Scalable Logic Defined Static Analysis

Pavle Subotić

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

Tuesday 10th July, 2018

I, Pavle Subotić, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Logic languages such as Datalog have been proposed as a method for specifying flexible and customisable static analysers. Using Datalog, various classes of static analyses can be expressed precisely and succinctly, requiring fewer lines of code than hand-crafted analysers. In this paradigm, a static analysis specification is encoded by a set of declarative logic rules and an off-the-shelf solver is used to compute the result of the static analysis. Unfortunately, when large-scale analyses are employed, Datalog-based tools currently fail to scale in comparison to hand-crafted static analysers. As a result, Datalog-based analysers have largely remained an academic curiosity, rather than industrially respectful tools.

This thesis outlines our efforts in understanding the sources of performance limitations in Datalog-based tools. We propose a novel evaluation technique that is predicated on the fact that in the case of static analysis, the logical specification is a design time artefact and hence does not change during evaluation. Thus, instead of directly evaluating Datalog rules, our approach leverages partial evaluation to synthesise a specialised static analyser from these rules. This approach enables a novel indexing optimisations that automatically selects an optimal set of indexes to speedup and minimise memory usage in the Datalog computation. Lastly, we explore the case of more expressive logics, namely, constrained Horn clause and their use in proving the correctness of programs. We identify a bottleneck in various symbolic evaluation algorithms that centre around Craig interpolation. We propose a method of improving these evaluation algorithms by a proposing a method of guiding theorem provers to discover relevant interpolants with respect to the input logic specification.

The culmination of our work is implemented in a general-purpose and high-performance tool called Soufflé. We describe Soufflé and evaluate its performance experimentally, showing significant improvement over alternative techniques and its scalability in real-world industrial use cases.

Impact Statement

The contributions presented in this thesis have the potential to impact a wide range of applications in computer science. The first contribution of this thesis, the tool Soufflé, makes a significant contribution to the field of static analysis and database technology. This work builds on a line of research aimed improving the performance of state-of-the-art logic defined static analysers to that of industrially respected tools. The second contribution, an auto-indexing scheme, has impact on the performance and usability of Datalog engines in general, and in particular those aimed at large scale computations. The last contribution, improves the performance of interpolation based symbolic model checkers, by definition the semantic space of interpolants , and providing a solver independent mechanism of injecting domain specific knowledge into a theorem provers interpolation algorithm. In that sense, the impact exceeds that of performance improvements in model checking and has the potential to impact the logic, database and verification communities. In addition, this thesis has had direct industrial impact as outlined by the experimental sections of the contribution chapters and the application chapter of this thesis. Soufflé is used in production by Amazon Web Services on virtual network security analysis and Oracle for large scale Java program analysis.

Acknowledgements

I would like to thank my supervisor Byron Cook for giving me the opportunity to apply my research at Amazon, his encouragement at Amazon and his detailed insight/feedback on this PhD. I would like to thank my co-supervisor James Brotherston for his support in London, assistance in proof reading my thesis and positive feedback in my pre-viva examination. I would like to thank Ilya Sergey for his positive feedback of my thesis draft as my pre-viva examiner. I would like to thank Bernhard Scholz for his long time technical collaboration with Soufflé. I thank Daniel Kroening and Peter Schrammel for hosting me at the University of Oxford on a COST sponsored short term scientific mission. I thank all the people I met (and shared a drink with) at the Marktoberdorf Summer School. I would like to thank, in no particular order, all my co-authors, namely, Bernhard Scholz, Herbert Jordan, Andrew Santosa, Philipp Rümmer, Alan Feteke, Lijun Chang, Peter Schrammel, Hoissein Hojjat, Byron Cook, Evgeny Kotelnikov. I would also like to thank all my co-workers at the Automated Reasoning Group at Amazon in New York, Seattle and San Francisco (Byron, Daniel, Christopher, Jason, Sean, Catherine, Teme), and at the Program Analysis Group at Oracle Labs (Cristina, Bernhard, Herbert, Raghavendra, Paddy). I would like to thank Moshe Vardi, Yannis Smaragdakis, and many other researchers for their positive feedback. I would like to thank all the researchers who gave me great (and sometimes tough!) questions and discussions at my conferences talks and tutorials (COST meetings, FMCAD, CC, CAV, Soufflé PLDI tutorial). All these resulted in furthering my insight into my research and computer science. Finally, I would like to thank my family (Jasmina, Joe, Jovana, Zoran, Olja) for their understanding and support during both the many highs and

lows of this journey. I dedicate this thesis to my grandmother, Nadezda Subotić who I sadly lost in December 2016.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores the use of logic as a mechanism to create extensible and scalable static analysis tools.

Static analysis tools are now mainstream, and are employed on a variety of applications, including security, operational readiness and performance of embedded systems and general bug finding in software. In fact, static analysis tools are increasingly being used within niche domains, including network security [1], analysis of blockchain technology [2], and biomedical technology [3].

Historically, static analysers can be traced back to tools such as LINT [4] that was developed in the late 1970s at Bell Labs by developers dissatisfied with the support that compilers provided at that time. Such tools were fairly simple, performing limited syntactic and semantic checks similar to today's compiler warnings [4]. As the state-of-the-art progressed due to a growing body of research from the compiler and logic communities [5, 6], the next generation of static analysis tools became more powerful. These tools were able to perform increasingly complex semantic analyses [7, 8] and discover bugs such as integer overflows [9], null pointers [10], and many more. As computing became exceedingly more ubiquitous in society, e.g., with the emergence of e-commerce, smart phones etc in the 2000s, much attention was placed on software correctness, underscored by several notable software failures [11]. As a result, software verification began to play a more prominent role in many organisation's software development life cycles (SDLC). Today, organisations commonly employ some form of static analysis in their SDLC. For example, in the application security industry, software security life cycles involving static analysis, are commonly used; as defined by Microsoft [12]. Moreover, several regulatory authorities, including the FDA [13] and the ONR [14] now recommend the use of static analysis in software development. As a result, there are an abundance of industrial-strength static analysis tools available including Checkmarx [15], Coverity [16], Fortify [17], and Infer [18]. These tools are comprehensive, often supporting multiple programming languages [16, 19], covering an increasing range of implementation bugs [20] and scaling to millions of lines of source code.

The major challenge of performing static analysis stems from the fact that most

static analysis problems are undecidable [21]. As a result, static analysers perform sound approximations encoded in the property that one wants to prove. Each tool must therefore implement a given property of interest which will result in a some degree of false positives [22] (resp. negatives) on the correctness (resp. incorrectness) of the target system (e.g., program, network, etc.). Given, the large number of potential properties [9, 7, 8], including ones targeting niche, domain specific bugs, it is difficult to incorporate every type of property in a static analyser. While a static analyser may even detect all implementation level software errors in a program within respect to a given programming language, a defect may not necessarily be attributed to implementation level bugs. Instead, the defect could be attributed to a what we call a *software flaw* [20, 23]. That is, a software defect that may not cause crashes or any undefined behaviour, but may not fulfil part of its functional specification; e.g., relating to security [23]. In [20], the authors report that at Cigtal Inc., defects found by static analysis tools account for no more than 15% of all defects in their source code reviews. The majority of defects found were failures of the software to implement certain security requirements; e.g. ensuring code applies an authorisation before executing a particular functionality of an application. In 2013, Oracle reported [24] that despite using various static analyses on the Java source code [25], a spike in Java vulnerabilities was observed from 0-day attacks, including one which was allowed to bypass the Java sandbox. These vulnerabilities were based on the unsafe use of the `doPrivileged` [24] operation in the Java Development Kit (JDK). This type of error allows the bypassing of access control. Such vulnerabilities, unlike more general, well studied implementation bugs (e.g., buffer overflow), lacked a clear definition and their detection was not supported in any static analyser on the market.

1.1 Logic Defined Static Analysis

As a result of the above scenarios, the use of customisable static analysers have been proposed. One proposal is the use of static analyser frameworks such as Clang [26]. These frameworks use extensible software engineering design patterns to ease the

burden of much of the development overhead (e.g. parsing, data structures, etc.) typically required when crafting a static analyser. However, this approach still requires users to program in a low-level language (e.g. C++/Java) and be familiar with aspects of static analyser internals. A more user friendly alternative proposes the use of a high-level Domain Specific Language (DSL) [27, 28, 29] to express the properties of the analysis without specifying the implementation details of a static analyser. The choice of DSL is particularly important. The right level of abstraction must be chosen so that a user can express the analysis specification in a language congruent with their understanding and not be burdened with implementation details. On the other hand, the language must be expressive enough to adequately encode the static analysis problem.

A recent approach that has gained popularity in the static analysis community [30, 27, 31, 32] is the use of declarative, logic-based DSLs to specify a static analysis in the form of logical rules and to rely on an existing off-the-shelf logic constraint solver/decision procedure to perform computations. These logic-based DSLs provide declarative semantics for programs, resulting in succinct program representations and rapid-prototyping capabilities. Rather than specifying the computational steps imperatively, they allow users to specify the intended result declaratively, and thus are able to express computations in a more concise manner. Furthermore, the logical specification is decoupled from the solver evaluation algorithms, thus allowing for a separation of concerns between the static analysis domain expert and the engine designer. Therefore, any improvements to state-of-the-art solvers can be easily leveraged by users.

Many approaches have been proposed that use various logics [33, 34, 35] to encode static analysis problems. Among them, Datalog has shown to be particularly useful in specifying various program analyses due to its balance between expressivity and evaluation complexity. In the Datalog paradigm, a user specifies a static analysis problem through a set of Horn clauses that do not allow function symbols and assume finite domains. *A popular extension that is often added to increase the expressivity of Datalog is the limited use of negation in the body of a*

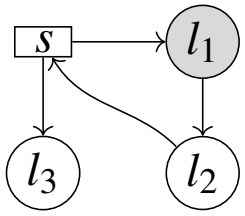
clause. This fragment of logic allows users to specify various reachability properties that can be solved in polynomial-time. Given an encoding of the analysis, the system to be analysed, e.g., a program, is abstracted to a set of input relations. A Datalog engine then executes the logic specification along with the input relations and produces an output relation(s) that represent analysis results. This paradigm is supported by a plethora of state-of-the-art Datalog engines that specifically target static analysis. Examples of recent Datalog engines are BDDBDDB [36], μZ [37] and `PA-Datalog` [30], which use fast/compressed data structures and have advanced the capabilities of logic-based static analysis [38, 39].

Example 1 (Java Access Analysis)**.** *In Figure 1.1a a simple scenario based on the do-privileged analysis presented in [24]. In our analysis, we assumes a low and high security state. We wish to assert that the invocation of a security sensitive method* `vulnerable` *is permitted only in the high-security state,* ie., the caller is authorised to make the call. *A call to the method* `protect` *transfers the security state from low to high if permitted. The example code of Figure 1.1a would not violate the imposed security policy if it can be assumed that* $i < j$ *whenever* `m` *is invoked. However, since this can not be ensured,* `m` *exhibits a security violation which we would like to detect.*

The control-flow graph of `m` *is shown in Figure 1 to the code fragment. It has a start node s, and nodes* l_1*,* l_2*, and* l_3 *representing statements in the input program. An edge* $(x, y) \in E$ *between two nodes represents a potential transfer of control. A statement* $x \in P$ *raises the security level.*

```
void m(int i, int j){
 s: while (i < j){
l1:  protect();
l2:  i++;
    }
l3: vulnerable();
}
```

(a) *Program to analyse*



(b) *Abstract CFG of program*

Figure 1.1: *Java-like input program, a graphical representation of its control-flow, and Datalog security specification*

To discover the potential of calling a vulnerable method in a low security state, we employ a simple Datalog analysis.

```
.type N
E(s:N,d:N) .input E
S(s:N) .input S
P(node:N) .input P
I(node:N) .output I

I("s").
I(y) :- I(x), E(x,y), !P(y).
```

Firstly, a Java program is abstracted into a set of input relations called the extensional database (EDB). For example, the Java program in Figure 1.1a is encoded into two EDB relations, namely, `E` *which encodes the edges of the CFG while* `P` *which encodes the set of protected notes in the CFG. We represent the two relations as a CFG in Figure with the gray node representing a node calling* `protect`*, i.e.,* l_1.

In the analysis we define a type `N`*, representing nodes, and* `E` *is defined as a binary relation between two* `N` *elements so as to represent edges between two nodes. The sets* `P` *and* `I` *are defined such that they also contain elements of type* `N`*. The qualifier* `.input` *denotes that the relations are an EDB and are provided as an input when executing the analysis. The set* `I` *is an output relation that will contain all nodes in the control-flow that are not secure and hence is marked with the qualifier* `output`*. If node* ℓ_3*, which is a* `vulnerable` *call, is in set* `I`*, the method* `m` *does not fulfil the security policy to be enforced and would thus be identified as insecure. The analysis always assumes the entry node* `s` *to be insecure by adding it to set* `I` *via* `I("s")`*. The propagation rule*

```
I(y) :- I(x), E(x,y), !P(y).
```

adds node y to the set of insecure nodes if (1) node x is insecure, (2) there is a control-flow from x to y, and (3) the target node y does not raise the security level.

If we execute our analysis we will discover a potential insecure path, i.e., a path that starts at node s, bypasses l_1 and yet reaches l_3. This can be easily seen in the Java program for the case that $i \geq j$. While this is a very simple analysis for demonstration purposes, Datalog analysis that check various properties can consist of many hundreds of Datalog rules e.g., [28] and analyse code bases of over a billion tuples [31, 40].

□

1.1.1 Open Problems

Despite the vast potential of Datalog-based static analysis and the continuing advances in Datalog solver technology [37, 41, 30, 36, 42], Datalog solvers fail to perform at the level of hand-crafted analysers. Consider the reasonable attempt at a hand-crafted C++ implementation shown in Figure 1.2. Here a worklist-based algorithm computes the analysis result in the *I* relation. The relation *I* is initialised with the `s` value and each `for` loop iteration attempts to add new information to *I* until no further information can be inferred, i.e., a fixpoint is reached. Of particular significance is that the performance disparity between the hand-crafted version that is equivalent to the Datalog analysis in Example 1. Despite the Datalog version being obviously simpler and more concise, if it is executed on the μZ (a state-of-the-art Datalog engine), a difference is observed in both run-time and memory usage[1]. For a CFG with 1 million generated nodes, the C++ program performs 4.5 times faster than μZ, while consuming 3.2 times less memory[2]. In general, because of the performance exhibited by above example, static analysers expressed by declarative DSLs (like the Datalog) tend to be limited in size, complexity and precision of the analyses performed, and as a result, have largely remained an academic curiosity.

Given the above, the question arises: **Can logic-based analysers be made to perform on par with hand-crafted alternatives?**

[1] On a Intel(R) Core(TM) i7-6600U CPU 2.60GHz 20GB of memory
[2] 8x less memory compared to BDDBDDB

```
using Tuple1 = std::array<int,1>;
using Tuple2 = std::array<int,2>;

using Relation1 = std::set<Tuple1>;
using Relation2 = std::set<Tuple2>;

Relation1 I,P;
Relation2 E;

E = someSourceForE();
P = someSourceForP();

I.insert(Tuple1{0});

auto dI = I;
while(!dI.empty()) {
  Relation1 newI;
  for(const auto& t1 : I) {
    auto x = t1[0];
    auto l = E.lower_bound({x,0});
    auto u = E.lower_bound({x+1,0});
    for(auto it = l; it != u; ++it) {
      auto y = (*it)[1];
      if (!contains(P,{y}) &
            !contains(I,{y})) {
          newI.insert({y});
      }
    }
  }

  I.insert(newI.begin(),newI.end());
  newI.swap(dI);
}
```

Figure 1.2: *Manual C++ analysis*

This thesis investigates methods for improving the performance and scalability of the logic-based static analysers with the goal of achieving best-in-class performance. First the thesis focuses on Datalog-based static analysers, and later investigates efficiently solving more expressive logics, namely, constrained Horn clauses using predicate abstraction and interpolation.

1.1.1.1 Thesis Hypothesis

To achieve the techniques presented in this thesis, we proceed from the hypothesis that in order to increase solver performance, solvers need to leverage the information present in the input logic specification and specialise their otherwise generic methods of evaluation. This approach allows engines to perform input specific optimisations instead of the relying on the solver designer to determine the most beneficial optimisation for average case inputs.

In the case of Datalog engines, state-of-the-art evaluation algorithms do not make any assumption about the input logical specification and performance is highly sensitive to the encoding of the Datalog rules. Any instance specific optimisations are typically performed at evaluation-time, incurring runtime overheads [43]. Therefore in order to further improve their performance, manual optimisations are required to modify the Datalog input to suite the underlying engine. For example, this approach can be observed in the Datalog static analysis library Doop [44] which requires a range of optimisations to enable the `Logicblox` engine to perform static analysis. In fact most Datalog engines require annotations to the Datalog program to provide hints for indexing [45], data structure order [36] etc. While these approaches can indeed result in runtime improvements, they are tedious to perform, are only good for the current input specification, and require detailed knowledge of the internals of the used Datalog engine, e.g., evaluation algorithm, indexing policy, scheduling policy etc.

A similar problem can be observed for engines that are used to solve static analysis problems encoded as recursive constrained Horn clauses. These solvers rely on external theorem provers to compute Craig interpolants [46], a mechanism that explains unsatisfiable formulae. However, for any given unsatisfiable formu-

lae, there may be an infinite set of interpolants. Therefore the theorem prover may return an interpolant that results in the solver exhibiting poor performance or even non-termination. Existing approaches have resorted to modifying the way theorem provers internally compute interpolants [47, 48]. However, there is no universal notion of what is a good interpolant, and while these approaches can improve performance for some targeted use cases, they may not always return a good interpolant for others.

In this thesis, we propose several novel techniques that incorporate knowledge of the input logical specification to the underlying engine/solver. In the case of Datalog evaluation, we leverage the fact that in the static analysis use case, the analysis specification, encoded as Datalog rules, is a design time artefact that remains fixed during evaluation. Therefore we employ partial evaluation to to synthesise a specialised static analyser. This approach leads to several optimisations during specialisation time. Among these the ability to automatically infer the optimal number of indexes required to speed up search operations for a given analysis specification. As a result these techniques which are implemented in Soufflé, enable the evaluation of complex static analyses performed on giga-tuples sized relations, typically deemed too difficult for Datalog based analysers.

We extend the expressivity of Datalog and investigate evaluation techniques for more expressive logics, such as constrained Horn clauses. For engines that solve recursive constrained Horn clauses, we present a technique that incorporates domain specific knowledge e.g., obtained from the input logic, and forces the theorem prover to compute interpolants with desired characteristics that improve CEGAR convergence and limit non-termination of Horn clause engines such as Eldarica [49]. Unlike other techniques, we do not resort to modifying theorem prover internals, but rather, present a theorem prover agnostic approach which abstracts the interpolation problem to force the theorem prover to give us the interpolants we deem useful w.r.t the abstraction. We have demonstrated that our technique enables solvers such as Eldarica to solve problems which previously it could not solve.

1.2 Thesis Contributions and Structure

The thesis is structured as follows. In Chapter 2, preliminary definitions and baseline knowledge are presented.

Contribution I: In Chapter 3, the first contribution of this thesis is presented. The chapter describes the Soufflé tool which is a Datalog based analyser that performs on a par with hand-crafted tools. For example, compared to the C++ hand-crafted solution to 1, Soufflé is able to perform the analysis 17x faster then the C++ program, using 20x less memory. Instead of evaluating a Datalog program directly, the framework proposes a paradigm shift. In the presented approach, the Datalog rules are treated as a logical specification which is used to synthesises an efficient C++ static analyser that adheres to this specification. The synthesis is performed using a hierarchy of partial evaluation stages which use the interpreter of various representations of the specification to lower the analysis to a more concrete form, eventually resulting in a parallel C++ version of the analyser. In addition, we extend Datalog to include more user friendly language constructs which aid users in engineering large analysis specifications. The framework has been implemented in the Soufflé tool and has been shown to perform on par with state-of-the-art hand-crafted analysers [50] on large industrial benchmarks. Contribution I can be broken down into the following sub-contributions:

- A novel approach to synthesising C++ analysers from Datalog via staged partial evaluation
- The definition of the relational algebra machine (RAM) intermediate representation
- The parallelisation of Datalog within the Soufflé framework
- Experiments showing the effectiveness of our approach using large, real world rulesets and factsets

This work has been published in Compiler Construction (CC) [40], Computer Aided Verification (CAV) [31], and the Symposium on Principles and Practice of Paral-

lel Programming (PPoPP) [51]. I am the main co-researcher, co-author and co-implementer with Bernhard Scholz and Herbert Jordan.

Contribution II: In Chapter 4, we present a new automatic indexing mechanism tailored for large scale Datalog computations. We present our novel indexing scheme that unlike previous Datalog indexing schemes, allows for an *optimal* number of indexes to be *automatically* inferred *statically*. The indexing approach exhibits negligible overhead during specialisation/compilation time while resulting in both runtime improvements and low memory usage compared to existing indexing schemes. Apart from boosting performance this technique avoids the need for users to manually reorder attributes in relations, as is required to scale program analysis frameworks such as Doop on single indexing schemes in state-of-the-art engines such as `Logicblox`/`PA-Datalog`. Contribution II can be broken down into the following sub-contributions:

- The formulation of an automatic indexing for large-scale Datalog computation based on this theory
- A formal definition of the minimum index selection problem (MISP) that finds the minimum number of indexes to cover all primitive searches
- A polynomial-time algorithm to solve MISP optimally via computing search chains
- Experiments showing the effectiveness of the indexing scheme in Soufflé, using large, real world rulesets and factsets

This work has been published in Very Large Databases (VLDB) [52]. I am the main researcher, co-author with Bernhard Scholz, Alan Feteke and Lijun Chang and sole implementer of this research.

Contribution III: In Chapter 5, In this chapter of this thesis we present a semantic and solver-independent approach for systematically exploring interpolant lattices, based on the notion of interpolation abstraction. We discuss how interpolation abstractions can be constructed for a variety of logics, and how they can be exploited

to improve solver performance. Contribution III can be broken down into the following sub-contributions:

- An approach to guiding theorem provers to discover specialised interpolants
- The theory of interpolant abstractions and their properties
- A number of exploration algorithms for finding feasible and cost effective interpolant abstractions
- Experiments demonstrating the feasibility of our technique on a set of diverse benchmarks

This work has been published at Formal Methods Computer Aided Design (FM-CAD) [53] and Acta Informatica [54]. I am the main co-author/researcher with Philipp Ruemmer and sole implementer of the work.

In Chapter 6, two use cases are presented that evaluate Soufflé on two industrial use cases. The first evaluates Soufflé in the context of security analysis of Amazon virtual networks. This use case demonstrates Soufflé's ability to scale to very large networks that are typically too difficult for alternative solvers. The second use case evaluates the Soufflé in the evaluation of Datalog encoded points-to analyses of the Java Development Kit (JDK) version 7. This use case is typically too complicated for existing Datalog engines and has been only recently been solved using highly specialised hand-crafted analysers, namely, Dietrich etal., in OOPSLA'15 [50]. The Amazon virtual network analysis use case is based on an yet to be published technical report that was co-authored with Evgeny Kotelnikov and Byron Cook at Amazon Web Services. The program analysis use case was done in collaboration with Bernhard Scholz at Oracle Labs.

In Chapter 7, we conclude with a discussion of future work resulting from this thesis.

Chapter 2

Background

In this chapter, we present the contextual and technical foundations required for understanding the approaches presented in later chapters of this thesis.

2.1 Datalog

In this section, we provide background on aspects of Datalog used in this thesis. This section is background knowledge for Chapter 3 and 4.

Datalog's origins date back to the 1977 *Symposium on Logic and Databases*, where David Maier is credited with coining the term *Datalog*. Datalog became an active area of interest in the database systems community in the eighties and early nineties with several seminal works investigating the pros and cons of various evaluation techniques [55, 56] language extensions [57, 58, 59], pragmatics [60, 61] etc. This research resulted in several early Datalog tools such as LDL [61], LOLA [62], Nail [61], and Coral [60]. However, due to a perceived lack of compelling applications at the time [63] Datalog research remained largely dormant [64].

Recently, Datalog has reemerged at the centre of several computer science communities resulting from a wide range of new applications, including data integration [65, 66], networking [67], security [68, 69], and, particularly important to this thesis, static analysis [28, 36, 70, 71, 72, 73].

Each of these application domains use Datalog language as a core, and further customise its syntax, expressivity and engine design to meet the particular needs of the given use case. The tool Soufflé, described in Chapters 3 and 4 of this thesis, is primarily focused on static analysis.

2.1.1 Datalog Engines

Below we survey several Datalog engines targeting static analysis and beyond:

2.1.1.1 Datalog Engines and Static Analysis

Datalog has been used a language to specify various classes of program analyses. Early work by Reps [38] and Dawson etal. [39] considered small programs (hundreds of lines of code) and was not viable for industrial sized code bases. In recent years, there have been efforts to apply Datalog program analysis to much larger code bases (e.g., thousands of lines) and more complex analysis problems (e.g., context

sensitive points-to).

The Prolog based analysis framework Dimple [74] demonstrated reasonable performance for simple context-insensitive pointer analysis using tabled Prolog.

Whaleys BDDBDDB engine [36] demonstrated encoding context-sensitive pointer analysis using Datalog and BDDs. The main challenge in using BDDBDDB for large systems relates to the issue of variable ordering. As it is uses BDDs (binary decision diagrams) as the underlying structure, choosing the right ordering is of paramount importance to performance. Given an unfavourable order, the analysis does not terminate within reasonable bounds. In our experimentation the default variable ordering rarely works, e.g., in the case of the JDK (see Chapter 6) the default order could not compute a context insensitive analysis. However, after significant exploration we were able to get a variable ordering do that BDDBDDB scaled for less precise analyses on the JDK. This variable ordering was not useful for the analysis of a different version of the JDK and other code. Such repeated exploration to find suitable variable orderings is too time consuming for BDDBDDB to be useful in our context.

μZ [37] is another tool that does very well on small examples. We have found that on very small Datalog programs, μZ performs on a par and occasionally better than SOUFFLÉ. However, the μZ performance quickly falls off as the input size and analysis complexity increases. In particular μZ does not seem to handle the large data sets generated during the analysis of the JDK. A common difference between the aforementioned Datalog engines and SOUFFLÉ is that they perform Datalog evaluation whereas we use Datalog as a specification to synthesise a C++ program.

In the case of static analysis, a number of engines have been proposed that are primarily used for analysing programs, networks, etc. Among them is Network Optimised Datalog(NoD [72]), implemented within the μZ engine, which provides customised data structures for network analysis. In our experience (see Chapter 6), NoD/μZ does not scale for large-scale static analyses.

`Logicblox` is a state-of-the-art proprietary tool that is used primarily for business use cases. However, modified variant of `Logicblox` version 3, `PA-Datalog` specifically targets program analysis. The popular program analysis library/frame-

work Doop uses `PA-Datalog` as its backend engine [28]. While Doop can exhibit good performance, it requires manual optimisations, as described in [44]. The syntax and some semantic aspects of `PA-Datalog` differs from Soufflé. As such, comparisons between the two tools have been difficult for large, complex analyses. However, a recent Doop port to Soufflé (See [75]) has made comparisons possible for program analysis benchmarks. In Chapters 3 and 4 we compare Soufflé to `PA-Datalog`.

A recent tool Flix [27] extends the semi-naïve evaluation to include lattices. This enables Datalog based solvers to perform abstract interpretation [5] with various abstract domains. This is an approach that contrasts to constraint databases [76] which have been linked to abstract interpretation. We believe that records implemented in Soufflé can be used as a foundation to encode lattices, this however is left to future work.

Semmle is a software engineering analytics and code exploration provider whose SemmleCode [77, 71] and QL [78] technology is built on Datalog research. QL [78] is a meta language that compiles to Datalog, similar in spirit to the Soufflé language extensions in Chapter 3. The approach described in [71] uses existing RDBMS engines to perform evaluation. For large scale static analyses described in Chapter 6 such approaches, including SQL-Lite [79], LOLA [80], SDS/DECLARE [81] and Logres [82] which use a RDBMS e.g., via source-to-source translators from Datalog to SQL, do not scale. In our experience, current relational database management systems cannot cope with the vast amounts of data and complex queries that arise translating Datalog to SQL (e.g., JDK experiments in Chapter 6).

2.1.1.2 Other Datalog Engines

Other systems that do not target static analysis but provide support for Datalog execution include IRIS [83] and DLV [84]. Both of them are bottom-up rule inference engines. However, they cannot be used as a stand-alone system. They provide the basic knowledge base component and the actual application needs to be written in a language like C++. Datalog Education System [85] (DES) is a deductive database

system that supports querying via both Datalog and SQL. Their focus is to support SQL queries in Datalog and thus translate SQL into Datalog. Socialite [86] provides extensions to Datalog to facilitate parallel execution. The aim is to speed up various graph algorithms and hence provide support for features such as aggregation. The programmer needs to provide suitable annotations to enable effective parallelisation on distributed systems. A similar approach is provided by [87]. Liu and Stoller [88] describe a general method for transforming Datalog rules to SETL programs. Their focus is on guaranteed worst-case time and space complexities. They use a mixture of arrays and linked list to manipulate the various sets. While this is useful in guaranteeing worst-case complexities their experiments are on relatively small data sets. Thus, it is not clear if their approach can handle large data sets. There are other approaches to implementing Datalog engines using GPUs [89] that harness the parallel capabilities of accelerators. In their work, tables may store the same tuple several times, and enforcing a set constraint at a later stage becomes costly, dominating the overall execution time. The duplication of tuples depletes the GPU memory quickly, and memory limitations of contemporary GPUs just amplify the short-coming of their approach for large-scale program analysis.

2.1.2 Datalog Preliminaries

2.1.2.1 Datalog Programs

A *relation* R is a subset of a m-ary cartesian product $\mathcal{D} = \mathbb{D}_1 \times \cdots \times \mathbb{D}_m$ (i.e., $R \subseteq \mathcal{D}$), where $\mathbb{D}_i$ $(1 \leq i \leq m)$ are the finite *domains* of the relation. Elements of a relation R are referred to as *tuples*. Each tuple $t = \langle e_1, e_2, \ldots, e_m \rangle \in R$ has a fixed length m, and e_i is an element of the domain $\mathbb{D}_i$ for $1 \leq i \leq m$. Given a relation R, *attributes* are used to refer to specific element positions of tuples of R. The set of attributes of R, denoted by $A = \{x_1, \ldots, x_m\}$, are m distinct symbols, and we write $R(x_1, \ldots, x_m)$ to associate symbol x_i to the i-th position in the tuples. The elements of a tuple $t = \langle e_1, \ldots, e_m \rangle$ can be accessed by *access function* $t(x_i)$ that maps tuple t to element e_i. For example, given a relation $R(x, y, z)$ and a tuple $t = \langle e_1, e_2, e_3 \rangle \in R$, the access function is $\{t(x) \mapsto e_1, t(y) \mapsto e_2, t(z) \mapsto e_3\}$.

A Datalog program P consists of a finite set of Datalog *rules* $\{r_1, r_2, \ldots\}$, each

of the form:

$$r : R_0(X_0) \text{ :- } L_1(X_1), L_2(X_2), \ldots, L_d(X_d), C(X_0, \ldots, X_d).$$

Each L denotes a literal which is either a positive or negative *atom*, i.e., $R_j(X_j)$ or $\neg R_j(X_j)$, where each R_j is a relation name, and each X_j is a sequence of variables and constants, and symbol "_" indicating irrelevance; for example, $R(u, \_, 1)$ where u is a variable. We assume each X_j is of correct arity and each $R_j(X_j)$ is a predicate or named relation. C is an equality constraint, i.e., a conjunction of equalities between variables ranging over $\{X_0, \ldots, X_d\}$.

$R_0(X_0)$ is called the *head* of the rule, and other atoms form the *body* of the rule.

A rule with only a head is called a *fact* (it must be instantiated) and a rule with only a body literal is called a query.

The set of relations that appear in the heads of P's rules are referred to as the intensional database (IDB). The set of input relations are referred to as the extensional database (EDB) or *dataset*. The extensional schema of P, $edb(P)$, consists of all extensional predicates of P. The intensional schema of P, $idb(P)$, consists of all intensional predicates of P. The set of all predicates in a Datalog program are referred to as a *schema* and denoted $sch(P) = idb(P) \cup edb(P)$. The set of rules in a Datalog program are called the *ruleset*.

The semantic meaning of a Datalog rule is that given a binding of all variables to constants, the head of the rule holds if each atom in the body of the rule holds.

In this thesis, we allow negated predicates in the body, but we limit its usage by the semantics of *stratified* and semi-definite Datalog (see [90] for the details of stratified Datalog), meaning that negated predicates must have an EDB relation or already computed IDB relation (hence in a lower strata), and a negated predicate holds if the positive version does not hold.

Example 2 (Datalog Program)**.** *The example program below computes the transitive closure of a graph in relation Path given an input edge set in Edge. The first clause is the base case and the second clause is the inductive case.*

$$Path(x,y) \text{ :- } Edge(x,y).$$
$$Path(x,z) \text{ :- } Edge(x,y), Path(y,z).$$

A concrete instantiation of a predicate $R(X)$, with variables replaced by appropriate constants, is denoted as a fact or a tuple, R(u). An instance of a Datalog program P, denoted $inst(P)$ is a set of tuples with relations from P, we use I to denote inst(P). An instance is in the EDB if all tuples contained are in the EDB, otherwise it is in the IDB.

2.1.2.2 Datalog Semantics

We now briefly describe the Herbrand interpretation of a Datalog program. The Herbrand universe U of a Datalog program P is the set of all possible ground terms. A ground term is a non-variable term, i.e. a constant value appearing somewhere in the program P. In important characterisation of Datalog is that the Herbrand universe is *finite* since the program P contains a finite number of constants.

The Herbrand base B of a Datalog program P is the set of all ground atoms. A ground atom is a predicate symbol that occurs in P with its arguments drawn from the Herbrand universe. Note that the Herbrand base respects the arity of the predicates. The Herbrand base is also finite.

An interpretation I of a Datalog program P is a subset of the Herbrand base B. A ground atom R is true w.r.t. an interpretation I if $R \in I$. A conjunction of atoms $R_1, \ldots, R_n$ is true w.r.t. an interpretation, if each atom is true in the interpretation. A ground rule is true if either the body conjunction is false or the head is true. A ground rule is a rule where all atoms are ground. A model M of a Datalog program P is an interpretation, i.e. a subset of the Herbrand base B, that makes each ground instance of each rule in P true. A ground instance of a rule is obtained by replacing every variable in a rule with a term from the Herbrand universe. A model M is minimal if there is no other model M_1 such that $M_1 \subset M$.

2.1.3 Bottom-up Datalog Program Evaluation

The model theoretic semantics above to not tell us how to compute a minimal model. For this, Datalog has a wide variety of evaluation approaches [90]. Among them is the bottom-up evaluation scheme. The bottom-up scheme computes the minimal model or solution by applying a monotonic Γ_P operator until a fixpoint is reached.

Definition 1 (Immediate Consequence Operator)**.** *We define an* immediate consequence *of I to be a fact $R(u)$ such that either $R(u) \in I$, or $R(u)$:- $R_1(u_1), \ldots, R_n(u_n)$ is a valid instantiation of a rule with each $R_i(u_i) \in I$. We then define the immediate consequence operator as a function $\Gamma_P : 2^{inst(P)} \rightarrow 2^{inst(P)}$ such that*

$$\Gamma_P(I) = \{t : \ t \text{ is an immediate consequence of } I\}$$

The process starts from an instance I of P that consists only of EDB tuples (also called facts). Then, an *immediate consequence operator* Γ_P in Definition 1 is repeatedly applied to I to generate new IDB tuples to be included into I. The process completes when a fixed-point is reached, i.e. no more IDB tuples can be generated. Due to the monotonicity of Γ_P, we can show using *Tarski's Fixpoint Theorem* [91], that there exists a least fixpoint of Γ_P. The resulting least fixpoint is denoted the model of P given I, or $P(I)$, and is the final result of bottom-up evaluation.

