates to the e-graph, the equality explanation that explains a synthetic equality might eventually become invalid. Blaming such an invalid synthetic equality would thus be misleading in the instantiation graph. For instance, we might encounter a situation where `#a`, `#b`, `...`, `#e` are part of the same equivalence class and `#f`, `#g` are part of another equivalence class. But at some earlier point during the solver run, Z3 might have recorded that `#c`, `#e`, and `#f` belonged to the same equivalence class, which is no longer valid. If we did not remove any synthetic equalities from the equality graph, the synthetic equalities `#c=#f` and `#e=#f` would still be in the equality graph. This situation is illustrated in 3.51. Note that nodes of the same color correspond to terms that belong to the same current equivalence class, and the dashed edges represent synthetic equalities.

In such a situation, finding the shortest path from `#c` to `#e` would erroneously yield the path `[#c, #f, #e]` even though the equalities `#c=#f` and `#f=#e` are no longer valid. To solve this problem, we can store in each node of the equality graph the root of the most recent equivalence class that it belongs to and only find the shortest path in the equality graph with respect to the nodes that have the same equivalence class root. This information is updated whenever we add the path from the e-graph explaining the equality between two terms into the equality graph.

Pruning Filter for Equality Nodes

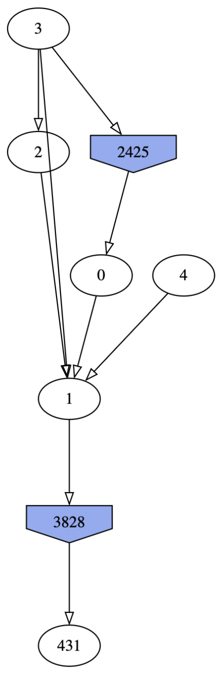
For convenience, a *pruning filter* was implemented such that we can only retain those equality nodes that have both a visible ancestor and a visible descendant that corresponds to an instantiation node. To this end, each node in the instantiation graph has two fields `has_inst_ancestor: bool` and `has_inst_descendant: bool` which are set to true if a node has a visible ancestor, which is an instantiation, and if a node has a visible descendant which is an instantiation, respectively. We do two passes through the currently visible nodes of the graph, once in topological order and once in reverse topological order. During the traversal in topological order, we perform the following operations on each node:

- If an instantiation node is encountered, `has_inst_ancestor` is set to true since the node itself is an instantiation and hence its ancestor.
- If an equality node is encountered, `has_inst_ancestor` is set to true if any of its parents has it set.

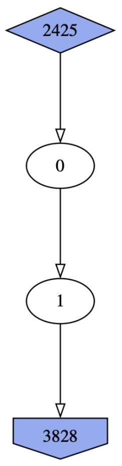
During the traversal in reverse topological order, we perform the following operations on each node:

- If an instantiation node is encountered, `has_inst_descendant` is set to true since the node itself is an instantiation and hence its descendant.
- If an equality node is encountered, `has_inst_descendant` is set to true if any of its children has it set.

After both passes, the `visible`-fields of only those nodes are set to true where both fields, `has_inst_descendant` and `has_inst_ancestor`, are set. Figure 3.53 shows the same graph as in figure 3.41 but with pruned equality nodes.

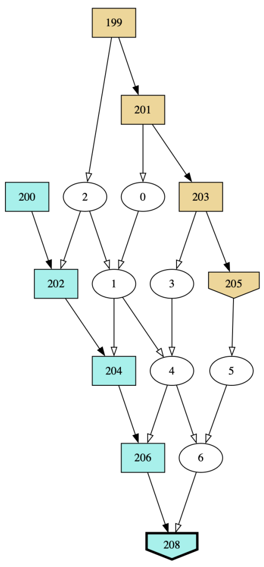


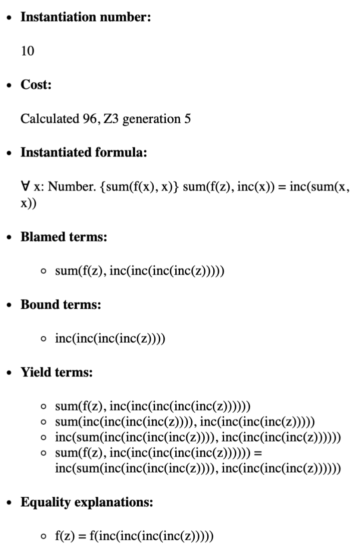
(a) Instantiation graph before applying pruning filter.



(b) Instantiation graph after applying pruning filter.

Figure 3.52: Note how the pruning filter only keeps the equality nodes 0 and 1 since they are the only nodes that have a descendant *and* an ancestor corresponding to an instantiation node as opposed to an equality node.





- **Instantiation number:**

  10

- **Cost:**

  Calculated 96, Z3 generation 5

- **Instantiated formula:**

  ∀ x: Number. {sum(f(x), x)} sum(f(z), inc(x)) = inc(sum(x, x))

- **Blamed terms:**

  - sum(f(z), inc(inc(inc(inc(z)))))

- **Bound terms:**

  - inc(inc(inc(inc(z))))

- **Yield terms:**

  - sum(f(z), inc(inc(inc(inc(inc(z))))))
  - sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z)))))
  - inc(sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z))))))
  - sum(f(z), inc(inc(inc(inc(inc(z)))))) = inc(sum(inc(inc(inc(inc(z)))), inc(inc(inc(inc(z))))))

- **Equality explanations:**

  - f(z) = f(inc(inc(inc(inc(z)))))

Figure 3.53: Instantiation graph corresponding to the log generated by Z3 when running the problem in 3.40 but with equality nodes. The right panel shows the information associated with the selected node 208 which corresponds to node 9 in figure 3.41. Note how node 208 only blames the equality `f(z)=f(inc`^{4}`(z))` whereas in figure 3.41 the corresponding node blames 4 equalities.

Chapter 4

Evaluation

In this chapter, we will qualitatively and quantitatively compare AP2 to AP1. In section 4.1 we will compare and evaluate the differences in the visual design of the instantiation graph in both tools. In section 4.2 we will evaluate the processing speed in AP2 and compare it with processings speeds in AP1. Finally, in section 4.3 we will compare our approach for analyzing matchings loops with the approach used in AP1.

The main questions that we want to address to evaluate our tool are the following:

- How fast does AP2 process logs in comparison to the AP1?
- How does the processing speed scale with log size?
- What are the performance bottlenecks in processing the log?
- How does using equality nodes impact the processing speed?
- Can the AP2 efficiently find potential matching loops?