Example 3 (Bottom-up Evalaution)**.** *The program in Example 2 can be evaluated in the following steps assuming the Edge relation holds tuples* $(1,2)$ *and* $(2,3)$*:*

$$\begin{aligned} I &: Edge(1,2), Edge(2,3) \\ \Gamma_P(I) &: Edge(1,2), Edge(2,3), Path(1,2), Path(2,3) \\ \Gamma_P^2(I) &: Edge(1,2), Edge(2,3), Path(1,2), Path(2,3), Path(1,3) \end{aligned}$$

This process can therefore be seen as constructing an instance of the program, starting from the EDB and applying rules to compute new facts until no more new facts can be computed.

2.1.4 Properties of Bottom-up Datalog Evaluation

In Chapter 3 we base our evaluation strategy on bottom-up methods. In this section we discuss our justification our decision. However we first briefly discuss an alternative approach below.

An alternative to bottom-up evaluation, top-down evaluation describes a dual, proof theoretic method where proofs are explicitly constructed from queries to facts. These approaches [92] start from a query, and checking in the program whether there are rules and facts that make the query satisfiable. The query takes the form of a *goal clause*, which is a sequence of atoms:

$$\leftarrow R_1, \ldots, R_n$$

We consider each atom R_i in the goal clause, and search for some rule with R_i as the head. Once found we perform unification where we substitute constants for variables so they match. We then replace R_i in the goal clause with the body of that rule. We can apply this step repeatedly until either we reach EDB facts, in which case we show that the original goal clause holds, or we fail to find a valid instantiation at some point, in which case the original goal clause does not hold. An example of top-down evaluation is given in Example 4

Example 4 (Top-Down Evaluation)**.** *The program in Example 2 can be evaluated as follows in a top-down approach:*

$$\leftarrow path(1,3)$$
$$\leftarrow edge(1,y), path(y,3)$$
$$\leftarrow edge(1,2), path(2,3)$$
$$\leftarrow edge(1,2), edge(2,3)$$
$$\leftarrow \square$$

Top-down is rarely employed in Datalog for large-scale problems due to its inefficiency when non-ground terms are not present [93, 55, 56] due to the cost of

memoising facts. Moreover, top-down provides no guarantee that minimal proof-trees are constructed during its evaluation, and the full proof tree must be constructed in order to prove the existence of a tuple. For large-scale problems, this may impose a serious obstacle. We further elaborate in this problem below:

Observation 1 (Proof Derivation Level)**.** *We therefore prove that $R(u)^l$ iff the Γ_P was applied l times to derive $R(u)$.*

Proof. By induction on l:

- The base case ($l = 0$): Since very tuple in a EDB has $l = 0$ if $R(u)$ was derived then it either exists already and is thus by definition of the set semantics of relations cannot exist as a duplicate with another level or $l \neq 0$.

- Inductive case: We assume $R(u)$ was derived by a set of body tuples $R_1^{l1}(u_1), \ldots R_n^{ln}(u_n)$. Our IH assumes $l1$ to ln are equal to the applications of Γ_P for their existence. By definition of derivation and its monotonicity property, $R(u)$ must either not already have been derived if it is derived and hence its existence depends on $R_1^{l1}(u_1), \ldots R_n^{ln}(u_n)$. Hence the number of applications of Γ_P needed are the maximum of $l1$ to ln plus the one current application.

□

In the proof above, we observe that the top-down evaluation explicitly acts on a proof tree, where each branch has a height or derivation level. Below we connect the two notions via a derivation level on tuples. Assume we assign a proof derivation level l to each tuple as follows: $R(u)^l$. For every application of Γ_P operator a derived tuple is assigned a derivation level of 0 if it is from the EDB or the incremented level of the maximal height of tuples used to derive it. Given a tuple $R(u)^l$ its derivation level is therefore l. We therefore prove that $R(u)^l$ iff the Γ_P was applied l times to derive $R(u)$. Next we use this result to prove that bottom-up evaluation produces minimal proof heights.

Theorem 1 (Bottom-up Produces Minimal Height Derivations)**.** *Bottom-up Produces Minimal Height Proof Derivations, i.e., given a tuple $R(u)^l$ of a Datalog program P, there is no $l' < l$ such that $R(u)^{l'} \in P(I)$.*

Proof. Given Observation 1, we give a proof by contradiction. Assume $R(u)^l \in P(I)$, and there is an $l < l$ such that $R(u)^{l'} \in P(I)$. Then, $R(u)^{l'}$ is produced in fewer applications than $R(u)^l$, and so will have been added to the model earlier in evaluation. Due to the monotonously and definition of Γ_P, $R(u)$ is added only when first encountered. Hence as $R(u)^{l'}$ is in $P(I)$, $R(u)^l$ cannot be in $P(I)$ and we have a contradiction. □

On the other hand, for Datalog programs that query small number of tuples top-down may be more efficient as the entire IDB does not need to be computed to assert the existence of a single tuple. However, by combining a magic set transformation, a technique that statically simulates top-down evaluation, bottom-up can mitigate this drawback. Such an approach has been shown to be strictly better for Datalog programs compared to top-down [55]. Another limitation of bottom-up is its need to perform copies and existence/subsumption checks. For large amounts of data this can become costly [94]. We mitigate these issues in Soufflé with a variation of bottom-up algorithm and heavy use of indexing in searches.

2.1.5 Implementation of Bottom-up Datalog Evaluation

Recall, the result of program evaluation is attained when Γ_P reaches a fixpoint, i.e., when $\Gamma_P(I) = I$. Note that this evaluation appears closely related to the inductive construction of proof trees, and as we proved above the set of tuples represented by the proof tree T of height i is equal to the set of tuples generated by the i-th application of Γ_P . However, this naïve evaluation will repeat computations, since a tuple computed in some iteration will then be recomputed in every subsequent iteration. Therefore, the standard implementation of bottom-up evaluation in real world engines is typically based on semi-naïve. Semi-naïve evaluation contains two main optimisations over naïve bottom-up evaluation:

- Precedence graph optimisation: the Datalog program is split into strata. Firstly, a precedence graph of relations is computed, then each strongly connected component of the precedence graph forms a stratum. Each stratum is evaluated in a bottom-up fashion as a separate fixpoint computation in order based on the topological order of SCCs. The input to a particular stratum is

the output of the previous stratum.

- New knowledge optimisation: within a single stratum, the evaluation is optimised in each iteration by considering the new tuples generated in the previous iteration. A new tuple is generated in the current iteration only if it directly depends on tuples generated in the previous iteration. This avoids the recompilation of tuples already computed in prior iterations. The process is described in further detail below.

With these two optimisations, semi-naïve performs less repeated computations than the naïve algorithm. We present the semi-naïve algorithm in Algorithm 1 following the outline given in [90].

In this description we assume a set of rules and relations in a Datalog program such that each relation R_i, which is the i-th relation in a component C has a j-th rule, i.e., its head. A non-recursive rule consists of a singleton component with one relation R_1.

In a Datalog program evaluation we differentiate between recursively defined relations and non-recursive relations. Recall, recursively defined relations are defined such that the relation itself shows up immediately or intermediately in the body of its clauses (where it is the head). To avoid recurring computations, the semi-naïve evaluation of Datalog programs keeps track of the previous, current, delta and new knowledge of a recursively defined relation as depicted in Figure 2.1. The general observation is that only new knowledge in the previous iteration (i.e. delta knowledge) can generate new knowledge in the current iteration. Hence, for each iteration in a fixed-point iteration relations are sliced into (1) current knowledge, which includes the previous knowledge describing the knowledge of the previous iteration and the delta knowledge that is the new knowledge of the previous iteration, and (2) in the new knowledge gathered in the current iteration. With this partitioning of relations, the fixed-point will converge faster, however, with the disadvantage of keeping track of previous, current, delta, and new knowledge of a relation by copying data.

To avoid computing all recursively defined relations in a single fixed-point, the

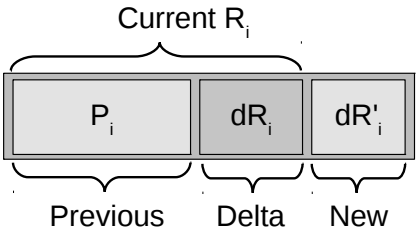


Figure 2.1: *The semi-naïve algorithm splits recursively defined relations into subsets per fixed-point iteration called previous, current, delta, and new knowledge*

semi-naïve algorithm computes the *data dependencies* between relations in form of a *precedence graph*. Strongly-connected components of the precedence graph resemble mutually recursive relations and are computed by a Tarski-Knaster style fixed-point algorithm. The strongly-connected component graph of the precedence graph represents a partial order dictating orders among components in the graph. For example, a relation in component C_1 that requires another relation in component C_2 for its evaluation, will enforce the semi-naïve algorithm to compute the component C_2 prior to component C_1.

Example 5 (Semi-naïve Bottom-up Evaluation)**.** *Observed when we apply the semi-naïve algorithm to Example 2 we compute only new information in the* Δ *relations. We have only one component. We first compute path from the non-recursive rule and the we compute only new knowledge, i.e., tuple* $(1,3)$.

$$\Delta path_0 : \{(1,2),(2,3)\}$$
$$\Delta path_1 : \{(1,3)\}$$

In the case we have more tuples to compute, we would never recompute $(1,3)$.

Algorithm 1 computes the strongly connected component graph in the first step. The collection of relations I represent relations that have been computed so far and are stable.

Initially, the collection of already computed relations is empty (line 2). After computing the SCC graph and initialising I, the SCC graph is traversed in topo-

```
1  Compute SCC dependence graph G;
2  I := ⟨⟩;
3  foreach component C = [R_1,...,R_n] in G in topological order do
4      if C is recursive then
           // iterate over all relations in C
5          for i=1 to n do
6              R_i := ∅; // Initialise relation R_i
               // Iterate over all non-recursive rules of R_i
7              for j=1 to m do
                   // Eval non-recursive rules
8                  R_i := R_i ∪ Eval_i^(j)(I) ;
9              end
               // Set new knowledge to current knowledge
10             ΔR_i := R_i;
               // Set previous knowledge to empty set
11             P_i := ∅;
12         end
           // Compute recursive rules until fixed-point is
              reached.
13         while Σ_{i=1}^{n} |ΔR_i| > 0 do
14             for i=1 to n // iterate over all relations in C
15             do
16                 ΔR'_i := ∅ ;
17                 for j=1 to m // iterate over all recursive rules
                       of R_i
18                 do
19                     ΔR'_i := ΔR'_i ∪ ⋃_{1≤k≤n} Eval_i^(j)(I ∪
                         ⟨R_1,...,R_{k-1},ΔR_k,P_{k+1},...,P_n⟩) \ R_i ;
20                 end
21             end
               // Update previous and current knowledge
22             for i=1 to n // iterate over all relations in C
23             do
24                 P_i := R_i ;
25                 R_i := R_i ∪ ΔR'_i ;
26                 ΔR_i := ΔR'_i ;
27             end
28         end
29     else
           // Component is a single non-recursive relation
30         R_1 := ⋃_j Eval_1^(j)(I) ;
31     end
32     I := I ∪ ⟨R_1,...,R_n⟩ ;
33 end
```

Algorithm 1: Semi-naïve algorithm

logical order (line 3), hence, ensuring that relations are computed before their use (except mutually recursive ones). We assume n relations in a component where $1 \leq n$.

Case (recursive components): We first explain the recursive case of Algorithm 1, defined in lines 4-31. Here we assume a recursive set of relations in a component. First the previous knowledge P_i, current knowledge R_i and delta knowledge ΔR_i are initialised for each relation $R_1 \ldots R_n \in C$. The current (R_i) and delta knowledge (ΔR_i) for each relation in the component is initialised by the facts and the result of evaluating the non-recursive rules on the relations (line 10). The j-th facts/rule of the i-th relation in the component C is evaluated by the evaluation function $Eval_i^{(j)}[I]$ using the already computed relations I of the program. Since, the j-th rules are only non-recursive rules for relation R_i, we can evaluate the j-th rule with I only. The non-recursive evaluation of the component C iterates over all relations and over all rules of a relation. After the end of the inner for-loop, the delta-knowledge and the previous knowledge is updated for the fixed-point calculation. In the second part of the evaluation of recursive component (line 15), a fixed-point is computed for the mutual recursive rules in a component.

The `while` loop continues if new knowledge in the previous iteration could be found, i.e., $\sum_{1 \leq i \leq n} |\Delta R_i|$ has to be greater than zero for another iteration of the fixed-point calculation. Inside the fixed-point loop the new knowledge of the relations in C are computed and stored in $\Delta R'_i$. The evaluation of a single recursive rule requires to keep track of the positions of the recursively defined relations in a rule. For each position a new relational algebra operation is issued. All relations before the position use the current knowledge, for the position k the delta knowledge is used, and for all succeeding positions the previous knowledge is used (cf. Chapter 12, page 314 [95]). After updating previous, current, and delta knowledge the next iteration of the fixed-point loop is started.

Case (non-recursive components): In the non-recursive case of Algorithm 1, non-recursive relations components will only contain a single relation R_1, and the evaluation is straight-forward by evaluating all facts and rules of the relation. After

computing the results of the relations $R_1, \ldots, R_n$ of the component C, the relations are added to the collection of already computed relations I.

2.2 Partial Evaluation

In this section we give an overview of partial evaluation that is used in Chapter 3.

Partial evaluation follows from the observation that a one-argument function can be obtained from one with two arguments by fixing one of the input arguments. Partial evaluation performs this process to programs proceeding as follows: a partial evaluator is given a subject program p together with part of its input data, $in1$. It constructs a new specialised program p_{in1} which, when given p's remaining input $in2$, will yield the same result that p would have produced given both inputs $in1$ and $in2$. We provide an example of this process below.

Example 6 (Partial Evaluation)**.** *Below we define a recursive program that computes x^n. The program contains two inputs, namely, x and n.*

```
int f(n, x) {
 if(n == 0) then return 1;
 else if (even(n)) then return f(n/2,x) ^ 2
 else x * f(n-1,x)
```

Given we fix n = 5, we obtain the program below.

```
int f5(x) {
  return x * ((x ^ 2) ^ 2)
}
```

We are able to precompute all expressions involving n, to unfold the recursive calls to function f, and to reduce $x * 1$ to x. This optimisation was possible because the program's control is completely determined by n. If on the other hand $x = 5$ but n is unknown, specialisation gives no significant speedup.

Lombardi was first to coin the term Partial Evaluation in 1964 in reference to discussing Lisp's ability to compute with incomplete information [96]. However, as a theory, partial evaluation has its foundations in Kleene's s-n-m theo-

rem [97]. Kleene proved that for any given Turing machine for a general m+n-argument function f, and given values a_1 to a_m of the first m arguments, there exists a Turing machine for the specialised function $g = f_{a_1,\ldots a_m}$ which satisfies $g(b_1,\ldots,b_n) = f(a_1,\ldots,a_m,b_1,\ldots,b_n)$ for all b_1 to b_n. Futamura [98] was the first to use partial evaluation in the context of program transformation and considered the application of the partial evaluator to itself, thus deriving compilers, compiler generators and compiler generator generators in form of semantic equations. The first implemented partial evaluator can be traced to Redfun, a partial evaluator for Lisp. This work also mentioned the possibility of a compiler generator (generator), similar to Futamura. Around the same time Turchin proposed the idea of partial evaluation in the context of symbolic computation of functional languages. In the mid-80s Jones, Sestoft, and Søndergaard developed implemented first self-applicable partial evaluator written in Lisp and used as a compiler generator [99]. This line of work propelled a wide variety of research and applications of partial evaluation. These applications include parser and compiler generators [100], program transformations [96], abstract interpretation [101, 102], security analysis [103], implementation of Virtual Machines [104], and Model driven development [105, 106] among many other applications. A particularly interesting application of partial evaluation that is in the spirit of the approach presented in Chapter 3 is the partial evaluation of interpreters in model driven development to turn an interpreter into a translator. Here a partial evaluator is used to specialising a model interpreter with respect to a model to create a compiled model interpretation. Another related approach is observed in [104] allows uses to define languages solely by defining an interpreter. The system then uses the interpreter and partial evaluation to perform compilation independent of the language. Techniques relating to Datalog program transformation such as [42, 107] can be seen as instances of partial evaluation.

2.2.1 Trivial Partial Evaluation

A partial evaluator is typically denoted as a `Mix` operator defined in Definition 2. The term `Mix` was given after *mixed computation*.

Definition 2 (`Mix` operator)**.** *An operator* `Mix` *is a partial evaluator iff*

$$\forall p, s, d : [\![p]\!](s,d) = [\![[\![\texttt{Mix}]\!](p,s)]\!](d)$$

We say the program produced by $[\![\texttt{Mix}]\!](p,s)$ *is the* residual program*.*

A partial evaluator thus performs a mixture of code execution and code generation in the `Mix` operator. Thus the specialisation can be shown in our previous example as the following equation: $p_5 = [\![\texttt{Mix}]\!](p,5)$. The execution can be described as the following equation: $out = [\![p_5]\!](x)$.

2.2.2 Interpreter Partial Evaluation

A special case of partial evaluation that follows the work of Futamura [108] is when the program that is being partially evaluated is an interpreter. It follows then that we can construct a first Futamura projection:

Definition 3 (First Futamura Projection)**.** *Let int be an interpreter for the language L itself written in the language M. Then, for an arbitrary program p written in L and its input d we have:*

$$[\![P]\!]_L(d) = [\![int]\!]_M(p,d) = [\![[\![\texttt{Mix}]\!](int,p)]\!]_M(d)$$

The implementation language of `Mix` is irrelevant for the purpose of this equation. The equation in particular means that the residual program, i.e., $[\![\texttt{Mix}]\!](int,p)$, is an M-program with the same operational behaviour as the L-program p.

2.2.3 Considerations in Partial Evaluation

2.2.3.1 Online vs Offline

There are two approaches to partial evaluation: online and offline. An online partial-evaluator is a non-standard interpreter. The treatment of each expression is determined at partial evaluation time. Online partial evaluators in general are very accurate but at the price of a considerable interpretive overhead. Offline partial evaluators are structured with a preprocessing phase and specialisation phase. The

preprocessing phase employs a binding-time analysis to determine for each expression whether it can be evaluated at partial-evaluation time or whether it must be evaluated at run-time. Once this information is determined, the specialisation is performed. The approach in Chapter 3 can be is offline (in a single specialisation phase) as the static parts are known ahead of time.

2.2.3.2 Termination

In the case of unfolding calls during function specialisation, partial evaluation can loop in two ways: either by unfolding infinitely many function calls or by creating infinitely many specialised functions. Both of these issues can be avoided by defining a bound on the number of unfolded calls and the number of specialised functions, but often this strategy appears unsatisfactory. Hence more sophisticated techniques have been proposed, e.g., [109, 110]. As will be explained, in the approach of Chapter 3 termination issues do not manifest due the fact that we do not unfold recursive rules, and instead, replace the rule with an interpreter definition which has termination characteristics on Datalog's finite domain.

2.2.3.3 Performance Benefits

It is not always apparent if partial evaluation is beneficial for a given application. Given a time function t and a program p we say the execution time of p is $t(p)$. Therefore, for a fixed two input program p with static input s and dynamic input d, the speedup function is defined as:

$$speedup_s(d) = \frac{t_p(s,d)}{t_{ps}(d)}$$

In general, if $speedup_s(d) > 1$ for all s, and d changes more than s then partial evaluation is advantageous. In cases where, s and d both change frequently, the time to do specialisation must be accounted for and thus we desire the following to hold:

$$t_{mix}(p,s) + t_{ps}(d) < t_p(s,d)$$

Remark. We note that many uses of partial evaluation are not motivated primarily by performance [103, 105, 106]. For example, the approach taken in [105] use partial evaluation as a method of correct compilation with respect to a model and an interpreter. A virtue is that the method yields target programs that are always correct with respect to the interpreter. Thus the problem of compiler correctness seems to have vanished. This approach is clearly suitable for prototype implementation of new languages from interpretive definitions (known as meta-programming in the Prolog community).

Our approach uses both benefits of partial evaluation: On one hand we perform interpreter guided compilation, our residual program is correct by using an interpreter. On the other hand, we remove expensive run-time aspects (e.g., virtual dispatch) in the interpretor to produce faster residual programs.

2.3 Symbolic Solving of Horn Clauses

In this section, we provided background on the symbolic evaluation of recursive constrained Horn clauses [33]. Unlike Datalog evaluation, we compute symbolic formulae solutions to predicates instead of assigning sets of tuples to relations names.

The use of logic to model programs has early roots that can be traced back to Floyd's seminal work *Assigning Meaning to Programs* [111]. This was followed up by Tony Hoare's, *An Axiomatic Basis for Computer Programming* [112]. Later, in 1987, Blass and Gurevich [113] proposed Existential Positive Least Fixed-Point Logic (E+LFP) to produce the partial correctness of simple procedural imperative programs by satisfiability checking. In 1977, Clarke [114] established boundaries for relative completeness. Reasoning about constrained Horn clauses is paramount to the field of constraint logic programming [115]. Despite the fact that CLP is typical targeted as a declarative programming language, the uses of CLP for static analysis is extensive e.g., [116]. It relies on an execution engine that finds a set of substitutions that are solutions to a query, which can be seen as an extension to top-down engines found SLD resolution.

A number of sophisticated methods have recently been developed for solving Horn clauses in the context of static analysis and verification. The which can be seen come in two main varieties: (1) Top-down derivations, in the spirit of SLD resolution, start with a goal and resolve the goals with clauses. Derivations are cut off by using cyclic induction or interpolants. If the methods for cutting off all derivation attempts, one can extract models from the failed derivation attempts. Examples of tools based on top-down derivation are [117, 118]. (2) Bottom-up derivations, start with clauses that dont have uninterpreted predicates in the bodies. They then derive consequences until sufficiently strong consequences have been established to satisfy the clauses. Examples of tools based on bottom-up derivation are [119, 49].

2.3.1 Constrained Horn Clauses (CHC)

2.3.1.1 Constraint Languages

We assume that a first-order vocabulary of interpreted symbols has been fixed, consisting of a set of fixed-arity function symbols $\mathcal{F}$, and a set of fixed-arity predicate symbols $\mathcal{P}$. Interpretation of $\mathcal{F}$ and $\mathcal{P}$ is determined by a class $\mathcal{S}$ of structures (U, I) consisting of non-empty universe U, and a mapping I that assigns to each function in $\mathcal{F}$ a function over U, and to each predicate in $\mathcal{P}$ a set-theoretic relation over U. We assume an equation symbol = in $\mathcal{P}$, with the usual interpretation. Given a countably infinite set $\mathcal{X}$ of variables, a constraint language is a set $Constr$ of first-order formulae over $\mathcal{F}$, $\mathcal{P}$, $\mathcal{X}$. For example, the language of quantifier-free Presburger arithmetic has $\mathcal{F} = \{+, -, 0, 1, 2, \dots\}$ and $\mathcal{P} = \{=, \leq, \dots\}$).

A constraint is called *satisfiable* if it holds for some structure in $\mathcal{S}$ and some assignment of the variables $\mathcal{X}$, otherwise unsatisfiable. We say that a set $\Gamma \subseteq Constr$ of constraints entails a constraint $\phi \in Constr$ if every structure and variable assignment that satisfies all constraints in Γ also satisfies ϕ; this is denoted by $\Gamma \models \phi$. $fv(\phi)$ denotes the set of free variables in constraint ϕ. We write $\phi[x_1, \dots, x_n]$ to state that a constraint contains (only) the free variables $x_1, \dots, x_n$, and $\phi[t_1, \dots, t_n]$ for the result of substituting the terms $t_1, \dots, t_n$ for $x_1, \dots, x_n$. Given a constraint ϕ containing the free variables $x_1, \dots, x_n$, we write $Cl_\forall(\phi)$ for the universal closure $\forall x_1, \dots x_n.\phi$

2.3.1.2 Horn Clauses

To define the concept of Horn clauses, we fix a set $\mathcal{R}$ of uninterpreted fixed-arity relation symbols, disjoint from $\mathcal{P}$ and $\mathcal{F}$. A *constrained Horn clause* (CHC) is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where:

- C is a constraint over $\mathcal{F}, \mathcal{P}, \mathcal{X}$
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms over $\mathcal{F}$, $\mathcal{X}$;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in \mathcal{R}$ to first-order terms, or is the constraint false.

Similarly to Datalog, H is called the head of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the body. In case $C = true$, we usually omit C. First-order variables in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in S$. Notions like (un)satisfiability and entailment generalise straightforwardly to formulae with relation symbols.

2.3.1.3 Solvability

A relation symbol *assignment* is a mapping $SOL : \mathcal{R} \rightarrow Constr$ that maps each n-ary relation symbol $p \in \mathcal{R}$ to a constraint $SOL(p) = C_p[x_1, \ldots, x_n]$ with n free variables. The instantiation SOL(h) of a Horn clause h is defined by:

- $SOL(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t}))$=$C \wedge SOL(p_1)[\bar{t}_1] \wedge \cdots \wedge SOL(p_n)[\bar{t}_n] \models SOL(p)[\bar{t}]$
- $SOL(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow false)$=$C \wedge SOL(p_1)[\bar{t}_1] \wedge \cdots \wedge SOL(p_n)[\bar{t}_n] \models false$

Definition 4 (Solvability)**.** *Semantic and syntactic solvability is defined as follows:*

(i) A $\mathcal{HC}$ is called semantically solvable if for every structure $(U, I) \in \mathcal{S}$ there is an interpretation of the relation symbols $\mathcal{R}$ as set-theoretic relations over U such the universally quantified closure $Cl_\forall(h)$ of every clause $h \in \mathcal{HC}$ holds

in (U, I)*. In other words, if the structure* (U, I) *can be extended to a model of the clauses* $\mathcal{HC}$*.*

(ii) A $\mathcal{HC}$ *is called syntactically solvable if there is a relation symbol assignment SOL such that for every structure* $(U, I) \in \mathcal{S}$ *and every clause* $h \in \mathcal{HC}$ *it is the case that* $Cl_\forall(SOL(h))$ *is satisfied*

Lemma 1. *A set* $\mathcal{HC}$ *of Horn clauses is semantically solvable if and only if* $\mathcal{HC}$ *does not have any counterexamples.*

Proof. Counterexamples correspond to satisfying assignments of the expansion of recursion-free unwinding of $\mathcal{HC}$, [120]. It is clear that if $\mathcal{HC}$ is solvable, then every recursion-free unwinding is solvable; for the converse, construct a solution of $\mathcal{HC}$ as the union of minimal solutions of all recursion-free unwinding of $\mathcal{HC}$. □

Lemma 2. *A set* $\mathcal{HC}$ *of Horn clauses has a closed ARG(S,E) (see Definition 5) iff* $\mathcal{HC}$ *is syntactically solvable.*

Proof. We show both cases for the iff: Case $\Rightarrow$: We define each relation symbol p as the disjunction $\bigvee_{(p,Q)\in S} \bigwedge Q$. Given S is closed under the edge relation E, this yields a solution for the set of Horn clauses $\mathcal{HC}$ Case $\Leftarrow$: Suppose $\mathcal{HC}$ is syntactically solvable, with each relation symbol mapped to a Constraint C_p. We can define the mapping $\Pi(p) = \{C_p\}$, and construct the ARG with nodes $S = \{(p, C_p)\}$ and the maximum edge relation E, which is be definition closed. □

Remark. We note, if a set of Horn clauses is syntactically solvable, then it is also semantically solvable. The converse is not true in general, because the solution need not be expressible in the constraint language.

Example 7 (Greatest Common Divisor (GCD))**.** *The example below shows a system of constrained Horn clauses that compute the* greatest common divisor *of the first and second argument and store the result in the third argument. The algorithm is based on Euclid iterative GCD algorithm. The analysis property is expressed in the final Horn clause. After invoking the gcd operation on equal positive numbers M*

and N, we wish to check whether it is possible for the result R is larger than M. This error condition is encoded as a Horn clauses with false in its head.

$$(1)\ gcd(M,N,R) \leftarrow M = N \wedge R = M$$

$$(2)\ gcd(M,N,R) \leftarrow M > N \wedge M1 = M - N \wedge gcd(M1,N,R)$$

$$(3)\ gcd(M,N,R) \leftarrow M < N \wedge N1 = N - M \wedge gcd(M,N1,R)$$

$$(4)\ false \leftarrow M \geq 0 \wedge M = N \wedge gcd(M,N,R) \wedge R > M$$

2.3.2 Craig interpolation

We assume familiarity with standard classical logic, including notions like terms, formulae, Boolean connectives, quantifiers, satisfiability, structures, models. For an overview, see, e.g., [121]. The main logics considered in this Chapter 5 are *classical first-order logic* with equality (FOL) and *Presburger arithmetic* (PA).

Given any logic, we distinguish between *logical symbols,* which include Boolean connectives, equality $\doteq$, interpreted functions, etc., and *non-logical symbols*, such as variables and uninterpreted functions. If $\bar{s} = \langle s_1, \ldots, s_n \rangle$ is a list of non-logical symbols, we write $\phi[\bar{s}]$ (resp., $t[\bar{s}]$) for a formula (resp., term) containing no non-logical symbols other than $\bar{s}$. We write $\bar{s}' = \langle s'_1, \ldots, s'_n \rangle$ (and similarly $\bar{s}''$, etc.) for a list of primed symbols; $\phi[\bar{s}']$ ($t[\bar{s}']$) is the variant of $\phi[\bar{s}]$ ($t[\bar{s}]$) in which $\bar{s}$ has been replaced with $\bar{s}'$. With a slight abuse of notation, if $\phi[x_1, \ldots, x_n]$ is a formula containing the free variables $x_1, \ldots, x_n$, and $t_1, \ldots, t_n$ are ground terms, then we write $\phi[t_1, \ldots, t_n]$ for the formula obtained by substituting $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$.

An interpolation problem is a conjunction $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ over disjoint lists $\bar{s}_A, \bar{s}, \bar{s}_B$ of symbols. An *interpolant* is a formula $I[\bar{s}]$ such that $A[\bar{s}_A, \bar{s}] \Rightarrow I[\bar{s}]$ and $B[\bar{s}, \bar{s}_B] \Rightarrow \neg I[\bar{s}]$; the existence of an interpolant implies that $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is unsatisfiable. Graphically, an interpolantion problem $A \wedge B$ with an interpolant I can be represented by the Venn diagram in Figure 2.2.

We say that a logic has the *interpolation property* if also the opposite holds: whenever $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is unsatisfiable, there is an interpolant $I[\bar{s}]$. For sake of

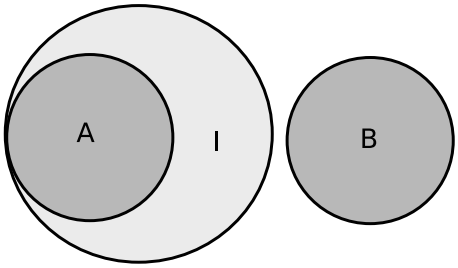


Figure 2.2: *Graphic representation of interpolation problem $A \wedge B$ with interpolant I*

presentation, we only consider logics with the interpolation property (see e.g., [122] for unhandled logics); however, many of the results hold more generally.

We represent *binary relations* as formulae $R[\bar{s}_1, \bar{s}_2]$ over two lists $\bar{s}_1, \bar{s}_2$ of symbols, and relations over a vocabulary $\bar{s}$ as $R[\bar{s}, \bar{s}']$. The identity relation over $\bar{s}$ is denoted by $Id[\bar{s}, \bar{s}']$.

Practically interpolants can be computed using a wide range of mechanisms, including resolution proof annotations [123], constraint solving [124] and SAT/SMT solving (using the unsat core) [125]. We have described binary interpolants. In fact a taxonomy of interpolants have been described by Ruemmer etal. [126], including inductive sequences of interpolants, tree interpolants and DAG interpolants. In any case, these interpolants can be solved by repeated computations of binary interpolants.

2.3.3 Solving Recursive Horn Clauses via Predicate Abstraction

Solutions for systems of Horn clauses as described in Example 7, can be constructed using a predicate abstraction based algorithm [119, 127]. Predicate abstraction depicted in Figure 2.3, computes a sound *over-approximation* of a transition system (represented by Horn clauses) and verifies whether an error state is reachable in the abstract system. If no error occurs in the abstract system, the algorithm reports that the original system is safe. Otherwise, if a path to an error state (counterexample) is been found in the abstract system, the corresponding concrete path is checked. If this latter path corresponds to a real execution of the system, then a real error has been found and is reported. Otherwise, the abstraction is refined in order to exclude the counter example, and the procedure continues.

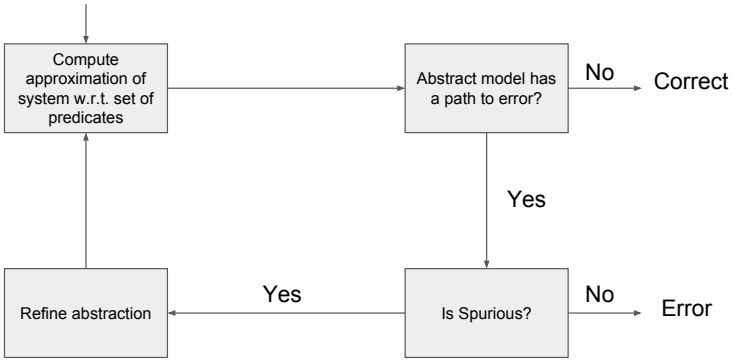


Figure 2.3: *Predicate abstraction counter-example guided approach*

By refinement we understand the process of enriching the predicate mapping used to construct the abstract system. The goal of refinement is to prevent spurious counterexamples (paths to an error state) from appearing in the abstraction. A key difficulty in the predicate abstraction approach is to automatically find predicates to make the abstraction sufficiently precise. A breakthrough technique [128, 129] is to generate predicates based on Craig interpolants [46] derived from the proof of unfeasibility of a spurious trace. To this end, an effective technique used in many predicate abstraction tools is that of interpolation.

Example 8 (Solving GCD)**.** *We return to the system of Horn clauses in Example 7. In this example, the abstraction of Horn clauses starts with a trivial set of predicates, each one assigned to $false$. We assume this to be a valid approximation and try to prove otherwise. Due to the existence of a clause that has a concrete satisfiable formula in its body (e.g. $M = N \wedge R = M$), we rule out $false$ as the approximation of gcd. In the absence of other candidate predicates, the approximation true is used (unassigned predicates). Using this approximation, we find that the error clause is no longer satisfied. At this point the algorithm checks whether a true error is reached by directly chaining the clauses involved in computing the approximation of predicates. This amounts to checking whether the following recursion-free subset of clauses has a solution:*

$$gcd(M,N,R) \leftarrow M = N \wedge R = M$$

$$false \leftarrow M \geq 0 \wedge M = N \wedge gcd(M,N,R) \wedge R > M$$

The solution to above problem is any formula $I(M,N,R)$ such that

$$I(M,N,R) \leftarrow M = N \wedge R = M$$
$$false \leftarrow M \geq 0 \wedge M = N \wedge I(M,N,R) \wedge R > M$$

This is in fact an interpolant of $M = N \wedge R = M$ and $M \geq 0 \wedge M = N \wedge R > M$. A valid interpolant is $P1(M,N,R) \models M \geq R$. Choosing this interpolant eliminates the current contradiction for Horn clauses and P1 is added into a list of abstraction predicates for the relation gcd. Because the predicates approximating gcd are now updated, we consider the abstraction of the system in terms of these predicates. The predicate $P1$ is not a conjunct in a valid approximation for gcd in the second clause*, so the following recursion-free unfolding is not solved by the approximation so far:*

$$gcd(M,N,R) \leftarrow M = N \wedge R = M$$
$$gcd'(M,N,R) \leftarrow M > N \wedge M1 = M - N \wedge gcd(M1,N,R)$$
$$false \leftarrow M \geq 0 \wedge M = N \wedge gcd'(M,N,R) \wedge R > M$$

Again an interpolant is used to update the set of predicates, and the process is repeated until 1) a genuine counter-example is found, or, 2) with our approximation we can not find a path to the head $false$. By following this approach eventually we will obtain an approximation of $gcd(M,N,R)$ assigned to $(M = N) \rightarrow (M \geq R)$ which will be a solution to these Horn clauses and hence we will prove safety.