4.1 Instantiation Graph Design

We can compare the instantiation graph design of AP1 with the design used in AP2. Figure 4.1a shows the same graph as in 4.2a but the former figure corresonds to AP2 and the latter to AP1. Likewise, 4.1b shows the same graph as in 4.2b.

Notice that in our tool, the node styles depend on whether or not there are filtered children, parents or both (see figure 3.20) whereas in AP1, all nodes have the same shape. For instance node 115 in figure 4.1a has a different shape than in 4.1b because in the latter it has a filtered-out parent. This difference is not evident in the original Axiom Profiler (see figure 4.2). Our intuitive node shapes help the user in identifying nodes to which the filters for displaying the children or parents are applicable.

Furthermore, we use different edge styles indicating whether they represent a blame-term dependency (filled arrowhead) or whether they represent an equality dependency (empty arrowhead). Unlike in AP1, in our tool the edges are clickable such that the user can directly understand how any two instantiations depend on each other. Furthermore, AP1 does not implement a reconnecting algorithm for representing indirect dependencies. Consider the example in figure 4.2. There is a path from the third lowest to the lowest blue node in figure 4.2a but this indirect dependency is not represented in the filtered graph in figure 4.2b. In our tool, this indirect dependency is represented via a dashed edge between nodes 70 and 115 as shown in figure 4.1b.

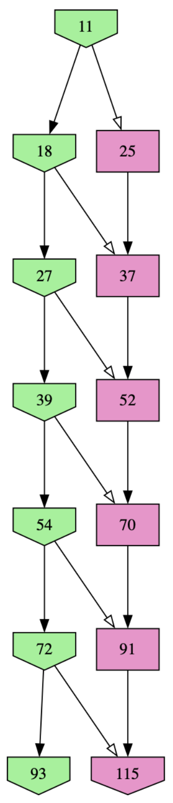
In our tool, the colors between nodes of different instantiations are chosen such that they are clearly visually distinguishable. This effect can be seen in figure 4.1 as the two nodes almost have complementary colors hence optimizing the contrast. Compare this to the colors used in AP1 shown in figure 4.2 where the contrast is not as stark.

Furthermore, the original Axiom Profiler does not implement equality nodes. This can be a problem in situations where there are many equality dependencies between nodes as discussed in section 3.4.

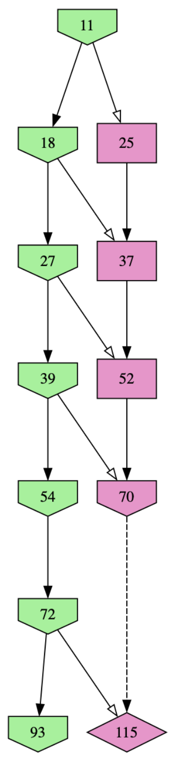
4.2 Performance Analysis

There are various processing stages that happen in the time between when a user selects a file until the instantiation graph filtered with the default filters is displayed:

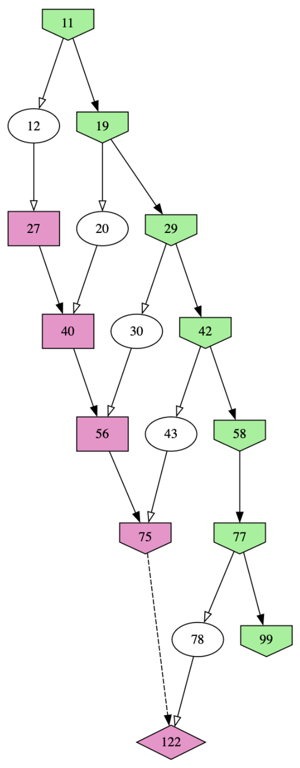
1. Parse the log file (see section 3.1 and 3.1 on page 14).
2. Construct the instantiation graph (see section 3.1).
   - This involves precomputing various data structures (transitive closure for reconnecting algorithm, number of parents and children for nodes for filtering operations, etc.).
3. Apply the default filters (see section 3.2.3).
   a) Filter out theory-solving instantiations.
   b) Filter out all but the 125 most expensive instantiations.
4. Remove the nodes marked as invisible and reconnect the nodes (see section 3.2.2).
5. Convert the filtered `petgraph` into dot format (see section 3.2).



(a) Instantiation graph design from Axiom Profiler 2.0. Figure 4.2a shows the same graph but in the original Axiom Profiler.



(b) The same instantiation graph as in figure 4.1a but with filtered out node 91. Figure 4.2b shows the same graph but in the original Axiom Profiler.



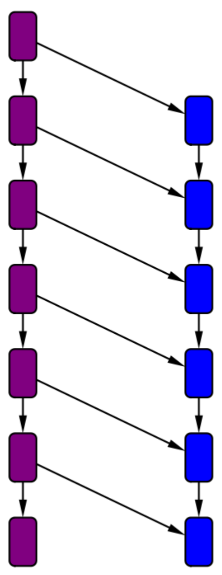
(c) The same instantiation graph as in figure 4.1b but with equality nodes. The original Axiom Profiler does not implement equality nodes.

Figure 4.1: Instantiation graph design in Axiom Profiler 2.0. The green nodes in these graphs correspond to the purple nodes in figure 4.2. Note that there is a dashed edge from node 70 to node 115 in 4.1b (corresponding to the dashed edge from 75 to 122 in figure 4.1c) as in figure 4.1a there is a path which directly connects them.

6. Convert the computed dot file into SVG format such that it can be displayed in the browser (see section 3.2).

An important criterion to evaluate our tool is to measure how long it takes to display an instantiation graph. For this, we conducted a detailed performance analysis measuring how long the previously listed processing stages take. Figure 4.3 and 4.4 show the results. Note that we conducted this analysis both with the code that uses equality nodes introduced in section 3.4 and with the code that does not use equality nodes. More details concerning reproducibility of our analysis can be found in appendix D.

In sequences-20, the total time to process the log (without equality nodes) from selecting the file until it is displayed is 45.78 seconds of which almost


(a) Instantiation graph in the original Axiom Profiler [6].

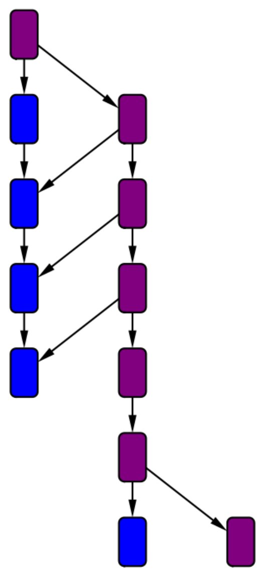
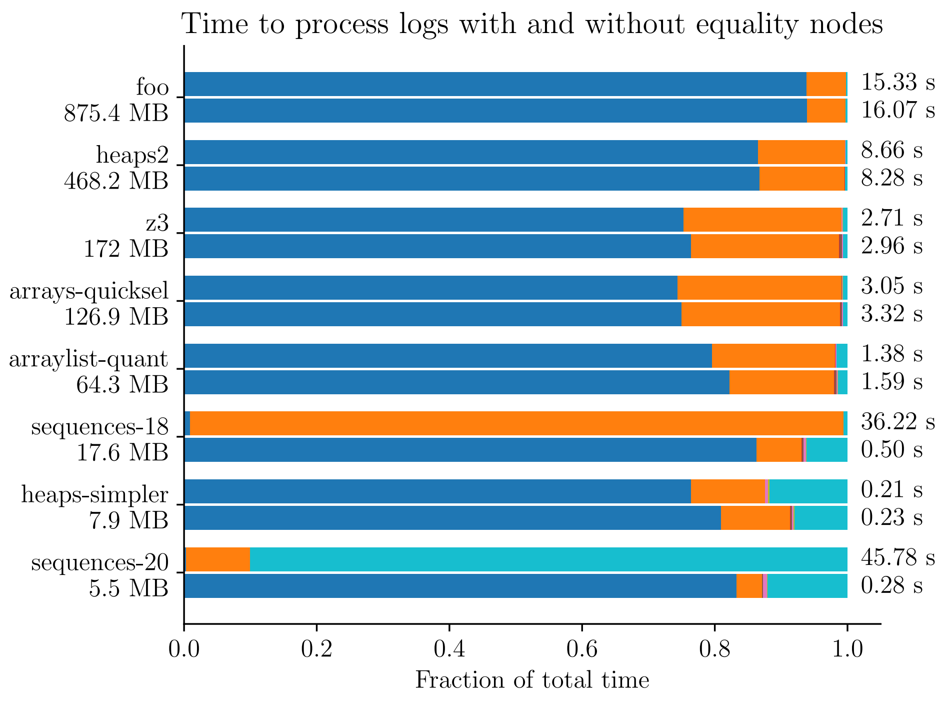

(b) Same instantiation graph as in figure 4.2a but where the second lowest blue instantiation has been filtered out.

Figure 4.2: Instantiation graph design in original Axiom Profiler. Note how the indirect dependency between the third lowest and lowest blue node is not represented.

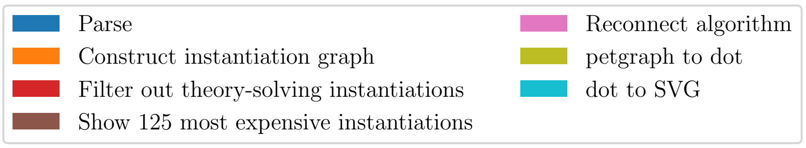
90% are spent converting the dot format into an SVG. In this step, Graphviz computes the Sugiyama layout for the graph specified in dot format. This step can take a long time if there are for instance many edges. To understand how this can happen we can study the example in figure 3.41. Each node which is an instantiation of Q_2 blames all previous instantiations of Q_1 and therefore the total number of edges grows quadratically in the total number of instantiations of Q_2. As explained in section 3.4 our improved representation with equality nodes ensures that instead of blaming all equalities that are found on the path to the root of the equivalence class we make more economic use of computed equality explanations by creating nodes for them that can be blamed (see figure 3.53). This reduction in total edge count can in some cases significantly reduce the time to process the log, in this case by 99.4% to 0.28 seconds.

Precisely because of such cases where a filtered graph can contain many nodes and edges hence slowing down the computation of the SVG of the filtered graph, we have implemented a warning prompt (see section 3.2.3) that gets displayed whenever the current filter chain leads to a graph with an excessive amount of edges and nodes. Importantly, this prompt gets displayed before the computation of the SVG to ensure responsiveness.

In sequences-18, the total time to process the log is 36.22 seconds of which



(a) The left labels indicate the name of the processed log along with its size. The right labels indicate the time it takes to process the log from selecting the log to displaying it to the user. Each log was once processed with the code that uses equality nodes and once with the code that does not. The upper number denotes the time without equality nodes and the lower number the time with equality nodes.


Parse
Construct instantiation graph
Filter out theory-solving instantiations
Show 125 most expensive instantiations
Reconnect algorithm
petgraph to dot
dot to SVG

(b) Legend for figure 4.3a.

Figure 4.3: Plots illustrating what fraction the various stages take when processing logs of various size. See appendix D for the raw data.

almost 99% are spent constructing the instantiation graph. After a more detailed analysis, we found that almost 99% of the time constructing the instantiation graph was spent computing the transitive closure as described in section 3.2.2. Using equality nodes, the total time to process the log decreased by 98.6%. A more detailed analysis of the structure of the corresponding instantiation graph reveals that there are many nodes with many children due to equality dependencies. As described in section 3.2.2 the computation of the transitive closure involves doing a bitwise OR of all children of each node.