2.3.3.1 Predicate Abstraction in Eldarica

We now describe in detail how the above process works in general on recursive constrained Horn clauses in the algorithm of the Eldarica tool. For readability we present a simplified version of some aspects of the algorithm, which can be found in full in [127]. In Chapter 5 we investigate how to improve the performance and convergence of such algorithms by improving the interpolants a theorem prover

produces. We also note the technique described in Chapter 5, can be used in various other methods that use Craig interpoalants e.g., [130, 131, 132] including top-down algorithms [132, 130] which use interpolants as a subsumption mechanism.

We first define the abstraction of the set of Horn clauses, namely, an abstract reachability graph (ARG). An ARG, is defined in Definition 5 and represents an over-approximated representation of our system of Horn clauses. The construction of the ARG is guided by the the mapping $\Pi : \mathcal{R} \rightarrow 2^P$ which maps relation symbols to a set of predicates that approximate the relation symbol.

Definition 5 (Abstract Reachability Graph (ARG))**.** *An ARG is a* hyper-graph *(S, E) where:*

- $S \subseteq \{(p, Q) \mid p \in \mathcal{R}, Q = \Pi(p)\}$ *is the set of nodes, each a pair consisting of relation symbol and a set of predicates*
- $E \subseteq S^* \times \mathcal{HC} \times S$ *is a hyper-edge relation, which each being labelled with a clause. For example, let* $E((s_1, \ldots, s_n), h, s)$ *then each* B_i *is* $p_i(\bar{t}_i)$ *in h, i.e., the body relational symbols and H is* $p(\bar{t})$*, and* $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$*.* $Q_i[\bar{t}_i]$ *is the predicate* Q_i *instantiated for the arguments* $\bar{t}_i$*.*

The algorithm in Algorithm 2 follows the predicate abstraction approach. It explores unwindings of Horn clauses and attempts to either find an real counter example or by a CEGAR algorithm, build on the predicate mapping Π in order to construct a closed ARG.

To find the solvability or unsolvability of $\mathcal{HC}$ Algorithm 2 selects a node and clause and builds an ARG until it finds a clause with a false head. It proceeds to build an ARG by adding edges that don't lead to false to an initially empty graph until it find a system of Horn clause that lead to the error, i.e., a clause with false as its head. The algorithm will build a potential interplation problem, or set of clauses forming a counter example, *cex* and determine if they are satisfiable or not. If they are satisfiable then it represents a *real* counter-example, i.e., a witness to the unsolvability of $\mathcal{HC}$, otherwise we have a real interpolation problem, and we call an interpolating theorem prover to extract an interpolant, which we use to

update Π and we remove the clauses in *cex* from the ARG. For certain logics, such as Presburger, the number of iterations, and its termination largely depends on the choice of interpolants we obtain from a theorem prover, i.e., the predicates we have in Π. Note, for infinite domain logics there are not termination guarantees in the procedure. In Chapter 5 we investigate this issue in detail.

```
   // Empty ARG
1  ARG(S = ∅, E = ∅)
2  Function Solve is
3      while true do
4          select clause h = (C ∧ p1(t̄1)∧, ..., ∧pn(t̄n) → H) ∈ HC;
5          and select node = (p1, Q1), ..., (pn, Qn) ∈ S;
6          s.t. ¬∃ edge = (((p1, Q1), ..., (pn, Qn)), h, s) ∈ E and
             C ∧ Q1[t̄1] ∧ ··· ∧ Qn[t̄n]¬ ⊨ false;
7          if such clause and nodes no not exist (closed) then
8              return HC solvable
9          end
10         if H = false then
11             cex = getCEX(h, node);
12             if cex is unsat then
                   // Interpolation problem
13                 preds = Interpolate(cex);
14                 add preds to Π;
15                 delete cex clauses from (S, E);
16             end
17             else
                   // Real counter example
18                 HC unsolvable, return cex;
19             end
20         end
21         else
               // Add edge to ARG
22             Let H = p(t̄);
23             Q = {φ ∈ Π(p) | {C} ∪ Q1 ∪ ··· ∪ Qn ⊨ φ};
24             e = (((p1, Q1), ..., (pn, Qn)), h, (p, Q));
25             S = S ∪ {(p, Q)};
26             E = E ∪ e;
27         end
28     end
29 end
```

Algorithm 2: Algorithm for solving recursive Horn clauses

Chapter 3

Synthesising Static Analysers From Logic

In this chapter we present a Datalog analysis framework that synthesises high performance analysers from a Datalog specification using partial evaluation. Much of this chapter based on work published for the compiler and verification tools community in Computer Aided Verification (CAV) [31], Compiler Construction (CC) [40], the Symposium on Principles and Practice of Parallel Programming [51], and the Soufflé tutorial [133] at Programming Language Design and Implementation (PLDI). This chapter described the overall architecture of Soufflé and its use of novel compilation techniques such as partial evaluation to produce fast and memory efficient static analysers that are used for several industrial applications, that we further expand in Chapter 6.

3.1 Design Goals

Datalog is a multifaceted language. On the one hand, it is viewed as an unusually powerful query language by the database community [134] and on the other hand, a limited but a tractable logic, by the formal methods community [72, 44]. For that reason, apart from databases querying [85, 86, 135], Datalog has been used for a diverse range of applications, that including Data Integration [65], Declarative Networking [67], Program Analysis [44], Network Analysis [72], Software Engineering [71]. While a use case does not impact the general semantics of Datalog, each use case has subtle differences in its ruleset and dataset characteristics that benefit from the different Datalog engine design choices.

In this thesis chapter, we describe the design and implementation of the Soufflé Datalog engine that targets static analysis. To obtain high performance, the design of Soufflé re-evaluates the core of the Datalog evaluation paradigm. As a result, we propose novel evaluation techniques, optimisations and extend the Datalog language for improved usability for specifying static analyses in Datalog. The result, is a robust production-strength tool that has been successfully used as a core static analysis engine in several large-scale industrial projects, including security analysis of Amazon virtual networks and program analyses for the Oracle JDK™. Both of these use cases are further explored in Chapter 6.

3.1.1 Performance

The major feature of the design of Soufflé centres around the observation that, in the case of static analysis, the Datalog rules are a *design-time* artefact. That is, when a static analysis is employed, it is reasonable to assume that the Datalog rules (static analysis specification) will remain largely *constant*. On the other hand, the input tuples (EDB) that represent the system to be checked (e.g., a Java program) by the static analysis will *vary* considerably as a static analysis is built to analysers many different systems. Therefore we use Datalog as an input logic specification to synthesise C++ analysers that adhere to this specification. Our framework uses partial evaluation as a core mechanism to perform the synthesis. The approach taken by Soufflé hence provides a *best of both worlds solution*, incorporating the performance of low-level hand-crafted analysers with the usability of DSL-based analysers [136, 37, 41].

3.1.2 Expressibility

Soufflé provides a balance between expressiveness and performance. While most of the language constructs are included in the definition of stratified Datalog, which can be evaluated efficiently, advanced features, such as arithmetic functors and data structure constructors, are provided to allow users to break out of the finite world assumption. These constructs are *add-ons*, and users can continue to use the basic Datalog if such levels of expressibility are not required. When high-levels of expressivity are required, e.g., to verify more precise properties in programs, we employ techniques can described in Chapter 5.

3.1.3 Usability

Another major design goal of Soufflé is to provide practical usability. We do this by employing two main design decisions. Unlike most Datalog engines that are geared towards high performance, we do not necessitate user annotations (e.g., indexing) to achieve high performance. Instead, we employ *auto-optimisations* to avoid manual user intervention wherever possible. This is particularly important for the static analysis use case. Unlike database queries, the size of the rulesets in static analyses

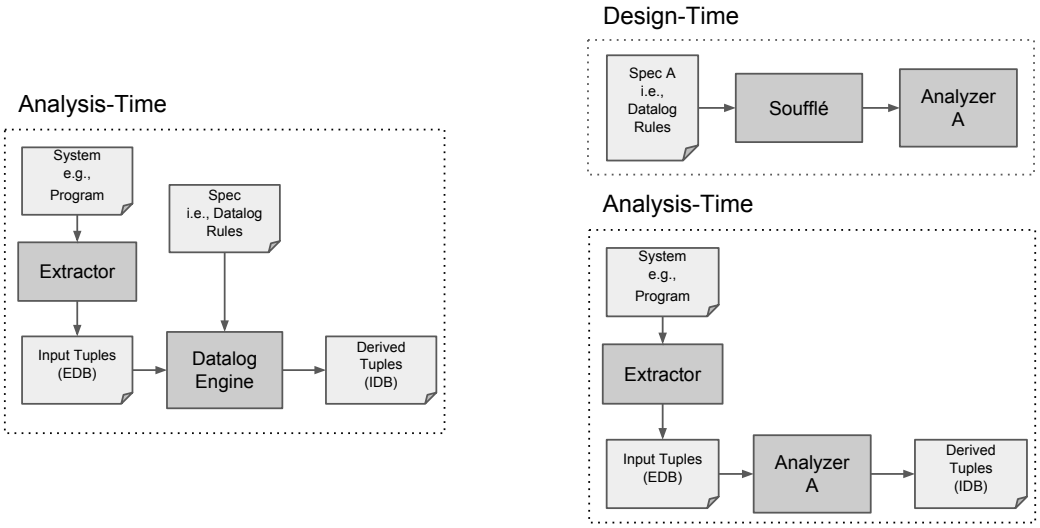
can run to hundreds or even thousands [28] of items. In Chapter 4 we elaborate on an important auto-optimisation that makes Soufflé unique compared to existing Datalog engines. Another aspect of usability that is provided by language constructs that allow modular specifications and promote code reuse, in addition to other good software engineering practices. Since static analysis rulesets are significantly larger than traditional database queries, this becomes a crucial usability feature.

3.2 Framework Overview

This section provides an overview of the Soufflé framework. In later sections we elaborate on individual components and stages. The overall paradigm employed in Soufflé is depicted in Figure 3.1. Here the Soufflé paradigm (Figure 3.1b) is contrasted with to the standard Datalog-paradigm used by typical Datalog engines for static analysis. In the standard Datalog-based static analysis setup, an analysis specification is defined by a, Datalog program, i.e., a set of Datalog rules. An input system to be analysed is translated to a set of facts (i.e., the EDB), by an extractor, e.g., Soot [137]. A Datalog engine then computes the analysis result from both the Datalog rules and input facts and produces a set of derived relations (IDB) from which the analysis answer can be obtained. The Soufflé approach, depicted in Figure 3.1b first creates an analyser from the Datalog rules. This analyser can then be executed to read in a set of EDB relations, for a matching schema and computes the set of IDB output relations.

To efficiently implement the approach, as shown in Figure 3.1b, Soufflé employs a multi-stage partial evaluation pipeline to translate Datalog rules to efficient, parallel C++ code. A notable advantage of this approach is that, after each partial evaluation is performed, further optimisation opportunities arise. As a result, an efficient static analyser is produced that correctly implements the logic specification and performs on a par with hand-crafted alternatives.

The stages of partial evaluations, are depicted as a hierarchy in Figure 3.2a. Here each stage in the hierarchy is formalised as a *first-order Futamura projection* [98] equation. The Futamura projection produces specialised code with respect



(a) *Datalog paradigm* ***(b)*** SOUFFLÉ *paradigm*

Figure 3.1: *Datalog analysis paradigms*

to an interpreter that computes the same results as if the input source program was evaluated with the interpreter. For example, Figure 3.2 derives a first-order Futamura projection. It asserts the equivalence between an interpretation and an execution of a partially evaluated program.

In terms of implementation, each stage is mapped to a translator component in the SOUFFLÉ architecture, as depicted in Figure 3.3. For each stage, a new language representation is needed, i.e., AST, RAM, or templatised C++, to model the analysers at a level of granularity required at that stage of specialisation.

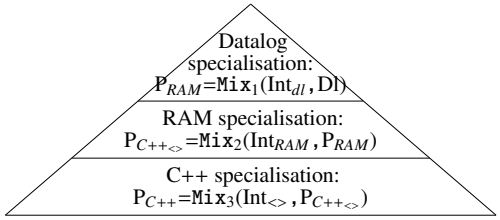
Datalog to AST: The framework proceeds by first parsing and translating a Datalog specification is into an abstract syntax tree (AST). In this translation step, semantic checks are conducted, including the asserting the relation symbols are used correctly, type checks for proper use of variables, and checks for cyclic, non-stratified negations among many other semantic checks. After the semantic checks are performed, several optimisations are applied to enable improved performance.

AST to RAM: Next, the AST which represents a declarative Datalog program is *specialised* into an imperative Relational Algebra Machine (RAM) program. The

lowering from a declarative Datalog program to an imperative RAM program is performed by re-interpreting the semi-naïve evaluation as a translation scheme [90] and applying a specialisation via a first-order Futamura projection. The RAM representation of an input program offers ample opportunity to apply optimisations. Unlike the case for the AST level, mid-level optimisations may target details of the evaluation process not visible in the declarative specification. Among the most important optimisations at this stage is join scheduling. Here, SOUFFLÉ employs the scheduler infrastructure too discover advantageous loop orders in the RAM using instance specific cost models.

RAM to Templatised C++: The next stage translates the RAM program to templatised C++. Here, a second specialisation is performed with respect to a RAM interpreter. The specialisation converts a simple traversal over relations to indexed searches that are derived from the input RAM program. The main challenge of this translation step is in the generation of efficient, high performance C++ code for processing and storing information into in-memory relations. In our framework, we obtain adequate performance by heavy use of C++ templates, tailored to relational algebra operations and efficient use of data structures including various types of index schemes. For specific instances of relations and operations we permit customisations of the code in the templates to achieve maximal performance. Thus, essentially, a large part of the actual code generation is deferred to the final translation, i.e., the C++ compiler that translates the heavily templatised input program to an executable program. The technique of scripting the generation of code using C++ templates is also known as template meta-programming [138]. For example, if the actual type of an object is known at compile-time the dynamic dispatch is converted to a static call, vastly improving vastly the performance of the compiler. In our specific use case, the meta-programming becomes a partial evaluator that pushes computations from runtime to compile-time.

Compilation: In the final stage, the resulting C++ code is compiled into a binary executable. The C++ compiler unfolds the template, producing highly efficient assembly code that is specialised for a given input program. Using C++ makes the



(a) Specialisation hierarchy

$$\begin{aligned}\texttt{result} &= [\![\texttt{Int}]\!](\texttt{Source},\texttt{Input})\\ &= [\![\texttt{Source}]\!]_{Int}(\texttt{Input})\\ &= [\![\texttt{Mix}(\texttt{Int},\texttt{Source})]\!]\texttt{Input}\\ &= [\![\texttt{Prog}]\!]\texttt{Input}\end{aligned}$$

Figure 3.2: *Application of Futamura's projection*

Datalog compilation independent of the actual target architecture.

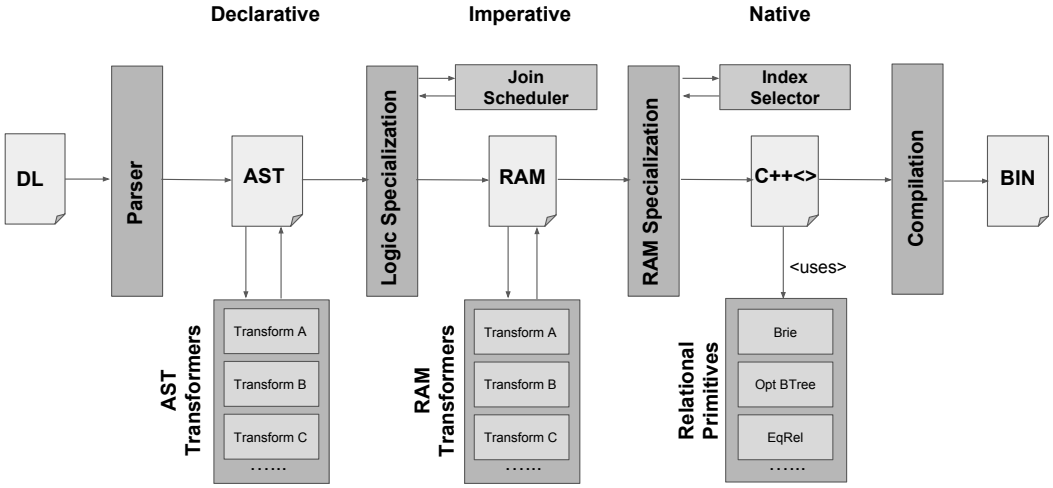


Figure 3.3: SOUFFLÉ *architecture*

3.3 Frontend

This section describes the SOUFFLÉ frontend, i.e., the SOUFFLÉ language and the conversion from syntax to an optimised abstract syntax tree (AST).

3.3.1 Extending Datalog for Static Analysis

The SOUFFLÉ language was designed with large industrial static analyses in mind. While the language implements the basic features of Datalog e.g., Datalog with stratified negation and aggregation (See [139, 90], [41] and [37]), the fact remains that Datalog was originally designed with database querying in mind. Non-trivial static analyses, on the other hand, require additional features to better express static analysis specifications as well as to aid in user productivity. Therefore the standard Datalog language is extended to accommodate large-scale static analysis use

cases [140]. In addition, the language design decisions take into account the fact that Soufflé, unlike most Datalog engines, synthesises an analyser and as such, does not dynamically execute queries. Therefore, considerations such as a solid I/O systems, static typing etc., are paramount to its design. The non-standard features of Soufflé are summarised below.

3.3.1.1 Type System

Types for logic programming are non-standard; however, for large Datalog specifications a rich type system is paramount. Large projects typically require several hundreds of relations (e.g. Doop) and tool support is needed to ensure that programmers don't bind wrong attribute types. For this reason, Soufflé provides a type system that is static. All attributes in a relation declaration need to be typed and these types are then enforced at translation time. We avoid dynamic checks at runtime as evaluation speed is paramount in static analyses.

Soufflé's type system is built with two primitive types, namely, the symbol type and the number type. The symbol type is defined as the universe of all strings. Internally, it is implemented by an ordinal number which can be accessed using the `ord(`<*string*>`)` construct. The number type is the universe of all numbers, i.e., simple signed numbers set to 32 or 64 bit. Symbol and number types can be declared with `.number_type` <*name*> or `.symbol_type` <*name*> constructs, respectively. In addition, Soufflé provides the means for defining user-defined types using the `.type` directive. Moreover, Soufflé allows the user to construct type hierarchies via union types using the following syntax:

`.type` <*ident*> `=` <$ident_1$> `|` <$ident_2$> `|` ... `|` <$ident_k$>

For example, the code `.type A = B | C`, creates a type A that is either a B or C but not both.

3.3.1.2 Functors

The Soufflé language has the ability to perform computations in numerical domains. Support for *functors* is thus provided to aid in computations in this domain, including arithmetic, bit-vectors etc. For example, the rule `P(a+1):- G(a)` is valid

in Soufflé. Additionally, several string functors are supported such as concatenation, length and substring. Functors significantly extend the Datalog semantics, allowing non-terminating Datalog programs, i.e., infinite relations can be defined. However, the underlying evaluation algorithm remains the same since properties such a monotonicity still hold, i.e., we have infinite, chains of increasing sets of tuples.

Functors can have several practical benefits for static analysis, including dealing with arithmetic operations in networks (See chapter 6), context increments in points-to analysis [28] among many other applications. However, they must be used sparingly when the semi-naïve evaluation algorithm is used, due to their potential to cause non-termination. In Chapter 5 we provide an approach to solving Horn clauses with numerical constraints using model checking techniques that is a potential candidate for rules that require heavy use of functors and numerical constraints.

3.3.1.3 Records

Relations are two dimensional structures in Datalog. Large-scale problems often require more complex structures. The Soufflé language has the ability to construct objects that break out of the flat Datalog world. A data structure can be generated using records with the following syntax:

```
.type <name> = [<name_1>: <type_1>, ..., <name_k> : <type_k>]
```

This construction can be used to form lists, trees and other data structures. Using logic rules, these data structures can be augmented, traversed, etc., similar to functional programming. In addition, records can be very powerful for implement complex domains of computation e.g., intervals.

```
.type list = [val: number , tail: list]
.type tree = [val: number , l: tree, r: tree]
```

Data structures are implemented by providing a hidden reference type in a relation for each data structure type. This translates the elements of a data structure into a number. During evaluation, if an element does not exist, it is created on the fly. Semantically, data structures are relations containing references that grow

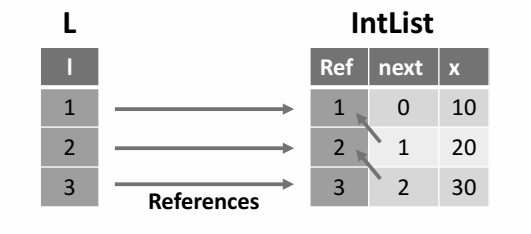


Figure 3.4: *Relational representation of a list using records*

monotonically and structural equivalence is determined by identity with new elements created on the fly. We note however, that Datalog lookup for data structures comes at the cost of performance, as an extra lookup is necessary. For example, the program below builds a list of numbers:

```
.type IntList = [next: IntList, x: number]
.decl L(l: IntList)
L([nil,10]).    L([r1,x+10]) :- L(r1), r1=[r2,x], x <
   30.
.decl Flatten(x: number)
Flatten(x) :- L([_,x]).
```

Figure 3.4 illustrates the layout of a list in a in-memory relation. Here, `intList` is a set of references (*Ref* field) that is used to as a value in the next field that is itself of type `intList`. In this way we can build/traverse the list data structure.

As we can see in extended example in Figure 3.5, records can have practical benefits such as defining traces, usage in context sensitive points to analysis [28], and even the potential to define lattices (See Chapter 7).

3.3.1.4 Components

Large logic programs often have little structure. Such programs consist of unstructured sets of rules. For large-scale static analyses specifications this creates serious software engineering challenges. To rectify this, Soufflé provides support for components. Components provide support for encapsulation, i.e., separation of concerns, replication of code and adaption of code. Components can be seen as a form of meta semantics for Datalog. Similar to C++ templates, the templatised Datalog

code is expanded at translation time but generates new instantiations of the templatised code substituted with input values. Components are first defined with the following syntax:

```
.comp <name>[<params, ...>][:<super-name>_1[<params, ...>],...
   ,<super-name>_k [<params,...>]]{<code>}
```

To use a component, a component needs to be instantiated:

```
.init <name> = <name>[<params,...>]
```

Each component instantiation has its own name to create a namespace and type and relation definitions inside the component inherit the namespace. Note that definitions permit embedded component definitions as well. Similar to classes in C++, this results in an embedded namespace. The translation of components to standard Datalog is shown in the example:

```
.symbol_type s
.decl A(x:s, y:s) .input A
.comp myC {
  .decl B(x:s, y:s) .output B
  B(x,y) :- A(x,y). }
.comp myCC: myC {B(x,z) :- A(x,y), B(y,z).}
.init c = myCC

// outer scope: no name space
.decl A(x:s, y:s) .input A
// name scoping
// B is declared inside myC/myCC
.decl c.B(x:s, y:s) .output c.B
c.B(x,y) :- A(x,y).
c.B(x,z) :- A(x,y), c.B(y,z).
```

Here, two components are defined where one component inherits from another. Component `myCC` adds an additional rule to `myC`. We instantiate `myCC` and label it as `c`. Soufflé then instantiates the rules from both components using `c` as a prefix.

Example 9. *In Figure 3.5, we extend the Datalog static analysis of Example 1.1 with an extended static analysis that uses* Soufflé *language extensions. While the static analysis in Example 1.1 gives us a list of insecure nodes, we may desire more information, such as a program trace. Using standard Datalog, this can be awkward to define. A user would be required to define several new relations and rules, many of which are redundant. However, using* Soufflé*'s extensions we can use components to extend the analysis and encode traces into a list data structure.*

We first wrap the analysis of the motivating example in a `Base` *component. Here we can instantiate the analysis of different types of data. The analysis is instantiated in the second file with the line* `.init A1 = Base<Node1>` *where the type* `Node1` *is given as an argument. In the analysis of* analysis2.dl*, we instantiate the* `Derived` *analysis as an* `A2` *object. The* `Derived` *component inherits from the* `Base` *component, meaning that all rules are accessible to* `A2` *as in* `A1`*. However, the* `Derived` *component defines an additional analysis. This analysis keeps track of the trace of all insecure nodes (in case several exist) and the length of edges traversed up to a user specified value of K. To do this, we define a constructor in the line* `.type Tr = [v : N, tail: Tr]`*. Note that this is a recursive list-like definition. Also note that, when we instantiate* `A2`*, we instantiate it with a super set type* `Node`*, which is a union of types* `Node1` *and* `Node2` *and a value K. The derived analysis contains two rules. The first rule represents the base case, and here we add s as the head of the list and keep the tail as* `nil` *(empty list). We initialise the edge size as* 0*. The next rule, increments the edge by* 1 *and adds a node to the head of the list, if it is not a protected node.*

□

3.3.2 Datalog to AST Transformation

The first stage of the pipeline in Figure 3.3 is the construction of the AST. This process is described in more detail in Figure 3.6. Here, a Datalog program along with configuration parameters is parsed using standard bottom-up parsing techniques and then converted into an *AST translation unit*. An AST translation unit represents an AST with its translation state, i.e., its symbol table, set of transformations, debug

```
// file analysis1.dl

.type Node1

.comp Base<N> {
  // Interface
  E(s:N,d:N)
  .input E
  S(s:N)
  .input S
  P(node:N)
  .input P
  I(node:N)
  .output

  I("s").
  I(y) :- I(x), E(x,y),
          !P(y).
}
```

```
// file analysis2.dl

#include "analysis1.dl"
.type Node2
.type Node = Node1 | Node2

.comp Derived<N, K> : Base<N> {
  .type Tr = [val : N, next: Tr]
  T(v: number, ls: Tr)
  .output T

  T(0, ["s", nil]).
  T(v+1, [y, r1]) :-
    T(v, r1),
    v < K,
    r1 = [x, tail]
    E(x,y), !P(y).
}

.init A1 = Base<Node1>
.init A2 = Derived<Node, 10>
```

Figure 3.5: *Extended static analysis from example 1.1*

and error reports.

Soufflé contains a set of transformations that aim to produce more efficient code. Where, improvements are always guaranteed, the transformation is executed by default, otherwise users typically must specify its use when invoking Soufflé. As in traditional compilers, Soufflé performs AST semantic checks. These transformers, do not transform the code per say, but instead perform various semantic analysis such as consistency checks, reporting an error if the AST representing the Datalog program is malformed. Likewise, Soufflé performs a lowering high-level ASTs (e.g., defining templates and components) and converts these high level syntactic features to low-level ASTs, i.e., vanilla Datalog. However, several transformer target performance. The key transformations are summarised below.

- Nullary relations transformation: this transformer avoids computation of a derived relation if only an existence of a tuple is required

- Constant propagation: this transformer forward propagates constant values within and among rules

- Alias elimination: this transformer unifies variables according to equality constraints

- Rule elimination: this transformer eliminates rules that are positive empty relations in bodies as the entire rule is not computed in this case.

- Relation elimination: this transformer eliminates relations and their rules if they do not contribute towards result

- Magic set transformation: syntactic top down propagation of queries [42]. Given literals in rules the literals are propagated from head to body thus resulting in more efficient evaluation.

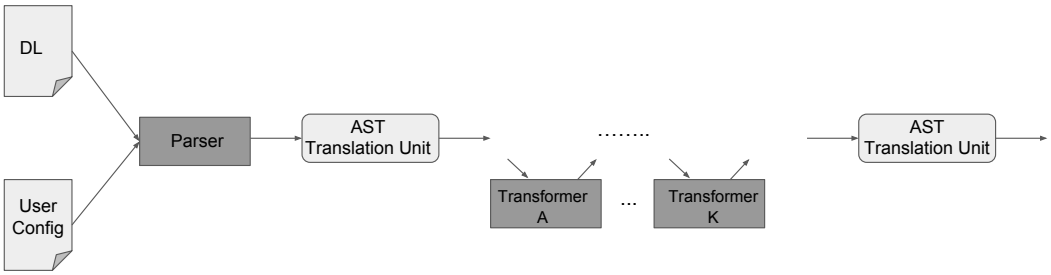


Figure 3.6: SOUFFLÉ *architecture*

3.4 Logic Specialisation

In this section, we describe the process of specialising Datalog (as an AST) into a RAM program. This is the first specialisation of the pipeline in Figure 3.3. The purpose of this step is to produce an imperative representation of the Datalog code, remove any interpreting overhead and enable further optimisations. To do this, we employ partial evaluation to inject aspects of the *evaluation algorithm* into each clause of the Datalog program.

3.4.1 Mechanisms for Datalog Evaluation

We first detail Datalog's evaluation mechanism, for which we propose a variation to the standard semi-naïve algorithm to achieve greater efficiency.

3.4.1.1 Relaxed Semi-naïve Algorithm

The existing Datalog evaluation algorithms were designed in the 80s and 90s with a focus on minimising computation time but do not consider data transfer as a cost

caused by copying large temporary tables for recursively defined relations. As a consequence, the presented standard semi-naïve algorithm in Algorithm 1 (see Chapter 2) does not perform well in regard to large-scale static analysis. The same observation has been made in [94]. Typically, the relation sizes can be as large as giga tuples. Hence, the *book-keeping* of the current, previous and delta knowledge becomes prohibitive due to the involved copy operations, e.g., current knowledge of the previous iteration becomes the previous knowledge of the current iteration and so on. To overcome these book-keeping costs of the standard semi-naïve algorithm, (1) we introduce a slight computational deficiency by replacing the previous knowledge by the current knowledge, and (2) we unroll the fixed-point iteration by two iterations to eliminate the copy operation for new knowledge and delta knowledge, i.e., a write-after-write dependency is resolved by this renaming.

The substitution of previous knowledge by current knowledge is performed by replacing the evaluation $Eval_i^{(j)}[I \cup \langle R_1, \ldots, R_{k-1}, \Delta R_k, P_{k+1}, \ldots, P_n \rangle]$ by $Eval_i^{(j)}(I \cup \langle R_1, \ldots, R_{k-1}, \Delta R_k, R_{k+1}, \ldots, R_n \rangle)$. The effect of this replacement is that more computations may be required, however, the relations P_i can be omitted from the semi-naïve algorithm and no copy operation $P_i := R_i$; for all relations in C are required.

The loop unrolling approach rewrites the fixed-point loop of the semi-naïve algorithm as outlined in Algorithm 3. The original fixed-point loop is unrolled twice. By unrolling the copy operations between delta and new knowledge can be eliminated. In one unrolled iteration the relation AR_i represents the delta of the previous iteration and BR_i the new knowledge. In the second unrolled iteration the roles of AR_i and BR_i are swapped to new knowledge and delta knowledge, respectively. To ensure correctness, we assume that in the initialisation phase the ΔR_i relations are renamed to AR_i so that the first unrolled iteration has initial data to process.

The proposed relaxed semi-naïve algorithm reduces the book-keeping overheads, since for each relation one of the intermediate relations is omitted (i.e. previous knowledge) and two copy operations are eliminated minimising the data traffic on the memory bus.

Lemma 3 (Correctness of Algorithm 3)**.** *Algorithm 3 produces the same result as*

```
1  for ever do
2      for i=1 to n do
3          BR_i := ∅ ;
4          for j=1 to m_j do
5              BR_i := BR_i ∪ ⋃_k Eval_i^(j)(I ∪ ⟨R_1, ..., R_{k-1}, AR_k, R_{k+1}, ..., R_n⟩) \ R_i
                ;
6          end
7      end
8      if Σ_{i=1}^n |BR_i| = 0 then
9          exit loop
10     end
11     for i=1 to n do
12         R_i := R_i ∪ BR_i ;
13     end
14     for i=1 to n do
15         AR_i := ∅ ;
16         for j=1 to m_j do
17             AR_i := AR_i ∪ ⋃_k Eval_i^(j)(I ∪ ⟨R_1, ..., R_{k-1}, BR_k, R_{k+1}, ..., R_n⟩) \ R_i
                ;
18         end
19     end
20     if Σ_{i=1}^n |AR_i| = 0 then
21         exit loop
22     end
23     for i=1 to n do
24         R_i := R_i ∪ AR_i ;
25     end
26 end
```

Algorithm 3: Improved semi-naïve algorithm for reducing copy overheads

Algorithm 1.

Proof. The correctness of the computation will not be affected since the invariant that the previous knowledge is a subset of the current knowledge is maintained. By unrolling we only compute 2 staged delta computations that are chained and hence do not affect the overall monotonicity of the computation. □

3.4.1.2 Clause Evaluation

The *Eval* function in Algorithms 1 and 3 evaluates a clause in a Datalog program. In modern query systems, clause evaluation is often implemented by a variation of a *nested-loop join*. For presentation simplicity, we partition the sequence of body

atoms of a Datalog rule into positive (referred to as R_i^+) and negative (referred to as R_j^-) occurrences (i.e., negative if it is negated in the body), and restate the above Datalog rule as:

$$R_0(X_0) \text{ :- } R_1^+(X_1), \ldots, R_h^+(X_h), R_{h+1}^-(X_{h+1}), \ldots, R_d^-(X_d).$$

where h is the number of positive atoms.

However, we note that the ordering may change due to levelling optimisations that hoist the negative predicates to outer loops for performance reasons.

```
1  for all t_1 ∈ σ_{φ_1(X_1)}(R_1^+) do
2      for all t_2 ∈ σ_{φ_2(t_1,X_2)}(R_2^+) do
3          ...
4          for all t_h ∈ σ_{φ_h(t_1,...,t_{h-1},X_h)}(R_h^+) do
5              if σ_{φ_{h+1}(t_1,t_2,...,t_h)}(R_{h+1}^-) = ∅ then
6                  ...
7                  if σ_{φ_d(t_1,t_2,...,t_h)}(R_d^-) = ∅ then
8                      if π(t_1,...,t_h) ∉ R_0 then
9                          add π(t_1,...,t_h) to R_0
10                     end
11                 end
12             end
13         end
14     end
15 end
```

Algorithm 4: Nested-loop joins for evaluating a Datalog rule

In the nested-loop joins, we iterate (denoted by the **for all** construct) over tuples that are obtained from a *primitive search*, which will be defined shortly, on a positive relation; this semantics comes from the implicit universal quantification in a Datalog rule. Then, negative occurring atoms are tested for emptiness with respect to primitive searches; this semantics stems from the implicit non-existence quantification of attributes of negative body literals in a Datalog rule. Finally, the most inner operation projects (denoted by π) the selected tuple into the head atom if the tuple does not already exist in the relation. This existence check is performed to ensure that tuples are not inserted twice into a relation; that is, it enforce the set

semantics of the relations.

Definition 6 (Primitive Search)**.** *A primitive search has the following form:*

$$\sigma_{x_1=v_1,\ldots,x_k=v_k}(R_i) = \{t \in R_i \mid t(x_1) = v_1, \ldots, t(x_k) = v_k\}.$$

Here, R_i is a relation and $x_1 = v_1, \ldots, x_k = v_k$ is a search predicate*, where $x_1, \ldots, x_k$ are attributes and $v_1, \ldots, v_k$ are constants.*

Note that, $\{x_1, \ldots, x_k\}$ does not necessary have to be the first k attributes of the relation R_i. A primitive search extracts all tuples from a relation that adhere to the *search predicate*. The constants $v_1, \ldots, v_k$ in a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R_i)$ are obtained either from X_i of the atom $R_i(X_i)$ or from other tuples in relations further up the nested-loop joins (i.e., R_j for $0 < j < i$). As an alternative notation, we denote $\sigma_{\varphi(t_1,\ldots,t_{i-1},X_i)}$, where $\varphi \equiv x_1 = v_1, \ldots, x_k = v_k$, as the substitution of $t_1, \ldots, t_{i-1}, X_i$ for appropriate constants v_1 to v_k.

3.4.2 Relational Algebra Machine

Next, we define the Relational Algebra Machine (RAM) language. RAM is the target language of the first specialisation. RAM is an abstract machine that we have developed for Soufflé that we use as a semantic model for evaluating translated input programs. The machine is specifically tailored to execute relational algebra programs that are produced by the semi-naïve evaluation. The RAM program contains relational algebra operations to compute results produced by clauses and has the ability to efficiently model Datalog fixpoint evaluation schemes through imperative constructs including statement composition for sequencing the operations, and loop construction with exit conditions. Additionally, RAM contains relation management operations to keep track of previous, current and new knowledge as required by efficient evaluation schemes such as the semi-naïve evaluation.

3.4.2.1 Execution Model

The abstract machines operates solely on relations and have no notion of variables and/or memory. Thus the evaluation of a RAM program entails maintaining a col-

lection of relations $\langle R_1, \ldots, R_k \rangle$ as a *state* for executing a RAM program. The relations $R_1, \ldots, R_k$ are fixed throughout the execution of a RAM program, i.e., no new relation is added to or deleted from the state whilst executing the program. However, the contents of a relation may change. There is a set of relations that the program operates on, some of which are pre-loaded with data, e.g., the tuples defined by the facts in the original input program. We define the RAM *state s* as a map between the relation names in the Datalog program and a sets of tuples the defining the relation, i.e., $(R_1 \mapsto \{t_1, t_2, \ldots\}, \ldots, R_n \mapsto \{t_1, t_2, \ldots\})$. Given a state s, $s[R]$ denotes a map access, accessing the element mapped to R. The notation $[R \mapsto e]$ denotes a *map update*, i.e., replacing the value mapped to R with e. Note, two Relations can be simultaneously updated as follows: $[R_1 \mapsto e_1, R_2 \mapsto e_2]$. Maps are closed under intersection, union, and compliment. τ denotes a variable to value set mapping. A mapping is assigned by $\tau \leftarrow_t S$ where t is mapped in τ to a set of values S.

3.4.2.2 Syntax and Semantics

In the design of RAM, we attempt to limit expressivity in order to avoid errors in translation and yet it must be expressive enough to model all required constructs. Therefore, we should employ sufficient constructs to represent the Datalog evaluation mechanisms described above. The RAM constructs are divided into control flow statements, operations, relational management and values and conditions. The control flow constructs allow a RAM program to model the iteration of the semi-naïve algorithm. Operations allow for the modelling of nested-loop joins for clause evaluation and relational management allows modelling of the book-keeping aspects of the semi-naïve algorithm.

Control Flow. The RAM syntax is defined in Figure 3.7. RAM has two statements for control flow, i.e., sequences of statements, and a loop statement with multiple exit statements. The sequencing of statements $S_1;S_2$ is necessary to order computations of relations that depend on each other. The order among relations stems from the strongly connected component graph of the dependencies between relations [90]. Loops constructions are necessary for computing fixpoints of recursively defined relations. Mutually recursive relations are congregated in a single strongly

$$
\begin{aligned}
&S \in \textbf{Stmt} \rightarrow \textbf{loop}\, S_1;\ [\textbf{exit}\ C_1;] \ldots S_n;\ [\textbf{exit}\ C_n;]\, \textbf{end} \\
&S \in \textbf{Stmt} \rightarrow S_1 \textbf{;} S_2 \\
\\
&S \in \textbf{Stmt} \rightarrow \textbf{merge}\, R_1\, \textbf{into}\, R_2 \\
&S \in \textbf{Stmt} \rightarrow \textbf{swap}\, R_1\, , R_2 \\
&S \in \textbf{Stmt} \rightarrow \textbf{purge}\, R \\
\\
&S \in \textbf{Stmt} \rightarrow \textbf{insert}\, O \\
&O \in \textbf{Oper} \rightarrow \textbf{search}\, R\, \textbf{as}\, t\, [\textbf{where}\, C]\, \textbf{do}\, O \\
&O \in \textbf{Oper} \rightarrow \textbf{project}\, (V_1, \ldots, V_k)\, \textbf{into}\, R \\
\\
&C \in \textbf{Cond} \rightarrow C_1\, \textbf{and}\, C_2 \\
&C \in \textbf{Cond} \rightarrow V_1\, rel\, V_2 \\
&C \in \textbf{Cond} \rightarrow \textbf{notexists}\, R(V_1, \ldots, V_k) \\
\\
&V \in \textbf{Value} \rightarrow R.v \\
&V \in \textbf{Value} \rightarrow t(v) \\
&V \in \textbf{Value} \rightarrow \textbf{count}(R) \\
&V \in \textbf{Value} \rightarrow const
\end{aligned}
$$

Figure 3.7: *RAM BNF grammar definition*

connected component and the computations of the clauses of the relations are iterated until no further knowledge can be obtained. The semantic function for control statements takes a function with a state s as an argument, which is defined as a mapping of the relation names to sets of tuples. The control flow loop is defined as the least fixpoint of the function $\mathcal{F} : (\mathcal{S} \rightarrow \mathcal{S}) \rightarrow (\mathcal{S} \rightarrow \mathcal{S})$, as shown below:

$$
\mathcal{F}(\alpha)(s) = \begin{cases} \alpha(\mathcal{S}[\![S_i]\!]s) & \text{if } \neg C[\![C_k]\!]s \text{ for all } C_k \text{ where } k < i \\ s & \text{otherwise} \end{cases}
$$

Here, we execute a statement if all of its previous conditions didn't trigger an exit. The sequence statement is defined by the composition of two statement executions. This type of control flow models the fixpoint characteristics of the loop in the semi-naïve algorithm.

$$\mathcal{S}[\![\textbf{loop}\, S_1; [\textbf{exit}\, C_1;] \ldots S_n; [\textbf{exit}\, C_n;]\, \textbf{end}]\!] ::= lfp(\mathcal{F})$$
$$\mathcal{S}[\![S_1; S_2]\!] ::= \lambda s.\mathcal{S}[\![S_2]\!](\mathcal{S}[\![S_1]\!]s)$$

$$\mathcal{S}[\![\textbf{merge}\, R_1\, \textbf{into}\, R_2]\!] ::= \lambda s.s[R_2 \mapsto s[R_2] \cup s[R_1]]$$
$$\mathcal{S}[\![\textbf{swap}\, R_1, R_2]\!] ::= \lambda s.s[R_1 \mapsto s[R_2], R_2 \mapsto s[R_1]]$$
$$\mathcal{S}[\![\textbf{purge}\, R]\!] ::= \lambda s.s[R \mapsto \emptyset]$$

$$\mathcal{S}[\![\textbf{insert}\, O]\!] ::= \lambda s.\mathcal{O}[\![O]\!]s\ \tau'$$
$$\mathcal{O}[\![\textbf{search}\, R\, \textbf{as}\, t\, [\textbf{where}\, C]\, \textbf{do}\, O]\!] ::=$$
$$\lambda s.\lambda\tau.\mathcal{O}[\![O]\!]s\ (\tau \leftarrow_t \{v \in R \mid \mathcal{C}[\![C]\!](v)\})$$
$$\mathcal{O}[\![\textbf{project}\, (V_1, \ldots, V_k)\, \textbf{into}\, R]\!] ::=$$
$$\lambda s.\lambda\tau.s[R \mapsto ([\![V_1]\!]\tau \times \ldots \times [\![V_k]\!]\tau)]$$

$$\mathcal{C}[\![C_1\, \textbf{and}\, C_2]\!] ::= \lambda s, \tau.\mathcal{C}[\![C_1]\!]\tau, s \wedge \mathcal{C}[\![C_2]\!]\tau, s$$
$$\mathcal{C}[\![V_1\, rel\, V_2]\!] ::= \lambda s, \tau.\mathcal{V}[\![V_1]\!]\tau, s \wedge \mathcal{V}[\![V_2]\!]\tau, s$$
$$\mathcal{C}[\![\textbf{notexists}\, R(V_1, \ldots, V_k)]\!] ::=$$
$$\lambda s, \tau.(\mathcal{V}[\![V_1]\!]\tau\ s, \ldots, \mathcal{V}[\![V_k]\!]\tau\ s) \notin s[R]$$

$$\mathcal{V}[\![R.v]\!] ::= \lambda s, \tau.R.v$$
$$\mathcal{V}[\![t(v)]\!] ::= \lambda s, \tau.\tau(t)(v)$$
$$\mathcal{V}[\![\textbf{count}(R)]\!] ::= \lambda s, \tau.card(s[R])$$
$$\mathcal{V}[\![const]\!] ::= \lambda s, \tau.const$$

Figure 3.8: *RAM semantics*

Relational Management. The RAM statements for relational management are defined as the next three constructs in Figures 3.7 and 3.8. The statement **merge** adds all of the tuples of relation R_1 to relation R_2. The statement **purge** deletes all tuples in relation R. The statement **swap** swaps the contents of two relations. Statements can be sequenced by a semicolon $S_1; S_2$ such that S_1 is executed prior to S_2.

Example 10 (Semi-Naive)**.** *Here we show how we can describe a semi-naïve iteration using RAM.*

```
1 insert (number(0)) into I
2 merge I into ΔI;
3 loop
4   ...
5   exit I' ≠ ∅;                                    □
6   merge I' into I;
7   swap ΔI, I';
8   purge I'
9 end loop;
```

Nested-Loop Joins. The insert statement is used to model rule evaluation. To evaluate inserts, we instantiate a new *loop state* τ', where the prime denotes a new empty map. This map stores tuple names to sets of tuples. An important feature of a RAM program is its ability to express *nested-loop joins*. To implemented nested-loop joins an **insert** statement contains a relational algebra operation O, that combines cross-product, selection and projection operations.

The operations in an insert are defined in the next two lines in Figure 3.7 and 3.8 which defines their syntax and semantics, respectively. The **search** traverses over all tuples in relation R, and tests whether, for a tuple t, the condition C holds. If it holds, the attached operation O is executed recursively, passing on the currently selected tuple of the traversal and the selected tuples of the outer traversals. If the condition does not hold, the operation O is skipped and the next tuple is assessed until the end of the relation is reached. The condition C is referred to as a *primitive search condition*. It is a restricted formula, as defined in Definition 6, consisting of a conjunction of equality predicates with right-hand-side attribute variables $t.v$ from the tuple $t \in R$ in the search, and left-hand-side *constant* $t_j.v_j$ obtained from a tuple from further up the nested-loop join. In the semantics of Figure 3.8 a search updates the nested-loop state by mapping the tuple t to the filtered tuples of the search.

The **project** operation selects a set of attribute variables $t_i.v_i, \ldots, t_k.v_k$ from the tuples in the relations in the nested-loop join and projects their values onto the target relation R_1. In the semantic definition, we now update the global state, as the

relation projected onto may not be in the nested-loop join traversal. The syntax of a condition used for the **search** operation as well as other statements is listed next in Figures 3.7 and 3.8.

A condition can be a conjunction of conditions, a binary relation over two values, for which *rel* is either one of the following binary relations: $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$, or a check on whether the tuple $(V_1, \ldots, V_k)$ can be found in relation R and represents an existence check of a tuple in a relation. We refer the reader to [141] to see that relational algebra is indeed expressed in the semantics of RAM operations.

A value can have the following syntax and semantics defined in the remaining of the definitions of Figures 3.7 and 3.8. Here, a value can be a reference to an attribute variable of a relation $R.v$, a tuple value $t.v$, a number of tuples in a relation or a constant value. We further clarify the semantics of nested-loop joins with an example:

Example 11 (RAM Nested-Loop Join)**.** *Consider a non-recursive Datalog rule*

$$P(x,y) \text{ :- } R_1(x,y), R_2(y,x).$$

*We can evaluated this rule with a cascading searches (**forall** loops) on R_1 and R_2 with a primitive equality on the attributes of R_1, R_2. This can be represented as the following RAM program:*

```
1 search R1 as t do
2  search R2 as t1 where R2.x = t1(y) ∧ R2.y = t1.x do
3   project (t.x,t.y) into P
4  end
5 end
```

□

3.4.3 Partial Evaluation of Datalog

To obtain a RAM program from a Datalog program we employ a partial evaluation mechanism. This step is characterise by the following equation:

$$P_{RAM} = \texttt{Mix}_1(Int^{dl}_{RAM}, P_{dl})$$

Here, P_{dl} denotes a Datalog program that models e.g., a static analysis. Int^{dl}_{RAM} is the operational semantics of the language of P_{dl} via an interpreter e.g., semi-naïve algorithm for Datalog in the language of RAM. P_{RAM} is a program in the same language as the interpreter such that given the same input, the same result is produced as for P_{dl} interpreted by Int^{dl}_{RAM}.

The semi-naïve [90] algorithms depicted in Algorithm 1 are defined with two input parameters, namely, the Datalog program P_{dl} and the set of input relations, which we denote as EDB and one output, i.e., the set of output relations, denoted as the IDB. The partial evaluation process merges the fixpoint aspects of the Datalog interpreter, which can be defined using RAM and specialised nested-loop joins for each evaluated rule. For each call to the **Eval**, a concrete instantiation of a nested-loop join described in RAM is syntactically substituted into the RAM fixpoint algorithm.

The specialisation is depicted in Fig. 3.9. We assume a single recursive rule *rule* that we wish to be specialise. The semi-naïve algorithm is depicted in RAM and its `LOOP-NEST` function dynamically evaluates a rule via a nested-loop join. The `Mix` function inserts the specialised nested-loop join code for the rule into the interpreter to update a single relation in the IDB. This process in done for all rules in the Datalog program resulting in a RAM program consisting of several RAM **loop** statements with many **insert** statements for rules in a given SCC.

Example 12 (Motivating Example (Cont.))**.** *The RAM program for the recursively defined rule in our motivating example is shown in Fig. 3.10. The set I is thereby supported by two auxiliary sets I' and ΔI. The set I' represents the newly gained knowledge within an iteration of the fix-point computation and set ΔI represents the newly gained knowledge over the previous iteration. The fix-point computation is performed in the loop from line 3 to line 13. The first Section of the loop body (lines 4 - 8) computes I' using ΔI as an input. The loop starting in line 5 iterates over all nodes in ΔI and the nested-loop starting in line 5 iterates over all edges*

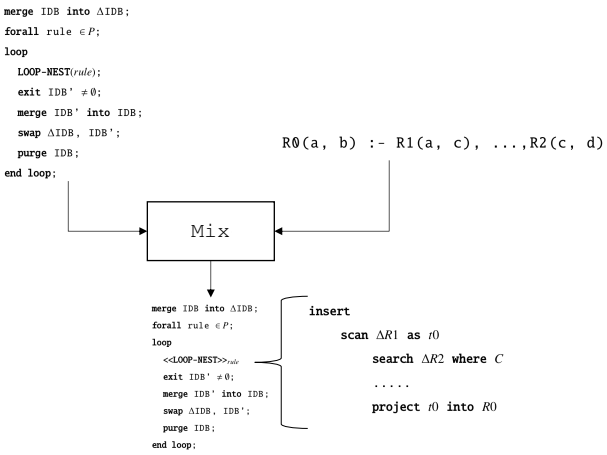


Figure 3.9: *Datalog specialisation process*

```
1  insert (number(0)) into I
2  merge I into ΔI;
3   loop
4    insert
5      search ΔI as t0
6        search E as t1
7          where t1.s=t0.c0 and (t1.d) ∉ P and (t1.d) ∉ I
8            project (t1.d) into I'
9    exit I' ≠ ∅;
10   merge I' into I;
11   swap ΔI, I';
12   purge I'
13 end loop;
```

Figure 3.10: *Running example: RAM program*

in the control flow graph. If any of those edges links some node x to a previously discovered insecure node y present in ΔI, where x is not a `protect` *call itself and has not been marked as insecure before, node y is add to the newly deduced set of insecure nodes I′. In the last two statements of the loop body (i.e. lines 10 and 11) the newly gained knowledge of relation I′ is added to relation I and I′ becomes ΔI. The fixed-point calculation terminates if no new insecure nodes could be identified.*

□

3.4.4 Nested Loop-Join Scheduling

The Datalog specialisation opens up many opportunities for important optimisations relating to nested-loop joins. Firstly, search conditions are hoisted to the outer-most

loop where they are still admissible in order to prune the iteration spaces effectively. This technique is also referred to as levelling [79]. Additionally, loops in the nested-loop join which have primitive searches subsumed by another loop can be coalesced into a single loop. The most impactful optimisation is the choice of good nested-loop join order. Here a scheduler (See Figure 3.3) selects a loop order, minimising the iteration space of the nested-loop join with the aid of a query planner [90]. After the translation to nested-loop joins we proceed to other optimisations in later specialisations like index selection. This optimisation has no baring on the semantic correctness of the evaluation and thus is a safe optimisation to perform, however, the analyser performance can be significantly impacted by the choice of loop schedule, and hence care must be taken when performing this optimisation. Unfortunately, this is an NP-Hard problem [142]. Several, solutions exist in the database literature mainly based on cardinality estimation [143]. However, cardinality estimation have mixed performance results [144]. In Soufflé a scheduling framework is employed to perform a variety of cardinality and cost estimations statically. An advantage of the staged specialisation architecture that is employ in Soufflé is that retain several schedules can be synthesised for a negligible synthesis time cost. In ongoing work (see Chapter 7) we are investigated auto literal scheduling combining with the technique of Chapter 4.

Example 13 (Motivating Example (Cont.))**.** *Consider the RAM program in Figure 3.10. An alternative version is to generate a* ***scan*** *on the E relation and a* ***search*** *on the I relation. This makes no different on the semantics of the RAM program, however, may impact efficiency. Say we have a metric that dictates that a rule is evaluated faster if smaller relations are in outer loops and large relations in inner loops (this a common metric) then if I is larger generally than E, we could improve performance by the changing the RAM program in Figure 3.10.* □

3.5 Relational Algebra Specialisation

In this Section we describe the third stage in Figure 3.3. In this stage, we further specialise Datalog program from its RAM representation to a C++ program.

3.5.1 RAM Interpreter

The RAM interpreter implements the RAM semantics of Figure 3.8. At this point, the Datalog program can be either evaluated using the RAM interpreter or be further specialised. We opt for the latter due to the fact that interpretation has several performance bottlenecks:

(i) any further optimisations needed for performance, require conditional checks at run-time. In other words, they must be performed *dynamically*, which results in slowdowns compared to further specialised programs

(ii) the AST traversal infrastructure relies on dynamic dispatch calls which result in lookup overheads

For this reason, the interpreter mode in SOUFFLÉ is mainly used for executing small Datalog programs or for testing purposes. The cost of interpretation is particularly highlighted by the search operation: for a typical program analysis in Section 4.4, a search operation is often executed many billions of times for large static analyses.

3.5.2 C++ Representation

3.5.2.1 C++ Constructs

Specialised RAM programs are expressed in templatised C++. The C++ programs have follow the control flow of RAM programs, as defined in the RAM interpreter implementation. For example, the RAM loop, exit construct is implemented with a `for(;;)` loop with `break` statements, merge and swap statements are implemented with copy operations of a relation class, purge deletes a relation object on the heap, nested-loop joins are implemented as a set of nested `for` loops, etc. that call search, contains and insert operators implemented in the relation class.

3.5.2.2 Relations and Indexing

The design of the relations are crucial for high performance. To this end, SOUFFLÉ assumes an execution model that keeps relations in-memory[1], as depicted in Figure. 3.12. Here, a relation stores its tuples in a set of indexed data structures. This

[1]Given the increasing availability of memory in computers (e.g., terabytes of memory), this is a viable option

```
1  ...
2   if ( keys == 0 ) { // sequential traversal of relation
3    for(Relation::iterator it=rel->begin(); it!= rel->end(); ++it) {
4     environment[level]=*it;
5     if ( cond == NULL ) {  // no condition
6       // evaluate
7       ...
8     } else if(cond->evaluate(environment)) {  // check condition
9       // evaluate
10      ...
11    }
12   }
13  } else { // indexed search of relation
14   if ( idx == NULL ) {  // has indexed be queried before?
15    idx = rel->getIndex(keys);
16   }
17   TupleElement tuple[rel->getArity()];
18   for(size_t i=0; i<rel->getArity(); i++) {
19    if( indexExpr[i] != NULL ) {
20      // evaluate
21      ...
22    }
23   }
24   Index::iterator it;
25   for(it=idx->begin(tuple); it!= idx->end(); ++it) {
26    environment[level]=*it;
27    if ( cond == NULL ) { // no condition
28      // evaluate
29      ...
30    } else if(cond->evaluate(environment)) { // check condition
31        // evaluate
32        ...
33    }
34   }
35  }
36  ...
```

Figure 3.11: *Implementation of interpreting RAM search*

data model thus requires two abstract data types (ADTs) in form of C++ classes to be defined as depicted in Figure. 3.13. Here, a `Relation` data structure which serves as *wrapper* for its indexes, provides operations e.g., search operation, required for interfacing with a Relation irrespective of its underlying implementation (e.g., B-Tree, Trie).

An important feature of the interface provided by the ADT is the search interface that is parameterised by a lexicographical order. The advantage of the lexicographical searches is that they are able to perform search operations more efficiency than in RAM. RAM searches have abstract notions of relations, e.g., a set of tuples. Thus they can be evaluated conducting a *linear scan* and checking the search predicate against each tuple of the relation. However, the time complexity of linear scan over a relation with n tuples is $O(n)$, which is too costly for large relations considering that each primitive search is invoked repeatedly many times. By applying an appropriate lexicographical order on tuples, a search can be performed with $O(log(n))$ time complexity (c.f B-Tree) and each call to a search operations accesses one of these indexes which contain specialised comparator functions.

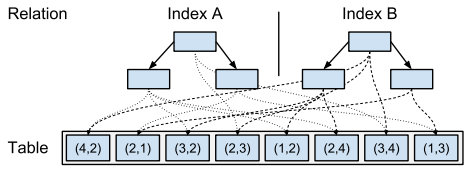


Figure 3.12: *Data structure layout of analyser*

Example 14 (Lex Search)**.** *Consider a ternary relation $R(x,y,z)$ where*

$$R = \{(1,2,3),(1,2,5),(2,3,3),(1,1,1)\}.$$

ADT Rel$\langle arity, Index_1\langle\ell_1\rangle, ..., Index_k\langle\ell_k\rangle\rangle$
Data:
Index[size(L)] indexes;
Public Operations:
equalRange$\langle\ell\rangle$(t)
insert(t)

(a) Relation Abstract Datatype

ADT Index$\langle\ell\rangle$
Data:
tuples size k
Public Operations:
range-search$\langle\ell\rangle$(a, b)
insert(a)
Private Operations:
compare$\langle\ell\rangle$(t1, t2)

(b) Index Abstract Datatype

Figure 3.13: *Data structure scheme*

Assume we have a lexicographical order, $x \prec y$ *and a call to a range search* `equalRange` $< x, y > ((1,2))$*, then the result of the range search is* $\{(1,2,3),(1,2,5)\}$.

□

Remark . The theoretical foundations of converting searches to indexes searches can be found in Chapter 4, in particular we point the reader to Definition 7 and Lemma 4 and its proof of correctness.

3.5.3 Partial Evaluation of RAM

The RAM program is specialised as follows: we use the RAM interpreter for another partial evaluation step (See Figure 3.3). As before, the partial evaluation can be characterised by the following first Futamura projection equation:

$$P_{C++<>} = \texttt{Mix}_2(Int^{RAM}_{C++<>}, P_{RAM})$$

Here, a RAM program P_{RAM} is transformed into to a *range program* $P_{C++<>}$. Using the RAM interpreter $Int^{RAM}_{C++<>}$ and P_{RAM}, we execute `Mix` to specialises P_{RAM} such that we obtain $P_{C++<>}$.

The partial evaluation defined above essentially strips unneeded code from the interpreter and uses already known compile time information from the RAM program to generate a C++ program that performs the computations of the interpret instantiated for the given RAM program. This use of partial evaluation solves numerous performance bottlenecks and thus improves the runtime for the generated C++ code when compared to the interpreter. The key performance bottlenecks resolved by the partial evaluation are:

- The removal of conditions in the RAM instruction parsing. For example, the search operation has an if-statement determining whether a relation is traversed sequentially (i.e., no condition) or via an indexed search. Since the condition is known at compile time it can be eliminated at compile time.

- The virtual dispatch in calls when traversing the RAM program. For example, the virtual dispatch for subsequent search operations or project can resolved

and nested-loop joins are constructed for all search/project operations of a statement, since subsequent operations are known at compile time.

- Conditions and values are expanded in C++ code such that an optimising compiler C++ can generate optimal code for evaluating conditions and values.

In the Figure 3.11 we illustrate the specialisation potential for searches. Here we present a code snippet from the execute method in the search handling portion of the interpreter. The highlighted lines indicate conditions known at compile time. The execute method implements both sequential traversal of a relations for unconditional searches and indexed search that convert primitive search predicates to an index in the presence of a conditional search. The specialisation phase collects all conditions of the search operations and specifies an index on various attributes on which the search is performed. The if-statement in line 2 decides whether the search has an index or not based on the member variable keys whose bits corresponds to attributes of relation `rel`. If the i-th bit is set, the i-th attribute is part of the index. If no bit is set, the search is performed by traversing the relation rel sequentially (cf. line 3 - 12) using a for-loop (cf. line 3). Each search operation has a level corresponding to the number of search operations that are prior to the current search operation. The level is used to store the current tuple in the environment as shown in line 4 so that subsequent evaluations of conditions/values can access the current tuple of relation rel. After retrieving the current tuple of the relation, it is checked whether a condition exists. If there is no condition associated to the search operation (cf. line 5), the next operation is executed (cf. line 6) that is stored with the member variable op. If there is a condition, the condition is evaluated (cf. line 7) and depending on the logical value of the condition the next operation is executed.

For indexed searches (cf. line 13 - 30), first we check if an index exists on a relation. If an index does not exist, a new index for the attributes specified in keys is created; otherwise the existing index is returned. In the next step (cf. line 15 - 19), the index expression is evaluated. The attributes that are not part of the index have no index expression stored in `indexExpr`. In the for-loop of line 22,

```
template<int...> struct Comparator;
template<int i, int ... tail  > struct Comparator<i,tail...> {
 static bool cmp(const tuple& a, const tuple& b){
  return a[i] < b[i] || (a[i] == b[i] && Comparator<tail...>::
      cmp(a,b));
 }
};
template<> struct Comparator<> {
  static bool cmp(const tuple&, const tuple&){ return true; }
};
```

Figure 3.14: *Implementation of comparison operator*

the relation is traversed for the given index expression and the current tuple (cf. line 23) is stored in the environment. In line 24, it is checked whether there is a condition associated to the search operation. If not, the next operation is executed (cf. line 25); otherwise the condition is evaluated (cf. line 26) and if the condition evaluates to true the next operation is executed (cf. line 26). Conditions that can be evaluated at compile time have been high-lighted in gray colour. For example, for non-indexed searches two conditions (i.e. line 2 and 5) are checked although the conditions can be decided as soon as the lowering of the Datalog program has completed. Similarly, the conditions in line 12, 17 and 24 can be decided after the lowering. A staged specialisation approach ensures that the generated code has resolved the conditions and only the conditions remain that can only be resolved at runtime. Similarly, the virtual dispatches of line 6, 7, 8, 12, 18, 25, 26, 27 calling either the method execute() or evaluate() are expanded by the code generator by the concrete loops or conditions to eliminate the runtime overheads of virtual dispatch.

This search thus becomes a part of the code of a data structure representing a relation. Moreover, to implement indices from the previous step, we employ templatised B-Trees that require a comparison function for two tuples in the relation. The comparison function is implemented as a lexicographical order in the form of a template as sketched in Figure 3.14.

The variadic template for the struct `Comparator` is parameterised by the columns in order. E.g., the call `Comparator<2,0>::cmp(a,b)` compares the tuples `a` and `b` by checking whether the third element of `a` is less than the

third element of `b`. If the comparison results in a tie, the first elements of both tuples are compared to determine the order between the two tuples `a` and `b`. The operator is defined recursively: the base case is given by the `struct Comparator<>` considering every tuple equal, and the inductive case by `struct Comparator<i,tail...>`, comparing the `i`-th components and, if equal, delegating the comparison to `Comparator<tail...>`. The expansion of the template for a given instance such as `Comparator<2,0>` is performed at compile time and delivers, in combination with function inlining, significant performance gains for index construction and retrieval. Without applying meta-programming techniques that rely on program specialisations, i.e., pushing computations from runtime to compile time, these performance gains would not be achievable.

The partial evaluation step results in very large run-time improvements. While for a single range search the improvement is quite small, on a typical Datalog program, lex searches are a major bottle neck due to the large number of range searches called. For example, a large program analysis benchmark could result in billions of search calls.

Motivating Example (Cont.). The RAM code of the previous specialisation is not optimal since it might have a worst-case complexity of $O(n \cdot m)$ where n is the number of nodes in the control-flow graph and m is the number of edges in the control-flow graph. To improve the performance of the program, we specialise the **search** in line 6 of Figure 3.15 by employing an index in line 2 of Figure 3.15 for the first attribute (0-th position). The index using a range search in line 19 filters out all pairs in the edge relation whose source is not node u, i.e., all the edges are selected which emanate of node u denoted by the set $E(u, \_)$.

3.5.4 Index Sets

A crucial performance question is the construction of the set of indexes for each relation. Engines such as `Logicblox/PA-Datalog` [136] require manual index construction if more than one index is required for a relation. The problem of automatically inferring the best set of indexes is discussed in Chapter 4 of the thesis.

Example 15 (Motivating Example (Cont.))**.** *Recall, to improve the performance of*

```
1  // -- Table: E
2  ram::Relation<Auto,2, ram::index<0>>* rel_1_E;
3  // -- Table: I
4  ram::Relation<Auto,1, ram::index<0>>* rel_2_I;
5  // -- Table: delta_I
6  ram::Relation<Auto,1>* rel_3_delta_I;
7  // -- Table: new_I
8  ram::Relation<Auto,1>* rel_4_new_I;
9  // -- Table: P
10 ram::Relation<Auto,1, ram::index<0>>* rel_5_P;
11 ....
12 ....
13 rel_2_I->insert(0);
14 rel_3_delta_I->insertAll(*rel_2_I);
15 for(;;) {
16   if (!rel_3_delta_I->empty()&&!rel_1_E->empty()) {
17     for(const auto& env0 : *it) {
18       const Tuple<RamDomain,2> key({env0[0],0});
19       auto range = rel_1_E->equalRange<0>(key);
20       for(const auto& env1 : range) {
21         if(((!rel_5_P->contains(Tuple<RamDomain,1>({env1[1]}))) &&
22           (!rel_2_I->contains(Tuple<RamDomain,1>({env1[1]}))))) {
23           Tuple<RamDomain,1> tuple({(RamDomain)(env1[1])});
24           rel_4_new_I->insert(tuple);
25         }
26       }
27     }
28   }
29   if(rel_4_new_I->empty()) break;
30   rel_2_I->insertAll(*rel_4_new_I);
31   {
32   auto rel_0 = rel_3_delta_I;
33   rel_3_delta_I = rel_4_new_I;
34   rel_4_new_I = rel_0;
35   ...
36   }
37 }
```

Figure 3.15: *Running example: range program C++*

the Datalog program we employed an index on relation E resulting in significantly reduced runtime complexity. Suppose we had another access to relation E on both u and v attributes, i.e., E(u,v). A naive implementation would be to have two indices defined by the lexicographical orders defined by the sequences of variables with lexicographical orders u and u ≺ v, respectively. However, the minimal solution would be to have only one index, namely, one with the lexicographical order u ≺ v as it subsumes the index with only u. □

3.6 Specialising Data Structures

The final phase of Soufflé yields a concrete analyser. Here the ADT of data structures are compiled with concrete data-structures implementations (e.g., B-Trees, Tries, etc.). Therefore, if B-Trees are selected for each ADT, a specialised set of B-Trees will be instantiated.

Among various types of balanced search trees, B-Trees, which were originally designed for secondary storage data-structures, are known to be the most memory efficient and cache effective data-structures. Therefore, we employ in-memory B-Trees as our primary data-structure for storing very large relations to obtain performance. We have found that Tries exhibit good performance on relations with small numbers of attributes. Both data-structures implement the same ADT and are interchangeable. For relations with a large number attributes, the table is stored in a blocked list and indices contain pointers pointing to the records in the list in order to save memory.

In the case, no annotations are provided by the user in the Datalog program, each concrete data structure implementations are inferred based on a heuristic criteria.

Unfortunately, it is not possible to decide based on the dimensions of a relation or the number of indices whether a B-Tree or Trie is better suited. It very much depends on the spatial distribution of the stored data points. And this is only known during execution. Thus, we argue, that we can only make a rough judgement to prune the options. Larger dimensional data is less likely to be dense, thus we pick B-Trees by default, Tries for lower-dimensional cases. But which option is best depends on the analysis data, and is thus to be fine-tuned for relevant relations by the user. From our empirical analysis, we derive the following decision table on what concrete data-structure implements which logical relation:

# of attributes	Number of Indices	
	$0-1$	≥ 2
0	flag	-
$1-2$	Trie	Trie
$3-5$	B-tree	B-tree
6+	B-tree	blocked list + indirect B-tree index

3.7 Parallelisation

To exploit parallelism we extend RAM to include the parallel statement **par**. The semantics of **par** $S_1 \| \ldots \| S_k$ **endpar** is that statements $S_1, \ldots, S_k$ are executed in parallel. The parallel statement is finished when all statements $S_1, \ldots, S_k$ have been terminated. The execution has three phases. In the first phase for each statement a thread is spawned, in the second phase the threads execute the statements, and in the third phase a barrier is imposed among all threads to ensure that the parallel statement does not finish execution before all statements of the parallel statement have terminated. The parallel execution assumes that there is no data race among relation read/write accesses. The synthesis process is responsible for producing code that does not contain races since the consistency is not enforced by the abstract machine.

Apart from specialised B-Tree and Trie data structures for non-parallel execution we fully support parallel execution. A Datalog program provides ample of opportunities for parallelisation. The most relevant code portions to parallelise are the executions of nested join loop. For instance, the nested-loop join:

```
1  forall(x ∈ R1)
2    forall((y,z,w) ∈ {(y,z,w) ∈ R2 | y = x ∧ w = x})
3      if(z ∈ R3 ∧ x ∉ R0)
4        R0 := R0 ∪ {x}
```

can be parallelised by partitioning the relation $R1$ and distributing the partition among multiple, parallel resources. However, to be a valid transformation, all operations conducted within the nested-loop join have to be thread safe. Note that

scanning, querying and checking for memberships are pure read-only operations which can always be processed safely in parallel. The only critical operation is the insertion of new values into $R0$ in the innermost loop. This update operation on the set-representation of $R0$ needs to be synchronised. However, the synchronisation only needs to protect concurrent inserts. A protection against e.g., concurrent scan and insert operations is not necessary since such combinations cannot occur in a RAM program produced by the semi-naïve evaluation strategy.