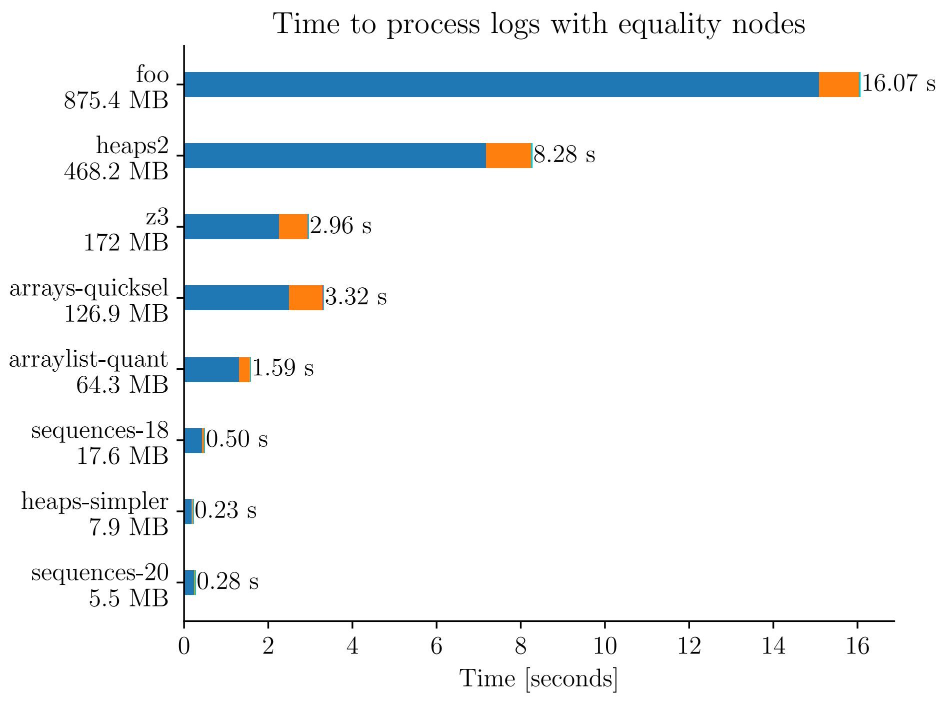


Figure 4.4: Comparison of processing times for logs of different size when using equality nodes. See appendix D for the raw data.

In cases such as this one, this can take a long time. Using equality nodes generally decreases the degrees of nodes as we reuse equality explanations. This significantly improves the structure of the graph hence speeding up the computation of the transitive closure. In this case, using equality nodes reduces the total time processing the log to 0.50 seconds.

Apart from the special cases sequences-18 and sequences-20, processing the log with equality nodes takes on average 7.4% longer than without equality nodes. Such an increase is expected because as discussed in section 3.4 we repeatedly use a shortest path algorithm to find the minimal number of nodes to blame.

In the common case, the total time to process the log is dominated by parsing, constructing the instantiation graph, and converting the dot file to SVG by Graphviz. Note that parsing a log and constructing the instantiation graph only have to be done once at the beginning. As parsing constitutes the largest fraction (see figure 4.3), the total time to process the log scales roughly linearly with log size. A more detailed analysis of the parsing speed can be found in figure 4.5.

In order to assess whether the processing times in AP2 are better than in

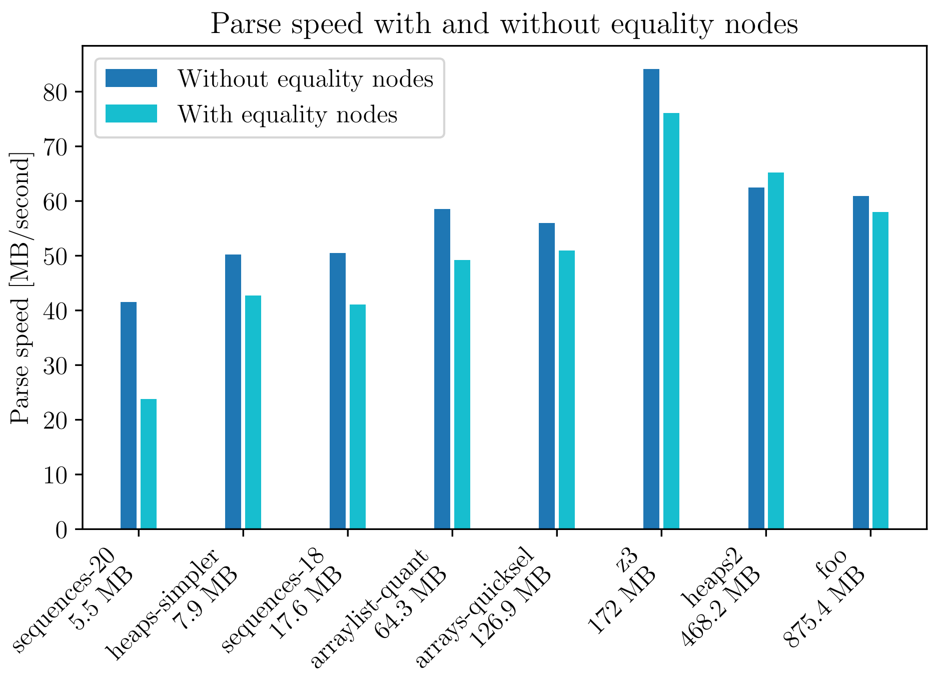


Figure 4.5: Computed parse speed with and without equality nodes. The parse speed decreased on average by 13.9% when using equality nodes. See appendix D for the raw data.

Name and size of log	AP2	AP1
running-example-fix-inj (55.3 MB)	1.3 s	10.1 s
running-example-fix-nxt (50.0 MB)	1.3 s	11.4 s
running-example-orig (50.9 MB)	1.3 s	12 s
z3 (172 MB)	4.3 s	crash after 105 s
file1 (16.1 MB)	2.2 s	crash after 22 s

Figure 4.6: Approximate times for opening log files on both the Axiom Profiler 2.0 (AP2) and on the previous Axiom Profiler (AP1) measured in seconds. For AP2 we used the Firefox browser v121.0. AP1 was compiled from source code (see [24]). All measurements in this figure were done on the same system: Ubuntu 22.04.3 LTS running on an Intel® Core™ i5-5250U CPU @ 1.60GHz $\times$ 4 with 4 GiB RAM.

AP1, we have also done approximate measurements by hand opening the same logs with AP2 and with AP1 both running on the same hardware and OS. The results are shown in figure 4.6. In some cases, AP1 crashed. In the cases where it did not crash processing the tool was roughly 10 times slower.

4.3 Matching Loop Analysis

AP1 does not provide a feature to search the entire instantiation graph for all matching loops. Instead, the authors devised a way to automatically select a *path* through the instantiation graph that represents a likely matching loop. Furthermore, by selecting a node the user could influence the choice of the path. To find a matching loop on the selected path, the path was reduced to a string such it can be analyzed to find a repeating substring representing the matching loop along with a path explanation as seen in figure 4.7a.

In AP2, we implemented an automatic search for the matching loops as discussed in section 3.3.1 and a way to reduce potential matching loops to matching loop graphs to better understand the terms involved in a matching loop as discussed in section 3.3.2.