To protect concurrent inserts for B-trees, several strategies are available. The simplest one is to protect concurrent insertion operations by locking the entire tree, thus sequentialising updates. Unfortunately, this also severely limits the parallel efficiency of the resulting code since due to lock contention, threads block each other in the execution of insert operations. Consequently, a locking strategy involving the underlying data-structure on a finer granularity is required.

For Tries the synchronisation operation for insertions can be implemented using atomic updates, thus realising a lock-free data-structure. Whenever a new node is inserted, a null-pointer somewhere in the structure will be atomically updated to point to the new node. If the update fails, the insertion procedure is simply restarted. This lead to a highly scalable parallel implementation. Moreover, we introduce an alternative data-structure that is based on geometrically encoded Tries [145] that further boost parallel performance.

For B-trees on the other hand, the synchronisation is a bigger challenge since insertions are not restricted to updating a single memory location. In the general case, keys and child pointers need to be shifted and potentially parent nodes split and re-balanced. The application of a fine-grained read/write locking scheme protecting all the nodes potentially affected by an insert operation and releasing locks as early as possible provided acceptable scalability on desktop systems. However, on multi-socket server systems the continued exchange of updates on the lock associated to the root node over the inter-chip buses caused a severe slow-down in performance and scalability. As a result, even with the fine-grained locking parallelism on multi-socket systems did not provide any net gains in performance.

To overcome this limitation we adapted an optimistic locking schema used in databases [51]. In this approach, every node in the tree is annotated by a version number which will be updated upon every modification. When a thread is reading a node while navigating the B-tree during an insert operation, it is recording the version number before starting its operation and comparing it after determining the next node to navigate to. If the version number remained unchanged, it continues by navigating to the resolved node. However, if the version number changed, some other thread has modified the content of the processed node while the read operation was in progress. Thus, the obtained result may be wrong. To correct, the thread simply restarts the read operation on the same node again.

Compared to the fine-grained locking, the optimistic locking approach does not update any memory location (or lock state) when there are no conflicts – which is the case in the vast majority of node traversals. Thus, communication between sockets is significantly reduced, leading to largely superior parallel scalability compared to the fine-grained locking solution.

The semi-naïve algorithm exposes a high-level of parallelism for Datalog programs as shown in the experiments. The parallelism can be exploited at various levels of the evaluation. There is a multitude of various parallelisation efforts of Datalog in the past [146, 147, 148, 149, 150, 151, 152] mainly focusing on rewriting techniques and top-down evaluations. We devise new parallelisation strategies for the semi-naïve algorithm ranging from fine grain-parallelism evaluating relational algebra operations to coarse-grain parallelism of components in the SCC graphs. The strategy at hand depends on the Datalog program and the nature of the underlying parallel computer architecture, e.g., distributed computer cluster, shared-memory multi-cores computers, and hardware accelerators including GPGPUs. In the following we list four parallelisation strategies found in Datalog programs using the semi-naïve algorithm for bottom-up strategies:

- Connected components in the SCC graph that are not dependent on each other, can be evaluated in parallel. The SCC graph represents a partial order that resembles a task dependency graph. The only condition for evaluating a

component is that its predecessors must be computed prior to itself.

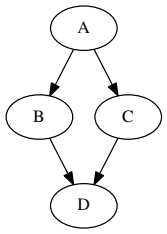
- The discovery of the new knowledge in a strongly connected component can be parallelised, i.e., if a component contains more than one recursively defined relation, the new knowledge for each of the relations in the component is computable in parallel.

- For each relation in a fixed-point iteration, the rules are evaluated in parallel. Using a thread safe insert operation in data structures such as [145], all concurrently processed rules can be insert into the same result relation.

- A relational algebra statement is performed in parallel. There are no loop-carried data-dependencies of the search operations in a relational algebra statement. To avoid data races writing the result, we use concurrent insert support from data structures [145].

3.7.1 Component Parallelism

One possibility to synchronise the evaluation of components in the SCC Graph are barriers. For each component there exists a barrier that can be passed as soon as all predecessor components in the SCC graph have been evaluated. However, our abstract machine has no notion to express general task dependencies. Nevertheless, the parallel statement is an implementation of a barrier, i.e., all statements of the parallel statement have to be finished before exiting the parallel statement. To exploit the component parallelism with the parallel statement, the SCC graph is converted to a series-parallel graph. The conversion gives an algebraic representation of the SCC graph using the sequence and parallel statement of the abstract machine. The algorithms for converting a generic task graph to a series parallel graph are introduced in [153, 154] such that the series-parallel graph still adheres to the dependencies of the SCC graph. An example is demonstrated in Figure 3.16. The SCC graph shows four components consisting of a single relation each. Relation B and C depend on A and the relation D depends on B, and C. By converting the graph to a series-parallel algebraic representation as shown in the figure, the best




```
<A>;
par
<B> ||
<C>
endpar;
<D>
```

Figure 3.16: *Example: Converting a SCC graph/task dependence graph to a series-parallel graph. The components <A> to <B> are further expanded for evaluating the relations in the components*

possible coarse-grain parallelism in components is given. Note that graphs that are not series-parallel graphs are approximated by a series-parallel graph, that adheres to the original dependencies. The coarse-grain component parallelism is suitable for clusters. However, the component parallelisation will be sensitive to the number of parallel components in the SCC graph and the computation time of the components. Since the SCC graphs depends on the Datalog program which is normally small, the available parallelism will be limited by the program analysis itself rather the input programs to be analysed.

3.7.2 Parallelising Relations in components

The fixed-point loop for a component permits the parallelisation in two parts: (1) the computation of the new knowledge for each relation can be computed in parallel since the computation does not cause a data-race, and (2) for all relations the new knowledge can be merged in parallel to the current knowledge after computing the new knowledge. For this parallelisation strategy only two synchronisation points are required: the point after computing the new knowledge and the point after merging the new knowledge with the current knowledge.

3.7.3 Parallelising Rules of a Relation

Parallelising the evaluation of rules of a relation is not for free. Each rule executed in parallel requires its own new/delta relation that imposes book-keeping overheads. After computing the rules in parallel, a merge operation filters out duplicates among the result relations. The filtering incurs a sequential cost that cannot be parallelised

and the parallelisation has to outweigh the costs of filtering. This strategy is suitable for multi-cores with shared-memory architectures. Profile-guided compilation can be used to guide the code generation, which rules should be parallelised (a benefit is expected) and which are not. However, by exploiting thread-save concurrent inserts the final merge phase is eliminated; all concurrent rules can insert into the same relation.

3.7.4 Parallelising Relational Algebra Operations

This parallelisation strategy is not specific to our Datalog engine and there exists a large body of related work (e.g., see [155]) to perform relational algebra operations in parallel. Various fine-grain architectures ranging from multi-cores to hardware accelerators including GPGPUs can be used to execute relational algebra operations in parallel.

3.8 Experiments

In this section we present the overall performance results of Soufflé. We follow these results up in Chapter 4 where we focus specifically on indexing schemes.

We perform an evaluation which compares Soufflé to a state-of-the-art Datalog engine, namely, `PA-Datalog`. All experiments are performed on two industrially motivated benchmarks. The experiments evaluate the Soufflé runtime, memory usage, index data structure usage and performance improvements using parallelisation.

3.8.1 Experimental Setup

3.8.1.1 Platform

Our experiments were performed on a 4 Core, 8 Hardware Threads, Intel(R) Core(TM) i7-7700K CPU at 4.20GHz with 64GB of physical RAM running Ubuntu 16.04.3 LTS on the bare-metal. The experiments were conducted in isolation without virtualisation so that runtime results are robust. Soufflé executables were generated using GCC 7.3.1.

Program	#Rules	#Relations
sec1	250	325
sec2	254	329
sec3	245	320

Table 3.1: *Datalog program sizes for cloud security analysis*

Dataset	#Facts
N1075	3,515
N2340	3,503
N3500	4,340
N3511	4,290
N9087	4,343

Table 3.2: *Virtual network dataset sizes*

Dataset	# Facts	Dataset	#Facts
lu-index	4,396,394	pmd	8,388,217
lu-search	4,396,394	fop	8,769,560
bloat	4,468,277	xalan	8,670,966
eclipse	4,389,763	hsqldb	9,007,087
antlr	8,319,095	chart	8,743,728
jython	5,203,400		

Table 3.3: *DaCapo dataset sizes*

3.8.1.2 Benchmarks

We perform our evaluations using two real-world sets of benchmarks: namely, network analysis and program analysis benchmarks. These benchmarks are based on the industrial case studies in Chapter 6. These use cases are of very large scale, where the Datalog programs contain hundreds of rules and relations and produce giga-tuple output relations.

Benchmarks-I: Cloud Security Analysis. The set of benchmarks are from several reachability properties of Amazon virtual networks that are manually encoded into Datalog. The benchmarks consist of three analysis workloads (i.e., three Datalog programs), each encoding specific reachability properties and security queries. We name these three programs as `sec1`, `sec2`, and `sec3`, where the numbers of rules and relations of these programs are shown in Table 3.1. We evaluate the programs on five virtual network datasets that vary in complexity: networks N1075 and N2340 have less complexity whereas networks N3500, N3511, and N9087 are more complex in terms of their network connectivity. The EDB sizes (i.e., total number of tuples in all input relations) of the five virtual network datasets are summarised in Table 3.2.

Benchmark-II: Program Analysis. The second set of benchmarks are from the

Doop program analysis library that performs points-to analyses for Java programs; Doop is publicly available and open source [28]. Specifically, a Java program is encoded as an EDB (i.e. input relations) and the points-to analysis is expressed as a Datalog program. Doop's points-to analysis has been used to analyse very large libraries such as the Oracle JDK [31]; as a result, *it requires very fast execution and low memory footprints* in order to be solved in a feasible time and with feasible resources.

The Doop analysis workloads have different parameterisable precision, which depend on (1) how concrete Java objects are abstracted to a finite set of objects in a sound fashion and (2) how much context is stored for each variable. For example, a context could be a trace over last few call-sites or receiver object of a method call. In our testings, we use three representative precision settings, 1-object-sensitive+1-heap (`1o1h`), 2-object-sensitive+2-heap (`2o2h`), and 3-object-sensitive+3-heap (`3o3h`).

Each of these precision settings corresponds to a Datalog program containing 496 relations and 469 rules. However, increased precision leads to some relations having more attributes and facts, and more complex rules. Each analysis program will be applied to 11 datasets from the DaCapo06 benchmark suite [156], where the sizes of these datasets are summarised in Table 3.3.

3.8.2 Experimental Results

3.8.2.1 Synthesis and Compilation

The analysis specified in Datalog has direct impact on the synthesis and compilation in the Soufflé framework. In Figure 3.17 we compare synthesis (code generation) and compilation times (compiling C++ to a binary) for each analysis benchmark. We can see that synthesis is negligible compared to compilation, requiring approx. 0.1 seconds to generate C++ code. Compilation times for the analyses range from 1.8 - 2.8 minutes. These times are typical for large scale static analyses, however the times vary depending on factors aside from code size, including number of unique indexes, index sizes etc.

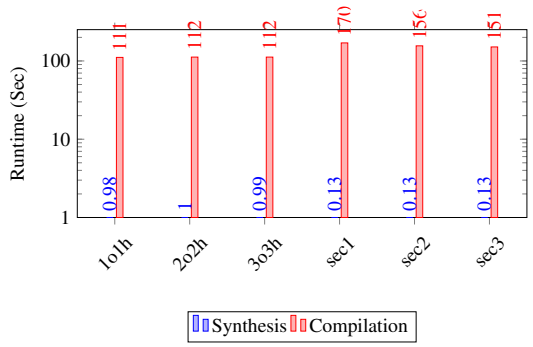
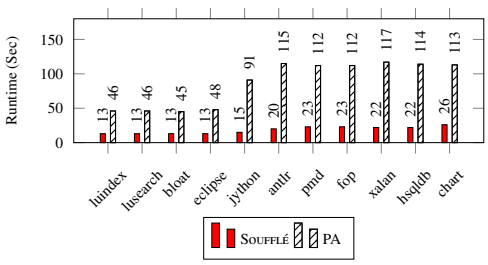
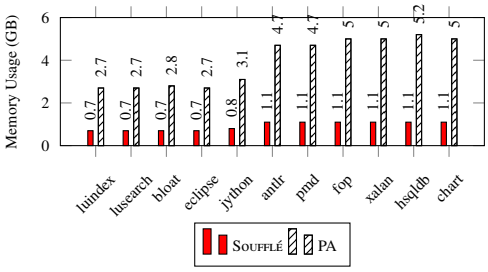


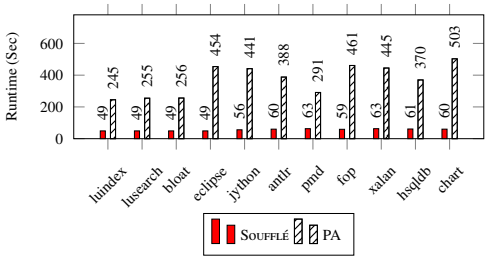
Figure 3.17: *Synthesis and compilation runtimes*



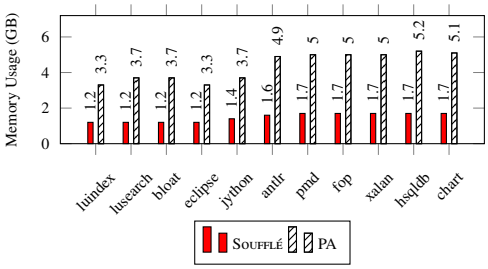
(a) *Runtime of 1o1h program*



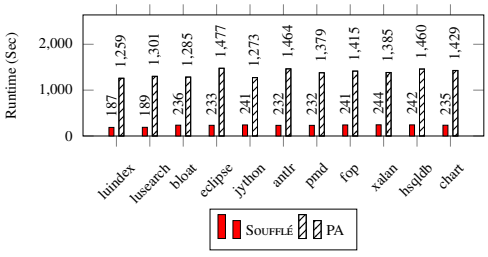
(b) *Memory usage of 1o1h program*



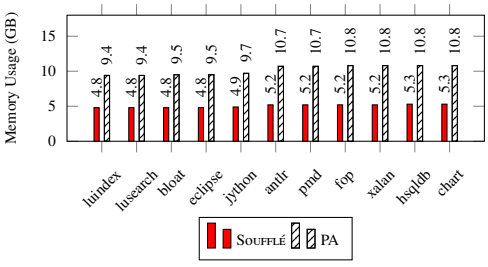
(c) *Runtime of* Doop *2o2h program*



(d) *Memory usage of 2o2h program*

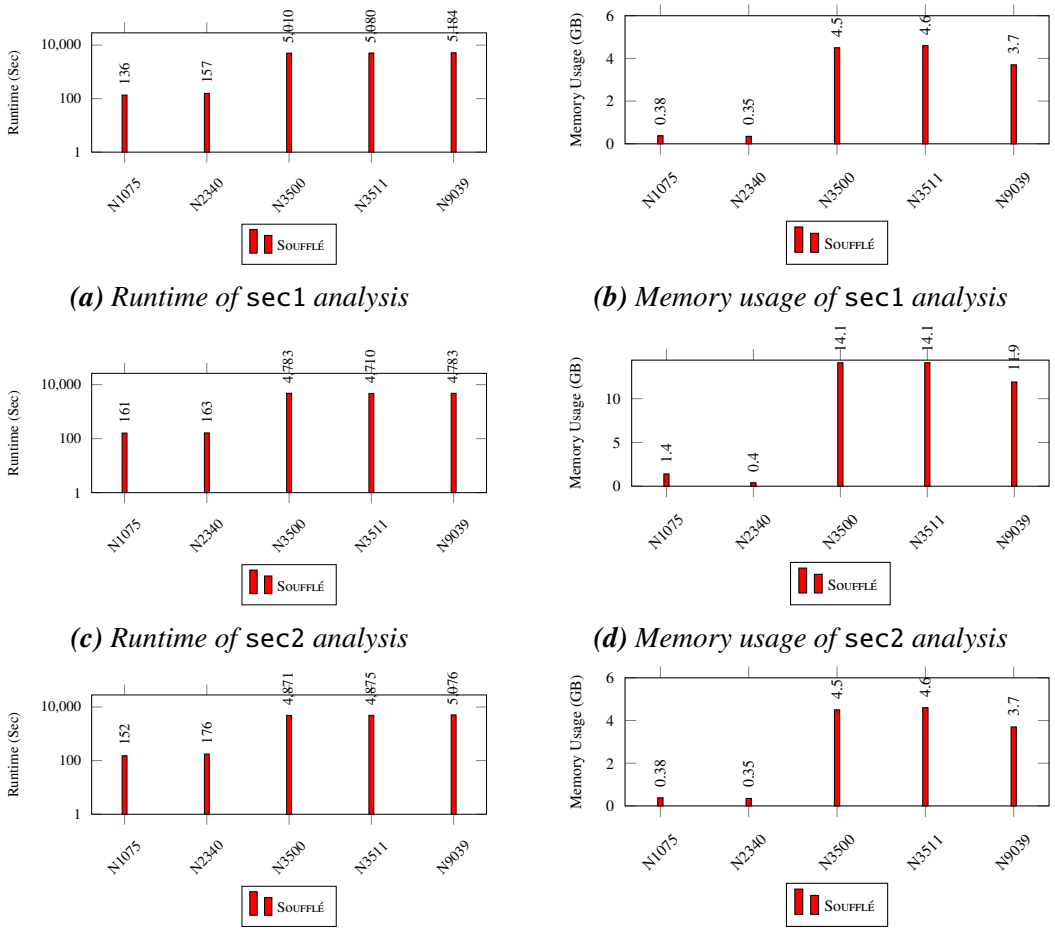


(e) *Runtime of 3o3h program*



(f) *Memory usage of 3o3h program*

Figure 3.18: Doop *program analysis experiments*



(a) *Runtime of* `sec1` *analysis*

(b) *Memory usage of* `sec1` *analysis*

(c) *Runtime of* `sec2` *analysis*

(d) *Memory usage of* `sec2` *analysis*

(e) *Runtime of* `sec3` *analysis*

(f) *Memory usage of* `sec3` *analysis*

Figure 3.19: *Network security experiments*

3.8.2.2 Performance Comparison of Indexing Data Structures

In this section we compare different types if indexing data structures. Here we justify our use of B-Trees compared to hashed indexes that are used in engines such as Flix [27] and μZ [37]. In Figure 3.20a and 3.20b we compare the average relative runtime, and memory usage respectively, of two hash maps implementations. Values lower than 1 indicate slower performance and more memory usage than B-Trees. We summarise the hashing data structures below:

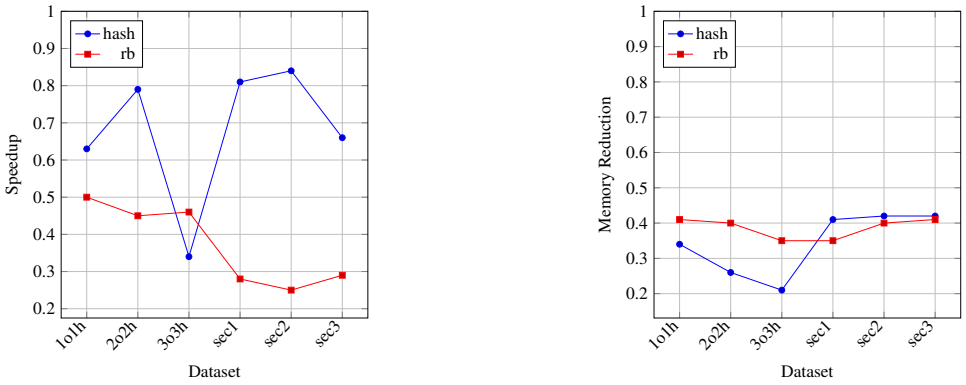
- Unordered Hashset (hash): a hash-based data structure using STL's unordered sets promising fast lookups; must recursively hash each relation/tuple
- Ordered Hashset (rb): Similar to the above, but uses STLs ordered sets; based on red-black trees

The results show that both hashing implementations exhibit sub-par performance in terms of speedup and memory usage compared to the B-Tree indexing approach of Soufflé. Unordered hashing (hash) has its peak in performance, on the 2o2h analysis but significantly drops off in runtime performance for the more heavy weight 3o3h. The ordered hashing (rb) remains stable for all program analyses, exhibiting a 100alternative approach consume a considerable amount of more memory and appear to degrade as the size of IDBs increase (i.e., in more precise analyses). Between datasets there was very little variation. We observed a runtime standard deviations ranging from 0.03-0.01 (hash) and 0.01 (rb). For memory usage we observed a standard deviation ranging from 0.1 - 0.008 (hash) and 0.008 - 0.002 (rb). The results hold (both for runtime and memory) when the amount of cores are increases, in fact the relative runtime performance of hashing degrades very slightly compared to B-Trees when more cores are used.

The cloud security benchmarks similarity demonstrate the superior performance of B-Trees compared to hashing. For all analyses, sec1, sec2 and sec3, both hashing approaches perform slower than B-Trees and consuming more memory. However, there is more variability in the hash results, i.e., a large standard deviation. We therefore, break down the results further. For the dataset N-2340, hash performs better for all analyses (sec1, sec2, sec3), rb performs very uniformly all all datasets, 3.4x more runtime and 3.1x more memory usage. For N-1075, hash performs at 1.5x slower (resp. 0.6) for sec1, 3.1x (resp. 0.32) for sec2 and 46.5x (resp. 0.02) for sec3. Memory variance is small, ranging from 2.8x - 3.5x (resp. 0.35 - 0.2) more memory. rb ranges from 2.8x (resp. 0.2) to 2.8x (resp. 0.35) more runtime and 2x (resp. 0.49) to 2.9x (0.34) more memory usage.

For the larger benchmarks on average sec1 took 1.8x (resp. 0.56) more runtime and 2.5x (resp. 0.39) more memory. For sec2 and sec3, both hash and rb timed outed out after 12 hours on larger datasets (N-3500, N-3511 and N-9087) hence we do not include them in the results in Figure 3.20a and 3.20b.

Overall, as shown in Chapter 6, hashing has a place in Datalog engine design, when limited to small sized, less computationally intensive data processing



(a) *Relative runtime alternative indexing data structures (< 1 indicate slowdown)*

(b) *Relative memory alternative indexing data structures (< 1 indicates more memory usage)*

Figure 3.20: *Relative alternative indexing data structure comparison*

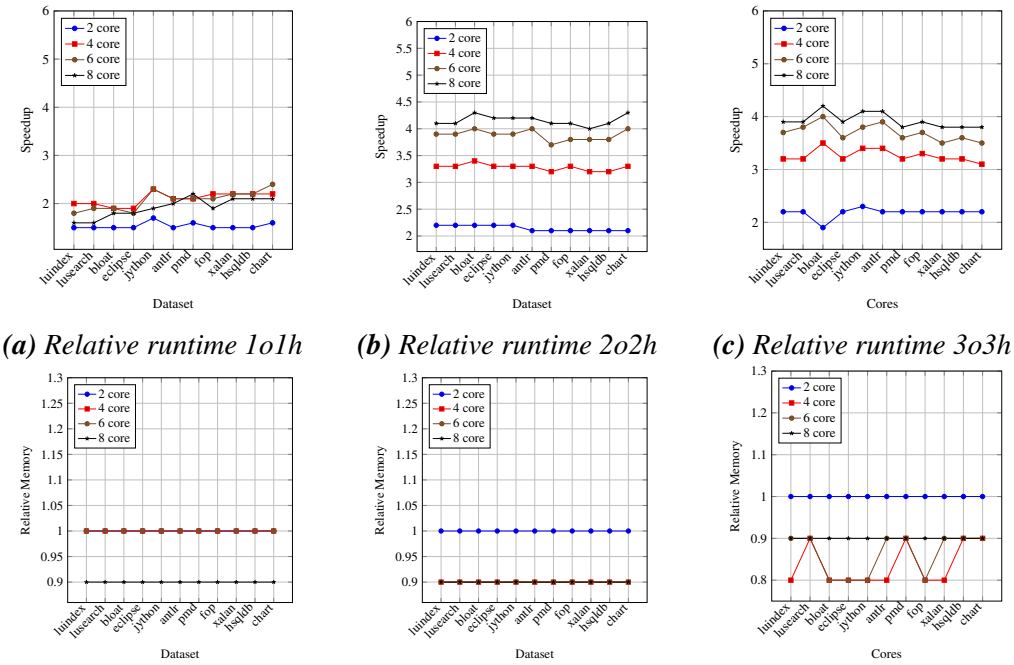
(cf. 7.2.4 [52]).

3.8.2.3 Performance vs `PA-Datalog`

The results in Figures 3.18 and 3.19 show significant speedups using Soufflé compared to `PA-Datalog`. For the `1o1h`, `2o2h` and `3o3h` analyses speedups ranging from 3.5-6.0, 5-9.2 and 5.4-6.8 are observed, respectively. Moreover, significant memory improvements are observed. For the `1o1h`, `2o2h` and `3o3h`, memory improvements of 3.9 - 4.7, 2.8 - 3.1 and 2-2.1 are observed. For the cloud security benchmarks `PA-Datalog` was not able to compute the network benchmarks in under 24 hours. We speculate this is due to a lack of code optimizations for the single indexing regiment of `PA-Datalog` and its inability to find an appropriate dynamic schedule. on analyses `sec1`, `sec2`, and `sec3`. Soufflé on the other hand, is able to compute the benchmarks ranging from 136-176 seconds and 0.35-1.4 gigbytes of memory on small benchmarks (N1075 and N2340) and 4710-5184 seconds and 3.7-14.1 gigabytes larger benchmarks (N3500, N3511, N9039).

3.8.2.4 Parallelism

We evaluate the performance of Soufflé with an increase in parallelism. We increase the number of utilised cores/virtual threads in the computation and measure the relative speed/memory usage improvement compared to a single core computation resulting from 2, 4, 6 and 8 threads. In Figure 3.21a,3.21b,3.21c the runtime



(a) *Relative runtime 1o1h* ***(b)*** *Relative runtime 2o2h* ***(c)*** *Relative runtime 3o3h*

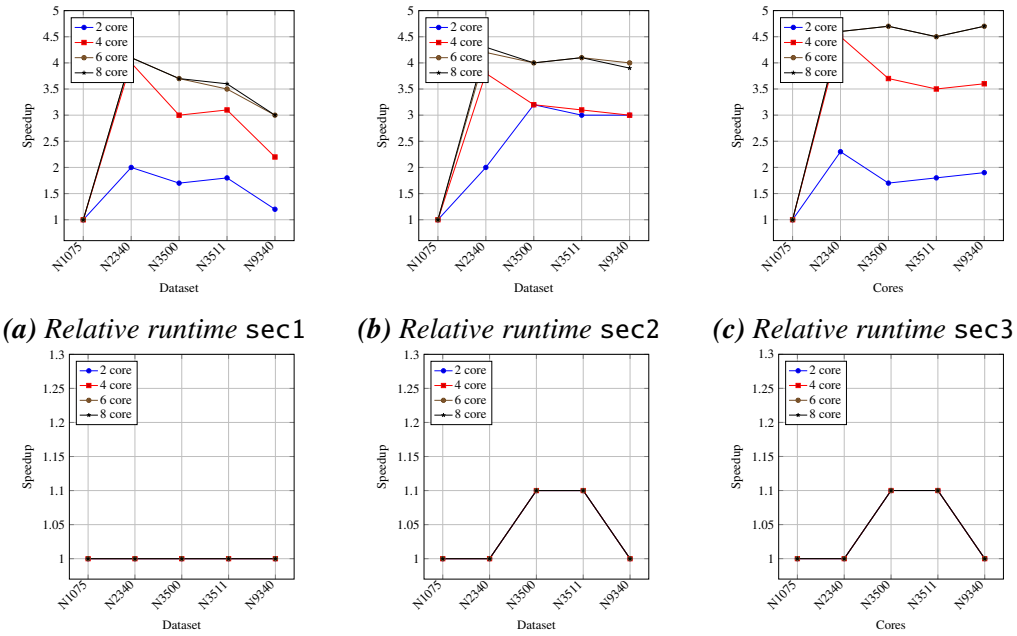
(d) *Relative memory 1o1h* ***(e)*** *Relative memory 2o2h* ***(f)*** *Relative memory 3o3h*

Figure 3.21: *Performance improvement of parallelisation for program analysis benchmarks*

improvements are shown for 2-8 core computations. The first observation is that larger parallelism results in greater improvements for the more complex analyses `2o2h` and `3o3h`, than for `1o1h`. Moreover, we can see that as more cores are used, the improvement decreases.

The memory usage improvements are shown in Figure 3.21d,3.21e,3.21f. The impact of parallel computation does not result in large memory usage variations, with a slight increase in memory usage for more cores observed, likely resulting from required overheads when utilising more than a single core.

For the cloud security benchmarks, a similar situation is observed. In in Figure 3.22a,3.22b,3.22c, the runtime improvements are shown. Benchmarks with the N1075 dataset (the smallest dataset) shows no improvement with parallelism. The other benchmarks show that as more cores are used, less runtime improvements are observed. For these benchmarks there is little difference between 8 core and 6 core computations. Similar to the Doop benchmarks, Figure 3.21d,3.21e,3.21f show little memory usages changes.



(a) Relative runtime `sec1` *(b) Relative runtime* `sec2` *(c) Relative runtime* `sec3`

(d) Relative memory `sec1` *(e) Relative memory* `sec2` *(f) Relative memory* `sec3`

Figure 3.22: *Performance improvement of parallelisation for network analysis benchmarks*

3.9 Discussion

3.9.1 Datalog Engines

Several engines such as [84, 45, 87, 68] claim to perform some compilation of Datalog code. For example, Socialite [45] compiles Datalog to Java and in some cases include parallelism to the semi-naïve algorithm [45, 87]. The details of this compilation in these engines are not described in detail in their publications. Our understanding is that instead of using partial evaluation, they incorporate elements of evaluation algorithms such as semi-naïve in their generated code. Often the code that is generated is then interpreted [68]. The work in [157] discuss the compilation of Datalog using a *push method*, the method also uses elements of semi-naïve and nested loop joins but does not explicitly use interpreters and partial evaluation.

3.9.2 Partial Evaluation of Systems

Our approach to using partial evaluation strongly aligns with the approaches in model-driven development [105, 106]. Here a model is a description of some desired behaviour. Typically models are compiled into code that generates the desired behaviour. Similar to our approach, in this work a model interpreter takes the

model as an input and performs the behaviours specified in the model. Likewise, the framework in [104] allows users to specify interpreters for which the system compiles code derived automatically form the interpreter using partial evaluation. In [158], an approach is described that uses partial evaluation to derive automatically implementations of operating system components from generic specifications. Similarly,the work in [159] uses partial evaluation to synthesise DSP circuits.

3.9.3 Methods of Synthesis of Analysers

In our context, program synthesis refers to the classical notion for it [160] i.e., constructing an executable program (i.e., program analyser) from a logical specification (i.e., in Datalog). We refer the reader to [161] for a survey of program synthesis techniques and uses. While our framework can also be used to generate C++ programs from Datalog specifications, our focus in this paper is efficient synthesis of program analysers, i.e, we generate C++ programs that take a program as a set of relations and produce analysis results in output relations. Several frameworks have been cast as a synthesis of analysers and/or verifiers e.g.,[162, 163], and to a lesser extent [164, 37, 30]. While our approach shares similarities of specifying the analysis in a logical specification (Datalog Horn clauses) we generate a standalone C++ analysis tool rather than solving clauses within the framework/engine itself. As shown in our experiments, this results in significant performance gains. The approach in [165], like us, uses partial evaluation of Datalog to improve performance of logic engines. Additionally, several compilers perform efficient code generation using synthesis techniques [166, 167]. This body of work, like our technique, synthesises efficient code form a logical specification; unlike our work, these approaches generate general programs optimised at the assembly level, where as we generate analysis tools optimised at the C++ level, adhering to a Datalog specification.

3.9.4 Correctness

The correctness of our approach depends on the faithful implementation of the RAM interpreter and the utilised C++ compiler. Proving the correctness of the

analyser being correct to the Datalog specification is a monumental task, akin to formal verification of compilers. Individual steps however can be reasoned with given some assumptions. For example, given a P_{dl} be an arbitrary Datalog program and a residual RAM program P_{RAM} from $\texttt{Mix}_1$ it can be seen that for all inputs, d, $[\![P_{RAM}]\!]_{Int_{RAM}}(d) = [\![P_{dl}]\!]_{Int_{dl}}(d)$. This can be shown by 1) demonstrating that non-recursive operations can be modelled with RAM programs. The correspondence between relational algebra and nested loop joins is well established (see [90, 168]). It is easy to see, and that RAM operations are equivalent to nested loop joins. 2) showing that the least fix point characteristics of semi-naïve can be modelled in RAM. By the semantics of the loop - exit construct (and book-keeping statements) this can seen. We can also reason that the C++ with templates model the RAM. Intuitively, this can be seen however a formal proof would require the verification of the code in the indexing scheme, C++ implementation of data structures modelling relations etc. Without any clear semantics of C++ (with templates) a completely formal proof is difficult. If it is deemed this avenue of research beneficial to the community, we may pressure such proofs in the future.

3.9.5 Limitations

For the static analysis use case, Soufflé presents a clear advantage over evaluation based approaches. However, for general Datalog programs Soufflé's synthesis approach may result in worse performance. As we detail in Chapter 2, Section 2.2, if specialisation (including C++ compilation) and runtime of the synthesised analyser is greater than Datalog evaluation time of a program, partial evaluation obviously is not worth it. For use cases when the ruleset of a Datalog program changes significantly and evaluation time is small, Soufflé will perform worse than general Datalog engines. For this Soufflé provides a RAM interpreter that allows small Datalog programs to be interpreted. At a certain point however, it is beneficial for the user to switch to synthesis/compilation mode, e.g., when the dataset becomes large.

Chapter 4

Auto-Index Optimisations

In this chapter, we propose an automatic method of selecting and assigning indexes to relations in order to speedup search operations in Datalog programs. This work is aimed provides a theoretical perspective of the auto indexing technique implemented in SOUFFLÉ. A large part of this chapter is described in [52] published in Very Large Databases (VLDB).

4.1 Indexing in Datalog

Indexing is a crucial method of speeding up data retrieval in query engines e.g., DBMS [169], Datalog [45] and Prolog [170]. The need for indexing becomes increasingly important for use cases such as static analysis. Since these use cases consist of hundreds of rules and result in giga-tuple sized IDB relations [31, 28], indexing is paramount to ensuring computations can be performed in a practical amount of time. As a result, high-performance Datalog engines store relations as *in-memory, index-organised tables* [136, 87].

4.1.1 Index Selection

The task selecting an appropriate index or set of indexes for a relation is non-trivial. Each index that is assigned to a relation results in data being replicated and incurred costs such as increased memory overhead, index maintenance etc.

The theory of the index selection has been formalised as the *Index Selection Problem* (ISP) in the database literature. Index selection for relational database management systems [171, 172, 173, 174] uses variants of the 0-1 knapsack problem, which has been shown to be NP-hard [175]. Deployed approaches such as [176] use heuristics and integrate with what-if query optimisation calculations. These techniques are surveyed in Bruno [177], but they are too computationally expensive for large Datalog analyses. Essential differences include (i) indexes are needed for both EDB and IDB relations, (ii) the Datalog relations are often wide (not normalised), and thus they offer a very large number of possible indexes, and (iii) the Datalog programs typically consist of *hundreds of relations and hundreds of deeply nested rules* (see Table 3.1 in Section 4.4). As a result, the high performance Datalog engines often require users to provide annotations to guide the choice of

Access	
Role	Operation
a	del
a	insert
a	select
rw	insert
rw	select
w	insert
r	select

Role		
Name	Role Doctor	Role Patient
M.Smith	a	a
L.James	rw	r
N.Jones	r	rw
D.Cousins	w	n

Figure 4.1: *EDB relations* `Access`*,* `Role`

```
(r1) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, _, "Con"),
       !Role(uid, _, _).
(r2) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, dbid, op),
       Zone(dbid, "Doctor"), Access(l, op), !Role(uid, l, _).
(r3) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, dbid, op),
       Zone(dbid, "Patient"), Access(l, op), !Role(uid, _, l).
(r4) Err(s, e) :- Src(uid, s), Path(s, e), Sink(e, dbid, "Priv"),
       Privileged(l1, l2), !Role(uid, l1, l2).
```

Figure 4.2: *Datalog rules for vulnerability detection*

```
loop1: for all t1 ∈ Src do
loop2:   for all t2 ∈ σ_{x=t1(y)}(Path) do
loop3:     for all t3 ∈ σ_{x=t2(y),z="Con"}(Sink) do
loop4:       if σ_{x=t1(x)}(Role) = ∅ then
               if (t1(y), t2(y)) ∉ Err then
                 add (t1(y), t2(y)) to Err
               end
             end
           end
         end
       end
```

Figure 4.3: *Nested loop joins for Datalog rule (r1)*

Figure 4.4: *Example Datalog analysis for vulnerability detection*

indexes; for example, the Doop framework [28] uses a code-rewriting technique that *manually* chooses an index for each relation and introduces "Opt" relations for building multiple indexes on a relation. To allow widespread use of program analysis, we must move beyond approaches that put the optimisation burden on the user, who requires painstaking trial and error over hundreds of rules and annotations.

Example 16. *To illustrate index selection in Datalog engines, we present an example in Figures 4.1 and 4.2 that depicts a simplified Datalog analysis, used for detecting the vulnerabilities of a web-based hospital management system. The source code of the management system is converted into EDB relations, e.g.,* `Src`*,* `Sink`*,* `Role`*,* `Access`*,* `Zone`*, and* `Priv`*, where relations* `Role` *and* `Access` *for access policy are shown in Fig. 4.1. Datalog rules are constructed for the security analysis, enumerating all possible vulnerability cases. This part of the ruleset is shown in*

Fig. 4.2; we omit the Datalog rules for computing the IDB relation `Path` *that defines the control flow of the source code. For example, the first rule asserts an error if a user connects to a database via operations grouped as "Con" (i.e., connect) but without a role. The results of the analysis, determining the set of error paths, are stored in the IDB relation* `Err`*.* □

4.1.2 Auto-Indexing

In this thesis chapter, we formulate an *automatic* indexing scheme for Datalog computations, aiming to achieve the best performance/memory usage while not requiring the intervention of end users. Our approach was motivated by experiences with industry use cases involving large scale static analysis performed with the state-of-art compilation-based Datalog engine Soufflé [31]. We found inadequate performance until we introduced our new technique into Soufflé, however the ideas should apply more broadly, to any engine that computes a Datalog program in successive phases: initially there are analysis phases that consider only the rules and produce code to perform a query evaluation plan resembling a nested loop join. These are followed by an evaluation phase that executes the compiled query on the facts, producing a materialised IDB considers only the rules. Our auto-indexing is conducted at one of the analysis phases, and it chooses indexes that improve the performance of the compiled code.

The key insights of this chapter are as follows. We identify that the compiled evaluation is built from frequently repeated calls to simple selections, each on a single relation (which might be in EDB or in IDB). We call these *primitive searches*, and a primitive search returns the tuples in a relation which satisfy a predicate that involves testing some of the attributes for equality to a given value. For example, Fig. 4.3 depicts the evaluation logic that is compiled for the Datalog rule (r1) in Fig. 4.2 where the first, second, and third attributes of a relation are assumed to be accessed by x, y, and z, respectively. There are three primitive searches $\sigma_{x=t_1(y)}(\texttt{Path})$, $\sigma_{x=t_2(y),z=\text{“Con”}}(\texttt{Sink})$, and $\sigma_{x=t_1(x)}(\texttt{Role})$, where the first one looks up all tuples in relation `Path` whose first attribute value is equal to $t_1(y)$ — the second attribute value of a tuple t_1 from relation `Src`. Note that each primitive search

is a very restricted kind of range query: for each attribute, we are either checking equality to a value, or else we accept any value in that attribute.

The next insight is that the evaluation of a primitive search can be greatly sped up if the relation has a clustered B-tree [178] index that *covers* the search predicate. This means that the set of attributes where equality is checked, forms a prefix of the sequence of attributes used to lexicographically define the index. For example, the primitive search $\sigma_{x=v_1,z=v_3}$ is covered by the index $\ell = x \prec z$ (that means, an index using x followed by z as its key) but not by $\ell' = x \prec y \prec z$. When a search is covered by an index, the tuples that match the search are a contiguous part of the scan of the index leaves. Accessing these can be much faster than a full table scan, which is what an engine would use in the absence of an index. Because the relations are so large, we find that queries are typically infeasible in practice unless there is some index to cover every primitive search among the rules. On the other hand, each index uses considerable space, and so we are driven to minimise the number of indexes constructed. Thus we define an abstract task, the *Minimum Index Selection Problem* (MISP), aiming to select the *minimum* number of indexes to cover all primitive searches used in the ruleset. We notice that this can be significantly fewer than one index for each primitive search on the relation. For example, the index $\ell = x \prec y \prec z$ covers primitive searches $\sigma_{x=v_1}$, $\sigma_{x=v'_1,y=v'_2}$, and $\sigma_{x=v''_1,y=v''_2,z=v''_3}$.

Finally, we are able to solve the MISP efficiently, using a relationship between the search space of indexes and the search space of *search chains* among lexicographic orders. We prove that the optimal MISP solution can be constructed from the optimal (i.e., with the minimum cardinality) search chains that cover all primitive searches. Then we apply the combinatorial result of Dilworth's theorem [179] to compute the minimum number of search chains, and thus the minimum number of indexes, in $O(|S|^{2.5} + |S|^2 \cdot m)$ time, for a set S of primitive searches on a relation with m attributes. This is much faster than a brute force examination of all possible sets of indexes on this relation, which would have a time complexity of $O(2^{m^m})$.

We have implemented our index selection approach as the default indexing technique of the Soufflé Datalog engine. We found that the computation over-

Literal	Primitive Search	Lex Search Predicate $\rho(\ell, a, b)$			
		Lower Bound a	Upper Bound b	Naïve ℓs	Minimum ℓs
`Role`$(v_1, \_, \_)$	$\sigma_{x=v_1}$	$\langle v_1, \bot, \bot \rangle$	$\langle v_1, \top, \top \rangle$	x	$x \prec y \prec z$
`Role`$(v_1, v_2, \_)$	$\sigma_{x=v_1, y=v_2}$	$\langle v_1, v_2, \bot \rangle$	$\langle v_1, v_2, \top \rangle$	$x \prec y$	$x \prec y \prec z$
`Role`$(v_1, \_, v_3)$	$\sigma_{x=v_1, z=v_3}$	$\langle v_1, \bot, v_3 \rangle$	$\langle v_1, \top, v_3 \rangle$	$x \prec z$	$x \prec z$
`Role`(v_1, v_2, v_3)	$\sigma_{x=v_1, y=v_2, z=v_3}$	$\langle v_1, v_2, v_3 \rangle$	$\langle v_1, v_2, v_3 \rangle$	$x \prec y \prec z$	$x \prec y \prec z$

Table 4.1: *Primitive and lex searches for relation* `Role` *in the nested loop joins for rules (r1)–(r4) in Fig. 4.2*

head for our index selection is negligible, i.e., no slowdowns were observed during compilation. Using our technique, Soufflé has managed to efficiently compute program analyses typically deemed too large for Datalog engines, and moreover, the performance exhibited by Soufflé has been on a par with recent state-of-the-art hand-crafted analyzers [180].

We remark that our scheme is based on clustered B-tree index structures kept in-memory. If multiple indexes are needed, we materialise replicas of the relation so that each index can be clustered. Experience with hash indexes in Soufflé has not been encouraging (see Chapter 3), and the highly specialised nature of the primitive searches are also not suited to spatial index structures such as R-trees.

4.2 Indexing Relations

In this section, we first introduce indexes to speed up primitive searches, and then formally define our problem of minimum index selection.

After constructing the nested loop joins for all rules in a Datalog program, the most critical factor to the performance of evaluating the Datalog program is how the primitive searches are conducted. Obviously, a primitive search can be achieved by conducting a *linear scan* of all tuples of the relation and checking the search predicate against each tuple. However, the time complexity of linear scan over a relation with n tuples is $O(n)$, which is too costly for large relations considering that each primitive search is invoked repeatedly many times. In this paper, we aim at creating indexes for relations to speed up the primitive searches, and we study the following problem whose formal definition will be given in Section 4.2.

Problem 1. *Given the primitive searches in the nested loop joins of all rules in a*

Datalog program, we study the problem of creating indexes for relations to speed up all the primitive searches.

4.2.1 From Primitive Search to Lex Search

Index-based Lex Search. To index on a relation, we first define an order among tuples in a relation to make them comparable. Since a tuple may have several elements, an order of tuples is imposed by element-wise comparison using a sequence over all attributes of the relation; that is, if the first elements of two tuples produce a tie, the second elements are used and so forth. This comparison is known as a *lexicographical order*. We denote an attribute sequence by $\ell = x_1 \prec x_2 \prec \cdots \prec x_m$ where $\prec$ denotes a chaining of elements to form a sequence. Then, given ℓ that is formed by all attributes of a relation, a lexicographical order $\sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$ is a total order (i.e., *reflexive, asymmetric, transitive*) defined over the domain $\mathcal{D}$ of the relation with respect to ℓ. For two tuples $a, b \in \mathcal{D}$, if $(a, b) \in \sqsubseteq_\ell \mathcal{D} \times \mathcal{D}$, then we write $a \sqsubseteq_\ell b$ and we say that a is smaller than b with respect to ℓ. Note that, (1) we have $a \sqsubseteq_\ell a$, and (2) for any two different tuples $a, b \in \mathcal{D}$, we have either $a \sqsubseteq_\ell b$ or $b \sqsubseteq_\ell a$ but not both.

Given an ordered set of tuples, tuple lookups can be performed efficiently using some notion of a balanced search tree, called an *index*, in which tuples can be found in logarithmic time rather than in linear time. In this paper, *we abstract away the underlying implementation details of an index with an attribute sequence*, and we use ℓ to denote both an index and the attribute sequence based on which the index is constructed. It is worth mentioning that different attribute sequences usually result in different lexicographical orders, and thus different indexes. That is, for two tuples $a, b \in \mathcal{D}$ and two attribute sequences ℓ and ℓ', it is possible that $a \sqsubseteq_\ell b$ and $b \sqsubseteq_{\ell'} a$. For example, for $a = \langle 1, 2 \rangle$ and $b = \langle 2, 1 \rangle$ in $R(x, y)$, we have $a \sqsubseteq_\ell b$ and $b \sqsubseteq_{\ell'} a$ where $\ell = x \prec y$ and $\ell' = y \prec x$.

Given an index ℓ, we define a lex search as follows.

Definition 7 (Lex Search)**.** *A lex search $\sigma_{\rho(\ell,a,b)}$ is defined for a relation $R \subseteq \mathcal{D}$ and*

its semantics is given by,

$$\sigma_{\rho(\ell,a,b)}(R) = \{t \in R \mid a \sqsubseteq_\ell t \sqsubseteq_\ell b\}.$$

$\rho(\ell, a, b)$ *is a* lex search predicate*, where ℓ is an index on R, and the lower bound a and the upper bound b are tuples in $\mathcal{D}$.*

Constructing Lex Searches from Primitive Searches. As lex searches can be efficiently conducted based on an index, we would like to transform each primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ into an equivalent lex search $\sigma_{\rho(\ell,a,b)}(R)$. A lex search contains two symbolic bounds a and b, as well as an index ℓ, in the lex search predicate. Thus, we need to construct a, b, and ℓ, which will be discussed in the following. We assume that the relation R has m attributes in total.

Firstly, we describe how to construct the lower bound a and the upper bound b. If $k = m$, then all attributes of R are in the search predicate, and $a = b$ and they are trivially defined by the search predicate. Otherwise, the primitive search does not specify all attributes of R in its search predicate, and unspecified values need to be padded with infima and suprema values for lower and upper bounds, respectively. We define an unspecified element for the lower/upper bound construction by an artificial constant[1] $\triangle$, and let $v_{k+1} = \triangle$. We define a surjective index mapping function $i : \{1, \ldots, m\} \to \{1, \ldots, k+1\}$ that maps the specified elements to their corresponding constant values, and maps the unspecified elements to $\triangle$ (i.e., v_{k+1}). The construction of the lower and upper bound is performed by the functions *lb* and *ub*, respectively,

$$a = lb(v_1, \ldots, v_k)$$
$$b = ub(v_1, \ldots, v_k)$$

that replace the unspecified $\triangle$ value with the infimum $\bot_j$ and the supremum $\top_j$ of the domain $\mathbb{D}_j$, respectively. Formally, the functions are defined as $lb(v_1, \ldots, v_k) =$

[1] We assume that $\triangle$ is not element of any of the domains $\mathbb{D}_i$.

$\langle v'_1, \ldots, v'_m \rangle$ where

$$v'_j = \begin{cases} v_{i_j} & \text{if } v_{i_j} \neq \triangle \\ \bot_j & \text{otherwise} \end{cases}$$

and $ub(v_1, \ldots, v_k) = \langle v''_1, \ldots, v''_m \rangle$ where

$$v''_j = \begin{cases} v_{i_j} & \text{if } v_{i_j} \neq \triangle \\ \top_j & \text{otherwise} \end{cases}$$

Secondly, we prove in Lemma 4 that given $a = lb(v_1, \ldots, v_k)$ and $b = ub(v_1, \ldots, v_k)$, we have $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R) = \sigma_{\rho(\ell,a,b)}(R)$ if the k-th prefix of l is $\{x_1, \ldots, x_k\}$. Before that, we first define prefix set.

Definition 8 (Prefix Set)**.** *Given an attribute sequence (i.e., an index) $\ell = x_1 \prec x_2 \prec \cdots \prec x_m$, the k-th prefix of ℓ is $\{x_1, \ldots, x_k\}$ if $k \leq m$, and it is $\{x_1, \ldots, x_m\}$ otherwise.*

Lemma 4. *Given $a = lb(v_1, \ldots, v_k)$, $b = ub(v_1, \ldots, v_k)$, and an index ℓ whose k-th prefix is $\{x_1, \ldots, x_k\}$, then*

$$\sigma_{x_1=v_1,\ldots,x_k=v_k}(R) = \sigma_{\rho(\ell,a,b)}(R),$$

holds for any $R \subseteq \mathcal{D}$.

Proof. Without loss of generality, we assume that $x_1, \ldots, x_k$ are the first k attributes of the relation R. Note that, if this is not the case, then we can conceptually reorganise the columns of R.

We prove the lemma by induction on k. For the base case $k = 1$, it is trivial that

$$\{t \in R \mid t(x_1) = v_1\} = \{t \in R \mid lb(v_1) \sqsubseteq_\ell t \sqsubseteq_\ell ub(v_1)\}$$

holds for ℓ whose first attribute is x_1, since $lb(v_1) = \langle v_1, \bot_2, \ldots, \bot_m \rangle$ and $ub(v_1) = \langle v_1, \top_2, \ldots, \top_m \rangle$. Assuming that this holds for $k = n$, we will show that it also holds

for $k = n + 1$. By the induction hypothesis, we know that

$$\{t \in R \mid t(x_1) = v_1, \ldots, t(x_n) = v_n\} =$$
$$\{t \in R \mid lb(v_1, \ldots, v_n) \sqsubseteq_\ell t \sqsubseteq_\ell ub(v_1, \ldots, v_n)\}$$

holds for index ℓ whose n-th prefix is $\{x_1, \ldots, x_n\}$. Now, consider

$$\{t \in R \mid lb(v_1, \ldots, v_n, v_{n+1}) \sqsubseteq_{\ell'} t \sqsubseteq_{\ell'} ub(v_1, \ldots, v_n, v_{n+1})\}$$

where the $(n+1)$-th prefix of ℓ' is $\{x_1, \ldots, x_n, x_{n+1}\}$, we can rewrite it as

$$\{t \in \sigma_{x_{n+1}=v_{n+1}}(R) \mid lb(v_1, \ldots, v_n) \sqsubseteq_\ell t \sqsubseteq_\ell ub(v_1, \ldots, v_n)\}$$

where ℓ is obtained from ℓ' by swapping x_{n+1} with the attribute at position $n + 1$. This is because

$$lb(v_1, \ldots, v_n, v_{n+1}) = \langle v_1, \ldots, v_n, v_{n+1}, \bot_{n+2}, \ldots, \bot_m \rangle$$
$$ub(v_1, \ldots, v_n, v_{n+1}) = \langle v_1, \ldots, v_n, v_{n+1}, \top_{n+2}, \ldots, \top_m \rangle$$
$$lb(v_1, \ldots, v_n) = \langle v_1, \ldots, v_n, \bot_{n+1}, \bot_{n+2}, \ldots, \bot_m \rangle$$
$$ub(v_1, \ldots, v_n) = \langle v_1, \ldots, v_n, \top_{n+1}, \top_{n+2}, \ldots, \top_m \rangle$$

As a result, we have

$$\{t \in R \mid lb(v_1, \ldots, v_n, v_{n+1}) \sqsubseteq_{\ell'} t \sqsubseteq_{\ell'} ub(v_1, \ldots, v_n, v_{n+1})\} =$$
$$\{t \in R \mid t(x_1) = v_1, \ldots, t(x_n) = v_n, t(x_{n+1}) = v_{n+1}\},$$

and the lemma holds. □

From Lemma 4, to transform a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ into an equivalent lex search, the index for the lex search can be any sequence of all attributes of R such that the first k attributes are $x_1, \ldots, x_k$ in an arbitrary order. Thus, *we also use $\ell = x_1 \prec \cdots \prec x_k$ which is only a subsequence of the attributes of R to denote an*

index, since the chaining order of the remaining attributes is irrelevant for the lex search.

Example 17. *Consider the primitive searches in the second column of Table 4.1, their corresponding lex searches are illustrated in the third to fifth columns, where the third column shows the lower bound a, the forth column shows the upper bound b, and the fifth column shows the index ℓ. Here, given a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$, the index is selected as $\ell = x_1 \prec \cdots \prec x_k$. Thus, each lex search uses a distinct index.* □

Remarks. The lex searches $\sigma_{\rho(\ell,a,b)}(R)$ constructed from primitive searches $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$, as discussed in above, are in a special form. That is, for any attribute $x_i \in \{x_1,\ldots,x_k\}$ we have $a(x_i) = b(x_i) = v_i$, and for any attribute $x_i \in A_R\backslash\{x_1,\ldots,x_k\}$ we have $a(x_i) = \bot_i$ and $b(x_i) = \top_i$. Thus, the results of a lex search form a consecutive interval in the lexicographical order of all tuples of R with respect to ℓ. As a result, any one-dimensional order-based index (e.g., B-tree) can be used to implement ℓ, and a lex search can be executed in linear-log time in the size of the output in the worst case, i.e., $O(|\sigma_{\rho(\ell,a,b)}(R)|\log n)$ where n is the number of tuples in the relation R. It is worth mentioning that for general range searches, we will need a multi-dimensional index (e.g., R-tree) to implement ℓ, which has a higher time complexity and runs slower than one-dimensional index such as B-tree. Thus, in this paper we only consider the special range searches, which we refer to as lex searches. Lex searches can be supported by one-dimensional indexes.

On the other hand, it is easy to construct an example such that the k-th prefix of ℓ is not $\{x_1,\ldots,x_k\}$, and the results of $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ do not form a consecutive range in the lexicographical order of all tuples of R with respect to ℓ. Thus, this primitive search cannot be transformed into a lex search using index ℓ, and thus cannot be sped up by ℓ. For example, for $R(x,y) = \{\langle 1,1\rangle, \langle 1,2\rangle, \langle 2,1\rangle\}$ and $\ell = x \prec y$, we have $\sigma_{y=1}(R) = \{\langle 1,1\rangle, \langle 2,1\rangle\}$ which is the first and third tuple in the lexicographical order ℓ. In view of this, we say that an index *covers* a primitive search if it can be used to speed up the primitive search by the special range search. We have the following corollary.

Corollary 1 (INDEX COVER)**.** *An index ℓ covers a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}(R)$ for all $R \subseteq \mathcal{D}$* ***if and only if*** *the k-th prefix of ℓ is $\{x_1, \ldots, x_k\}$.*

As the lex search that is transformed from a primitive search is uniquely determined by the index and the primitive search, we focus our discussions on indexes rather than lex searches in the remainder of the paper.

4.2.2 Minimum Index Selection

Due to the lower look-up time complexity of lex searches compared with that of linear scan, indexes are essential for efficient Datalog program computations. However, when constructing indexes, the question remains: *what is the best set of indexes needed to cover all primitive searches* for a given relation. In this Section we define the minimum index selection problem.

Before formally defining our problem, we first establish some additional notations. Firstly, we abstract a primitive search $\sigma_{x_1=v_1,\ldots,x_k=v_k}$ as its set of search attributes, which we refer to as a *search* and is denoted by $s = \{x_1, \ldots, x_k\}$. This is because the constants $v_1, \ldots, v_k$ are irrelevant to index creation. Secondly, given a set S of searches and a set $\mathcal{L}$ of indexes on a relation R, we would like to know whether $\mathcal{L}$ can cover S. Note that, since all primitive searches with the same set of attributes (i.e., the same search) can be covered by the same index, in the following when referring search set we use the set-based semantics. We formalise this via the **l-cover** predicate.

Definition 9 (L-COVER)**.** *Given a set S of searches and a set $\mathcal{L}$ of indexes on a relation R, we define a predicate* ***l-cover***$_S(\mathcal{L})$ *which is true if for every search $s \in S$, there exists an index $\ell \in \mathcal{L}$ that covers s.*

Then, based on the definition of **l-cover**, we would like to find the smallest set of indexes that cover a search set S. The rationales of minimising the number of indexes are as follows. Firstly, following Corollary 1, an index represented by an attribute sequence ℓ may cover a multitude of searches assuming the elements of its prefixes coincide with the attributes of the searches. For example, two searches $s_1 = \{x\}$ and $s_2 = \{x, y\}$ on a relation can be covered by the same index $\ell = x \prec y$. Secondly,

for a search that can be covered by multiple indexes, the benefits of the different indexes are the same, i.e., they will result in the same running time. Thirdly, the fewer the indexes, the lower the creation and maintenance costs of these indexes.

As indexes and searches on different relations are independent, we consider each relation separately. We formulate our problem as follows.

Problem 2 (**Minimum Index Selection Problem (MISP)**). *Given a set S of searches on a relation R, the minimum index selection problem is to find a set of indexes with the minimum cardinality such that all searches of S are covered by the index set, i.e.,*

$$f_S = \arg \min_{\mathcal{L}:\textbf{\textit{l-cover}}_S(\mathcal{L})} |\mathcal{L}|.$$

Example 18. *Continuing Example 17, the set of searches in Table 4.1 is $S = \{\{x\},\{x,y\},\{x,z\},\{x,y,z\}\}$. It can be covered by two indexes $\ell_1 = x \prec y \prec z$ and $\ell_2 = x \prec z$, which is shown in the sixth column of Table 4.1; this is smaller than the four indexes used in Example 17. Indeed, two is the smallest number of indexes to cover S, since it is easy to see that $\{x,y\}$ and $\{x,z\}$ cannot be covered by the same index.*

4.3 Computing The Optimal MISP

In this Section, we propose an algorithm to solve MISP optimally in polynomial time. We begin with discussing the inviability of a brute-force approach.

4.3.1 Inviability of a Brute-force Approach

Before presenting our algorithm, we discuss the size of the search space of MISP. If it is very large, then a brute-force algorithm is not viable, especially for high performance engines.

Given a set S of searches on a relation R, let A be the set of attributes of R that are relevant for the searches, i.e., $A = \bigcup_{s \in S} s$. We use $\mathcal{L}_A$ to represent the set of all possible permutation/sequences that may be formed by the elements of A, i.e., $\mathcal{L}_A = \bigcup_{X \subseteq A, X \neq \emptyset} \mathsf{Pm}(X)$. Here, $\mathsf{Pm}(X)$ denotes the set of *permutations* of a set X which is the set of all possible sequences formed by all elements of X such that

each element occurs exactly once. Now, we bound $|\mathcal{L}_A|$. Although constructing a closed form is hard, it can be bounded by the following lemma.

Lemma 5. *Given a set A of m attributes (i.e., $A = \{x_1, \ldots, x_m\}$), the cardinality of the set $\mathcal{L}_A$ of all sequences of A is bounded by*

$$m! \leq |\mathcal{L}_A| \leq \mathsf{e} \cdot m!.$$

Proof. The lower bound is given by $|\mathsf{Pm}(A)| = m!$, since $\mathsf{Pm}(A) \subseteq \mathcal{L}_A$. The upper bound is computed as follows,

$$\begin{aligned}
|\mathcal{L}_A| &= \left| \bigcup_{X \subseteq A, X \neq \emptyset} \mathsf{Pm}(X) \right| = \sum_{X \subseteq A, X \neq \emptyset} |X|! \\
&= \sum_{1 \leq i \leq m} \binom{m}{i} i! = m! \sum_{1 \leq i \leq m} \frac{1}{(m-i)!} \\
&= m! \sum_{0 \leq i \leq m-1} \frac{1}{i!} \\
&\leq m! \sum_{i \geq 0} \frac{1}{i!} = \mathsf{e} \cdot m!
\end{aligned}$$

where the second equality follows from the fact that, for any $X \subseteq A$ and $Y \subseteq A$ with $X \neq Y$, we have $\mathsf{Pm}(X) \cap \mathsf{Pm}(Y) = \emptyset$. □

Note that, the absolute error of the over-approximation of $|\mathcal{L}_A|$ is small, i.e., $\mathsf{e} \cdot m! - |\mathcal{L}_A| = m! \sum_{i \geq m} \frac{1}{i!} = \sum_{i \geq 0} \frac{m!}{(i+m)!} \leq \sum_{i \geq 0} \frac{1}{i!} = \mathsf{e}$. The values of $|\mathcal{L}_A|$ and the relative error $\varepsilon = \frac{\mathsf{e} \cdot m! - |\mathcal{L}_A|}{|\mathcal{L}_A|}$ of its over-approximation, for m varying between 1 and 9, is given in the table below:

m	$\lvert\mathcal{L}_A\rvert$	$\varepsilon \cdot 100$
1	1	171.828
2	4	35.914
3	15	8.731
4	64	1.936
5	325	0.367
6	1956	0.059
7	13699	0.008
8	109600	0.001
9	986409	≈ 0.000

Recall that, MISP searches for the smallest subset of $\mathcal{L}_A$ that covers all primitive searches on a relation. Thus, a brute-force approach would require to iterate through all subsets of $\mathcal{L}_A$. Then, the search space of a brute-force approach is $2^{\mathcal{L}_A} = \{\mathcal{L} \mid \mathcal{L} \subseteq \mathcal{L}_A\}$, and its size is $|2^{\mathcal{L}_A}| = 2^{|\mathcal{L}_A|}$. Using the approximation of $|\mathcal{L}_A|$ in Lemma 5, we obtain a complexity of $O(2^{\mathrm{e}\cdot m!})$.

Theorem 2. *A brute-force approach for MISP exhibits a worst-case time complexity of* $O(2^{m^m})$.

Proof. As discussed above, the time complexity of a brute-force approach for MISP is $O(2^{\mathrm{e}\cdot m!})$. Then, this theorem follows from Sterling's approximation of $m!$. Note that, the approximation becomes more precise for a large m. □

As a result, a brute-force approach becomes intractable very quickly. For example, for a relation with 4 attributes, a brute-force MISP algorithm has to test $2^{64} \approx 1.8 \times 10^{19}$ different subsets of $\mathcal{L}_A$ for coverage and minimality.

4.3.2 Computing MISP via Chain Cover

In view of the inviability of a brute-force approach, we propose to solve MISP via computing a chain cover of the searches. In the following, we first formulate the minimum chain cover problem (MCCP) and prove that an optimal MISP solution can be obtained from an optimal MCCP solution. Then, we propose a polynomial-time algorithm MinIndex that solves MISP optimally.

4.3.2.1 Minimum Chain Cover Problem

We define a search chain C as a set of searches $\{s_1, \ldots, s_k\}$ that subsume each other and form a total order, i.e., $C \equiv s_1 \subset s_2 \subset \cdots \subset s_k$. A search chain is related to an index as follows.

Lemma 6. *Given a search chain $C = s_1 \subset s_2 \subset \cdots \subset s_k$, we can construct an index to cover all searches of C.*

Proof. We prove this lemma by constructing such an index that covers all searches of C. Let $s_i - s_{i-1}$ denote the set of attributes of s_i that are not in s_{i-1}. Then, it is easy to see that any index conforming with $s_1 \prec (s_2 - s_1) \prec \cdots \prec (s_k - s_{k-1})$ is such an index, i.e., attributes of $s_{i+1} - s_i$ appear later than attributes of $s_i - s_{i-1}$. Note that, the attributes of s_1 and the attributes of $s_i - s_{i-1}$ can be ordered arbitrarily, respectively, within their sets of attributes. □

Following Lemma 6, we say that a search chain C covers all its searches, i.e., C covers s for every search $s \in C$. Then, we would like to know whether a set $\mathcal{C}$ of search chains can cover all searches in a search set S. We formalise this via the **c-cover** predicate.

Definition 10 (C-COVER)**.** *Given a set S of searches and a set $\mathcal{C}$ of search chains on a relation R, we define a predicate $\textbf{\textit{c-cover}}_S(\mathcal{C})$ which is true if for every search $s \in S$, there is a search chain $C \in \mathcal{C}$ that covers s, i.e.,*

$$\textbf{\textit{c-cover}}_S(\mathcal{C}) = \forall s \in S : \exists C \in \mathcal{C} : s \in C.$$

Now, we are ready to define our minimum chain cover problem, which aims to find the smallest set of search chains to cover all searches in a given set of searches.

Problem 3 (**Minimum Chain Cover Problem (MCCP)**)**.** *Given a set S of searches on a relation R, the minimum chain cover problem is to find the minimum set g_S of search chains to cover S, i.e.,*

$$g_S = \arg \min_{\mathcal{C}:\textbf{\textit{c-cover}}_S(\mathcal{C})} |\mathcal{C}|$$

The rationality of defining MCCP is that given a set C of search chains covering all searches in a search set S, we can construct a set of indexes of cardinality $|C|$ to cover S by following Lemma 6. Thus, the smaller the cardinality of C the better.

Moreover, there is a one-to-one correspondence between solutions of MISP and solutions of MCCP, as proved by the following lemma.

Lemma 7. *Given any search set S on a relation R, there is a one-to-one correspondence between search chains C that cover S and indexes $\mathcal{L}$ that cover S, such that $|C| = |\mathcal{L}|$.*

Proof. Following from Lemma 6, we know that given any set C of search chains that cover S, we can construct an index set of cardinality $|C|$ to cover S. Thus, what remains to be proved in this lemma is that given any index ℓ, we can construct a search chain C to cover all searches that are covered by ℓ.

Given an index ℓ and a set S of searches, we let S_ℓ denote the subset of S that are covered by ℓ. We will show that S_ℓ is a search chain. Firstly, it is easy to see that for any $s, s' \in S_\ell$, we have $|s| \neq |s'|$. Secondly, following Corollary 1, we know that for any $s, s' \in S_\ell$, we have either $s \subset s'$ or $s' \subset s$, since the k-th prefix of ℓ is a subset of a $(k+1)$-th prefix of ℓ.

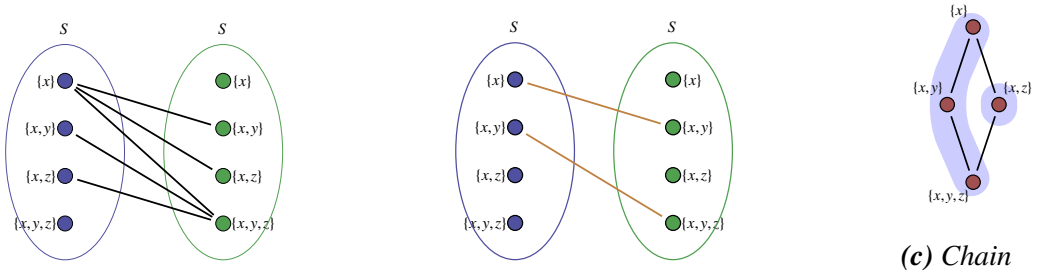
Thus, the lemma holds. □

Following from Lemma 7, we have the following corollary, which states that we can obtain an optimal MISP solution from an optimal MCCP solution.

Corollary 2. *Given any search set S on a relation R, an optimal MISP solution can be obtained from an optimal MCCP solution.*

4.3.2.2 A Polynomial-time MISP Algorithm

We have shown in Corollary 2 that we can obtain an optimal MISP solution from an optimal MCCP solution. The good news is that MCCP can be solved optimally in polynomial time by the Dilworth's Theorem [179], which states that in a finite partial order, the size of a maximum anti-chain is equal to the minimum number of chains needed to cover its elements. An anti-chain is a subset of a partially



(a) *Bipartite graph* ***(b)*** *Maximum matching* ***(c)*** *Chain cover*

Figure 4.5: *Running example of computing MCCP for searches* $\{x\}$, $\{x,y\}$, $\{x,z\}$, *and* $\{x,y,z\}$.

ordered set such that any two elements in the subset are unrelated, and a chain is a totally ordered subset of a partial ordered set. Although Dilworth's Theorem is non-constructive, there exists constructive versions that solve the minimum chain cover problem either via the maximum matching problem in a bipartite graph [181] or via a max-flow problem [182]. Both problems are optimally solvable in polynomial time.

The general idea of computing a minimum chain cover for a search set S is as follows. Firstly, a bipartite graph $G = (U, V, E)$ is constructed such that there is a vertex in both U and V for each search $s \in S$, and there is an edge between $s \in U$ and $s' \in V$ if s is a proper subset of s' (i.e., $s \subset s'$). Then, there is a one-to-one correspondence between sets of search chains of S and matchings of G. Recall that, a matching of G is a set $\mathcal{M}$ of edges of G such that each vertex of U and V appears at most once in $\mathcal{M}$. For example, given a matching $\mathcal{M} \subseteq E$, a set of search chains of cardinality $|S| - |\mathcal{M}|$ can be constructed. Specifically, chains are constructed from the matching set by finding the searches that start a chain, i.e., are the smallest element of a chain and do not have a predecessor. As a result, a minimum set of search chains can be constructed from a maximum matching of G. The pseudocode of computing a minimum chain cover is shown in Algorithm 5.

Then, given a set C of search chains that cover the search set S, a set $\mathcal{L}$ of indexes of the same cardinality as C can be constructed to cover S by following the proof of Lemma 6. The pseudocode is shown in Algorithm 6, and denoted by MinIndex.

Input: A set S of searches
Output: A minimum chain cover C of S

$\mathcal{M} \leftarrow \mathsf{MaximumMatching}(S, S, \{(s, s') \in S \times S \mid s \subset s'\})$;
Initialize C to be the empty set;
for all $u_1 \in S$ ***s.t.*** $\nexists (u_0, u_1) \in \mathcal{M}$ **do**
    Find max. set $\{(u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k)\} \subseteq E$;
    Add $u_1 \subset u_2 \subset u_3 \subset \cdots \subset u_{k-1} \subset u_k$ to C ;
end
return C

Algorithm 5: MinChainCover(S)

Input: A set S of searches
Output: A minimum set $\mathcal{L}$ of indexes to cover S

$C \leftarrow \mathsf{MinChainCover}(S)$;
Initialize $\mathcal{L}$ to be the empty set;
for all $s_1 \subset s_2 \subset \cdots \subset s_{k-1} \subset s_k \in C$ **do**
    Add to $\mathcal{L}$ an arbitrary index conforming with
    $s_1 \prec s_2 - s_1 \prec \cdots \prec s_k - s_{k-1}$;
end
return $\mathcal{L}$

Algorithm 6: MinIndex(S)

The correctness of MinIndex (Algorithm 5 and Algorithm 6) directly follows from Corollary 2 and the Dilworth's Theorem [179]. Let m be the number of distinct attributes in S; note that, m is at most the number of attributes in a relation. Then, the time complexity of MinIndex is bounded by the following theorem.

Theorem 3. *The time complexity of* MinIndex *(Algorithm 5 and Algorithm 6) is* $O(|S|^{2.5} + |S|^2 \cdot m)$.

Proof. The time complexity follows from the facts that, constructing the bipartite graph G takes $O(|S|^2 \cdot m)$ time, computing the maximum matching in G takes $O(|S|^{2.5})$ time, and both constructing chain cover from matching $\mathcal{M}$ and constructing indexes from chain cover take $O(|S| \cdot m)$ time. □

Note that, as both $|S|$ and m are not large in practice (e.g., they are at most hundreds), the running time of MinIndex usually is negligible compared with the total running time of a Datalog program.

Example 19. *Consider the search set* $S = \{\{x\}, \{x, y\}, \{x, z\}, \{x, y, z\}\}$ *in Table 4.1*

that needs to be covered by the smallest set of indexes. First, we construct a bipartite graph with nodes in S in both partitions. The edge set is given by the strict subset relationship between a search pair, i.e., $(\{x\},\{x,y\}),(\{x\},\{x,z\}),(\{x\},\{x,y,z\}),(\{x,y\},\{x,y,z\})$ *and* $(\{x,z\},\{x,y,z\})$. *The bipartite graph is depicted in Fig. 4.5a, and the matching set of the maximum matching solution is depicted in Fig. 4.5b. The solution of the maximal matching algorithm is given by the matching set,* $\mathcal{M} = \{(\{x\},\{x,y\}),(\{x,y\},\{x,y,z\})\}$. *With Algorithm 5 we obtain a chain cover* C *containing the following chains,* $\{x\} \subset \{x,y\} \subset \{x,y,z\}$ *and* $\{x,z\}$ *that are depicted in Fig. 4.5c.*

Then, Algorithm 6 converts the chain cover to indexes. The first chain is converted as follows: $\{x\} \subset \{x,y\} \subset \{x,y,z\} \Rightarrow \{x\} \prec \{x,y\} - \{x\} \prec \{x,y,z\} - \{x,y\} \Rightarrow x \prec y \prec z$. *The second chain consists of a single element* $\{x,z\}$. *This chain induces two possible indexes, i.e.,* $x \prec z$ *and* $z \prec x$, *and the choice is arbitrary to find an optimal solution for the MISP problem.*

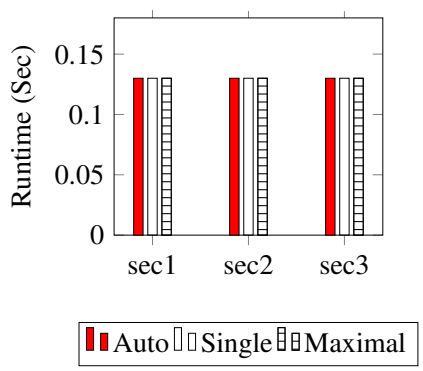
4.4 Experiments

In this Section, we evaluate our auto-indexing scheme by measuring an implementation of it, and also some alternative schemes, in a production-strength Datalog engine Soufflé [31]. The outcome of our evaluations is to validate the following claims.

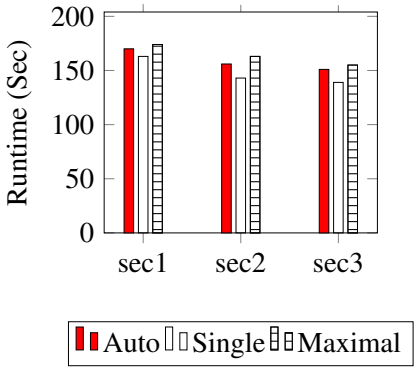
Claim-I: Negligible Index Selection Overhead. The time taken for selecting the indexes using our auto-indexing scheme does not substantially slow down the compilation phase compared to alternative indexing schemes.

Claim-II: Significant Performance Impact. Our auto-indexing scheme provides a good combination of fast runtime evaluation and low memory footprint.

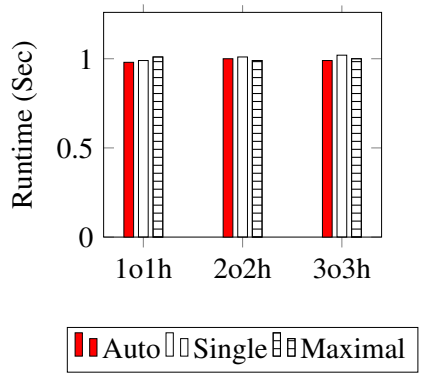
Claim-III: Competitive with Hand Optimisations. Our auto-indexing scheme delivers runtime evaluation speed and memory usage that compare well with what can be obtained by hand optimisations.



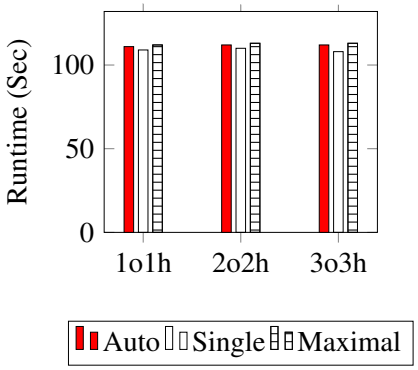
(a) Code generation for cloud security analysis



(b) Code compilation for cloud security analysis



(c) Code generation for Doop *program analysis*



(d) Code compilation for Doop *program analysis*

Figure 4.6: Soufflé *code generation and compilation time*

4.4.1 Experimental Setup

4.4.1.1 Platform

Our experiments were performed on a 4 Core, 8 Hardware Threads, Intel(R) Core(TM) i7-7700K CPU at 4.20GHz with 64GB of physical RAM running Ubuntu 16.04.3 LTS on the bare-metal. The experiments were conducted in isolation without virtualisation so that runtime results are robust.

4.4.1.2 Compared Indexing Schemes

We compare the following three indexing schemes, all implemented by us in Soufflé.

- Auto: our auto-indexing scheme presented in Algorithm 6.
- Maximal: one index for each distinct search on a relation.

- Single: only one index for each relation. To choose the best index for a relation R for a given workload, we first count the frequency of each individual search S on R which is obtained by instrumenting the search pattern while executing the Datalog program once. Then, the best single index is selected as the one whose set of covered searches has the maximum total frequency. This can be computed in quadratic time to the number of searches by dynamic programming (cf. Chapter 12, [183]); we omit the details.

Intuitively, these two alternative indexing schemes, Maximal and Single, should be especially good for the execution speed and the memory efficiency, respectively. However Maximal uses much memory for the numerous indexes, and Single doesn't cover every search, and thus could be very slow in evaluating the program. Our experiments in Section 4.4.2.2 validate these expectations, and show that Auto offers an excellent compromise, with runtime similar to Maximal and much less than Single, and using memory similar to Single, and substantially less than Maximal.

In addition, for the workloads of one use-case from program analysis, we also compare our auto-indexing scheme in Soufflé to another Datalog system `PA-Datalog`[2]. Note that, the ruleset used with `PA-Datalog` has been heavily hand-optimised through months of work by experts, specially for the use-case. Because these are different engines, the comparison of speed and memory is not truly apples-to-apples. Nevertheless, we will illustrate in Section 4.4.2.3 that our auto-indexing scheme in Soufflé results in better performance than the state-of-the-art for hand-optimised processing.

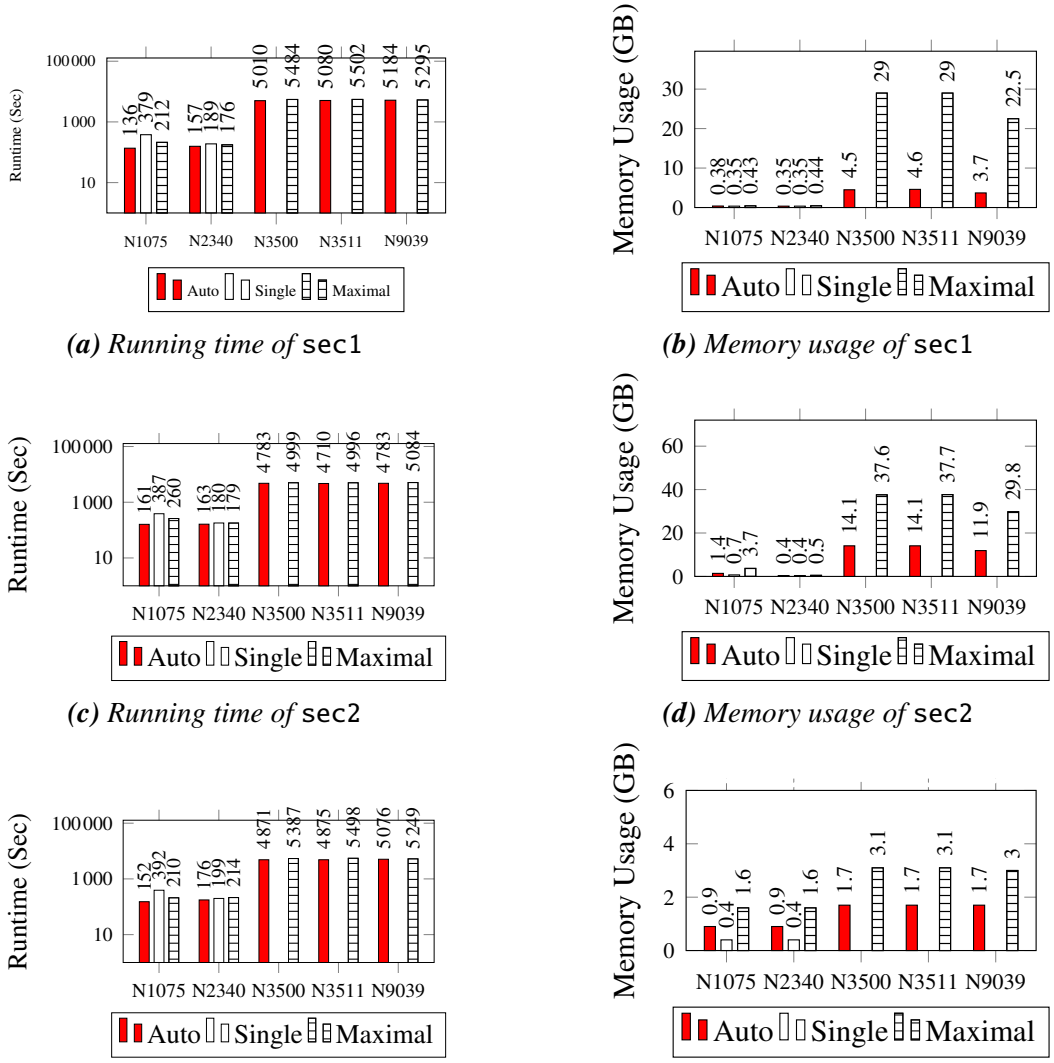
4.4.2 Experimental Results

We present experimental results to validate our claims in the following three subsections.

4.4.2.1 Index Selection Overhead

Soufflé translates a given Datalog program into C++ code (called the code generation phase), compiles the C++ code into binary executable code (called the code

[2] `PA-Datalog` is a variant of Logicblox Version 3 optimised for program analysis



(a) *Running time of* `sec1`

(b) *Memory usage of* `sec1`

(c) *Running time of* `sec2`

(d) *Memory usage of* `sec2`

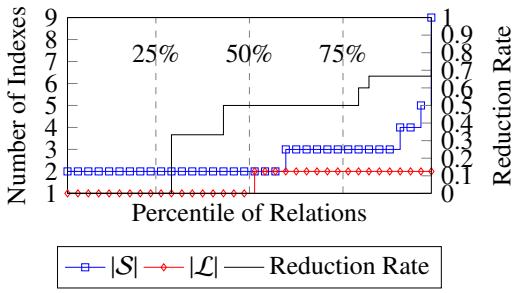
(e) *Running time of* `sec3`

(f) *Memory usage of* `sec3`

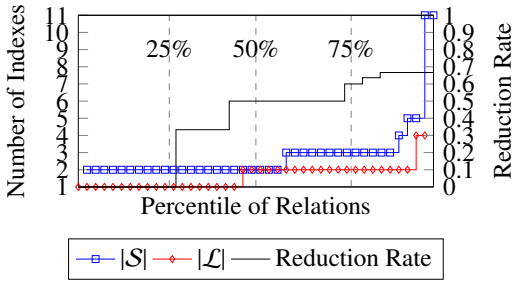
Figure 4.7: *Experimental results for cloud security analysis*

compilation phase), and then execute the binary code on the EDB (i.e., input relations) to compute the IDBs (called the code execution phase). For more details of SOUFFLÉ, please refer to [31, 40]. Index selection occurs in the code generation phase.

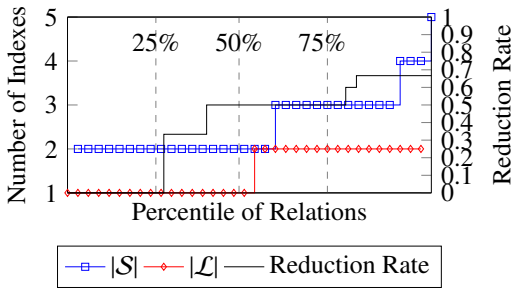
In order to quantify the overhead of our Auto indexing scheme, we report the code generation time as well as the code compilation time for all three indexing schemes, Auto, Single, and Maximal, for both use cases in Fig. 4.6. Recall that index selection occurs during code generation; however, different index choices may lead to different work in the code compilation phase too.



(a) Index distribution of `sec1`



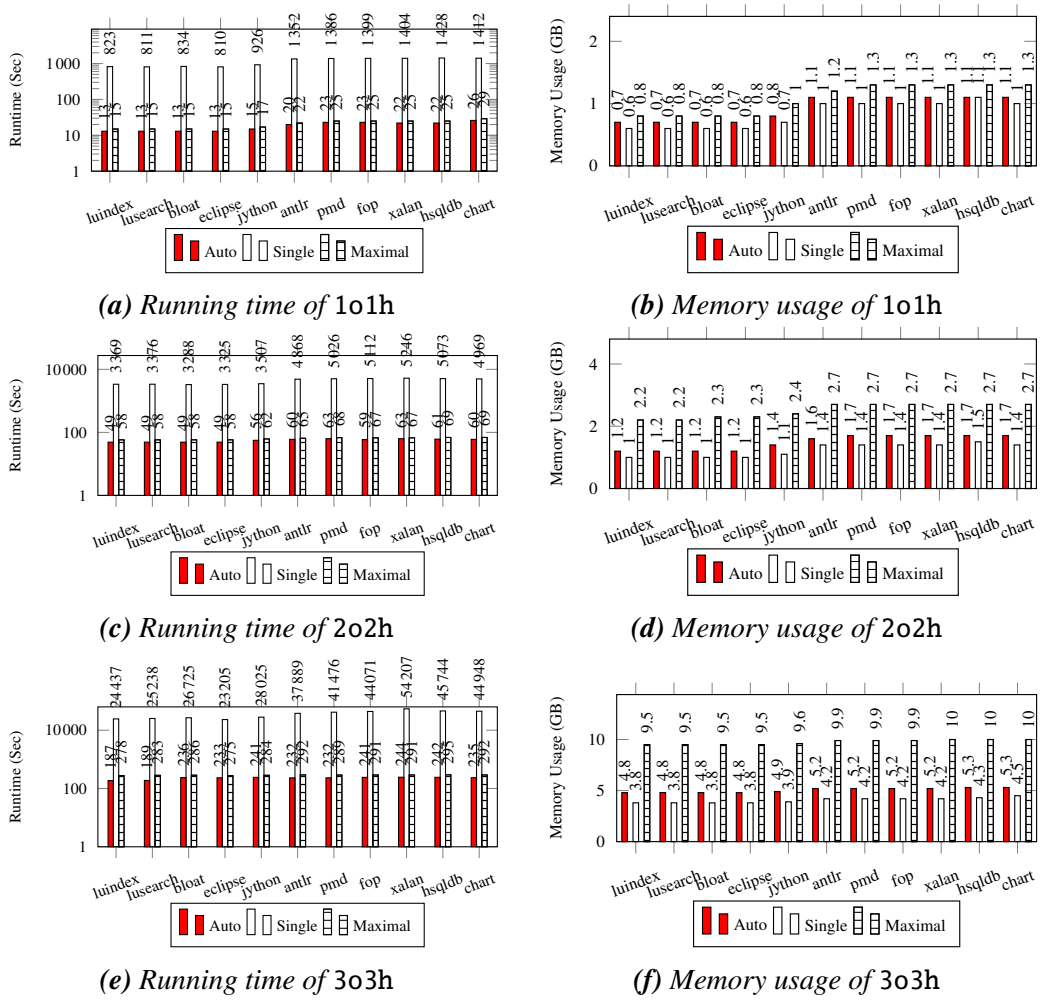
(b) Index distribution of `sec2`



(c) Index distribution of `sec3`

Figure 4.8: *Index Distributions*

As shown in Fig. 4.6a and Fig. 4.6c, the code generation time of the three indexing schemes are almost the same. Note that we have not included in the measurement for Single the extra preliminary activities that collect statistic information such as frequencies of searches. On the other hand, the code compilation time varies slightly for different indexing schemes as shown in Fig. 4.6b and Fig. 4.6d, and the more indexes to compile the longer the compilation time. The main reason is that each index requires additional templatised comparator functions that the C++ compiler needs to unroll at template instantiation time. Thus, in this phase Auto was 4% to 8% slower than Single, and 2% to 5% faster than Maximal. Overall we conclude that the differences in generation and compilation effort among the three indexing schemes are not significant, despite the computations needed to solve the MISP.



(a) Running time of 1o1h *(b) Memory usage of* 1o1h

(c) Running time of 2o2h *(d) Memory usage of* 2o2h

(e) Running time of 3o3h *(f) Memory usage of* 3o3h

Figure 4.9: Doop *program analysis experiment results*

4.4.2.2 Evaluation-time Performance

Eval-I: Running Time. The running time of the code execution phase for the three cloud security analyses on VPC networks are illustrated in Figures 4.7a, 4.7c, and 4.7e, each showing the three alternative index selection approaches: Auto, Single, and Maximal. Overall, both Auto is similar to Maximal and each outperforms Single, in the running time. Indeed, due to the lack of indexes to speed up all searches, Single takes an excessively long time (i.e., more than 24 hours) for processing the three large VPC networks, N3500, N3511, and N9039. Thus, Single is not able to process large-scale Datalog programs, and we omit from the figures the results for Single on these three large VPC networks.

The running time of Auto is almost the same as that of Maximal, and in fact

Auto is slightly faster, by 10% for sec1, 26% for sec2 and 13% for sec3. This is because when execution involves both constructing the indexes as the facts are read in, as well as doing the primitive searches. The latter aspect should in principle be the same for Auto and Maximal, but Maximal takes more time while it construct more indexes than Auto.

The running time of the execution phase for Auto, Single, and Maximal on the three Doop program analyses are illustrated in Figures 4.9a, 4.9c, and 4.9e. The general trend is similar to that for cloud security analysis. Although Single can complete all the Doop program analysis, it takes significantly more time than Auto and Maximal. The speedups of Auto over Maximal are 1.3 for 1o1h, 1.13 for 2o2h, and 1.7 for 3o3h.

Overall, we find that Auto is even a bit faster than Maximal, and it is significantly faster than Single. This validates our motivation to construct enough indexes so that every search is sped up.

Eval-II: Memory Usage. Now, we evaluate the memory usage of Auto compared to Single and Maximal. We define the memory usage improvement of an indexing scheme A over another scheme B as the ratio of memory usage or B compared to that of A.

The memory usages of Auto, Single, and Maximal for cloud security analyses, sec1, sec2, and sec3, are shown in Figures 4.7b, 4.7d, and 4.7f. We see that Single always consumes the smallest amount of memory, and Maximal always consumes the largest amount of memory. The memory usage improvement of Auto over Maximal can be up-to 6, e.g., see the memory usage for VPC networks N3500, N3511, N9087 in Fig. 4.7b. When compared to the memory usage lower bound as indicated by Single, Auto consumes at most two times more memory than the lower bound. Figures 4.9b, 4.9d, and 4.9f show the memory usage of Auto, Single, and Maximal for Doop program analysis. In general, the memory usage improvement of Auto over Maximal is around 2, and Auto consumes only around 20% more memory than Single.

Overall, Maximal consumes the largest amount of memory and Single con-

sumes the smallest. The memory usage of Auto is not far from that of Single, and much better than Maximal.

Eval-III: Distribution of Index Reduction. In this set of figures, we analyse the number of indexes constructed for the various Datalog programs. Recall that, given a set of searches S on a relation R, our Auto indexing scheme (i.e., Algorithm 5 and Algorithm 6 in Section 4.3) computes the smallest set of indexes $\mathcal{L}$ to cover/speed up all searches of S, while the Maximal indexing scheme constructs one index for each search in S, and Single constructs one index for each relation. Thus, the reduction ratio for the number of indexes of Auto over Maximal will be upper bounded by $|S|$ for a relation (which is the reduction ratio for Single over Maximal).

The distributions of $|S|$ among all relations that have at least two searches for the three cloud security analyses, `sec1`, `sec2`, and `sec3`, are shown as blue squares measured against the left-hand scale, in Figures 4.8a, 4.8b, and 4.8c, respectively. We can see that more than 50 percent of the relations have only two searches, and more than 80 percent of the relations have at most three searches; this means that for 80 of the relations, $|\mathcal{L}|/|\mathcal{S}|$ is at least $1/3$. In order to quantify the reduction ratio of Auto over Maximal, we define it as $1 - |\mathcal{L}|/|S|$. The distributions of the reduction ratio are shown as black line in Figures 4.8a, 4.8b, and 4.8c, measured against the right-hand scale. We can see that, for 25 percent of the relations, there is no reduction (i.e., $|S| = |\mathcal{L}|$), for another 25 percent of the relations, the reduction is around 50%, and for the remaining 50 percent of relations, the reduction is between 50% and 70%. Finally, the distributions of the actual number of indexes $|\mathcal{L}|$ constructed for the relations are shown as red diamond in Figures 4.8a, 4.8b, and 4.8c measured against the left-hand scale. For cloud security analyses `sec1` and `sec3`, the largest number of indexes constructed for a relation is only two, while the largest number searches on a relation is 9. For cloud security analysis `sec2`, the largest number of indexes constructed for a relation is three, while the largest number searches on a relation is 11. As shown in Fig. 4.10, similar results are also observed for the Doop program analysis.

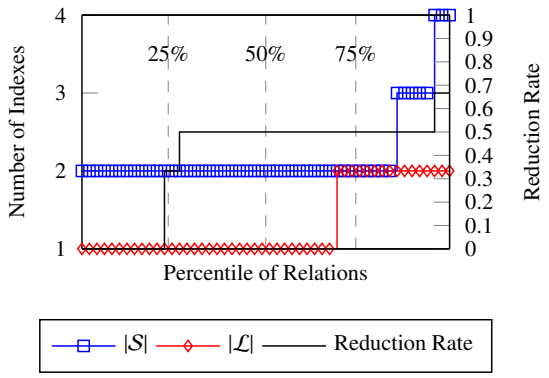


Figure 4.10: *Index distribution for* Doop *program analysis*

4.4.2.3 Against `PA-Datalog`

We refer to the experiments from Chapter 3, Figure 3.18. Here, we measured the heavily hand-optimised `PA-Datalog` system for the Doop program analysis. We can see that, the Soufflé measurements that use Auto are consistently faster and consumes much less memory than `PA-Datalog`. The running time improvement ranges from 3–5x, and the memory usage improvement ranges from 2–5x.

This demonstrates that our Auto indexing scheme also works well compared with hand optimisations.

4.4.2.4 Summary

Overall, the experimental results demonstrate the value of our Auto indexing scheme for large-scale Datalog computation. On running time, the Single index scheme is not able to compute some of our cloud security analyses within 24 hours. As shown in Figures 4.8a, 4.8b, and 4.8c, 50 percent of the relations need at least two indexes to cover all searches on a relation; thus, when only a single index is constructed for a relation, the uncovered searches need to be computed by linear scanning the entire relation. In addition, Auto also runs faster than Maximal, due to a reduction in the index construction/maintenance cost. As for memory usage, Auto consumes significantly less memory than Maximal, and only slightly more memory than Single. That is, Auto combines the desirable aspects of Single (low memory) and Maximal (speed) without the disadvantages of either. Finally, we have shown that Auto obtains performance that compares well with that of hand-optimised so-

lutions, constructed with great effort by experts.

4.5 Discussion

4.5.1 Index Selection in Datalog Engines

Datalog has been pro-actively researched in several computer science communities [184, 185, 186, 187], where a comprehensive introduction to Datalog can be found in [90]. A recent introduction text for optimising SQL queries is given by Bruno [177]. Driven by applications in data integration, networking, and program analysis, Datalog has recently regained considerable interest, e.g., see [64] for a survey of these developments. To facilitate these applications, general Datalog engines have been developed. For example, Logicblox version 3 [30], μZ [37], bddbddb [41], and Soufflé [31] are the state-of-the-art Datalog engines developed for program analysis. As shown in [34], indexes can greatly improve the execution efficiency of Datalog engines. However, the existing indexing scheme that uses order-based indexes has the limitation of one index per relation. To circumvent this limitation, a manual code-rewriting technique in the Doop[3] framework [28] was introduced that replicates a relation multiple times and creates a distinct index for each replica. This manual index creation, although resulting in an enormous speedup [34], requires end-users to be familiar with the underlying indexing mechanism of a Datalog engine. The manual code-rewriting technique is error-prone and painstakingly slow optimising Datalog programs by trial and error with hundreds of rules. The hand-optimised Datalog programs become obfuscated, and maintainability and readability are hampered. In this paper, we for the first time study the automatic optimal index selection problem for Datalog engines to accelerate their executions. We implemented our indexing technique in Soufflé, which results in significant runtime speedups. Our automatic index selection technique may also be beneficial to other Datalog engines, since the overhead for computing the optimal index selection is negligible while the selected indexes can significantly improve Datalog program evaluations.

[3]Note that Doop uses `PA-Datalog`, which is a variant of Logicblox version 3.

4.5.2 Index Selection in Relational Databases

In the context of relational databases, the problem of automatically selecting indexes for a set of database queries, referred to in the literature as the index selection problem (ISP) [171, 172, 173, 174], is well studied and has been shown to be NP-hard [175]. It is typically formulated as a variant of the 0-1 knapsack problem, which balances the overall execution time of queries for an index configuration (i.e., a subset of indexes that influence the performance of a query) and the cost of index maintenance. Our index selection problem differs from the classic ISP literature and to the best of our knowledge is the first formulation of such a problem. Firstly, in our case, we are restricted to support primitive searches only, which occur in equi-joins and simple value queries. Secondly, the nature of Datalog restricts the search predicate of each primitive search to be an equality predicate over the attributes of the relation. Thirdly, we further have the assumption that each primitive search benefits from being indexed, which is important for high-performance systems that need to accelerate all searches. Thus, we formulate our problem as automatically selecting the minimum number of indexes to cover all searches, and we present an algorithm to solve it optimally in polynomial time. Our index selection technique may be, in special cases, also applicable to general query engines that do not have Datalog as a front-end language; for example, it could work for bottom-up engines that use SQL defined queries.

4.5.3 Extensions

We discuss possible extensions of our auto-indexing scheme.

Single and Multiple Inequalities. Although we limited the search predicate in our primitive search to be equalities of left-hand-side attributes and right-hand-side constants, our techniques can be extended for inequality constraints on one attribute: First, the bounds of the lex search predicate are to be adapted for the attribute of the inequality. Second, the attribute has to be the last one among the attributes in the search with respect to the lexicographical order. The ordering restriction is encoded in the bipartite graph $G = (U, V, E)$ by omitting edges in the standard construction.

Specifically, there is an edge between $S \in U$ and $S' \in V$ if (1) S is a proper subset of S', (2) S has no inequality, and (3) if S' has an inequality on attribute x, then S does not have x.

To support multiple inequalities, we may need to resort to other forms of indexes, e.g., the multi-dimensional index R-tree. For example, the general primitive search $\sigma_{1\leq x\leq 3,2\leq y\leq 4}$ can be translated into a range search in a multi-dimensional space. However, multi-dimensional indexes are generally more expensive to build and also more expensive to query. We leave the study of extending our techniques to multiple inequalities, and also to richer variants of Datalog [57], as our future work.

Loop Scheduling. Some Datalog engines such as Logicblox version 4 [136] use a leapfrog join that, while requiring users to specify indexes manually, alleviates users from specifying join order. Integrating our technique into such an engine is not obvious as we assume a fixed literal order before our technique is applied. Typically, this can be identified using a profiler like Soufflé profiler, or alternatively, loop schedules can be automated using heuristic techniques [142]. Our technique then can compute the optimal index assignment for the given loop schedule. During performance tuning of large Datalog programs, only a few rules require manual loop scheduling. Therefore, our preference is to fix loop orders rather than indexes for a better user experience. Soufflé's auto-scheduler typically resolves this automatically for the user. Nevertheless, it will be an interesting future work to integrate automatic loop scheduling and automatic indexing selection.

4.5.4 Limitations.

As we have shown, our indexing technique performs well use cases such as static analyses, since they typically contain hundreds of rules and relations with complex access patterns. However, for small Datalog programs with large inputs (as is typically for databases querying), our performance will be the same as the Maximal indexing scheme. In the case that memory usage is paramount, it may be beneficial to use the Single indexing scheme with manual literal scheduling using a light weight profiler, such as the Soufflé profiler. It is therefore of no surprise that Data-

log engines, designed for use cases similar to database querying, employ the Single indexing scheme and require user annotations for any additional indexes.

Chapter 5

Symbolic Extensions

In this chapter we describe a method of improving the performance of evaluating recursive Horn clauses on infinite domains and complex constraints. This work is aimed at both the model checking and verification communities. Much of this work has been published in the Acta Informatica journal [54], and Formal Methods in Computer-Aided Design [53].

5.1 Symbolic Extensions

Static analyses such as var-points-to, do-privileged etc. are well suited to Datalog based analysers due to the finiteness of the abstractions they represent. However, often such abstractions are too imprecise for detecting general error conditions. An example of the need for a deeper analysis is given below.

Example 20 (C Program)**.** *We consider an example inspired by the program discussed in the introduction of [188]. The example exhibits a situation that requires the use of symbolic constraint solvers:*

```
i = 0; x = j;
while (i<50) {i++; x++;}
if (j == 0) assert (x >= 50);
```

Here we have three variables, i, j and x. The while loop iterates as long as i is less than 50*, and we would like to asset that when j is equal to* 0 *then x is greater than* 50*.*

Given the language extensions in Soufflé*, we can express the program above in terms of constrained Horn clauses as follows:*

```
P(1,i0,j0,x0):-i0=0,x0=j0.
P(1,i1,j0,x1):-P(1,i0, j0, x0),i1=i0+1,x1=x0+1, i0<50.
P(2,i1,j1,x1):-P(1,i1,j1,x1),i1>=50.
false:-x_1>=50,j1!=0, P(2,i1,j1,x1).
```

Each predicate denotes an invariant *at a particular program location, e.g., P is inductive invariant. To prove that the assertion holds, we must derive an inductive invariant, i.e., non-false values for P.* □

While these extensions, when used with care, provided more expressiveness to users when writing Datalog based static analyses, for examples as the one above, Datalog evaluation methods such as semi-naïve cannot effectively perform the analysis due to the domain of i, j and x being infinite (or impractically large). Datalog evaluation techniques employed on this scenario would compute all the sets values variables i, j and x can have at each loop iteration resulting in an impractical amount of memory usage.

Instead, we employ an algorithm based on predicate abstraction and counter example guided abstract refinement (CEGAR). The algorithm is described in Chapter 2, Section 2.3. Recall, the algorithm starts with an abstract representation of the system of Horn clauses using an accumulated set of predicates. The algorithm extracts a counter-example from the abstract representation and determines if it is spurious or not. If the counter-example is spurious, the algorithm further refines the abstract representation and repeats the process, otherwise gives a user a counter example as proof of an error. The counter example is checked for being spurious using Craig interpolants. For example, we may check the following set of chained clauses:

```
P(1,i0,j0,x0):-i0=0,x0=j0.
P(1,i1,j0,x1):-P(1,i0, j0, x0),i1=i0+1,x1=x0+1, i0<50.
P(2,i1,j1,x1):-P(1,i1,j1,x1),i1>=50.
false:-x_1>=50,j1!=0, P(2,i1,j1,x1).
```

In the example, we might consider the path to the assertion in which the loop terminates after one iteration. This path could lead to an assertion violation if the conjunction of assignments and guards on the path (in SSA form) is satisfiable:

The solution to P is precisely an interpolant of the conjunction:

$$i_0 \doteq 0 \wedge x_0 \doteq j \wedge i_0 < 50 \wedge i_1 \doteq i_0+1 \wedge x_1 \doteq x_0+1 \tag{5.1}$$

$$\wedge\, i_1 \geq 50 \wedge j \doteq 0 \wedge x_1 < 50 \tag{5.2}$$

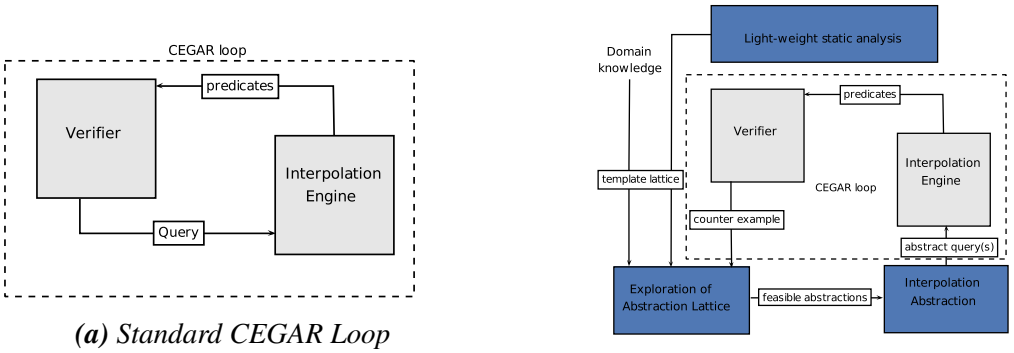
In the interpolation problem above, it is easy to see that the formula is unsatisfiable, and that the path therefore cannot cause errors. To improve our approximation we obtain predicates that prevent the path from being considered again in the CEGAR process. This is done by computing Craig interpolants for different partitionings of the conjuncts; we consider the case $(5.1) \wedge (5.2)$, corresponding to the point on the path where the loop condition is checked for the second time. An interpolant is a formula I that satisfies the implications $(5.1) \rightarrow I$ and $(5.2) \rightarrow \neg I$, and that only contains variables that occur in both (5.1) and (5.2); a model checker will use I as a candidate loop invariant.