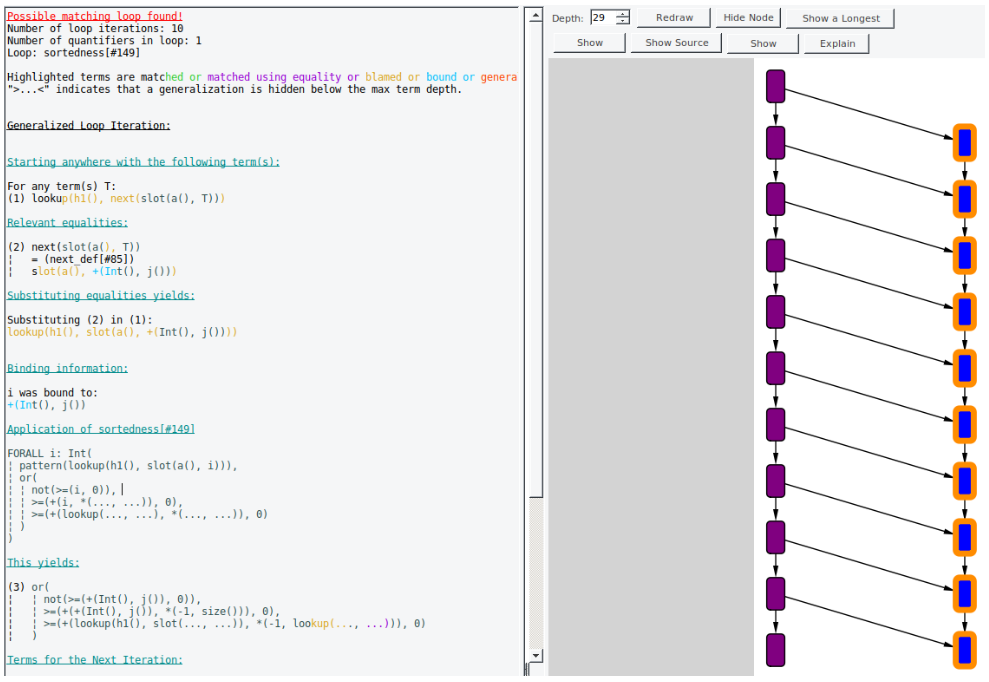
In the example of figure 4.7b, the generated matching loop graph can help the user understand that the instantiations of quantifier $q6$ repeatedly generate terms of the form `slot(a, _+j)` which again matches with $q6$'s trigger. This can give the user the idea that using a more restrictive trigger can avoid this matching loop.

Furthermore, we have implemented functionality that allows the user to view all found matching loops and manually select and anaylze suspicious matching loops (see section 3.3.1).

An important evaluation criterion in this context is how long it takes to search matching loops. The measurement results can be found in figure 4.8. More details for reproducibility can be found in appendix D. No clear dependency of matching loop search time on the log size was found as the search time in sequences-20 (5.5 MB) was measured to be 0.0594 seconds which is longer than in foo (875.4 MB) where the search took only 0.0177 seconds. This seems to indicate that the matching loop search time mainly depends on the graph structure rather than the overall size.

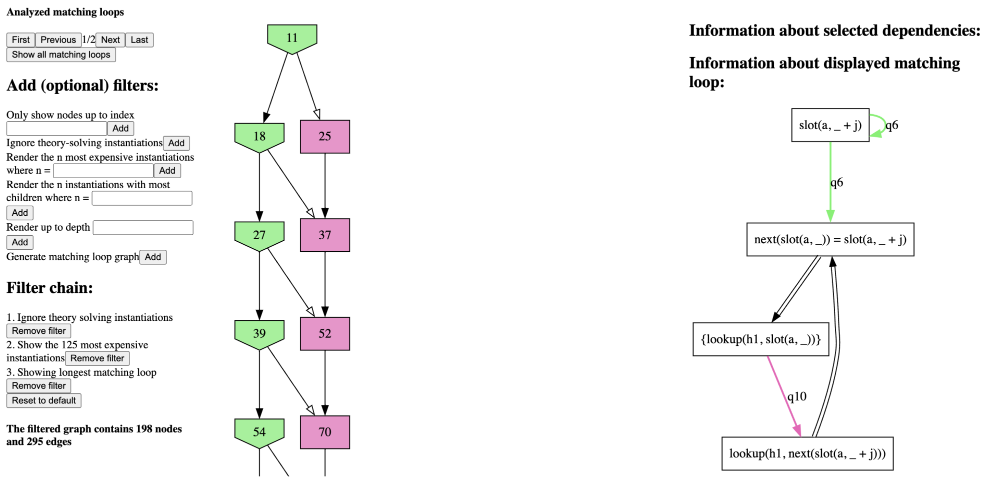
4.4 Summary

We have shown that processing times in AP2 are roughly ten times faster than in AP1. We have found that the time to process logs in AP2 scales roughly linearly with the log size. Parsing the log and constructing the instantiation graph were found to be the main performance bottlenecks. Fortunately, parsing and constructing the instantiation graph only occur once. After that, the main performance bottleneck was identified to be the conversion of the dot files to SVG done with the Graphviz library. This illustrates the significance of the warning prompt that is displayed after a user applies a filter before rendering in ensuring responsiveness.



```
Possible matching loop found!
Number of loop iterations: 10
Number of quantifiers in loop: 1
Loop: sortedness[#149]

Highlighted terms are matched or matched using equality or blamed or bound or genera
">...<" indicates that a generalization is hidden below the max term depth.


Generalized Loop Iteration:


Starting anywhere with the following term(s):

For any term(s) T:
(1) lookup(h1(), next(slot(a(), T)))

Relevant equalities:

(2) next(slot(a(), T))
¦   = (next_def[#85])
¦   slot(a(), +(Int(), j()))

Substituting equalities yields:

Substituting (2) in (1):
lookup(h1(), slot(a(), +(Int(), j())))


Binding information:

i was bound to:
+(Int(), j())

Application of sortedness[#149]

FORALL i: Int(
¦ pattern(lookup(h1(), slot(a(), i))),
¦ or(
¦ ¦ not(>=(i, 0)), |
¦ ¦ >=(+(i, *(..., ...)), 0),
¦ ¦ >=(+(lookup(..., ...), *(..., ...)), 0)
¦ )
)

This yields:

(3) or(
¦   ¦ not(>=(+(Int(), j()), 0)),
¦   ¦ >=(+(+(Int(), j()), *(-1, size())), 0),
¦   ¦ >=(+(lookup(h1(), slot(..., ...)), *(-1, lookup(..., ...))), 0)
¦   )

Terms for the Next Iteration:
```

(a) Matching loop explanation of previous Axiom Profiler [6]



Analyzed matching loops

First | Previous | 1/2 | Next | Last
Show all matching loops

Add (optional) filters:

Only show nodes up to index [ ] Add
Ignore theory-solving instantiations Add
Render the n most expensive instantiations where n = [ ] Add
Render the n instantiations with most children where n = [ ] Add
Render up to depth [ ] Add
Generate matching loop graph Add

Filter chain:

1. Ignore theory solving instantiations Remove filter
2. Show the 125 most expensive instantiations Remove filter
3. Showing longest matching loop Remove filter

Reset to default

The filtered graph contains 198 nodes and 295 edges

Information about selected dependencies:

Information about displayed matching loop:

11
18
25
27
37
39
52
54
70
slot(a, _ + j)
q6
q6
next(slot(a, _)) = slot(a, _ + j)
{lookup(h1, slot(a, _))}
q10
lookup(h1, next(slot(a, _ + j)))

(b) Matching loop explanation of Axiom Profiler 2.0

Figure 4.7: Comparison of the same matching loop in the previous and the current tool. The green nodes in figure 4.7b correspond to the purple nodes in figure 4.7a.

We have found that using equality nodes slows down parsing speed by roughly 13.9%. However, in some cases we have found that using equality nodes can reduce the total time to process a log by up to 99.4%. Therefore, using the equality nodes seems to be worth the slight reduction in parsing speed in the common case.

Name and size of log	Search time
sequences-20 (5.5 MB)	0.0594 s
heaps-simpler (7.9 MB)	0.0091 s
sequences-18 (17.6 MB)	0.2411 s
arraylist-quant (64.3 MB)	0.7678 s
arrays-quicksel (126.9 MB)	0.8983 s
z3 (172 MB)	0.4195 s
heaps2 (468.2 MB)	0.0589 s
foo (875.4 MB)	0.0176 s

Figure 4.8: Measured times for matching loop search for various log files. See appendix D for the raw data.

The algorithm for finding matching loops in AP2 was found to be fast regardless of the log size. No clear dependency on the log size was found.

All the available filters in AP1 were reimplemented in AP2 and some additional filters that were deemed useful during the development of the AP2 (see figure 4.10). Unlike the AP1, however, AP2 does not offer functionality for customizing printing rules of terms. AP1 and AP2 have very different approaches and hence very different features for analyzing matching loops. The approach used in AP1 is to find repeated sequences of the same quantifiers in long paths representing potential matching loops. This approach has the benefit that AP1 can distinguish between path prefixes that do not exhibit repeating behaviour from the actual start of the matching loop. In AP2, the user has to manually remove nodes that are not deemed to be part of the repeating pattern before generating a matching loop graph to analyze the terms involved in a potential matching loop.

Both tools, however, suffer from false positives in detecting matching loops. A major improvement over AP1 is that AP2 allows for a fast search of all matching loops. Furthermore, the approach used in AP2 can find matching loops that are not restricted to paths.

One of the features in AP2 that contributes to a major improvement in usability over AP1 is the filter chain that allows the user to view all applied filters and undo any of them or reset all filters to the default filter chain.

Feature	AP2	AP1
Display selected node information	✓	✓
Display arith theory solver inst.	✓	✓
Optional pretty printing of terms	✓	✓
Customizable pretty printing rules	✗	✓
Filter for showing equality expls.	✗	✓
Automated explanation of paths	✗	✓
Display selected edge information	✓	✗
Warning prompt before rendering graph	✓	✗
Option to undo applied filters	✓	✗
Overview of applied filters	✓	✗
Indirect edges for indirect dependencies	✓	✗
Alternative edge-styles	✓	✗
Alternative node-styles	✓	✗
Automatic matching loop search	✓	✗
Matching loop graph generation	✓	✗
Equality nodes	✓	✗

Figure 4.9: Comparison of available features in AP2 and AP1.

Filter	AP2	AP1
Show longest path through selected node	✓	✓
Show ancestors of selected node	✓	✓
Show children of selected node	✓	✓
Only show up to maximum depth d	✓	✓
Hide subtree rooted at selected node	✓	✓
Show n first instantiations in log	✓	✓
Show n last instantiations in log	✓	✓
Show n most expensive instantiations	✓	✓
Show n least expensive instantiations	✓	✓
Show n insts. with most children	✓	✓
Show n deepest insts.	✓	✓
Show n least deep insts.	✓	✓
Show roots of n longest paths	✓	✓
Show subtree rooted at selected node	✓	✗
Hide ancestors of selected node	✓	✗
Show descendants of selected node	✓	✗
Hide descendants of selected node	✓	✗
Only show nodes with same quant. as sel. node	✓	✗
Hide all nodes with same quant. as sel. node	✓	✗
Show parents of selected node	✓	✗
Only show up to node index n	✓	✗
Show all potential matching loops	✓	✗
Display ML ending at selected node	✓	✗
Only show nodes with index at least n	✓	✗
Ignore equality nodes	✓	✗
Prune equality nodes	✓	✗
Ignore chain equality nodes	✓	✗

Figure 4.10: Comparison of available filters in AP2 and AP1.

Chapter 5

Conclusion

In this project, AP2 was developed as an OS agnostic web application. It allows users to generate a graph representing the dependencies between instantiations. AP2 was measured to offer better performance for processing log files generated by Z3 compared to AP1 and supports versions 4.8.5 up to and including 4.12.6 whereas for AP1, we have found that some logs generated by Z3 versions newer than 4.8.9 are not supported.

We have improved the design of the instantiation graph to visually convey more information about dependencies and nodes. In addition to the filters present in AP1, various new filters were implemented. As an improvement to AP1, we implemented a reconnecting algorithm to also visualize indirect dependencies between instantiations.

One particular issue in the context of SMT solvers such as Z3 are matching loops. Our tool can efficiently search for potential matching loops that the user can then analyze further by computing a matching loop graph, which is an attempt to visually represent how the various terms are involved in a matching loop. Unlike AP1 which was limited to analyze matching loops restricted to a path, AP2 can find and analyze matching loops that exhibit a more complex structure.

We have furthermore identified classes of log files where due to the large number of equality dependencies processing the log was unacceptably slow. To address this issue, we introduced the concept of equality nodes which economically reuse computed equality explanations. Even though this comes at the cost of slightly slower parsing in the common case (parsing speed decreases by roughly 13.9%), there are cases where the overall time to process a log reduces by up to 99.4%.

5.1 Future work

There are various ways in which AP2 can still be improved. As discussed in section 3.3.2 the matching loop graphs are not yet well-behaved for all kinds of matching loop graphs. More specifically, if there are multiple dependencies of the same type between two abstract instantiations, the resulting matching loop graph erroneously generalizes these even though they might have very different roles.