The interpolation problem $(5.1) \wedge (5.2)$ has several solutions, including $I_1 = (i_1 \leq 1)$ and $I_2 = (x_1 \geq i_1 + j)$. What makes the example challenging is the fact that a theorem prover is likely to compute interpolants like I_1, recognising the fact that the loop cannot terminate after only one iteration as obvious cause of infeasibility. I_1 does not describe a property that holds across loop iterations, however; after adding I_1 as a predicate, a model checker would have to consider the case that the loop terminates after two iterations, leading to a similar formula $i_2 \leq 2$, and so on. The evaluation will only terminate after 50 loop unwindings; in similar situations with unbounded loops, picking interpolants like I_1 will lead to divergence (non-termination) of the CEGAR algorithm.

In contrast, the interpolant I_2 encodes a deeper explanation for infeasibility, the dependency between `i`, `x`, and `j`, and takes the actual assertion to be verified into account. Since I_2 represents an inductive loop invariant, adding it as predicate will lead to significantly faster convergence of the CEGAR algorithm.

The above example highlights a major performance problem in CEGAR based engines such as model checkers. The problem as highlighted in the thesis hypothesis, stems from the fact that the theorem prover is agnostic to a given input. Hence, similar in spirit to the evaluation in Chapter 3 we present a framework to specialise the interpolants obtained from the theorem prover.

In our solution we modify the existing CEGAR loop as shown in Figure 5.1. However, unlike the solution in Chapter 3 we do not use partial evaluation to syn-



(a) *Standard CEGAR Loop*

(b) *Modified CEGAR Loop*

Figure 5.1: *Architecture of Approach Compared to CEGAR*

thesise a theorem prover, instead we create a *template lattice* that is constructed manually or via a light weight static analysis (e.g., discover guards in loops). Using the templates we construct a lattice of abstract interpolation problems which we explore for feasibility. The set of feasible abstract problems are selected and given as input to a theorem prover which generates specialised interpolants that are incorporated into the set of accumulated predicates in CEGAR algorithm.

Our approach enables an CEGAR based engine e.g., a model checker to steer the theorem prover towards interpolants like I_2. A major advantage is that our approach is *solver-independent* and works by instrumenting the interpolation query, and therefore does not require any changes to the theorem prover.

The essence of our approach is observation that it is possible to over-approximate an interpolation problem. For instance, to obtain I_2, we over-approximate the interpolation query (5.1) $\wedge$ (5.2) in such a way that I_1 no longer is a valid interpolant:

$$(i_0 \doteq 0 \wedge x_0 \doteq j' \wedge i_0 < 50 \wedge i_1' \doteq i_0{+}1 \wedge x_1' \doteq x_0{+}1 \wedge \boxed{x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j})$$
$$\wedge (\boxed{x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j''} \wedge i_1'' \geq 50 \wedge j'' \doteq 0 \vee x_1'' < 50)$$

The rewriting consists of two parts: (i) the variables x_1, i_1, j are renamed to x_1', i_1', j' and x_1'', i_1'', j'', respectively; (ii) limited knowledge about the values of x_1, i_1, j is re-introduced, by adding the grey parts of the interpolation query. Note that the formula is still unsatisfiable. Intuitively, the theorem prover "forgets" the precise

value of x_1, i_1, ruling out interpolants like I_1; however, the prover retains knowledge about the difference $x_1 - i_1$ (and the value of j), which is sufficient to compute relational interpolants like I_2.

The terms $x_1 - i_1$ and j have the role of *templates,* and encode the domain knowledge that linear relationships between variables and the loop counter are promising building blocks for invariants. Template-generated abstractions represent the most important class of interpolation abstractions considered in this paper (but not the only one), and are extremely flexible: it is possible to use both template terms and template formulae, but also templates with quantifiers, parameters, or infinite sets of templates.

Templates are in our approach interpreted *semantically*, not *syntactically*, and it is up to the theorem prover to construct interpolants from templates, Boolean connectives, or other interpreted operations. To illustrate this, observe that the templates $\{x_1 - i_1, i_1\}$ would generate *the same* interpolation abstraction as $\{x_1, i_1\}$; this is because the values of $x_1 - i_1, i_1$ uniquely determine the value of x_1, i_1, and vice versa.

We have integrated interpolation abstraction into the model checker Eldarica [189], which uses recursion-free Horn clauses (a generalisation of Craig interpolation) to construct abstractions [119, 127]. Our experiments show that interpolation abstraction can prevent divergence and improve the speed of convergence of the model checker in cases that are often considered challenging.

Despite a simple implementation (requiring approx. 1000 lines of Scala code), interpolation abstractions are extremely flexible, and can incorporate domain-specific knowledge about promising interpolants, for instance in the form of *interpolant templates* used by the theorem prover. The framework can be used for a variety of logics, including arithmetic domains or programs operating on arrays or heap, and is also applicable for quantified interpolants.

5.2 Interpolation Abstractions

5.2.1 Basic Definitions

Lattices. A *poset* is a set D equipped with a partial ordering $\sqsubseteq$. A poset $\langle D, \sqsubseteq \rangle$ is *bounded* if it has a *least element* $\bot$ and a *greatest element* $\top$. We denote the *least upper bound* and the *greatest lower bound* of a set $X \subseteq D$ by $\bigsqcup X$ and $\bigsqcap X$, respectively, provided that they exist. Given elements $a, b \in D$, we say b is a *successor* (resp. *predecessor*) of a if $a \sqsubseteq b$ but $a \neq b$, and *immediate successor* if in addition there is no $c \in D \setminus \{a, b\}$ with $a \sqsubseteq c \sqsubseteq b$ (resp. *immediate predecessor*). Elements $a, b \in D$ with $a \not\sqsubseteq b$ and $b \not\sqsubseteq a$ are *incomparable*. An element $a \in X \subseteq D$ is a *maximal element* (resp., *minimal element*) of X if $a \sqsubseteq b$ (resp., $b \sqsubseteq a$) and $b \in X$ imply $a = b$.

A *lattice* $L = \langle D, \sqsubseteq \rangle$ is a *poset* $\langle D, \sqsubseteq \rangle$ such that $a \sqcup b = \bigsqcup \{a, b\}$ and $a \sqcap b = \bigsqcap \{a, b\}$ exist for all $a, b \in D$. L is a *complete lattice* if all non-empty subsets $X \subseteq D$ have a least upper bound and greatest lower bound. A complete lattice is bounded by definition. A non-empty subset $M \subseteq D$ forms a *sub-lattice* if $a \sqcup b \in M$ and $a \sqcap b \in M$ for all $a, b \in M$. A sub-lattice $M \subseteq D$ is *convex* if $a \sqsubseteq c \sqsubseteq b$ and $a, b \in M$ imply $c \in M$. A lattice is *distributive* if for all $a, b, c \in D$, $a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$. A *completely distributive lattice* is a complete lattice in which arbitrary joins ($\sqcup$) distribute over arbitrary meets ($\sqcap$). A function $f : D_1 \to D_2$, where $\langle D_1, \sqsubseteq_1 \rangle$ and $\langle D_2, \sqsubseteq_2 \rangle$ are posets, is *monotonic* if $x \sqsubseteq_1 y$ implies $f(x) \sqsubseteq_2 f(y)$ (resp., *anti-monotonic*) .

Stateless Logic. Some of the results presented in this paper require an additional assumption about a logic:

Definition 11. *A logic is called* stateless *if conjunctions $A[\bar{s}] \wedge B[\bar{t}]$ of satisfiable formulae $A[\bar{s}]$, $B[\bar{t}]$ over disjoint lists $\bar{s}, \bar{t}$ of non-logical symbols are satisfiable.*

Intuitively, formulae in a stateless logic interact only through non-logical symbols, not via any notion of global state, structure, etc. Many logics that are relevant in the context of verification are stateless, in particular quantifier-free first order logic, Pressburger arithmetic, logics based on the theory of arrays, etc. An example of a stateful logic is full FOL with equality. For instance, consider the conjunction $(\forall x, y.\ x \doteq y) \wedge (\exists x, y.\ x \not\doteq y)$ in full FOL. Although the individual con-

juncts $\forall x, y.\ x \doteq y$ and $\exists x, y.\ x \neq y$ are satisfiable, their conjunction is not: the first conjunct enforces a universe with only one element, whereas the second conjunct requires at least two elements.

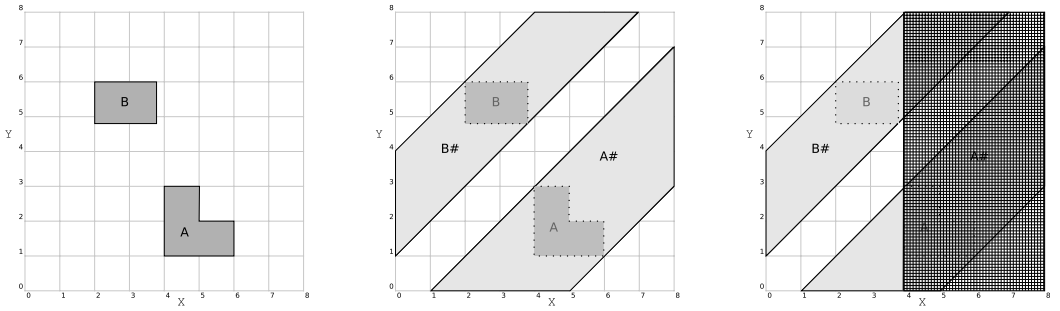
Other stateful logics are modal logics or separation logic; often, such logics can naturally be made stateless by enriching its vocabulary. Statelessness is important in this paper, since we use the concept of *renaming* of symbols to ensure independence of formulae.

Interpolation Abstractions. This section defines the concept of interpolation abstractions, and derives basic properties. Interpolation abstractions are represented by transformations of the formulae to be interpolated; in the most general formulation, this is represented via a pair of extensive functions on formulae:

Definition 12 (Interpolation abstraction)**.** *Suppose $\bar{s}$ is a list of non-logical symbols, for some arbitrary but fixed logic. An* interpolation abstraction *is a pair (T_A, T_B) of functions mapping formulae to formulae, with the following properties: We call $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ a* concrete interpolation problem*, and $T_A(A[\bar{s_A}, \bar{s}]) \wedge T_B(B[\bar{s}, \bar{s}_B])$ the corresponding* abstract interpolation problem *for the interpolation abstraction (T_A, T_B).*

In other words, interpolation abstractions define over-approximations of the conjuncts to be interpolated. Assuming that the concrete interpolation problem is solvable, we call the interpolation abstraction *feasible* if also the abstract interpolation problem is solvable, and *infeasible* otherwise. A simple illustration of the approach is depicted in Figure 5.2. In Figure 5.2a an arbitrary interpolation problem $A \wedge B$ is presented on its x and y dimensions. In Figure 5.2b an abstract interpolation problem $A\sharp \wedge B\sharp$ is shown when the template $x - y$ is applied. In Figure 5.2c the effect of the abstraction is shown by demonstrating that an interpolant valid for $A \wedge B$ is no longer valid.

Example 21. *An illustration is given in Figure 5.3. The concrete interpolation problem is solvable since the solution sets $A[\bar{s}]$ and $B[\bar{s}]$ are disjoint, i.e., $A[\bar{s}] \wedge B[\bar{s}]$ is unsatisfiable. An interpolant is a formula $I[\bar{s}]$ that represents a superset of $A[\bar{s}]$,*



(a) *Interpolation Problem $A \wedge B$ on Dimensions x and y*

(b) *Abstract Interpolation Problem $A\sharp \wedge B\sharp$ with Template $x - y$*

(c) *Abstract Interpolation Problem blocking of Interpolant $x \geq 4$*

Figure 5.2: *Applying Template $x - y$ to Interpolaiton Problem $A \wedge B$*

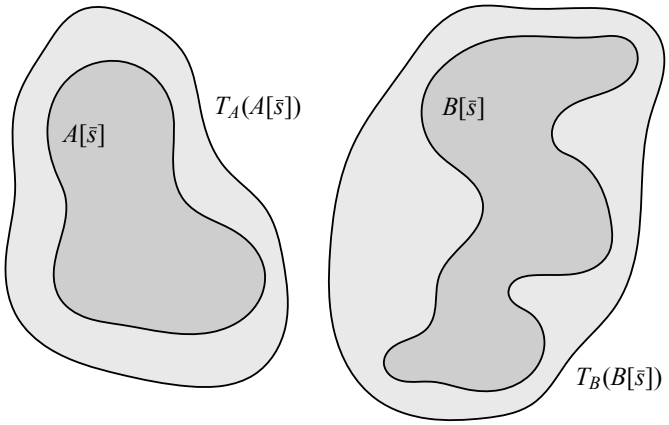


Figure 5.3: *Illustration of interpolation abstraction, assuming that only common non-logical symbols exist. Both concrete and abstract problem are solvable.*

but that is disjoint with $B[\bar{s}]$. By definition, the formula $T_A(A[\bar{s}])$ represents an over-approximation *of $A[\bar{s}]$; similarly for $T_B(B[\bar{s}])$. This ensures the soundness of computed abstract interpolants (see Lemma 8 below). In Figure 5.3, despite over-approximation, the abstract interpolation problem is solvable, which means that the interpolation abstraction is feasible.* □

While there are many ways to construct interpolation abstractions, in the scope of this paper we mainly concentrate on interpolation abstractions defined by means of *relations:*

Definition 13 (Relation abstraction)**.** *Suppose $\bar{s}$ is a list of non-logical symbols, and $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$. An* relation abstraction *is a pair $(R_A[\bar{s}', \bar{s}], R_B[\bar{s}, \bar{s}''])$ of formulae with the property that $R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid (i.e., $Id[\bar{s}', \bar{s}] \Rightarrow$*

$R_A[\bar{s}', \bar{s}]$ and $Id[\bar{s}, \bar{s}''] \Rightarrow R_B[\bar{s}, \bar{s}'']$). A relation abstraction defines an interpolation abstraction (T_A, T_B) by:

$$T_A(A[\bar{s}_A, \bar{s}]) \;=\; A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}], \qquad T_B(B[\bar{s}, \bar{s}_B]) \;=\; R_B[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\ .$$

Thus, the relation abstraction of a concrete interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is

$$\left(A[\bar{s}_A, \bar{s}'] \wedge R_A[\bar{s}', \bar{s}]\right) \;\wedge\; \left(R_B[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right).$$

Note that properties (i) and (ii) in Def. 12 are ensured by requiring that the relations $R_A[\bar{s}', \bar{s}]$ and $R_B[\bar{s}, \bar{s}'']$ subsume the identity relation ($R_A[\bar{s}, \bar{s}]$ and $R_B[\bar{s}, \bar{s}]$ are valid).

Example 22. *The interpolation abstraction applied in Example 20 is a relation abstraction. The common symbols of the interpolation problem are $\bar{s} = \langle x_1, i_1, j\rangle$, and the relation abstraction is defined by $R_A = (x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j)$ and $R_B = (x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'')$.* □

Finally, we can state a (straightforward) result about the correctness of interpolants computed using interpolation abstractions:

Lemma 8 (Soundness)**.** *Every interpolant of the abstract interpolation problem is also an interpolant of the concrete interpolation problem (but in general not vice versa).*

Proof. Suppose $A'[\bar{s}_{A'}, \bar{s}] = T_A(A[\bar{s}_A, \bar{s}])$ and $B'[\bar{s}, \bar{s}_{B'}] = T_B(B[\bar{s}, \bar{s}_B])$. An abstract interpolant only contains symbols from $\bar{s}$ (due to property (iii) of Def. 12), i.e., is of the form $I[\bar{s}]$. It also satisfies $A'[\bar{s}_{A'}, \bar{s}] \Rightarrow I[\bar{s}]$ and $B'[\bar{s}, \bar{s}_{B'}] \Rightarrow \neg I[\bar{s}]$, and thus $\exists \bar{s}_{A'}.\, A'[\bar{s}_{A'}, \bar{s}] \Rightarrow I[\bar{s}]$ and $\exists \bar{s}_{B'}.\, B'[\bar{s}, \bar{s}_{B'}] \Rightarrow \neg I[\bar{s}]$. Thanks to properties (i) and (ii) in Def. 12, this yields the implications $A[\bar{s}_A, \bar{s}] \Rightarrow I[\bar{s}]$ and $B[\bar{s}, \bar{s}_B] \Rightarrow \neg I[\bar{s}]$. □

5.2.2 Interpolant Lattices

Interpolation abstractions can be used to guide interpolation engines, by restricting the space $\mathit{Inter}(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])$ of interpolants satisfying an interpolation problem.

Recall that the set $\mathit{Inter}(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\equiv$ of interpolant classes (modulo logical equivalence) is closed under conjunctions (meet) and disjunctions (join), so that $(\mathit{Inter}(A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B])/\equiv, \Rightarrow)$ is a lattice. Figure 5.4 shows the interpolant lattice for Example 20; this lattice has a strongest concrete interpolant $I_\perp$ and a weakest concrete interpolant $I_\top$.[1]

For a feasible abstraction, the lattice $(\mathit{Inter}(T_A(A[\bar{s}_A, \bar{s}']),\ T_B(B[\bar{s}'', \bar{s}_B]))/\equiv, \Rightarrow)$ of abstract interpolants is a sub-lattice of the concrete interpolant lattice. The sub-lattice is *convex,* because if I_1 and I_3 are abstract interpolants and I_2 is a concrete interpolant with $I_1 \Rightarrow I_2 \Rightarrow I_3$, then also I_2 is an abstract interpolant. The choice of the function T_A in an interpolation abstraction constrains the lattice of abstract interpolants from below, the function T_B from above.

We illustrate two disjoint sub-lattices in Figure 5.4: the left box is the sub-lattice for the abstraction $(i_1' \doteq i_1, i_1 \doteq i_1'')$, while the right box represents the interpolation abstraction

$$(x_1' - i_1' \doteq x_1 - i_1 \wedge j' \doteq j,\ x_1 - i_1 \doteq x_1'' - i_1'' \wedge j \doteq j'')$$

used in Example 20 to derive interpolant I_2.

As the following lemma shows, there are no principal restrictions how fine-grained the guidance enforced by an interpolation abstraction can be; however, since abstraction is a semantic notion, we can only impose constraints *up to equivalence of interpolants:*

Lemma 9 (Completeness)**.** *Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem with interpolant $I[\bar{s}]$ in a stateless logic, such that both $A[\bar{s}_A, \bar{s}]$ and $B[\bar{s}, \bar{s}_B]$ are satisfiable (the problem is not degenerate). Then there is a feasible interpolation abstraction, definable as a relation abstraction in the same logic, such that every abstract interpolant is logically equivalent to $I[\bar{s}]$.*

Proof. Choose the relation abstraction $(I[\bar{s}'] \to I[\bar{s}],\ I[\bar{s}] \to I[\bar{s}''])$. Since $I[\bar{s}]$ is an interpolant of the abstract interpolation problem, the abstract problem is solvable.

[1] In general, the interpolant lattice might be incomplete and not contain such elements.

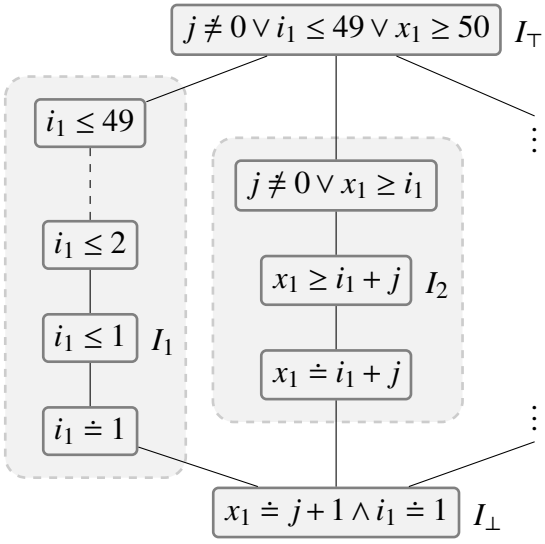


Figure 5.4: *Parts of the interpolant lattice for the Example 20 (up to equivalence). The dashed boxes represent the sub-lattices for the abstraction induced by the template terms $\{i_1\}$ (left) and $\{x_1 - i_1, j\}$ (right).*

Further, assume that $I'[\bar{s}]$ is an arbitrary abstract interpolant, i.e.,

$$A[\bar{s}_A, \bar{s}'] \wedge (I[\bar{s}'] \rightarrow I[\bar{s}]) \;\Rightarrow\; I'[\bar{s}] \quad \text{and} \quad (I[\bar{s}] \rightarrow I[\bar{s}'']) \wedge B[\bar{s}'', \bar{s}_B] \;\Rightarrow\; \neg I'[\bar{s}]\,.$$

By rewriting the left-hand sides of the entailments, we can conclude $I[\bar{s}] \Leftrightarrow I'[\bar{s}]$. We only show one of the directions:

$$\begin{aligned} A[\bar{s}_A, \bar{s}'] \wedge (I[\bar{s}'] \rightarrow I[\bar{s}]) \;&\Leftrightarrow\; (A[\bar{s}_A, \bar{s}'] \wedge \neg I[\bar{s}']) \vee (A[\bar{s}_A, \bar{s}'] \wedge I[\bar{s}]) \\ &\Leftrightarrow\; A[\bar{s}_A, \bar{s}'] \wedge I[\bar{s}] \end{aligned}$$

From $(A[\bar{s}_A, \bar{s}'] \wedge I[\bar{s}]) \Rightarrow I'[\bar{s}]$, it follows that $I[\bar{s}] \Rightarrow I'[\bar{s}]$, since $A[\bar{s}_A, \bar{s}']$ is satisfiable and does not contain any symbols from $\bar{s}$, and the considered logic is stateless. □

5.3 A Catalogue of Interpolation Abstractions

This Section introduces a range of practically relevant relation abstractions, mainly defined in terms of *templates* as illustrated in Example 20. For any interpolation abstraction, it is interesting to consider the following questions: (1) provided the

concrete interpolation problem is solvable, characterise the cases in which also the abstract problem can be solved (how *coarse* the abstraction is); (2) provided the abstract interpolation problem is solvable, characterise the space of abstract interpolants. The first point touches the question to which degree an interpolation abstraction limits the set of proofs that a theorem prover can find. We hypothesise (and explain in Example 20) that it is less important to generate interpolants with a specific syntactic shape, than to force a theorem prover to use the *right argument* for showing that a path in a program is safe.

We remark that interpolation abstractions can also be combined, for instance to create abstractions that include both template terms and template predicates. In general, the component-wise conjunction of two interpolation abstractions is again a well-formed abstraction, as is the disjunction.

5.3.1 Finite Term Interpolation Abstractions

The first family of interpolation abstractions is defined with the help of finite sets T of *template terms,* and formalises the abstraction used in Example 20. Intuitively, abstract interpolants for a term abstraction induced by T are formulae that only use elements of T, in combination with logical symbols, as building blocks (a precise characterisation is given in Lemma 11 below). For the case of interpolation in EUF (quantifier-free FOL without uninterpreted predicates), this means that abstract interpolants are Boolean combinations of equations between T terms. In linear arithmetic, abstract interpolants may contain equations and inequalities over linear combinations of T terms.

The relations defining a term interpolation abstraction follow the example given in Example 20, and assert that primed and unprimed versions of T terms have the same value. As a consequence, nothing is known about the value of unprimed terms that are *not* mentioned in T.

Definition 14 (Term interpolation abstraction)**.** *Suppose $\bar{s}$ is a list of non-logical symbols, $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground*

terms. The relation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ *defined by*

$$R_A^T[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}], \qquad R_B^T[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} t_i[\bar{s}] \doteq t_i[\bar{s}'']$$

is called term interpolation abstraction *over* T.

Term abstractions are feasible if and only if a concrete interpolant exists that can be expressed purely using T terms:

Lemma 10 (Solvability)**.** *Suppose* $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ *is an interpolation problem, and* $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ *a finite set of ground terms. The abstract interpolation problem for the abstraction* $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ *is solvable if and only if there is a formula* $I[x_1, \ldots, x_n]$ *over n variables* $x_1, \ldots, x_n$ *(and no further non-logical symbols) such that* $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ *is an interpolant of* $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.

Proof. "⇐": $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is also an abstract interpolant, which implies that the abstract interpolation problem is solvable.

"⇒": suppose the abstract interpolation problem

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}]\right) \wedge \left(\bigwedge_{i=1}^{n} t_i[\bar{s}] \doteq t_i[\bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right) \tag{5.3}$$

is solvable, which means that (5.3) is an unsatisfiable formula. Then also the following formula is unsatisfiable (for fresh variables $x_1, \ldots, x_n$):

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq x_i\right) \wedge \left(\bigwedge_{i=1}^{n} x_i \doteq t_i[\bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right) \tag{5.4}$$

Namely, suppose (5.4) is satisfied by the model S. The model can be extended to a model S' of (5.3) by interpreting the symbols $\bar{s}$ with the same value as the symbols $\bar{s}'$.

Given that (5.4) is unsatisfiable, due to the interpolation property there is an interpolant $I[x_1, \ldots, x_n]$ for (5.4). By the substitution theorem, then also $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is an interpolant for (5.3). Finally, by Lemma 8, $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is also an interpolant of the original interpolant problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$. □

Example 23. *Consider the interpolation abstraction used in Example 20, which is created by the set $T = \{x_1 - i_1, j\}$ of terms. The abstract interpolation problem is solvable with interpolant $x_1 \geq i_1 + j$, which can be represented as $(x_1 - i_1) \geq (j)$ as a combination of the template terms in T.* □

It would be tempting to assume that *all* interpolants generated by term interpolation abstractions are as specified in Lemma 10, i.e., constructed only from T terms and logical symbols. In fact, since our framework restricts the space of interpolants in a semantic way, only weaker guarantees can be provided about the range of possible interpolants; this is related to the earlier observation that interpolation can only be restricted *up to logical equivalence:*

Lemma 11 (Interpolant space)**.** *Suppose the abstract interpolation problem for the relation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ is solvable, and the underlying logic is EUF or PA. Then there is a strongest abstract interpolant $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$, and a weakest abstract interpolant $I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$, each constructed only from T terms and logical symbols. A formula $J[\bar{s}]$ is an abstract interpolant iff the implications $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]] \Rightarrow J[\bar{s}] \Rightarrow I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ hold.*

Proof. Again consider the interpolation problem (5.4), and observe that there is a strongest interpolant $I_\bot[x_1, \ldots, x_n]$ and a weakest interpolant $I_\top[x_1, \ldots, x_n]$. (For EUF, this is because there are only finitely many interpolants up to equivalence; for PA, this holds due to the quantifier elimination property).

We show that $I_\bot[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is the conjectured strongest interpolant. (The proof for the weakest interpolant $I_\top[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is symmetric.) Suppose $J[\bar{s}]$ is any abstract interpolant, which means

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq t_i[\bar{s}] \right) \Rightarrow J[\bar{s}]$$

and therefore also

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \doteq x_i \right) \Rightarrow \left(\bigwedge_{i=1}^{n} x_i \doteq t_i[\bar{s}] \Rightarrow J[\bar{s}] \right)$$

Since left-hand and right-hand side only share the (uninterpreted) symbols $x_1, \ldots, x_n$, and $I_\perp[x_1, \ldots, x_n]$ is the strongest formula over those symbols implied by the left-hand side, this entails:

$$I_\perp[x_1, \ldots, x_n] \Rightarrow \left(\bigwedge_{i=1}^{n} x_i \doteq t_i[\bar{s}] \Rightarrow J[\bar{s}] \right)$$

and therefore $I_\perp[t_1[\bar{s}], \ldots, t_n[\bar{s}]] \Rightarrow J[\bar{s}]$. □

Example 24. *Again, consider Example 20, and the interpolant lattice as shown in Figure 5.4. The strongest abstract interpolant for the interpolation abstraction induced by $T = \{x_1 - i_1, j\}$ is $x_1 \doteq i_1 + j$, the weakest one $j \neq 0 \vee x_1 \geq i_1$.* □

5.3.2 Finite Inequality Interpolation Abstractions

In the case of a logic with arithmetic operators, for instance linear rational arithmetic or Presburger arithmetic, it is possible to define interpolation abstractions on the basis of inequalities instead of equations, to achieve more fine-grained control over interpolants. Inequality interpolation abstractions can specify that interpolants can only give upper bounds (or only lower bounds) on the value of some term t, i.e., t can only occur on the left- or right-hand side of inequalities $\leq$, and not as part of equations. This degree of control is highly useful for model checking applications, where it is well-known that the quality of interpolants can be improved by abstracting equations to inequalities.

Definition 15 (Inequality interpolation abstraction)**.** *Suppose $\bar{s}$ is a list of non-logical symbols, $\bar{s}'$ and $\bar{s}''$ fresh copies of $\bar{s}$, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The relation abstraction $(R_A^{\leq T}[\bar{s}', \bar{s}], R_B^{\leq T}[\bar{s}, \bar{s}''])$ defined by*

$$R_A^{\leq T}[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} t_i[\bar{s}'] \leq t_i[\bar{s}], \qquad R_B^{\leq T}[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} t_i[\bar{s}] \leq t_i[\bar{s}'']$$

is called inequality interpolation abstraction *over T.*

Intuitively, the terms T can only occur only the right side of inequalities $\leq$ in interpolants, i.e., in the form of lower bounds. To specify upper bounds, it is

possible to specify negative terms $-t \in T$; when including both t and $-t$ in T, arbitrary occurrences of t in an interpolant are possible (also within equations). This shows that inequality interpolation abstractions strictly subsume term interpolation abstractions in the presence of arithmetic.

To characterise solvability, assume that interpolants only contain inequalities $\leq$ (and no $\geq$ or equations $\doteq$), and that no inequalities occur underneath negation $\neg$. An occurrence of a term is then called *positive* if the term (or a positive multiple of the term) is on the right-hand side of $\leq$, and negative if it is on the left-hand side.

Lemma 12 (Solvability)**.** *Suppose $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ is an interpolation problem in PA, and $T = \{t_1[\bar{s}], \ldots, t_n[\bar{s}]\}$ a finite set of ground terms. The abstract interpolation problem for the abstraction $(R_A^{\leq T}[\bar{s}', \bar{s}], R_B^{\leq T}[\bar{s}, \bar{s}''])$ is solvable if and only if there is a formula $I[x_1, \ldots, x_n]$ over n variables $x_1, \ldots, x_n$, all occurring only positively in $I[x_1, \ldots, x_n]$, such that the formula $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is an interpolant of $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.*

Proof. "$\Leftarrow$": as for Lemma 10, it can be observed that $I[t_1[\bar{s}], \ldots, t_n[\bar{s}]]$ is also an abstract interpolant, which implies that the abstract interpolation problem is solvable.

"$\Rightarrow$": again, as for Lemma 10, we consider the modified interpolation problem

$$\left(A[\bar{s}_A, \bar{s}'] \wedge \bigwedge_{i=1}^{n} t_i[\bar{s}'] \leq x_i\right) \quad \wedge \quad \left(\bigwedge_{i=1}^{n} x_i \leq t_i[\bar{s}''] \wedge B[\bar{s}'', \bar{s}_B]\right)$$

As a constructive way of showing the existence of interpolants $I[x_1, \ldots, x_n]$ of the desired form, the interpolating PA sequent calculus from [190] can be used. To this end, first assume that the conjuncts $A[\bar{s}_A, \bar{s}], B[\bar{s}, \bar{s}_B]$ are normalised in such a way that no equations, no quantifiers (or divisibility constraints), and no negations occur. Then, construct an interpolating proof by strictly ordering the applied rules: (i) eliminate all Boolean operators, (ii) apply arithmetic rules to the literals obtained from $A[\bar{s}_A, \bar{s}']$, $B[\bar{s}'', \bar{s}_B]$, (iii) finally, resolve with the inequalities $t_i[\bar{s}'] \leq x_i$ and $x_i \leq t_i[\bar{s}'']$ to obtain conflicts and close proof goals.

By checking the individual proof rules in [190], we can observe that the resulting interpolant $I[x_1,\ldots,x_n]$ will only contain variables $x_1,\ldots,x_n$ in positive positions. □

5.3.3 Finite Predicate Interpolation Abstractions

In a similar way as sets of terms, also finite sets of *formulae* induce interpolation abstractions. Template formulae can be relevant to steer an interpolating theorem prover towards (possibly user-specified or quantified) interpolants that might be hard to find for the prover alone. The approach bears some similarities to the concept of predicate abstraction in model checking [191, 129], but still leaves the use of templates entirely to the theorem prover.

Definition 16 (Predicate interpolation abstraction)**.** *Suppose $A[\bar{s}_A,\bar{s}]\wedge B[\bar{s},\bar{s}_B]$ is an interpolation problem, and $Pred=\{\phi_1[\bar{s}],\ldots,\phi_n[\bar{s}]\}$ is a finite set of formulae. The relation abstraction $(R_A^{Pred}[\bar{s}',\bar{s}],R_B^{Pred}[\bar{s},\bar{s}'])$ defined by*

$$R_A^{Pred}[\bar{s}',\bar{s}] \;=\; \bigwedge_{i=1}^{n}(\phi_i[\bar{s}']\rightarrow\phi_i[\bar{s}]),\qquad R_B^{Pred}[\bar{s},\bar{s}''] \;=\; \bigwedge_{i=1}^{n}(\phi_i[\bar{s}]\rightarrow\phi_i[\bar{s}''])$$

is called predicate interpolation abstraction *over Pred.*

Intuitively, predicate interpolation abstractions restrict the solutions of an interpolation problem to those interpolants that can be represented as a positive Boolean combination of the predicates in *Pred* (i.e., by combining elements of *Pred* using $\wedge$ and $\vee$, without negations $\neg$). Note that it is possible to include the negation of a predicate $\phi[\bar{s}]$ in *Pred* if *negative* occurrences of $\phi[\bar{s}]$ are supposed to be allowed in an interpolant (or both $\phi[\bar{s}]$ and $\neg\phi[\bar{s}]$ for both positive and negative occurrences).

Lemma 13 (Solvability)**.** *Suppose $A[\bar{s}_A,\bar{s}]\wedge B[\bar{s},\bar{s}_B]$ is an interpolation problem, and Pred a finite set of predicates. If the underlying logic is stateless, then the abstract interpolation problem for $(R_A^{Pred}[\bar{s}',\bar{s}],R_B^{Pred}[\bar{s},\bar{s}''])$ is solvable if and only if $A[\bar{s}_A,\bar{s}]\wedge B[\bar{s},\bar{s}_B]$ has an interpolant $I[\bar{s}]$ that is a positive Boolean combination of predicates in Pred.*

Proof. "⇐": via Boolean reasoning, it can be shown that the interpolant $I[\bar{s}]$ also is a solution of the abstract problem $(A[\bar{s}_A, \bar{s}'] \wedge R_A^{Pred}[\bar{s}', \bar{s}]) \wedge (R_B^{Pred}[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])$.

"⇒": suppose $(A[\bar{s}_A, \bar{s}'] \wedge R_A^{Pred}[\bar{s}', \bar{s}]) \wedge (R_B^{Pred}[\bar{s}, \bar{s}''] \wedge B[\bar{s}'', \bar{s}_B])$ is unsatisfiable. As a constructive way to show the existence of an interpolant that is a positive Boolean combination of *Pred* predicates, we use the propositional interpolating calculus from [192, Fig. 1]. Thanks to proof-confluency, we can start by splitting all implications from $R_A^{Pred}[\bar{s}', \bar{s}]$ and $R_B^{Pred}[\bar{s}, \bar{s}'']$, using rules OR-LEFT, NOT-LEFT. After that, all sequents containing complementary formulae $\phi_i[\bar{s}]$ can be closed with the rule CLOSE-LR; this leads to positive occurrences of $\phi_i[\bar{s}]$ in the interpolant.

All other sequents have the form $\ldots, \lfloor A[\bar{s}_A, \bar{s}'] \rfloor_