Generally, the way abstract instantiations that depend on equalities are represented in the matching loop graph could be improved by implementing a more detailed reconstruction of how equalities are used to rewrite blamed terms in an instantiation.

The matching loop graph construction feature was only implemented for the instantiation graphs without equality nodes hence one could try to extend the ideas introduced in section 3.3.2 to instantiation graphs with equality nodes.

Furthermore, the displayed matching loops reach all the way to the root of the instantiation graph. We have implemented filters for hiding all ancestors of a node such that the user can first manually filter out any instantiations that are deemed not to be part of a matching loop and then generate the matching loop graph for the filtered graph. To improve this, one could try to implement similar ideas as in AP1 to automatically detect the repeating pattern in a potential matching loop.

Another useful feature to implement would be a false positive detector for matching loops. In simple cases, this could be achieved by checking if the yield term of the last instantiation in a matching loop matches against the trigger of the repeated instantiation.

Like in the AP1, it might be useful to implement customizable pretty printing of terms to make large terms more legible.

Finally, the parser used in this project could be extended to handle more line cases of Z3 logs.

Appendix A

Z3 Log Documentation

A.1 Quantifier Instantiations

```
[new-match] fingerprint quant pattern bound_term+ ;
    {term_id | (lhs_id rhs_id)}+
```

- `fingerprint` is a 16-digit hexadecimal number
- `quant` denotes the identifier of the quantifier whose pattern was matched
- `pattern` denotes the identifier of the pattern used for this match
- `bound_term`$^+$ is a regular expression denoting the identifiers of the terms that were bound to the quantified variables
- `{term_id` | `(lhs_id rhs_id)}`$^+$ is a regular expression
  - `term_id` is a single term identifier indicating a blame term that either directly matches against the pattern or is rewritten with equalities to match against the pattern
  - `(lhs_id rhs_id)` indicates an equality between two terms that is used to rewrite blame terms to match against the pattern

```
[eq-expl] from (root|[lit eq;|cg arg_eqⁿ;|th theory;|ax] to)
```

- `from` is a term identifier
- `root` indicates that `from` is the root of its equivalence class
- `lit eq` indicates that `from` is equal to `to` due to the equality term `eq`
- `cg arg_eq`^{n} indicates that `from` is equal to `to` due to some n-ary function and the pairwise equalities between the arguments `arg_eq`
- `th theory` indicates that `from` is equal to `to` due to an equality obtained from the theory solver `theory`

Appendix B

Maximal difference

In section 3.2.1 we defined a permutation $p : \mathbb{Z}_n \to \mathbb{Z}_n$ and were interested in finding the parameter that maximizes the minimal distance between two permuted adjacent indices $k, k+1$ given by $\min\{|p(k+1) - p(k)| : k \in \mathbb{Z}_n\}$. This value can be derived as follows:

$$
\begin{aligned}
&\operatorname*{argmax}_c \min\{|p(k+1) - p(k)| : k \in \mathbb{Z}_n\} \\
&= \operatorname*{argmax}_c \min\{p(k+1) - p(k) : k \in \mathbb{Z}_n \wedge p(k+1) > p(k)\} \cup \\
&\{p(k) - p(k+1) : k \in \mathbb{Z}_n \wedge p(k+1) < p(k)\} \\
&= \operatorname*{argmax}_c \min\{(k+1)c - kc \bmod n : k \in \mathbb{Z}_n\} \cup \\
&\{kc - (k+1)c \bmod n : k \in \mathbb{Z}_n\} \\
&= \operatorname*{argmax}_c \min\{c \bmod n\} \cup \{-c \bmod n\} \\
&= \operatorname*{argmax}_c \min\{c \bmod n, (n-c) \bmod n\} \\
&= \{ \left\lfloor \frac{n}{2} \right\rfloor l : l \in \mathbb{Z}\}
\end{aligned}
\tag{B.1}
$$

Appendix C

Syntactically correct SMT2 problems

```
; some z3 options
(set-option :print-success false)
(set-info :smt-lib-version 2.0)
(set-option :smt.MBQI false)
(set-option :smt.QI.EAGER_THRESHOLD 100)
(set-option :smt.refine_inj_axioms false)
(set-option :trace true)
(set-option :trace_file_name cg_sort.log)

(declare-sort Number)

(declare-const z Number)
(declare-fun add_one (Number) Number)
(declare-fun f (Number) Number)

(assert (forall ((x Number)) (!(= (f x) (f (add_one x)))
    :pattern ((f x)) :qid loop)))

(assert (= (f z) z))
(check-sat)
```

Figure C.1: Syntactically correct encoding of the problem in figure 3.1.

```
; some z3 options
(set-option :print-success false)
(set-info :smt-lib-version 2.0)
(set-option :smt.MBQI false)
(set-option :smt.QI.EAGER_THRESHOLD 100)
(set-option :smt.refine_inj_axioms false)
(set-option :trace true)
(set-option :trace_file_name cg_sort1.log)

(declare-sort Number)

(declare-const z Number)
(declare-fun add_one (Number) Number)
(declare-fun f (Number) Number)
(declare-fun sum (Number Number) Number)

(assert (forall ((x Number)) (!(= (f x) (f (add_one x)))
     :pattern ((f x)) :qid loop)))
(assert (forall ((x Number)) (!(= (sum (f x) (add_one x)
   ) (add_one (sum x x))) :pattern ((sum (f x) x)) :qid
   loop)))

(assert (= (sum (f z) z) z))
(check-sat)
```

Figure C.2: Syntactically correct encoding of the problem in figure 3.8.

```
; some z3 options
(set-option :print-success false)
(set-info :smt-lib-version 2.0)
(set-option :smt.MBQI false)
(set-option :smt.QI.EAGER_THRESHOLD 100)
(set-option :smt.refine_inj_axioms false)
(set-option :trace true)
(set-option :trace_file_name cg_sort2.log)

(declare-sort Number)

(declare-const z Number)
(declare-fun add_one (Number) Number)
(declare-fun f (Number) Number)
(declare-fun sum (Number Number) Number)

(assert (forall ((x Number)) (!(= (f x) (f (add_one x)))
     :pattern ((f x)) :qid loop)))
(assert (forall ((x Number)) (!(= (sum (f z) (add_one x)
   ) (add_one (sum x x))) :pattern ((sum (f x) x)) :qid
   loop)))

(assert (= (sum (f z) z) z))
(check-sat)
```

Figure C.3: Syntactically correct encoding of the problem in figure 3.40.

Appendix D

Raw data of performance analysis

There are two kinds of measurements listed below. In the cells with a/b, a stands for the time measured with the code without equality nodes and b stands for the time measured with the code with equality nodes (in seconds). The code without equality nodes that was used has commit hash da02a07 in the linked repository [25]. The code with equality nodes that was used has commit hash 9649885 in said repository. In either case, the code was compiled with `trunk serve --release` to enable the highest level of compiler optimizations. The system specifications are given below:

- Browser: Google Chrome v121.0.6167.184
- OS: macOS Sonoma v14.2.1
- Chip: Apple M1
- RAM: 8 GB

Name	heaps-simpler	heaps2	foo	sequences-18
Size	7.9 MB	468.2 MB	875.4 MB	17.6 MB
a	0.1574/0.1849	7.4918/7.1797	14.3791/15.0879	0.3489/0.4283
b	0.0231/0.0239	1.1379/1.0652	0.9216/0.94	35.6563/0.0337
c	0/0.0002	0.0005/0.0004	0.0003/0.0004	0.0001/0.0001
d	0/0.0004	0.0003/0.0014	0.0005/0.0008	0/0013
e	0.0008/0.0005	0.0019/0.0029	0.0013/0.0016	0.006/0.0019
f	0.0005/0.0005	0.0002/0.0004	0.0003/0.0003	0.0027/0.0005
g	0.0242/0.0181	0.0258/0.0269	0.0244/0.0384	0.2091/0.0304
h	0.2833/0.2497	8.6857/8.3225	15.353/16.127	36.2604/0.5346
i	0.0091/na	0.0589/na	0.0176/na	0.2411/na

Figure D.1: Measured times for various processing stages in the Axiom Profiler 2.0. In each cell, the left value is for the code without equality nodes and the right column is with equality nodes.

Name	arraylist-quantified-permissions	arrays-quickselect-rec-index-shifting	z3	sequences-20
Size	64.3 MB	126.9 MB	172 MB	5.5 MB
a	1.0993/1.3064	2.2669/2.4914	2.0434/2.262	0.1324/0.2313
b	0.2565/0.2501	0.7539/0.7959	0.6444/0.6625	4.4239/0.0107
c	0.0001/0.0002	0.0003/0.0004	0.0003/0.0004	0/0
d	0.0002/0.0052	0.0003/0.0084	0.0003/0.0127	0/0.0004
e	0.0022/0.0027	0.0041/0.0048	0.0045/0.0047	0.0014/0.0017
f	0.0006/0.0007	0.0011/0.001	0.0009/0.0007	0.0045/0.0003
g	0.022/0.0224	0.0206/0.0211	0.0183/0.018	41.2201/0.0333
h	1.4058/1.6284	3.0717/3.3506	2.7431/2.9861	45.8374/0.296
i	0.7678/na	0.8983/na	0.4195/na	0.0594/na

Figure D.2: Summary of considered instantiations.

- **a**: Time to parse in seconds
- **b**: Time to construct instantiation graph in seconds
- **c**: Time to apply filter "Ignore theory solving instantiations" in seconds
- **d**: Time to apply filter "Render 125 most expensive instantiations" in seconds
- **e**: Time to filter out invisible nodes and reconnect in seconds
- **f**: Time to compute dot-String from petgraph in seconds
- **g**: Time to convert dot-String to SVG-element in seconds
- **h**: Approximate total time from selecting file to displaying graph in seconds
- **i**: Time for searching potential matching loops in seconds

Figure D.3: Measured times for various stages in Axiom Profiler 2.0. In each cell, the left value is for the code without equality nodes and the right column is with equality nodes.

Bibliography

[1] petgraph. URL: https://docs.rs/petgraph/latest/petgraph/index.html.

[2] Viz.js. URL: https://viz-js.com/api/.

[3] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972. arXiv:https://doi.org/10.1137/0201008, doi:10.1137/0201008.

[4] 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: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

[5] Mathieu Baudet. z3tracer. URL: https://github.com/facebookarchive/smt2utils/tree/main/z3tracer.

[6] Nils Becker, Peter Müller, and Alexander J. Summers. The axiom profiler: Understanding and debugging smt quantifier instantiations. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–116, Cham, 2019. Springer International Publishing.

[7] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[9] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*, pages 183–198. Springer, 2007.

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

[11] 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.

[12] Emden R Gansner and Stephen C North. An open graph visualization system and its applications to software engineering. *Software: practice and experience*, 30(11):1203–1233, 2000.

[13] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll (t): Fast decision procedures. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, pages 175–188. Springer, 2004.

[14] Juraj Hromkovic. Theoretical computer science: Introduction to automata, computability, complexity, algorithmics, randomization, communication, and cryptography, 2010.

[15] Thomas E Hull and Alan R Dobell. Random number generators. *SIAM review*, 4(3):230–254, 1962.

[16] Donald E Knuth. *The Art of Computer Programming: Seminumerical Algorithms, volume 2*. Addison-Wesley Professional, 2014.

[17] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pages 348–370. Springer, 2010.

[18] K Rustan M Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28*, pages 361–381. Springer, 2016.

[19] Richard Luo. Axiom profiler yew gui. URL: https://github.com/richardluo20/axiom-profiler-yew-GUI/tree/main.

[20] Richard Luo. Rust axiom profiler prototype. URL: https://github.com/richardluo20/axiom-profiler-rust-prototype/tree/main.

[21] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*, pages 41–62. Springer, 2016.

[22] Frederik Rothenberger. Integration and analysis of alternative smt solvers for software verification. Master's thesis, ETH-Zürich, 2016.

[23] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

[24] Programming Methodology Group (ETH Zürich). Axiom profiler. URL: `https://github.com/viperproject/axiom-profiler`.

[25] Programming Methodology Group (ETH Zürich). Axiom profiler 2.0. URL: `https://github.com/viperproject/axiom-profiler-2`.

[26] Programming Methodology Group (ETH Zürich). Log documentation. URL: `https://github.com/viperproject/axiom-profiler/blob/master/LogDocumentation/LogDocumentation.tex`.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

☐ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

[✓] I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Developing Tool Support for Understanding Quantifier Instantiations

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	First name(s):
Jyrkinen	Oskari

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	Signature(s)
Wettswil, 02.03.2024	[illegible]

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard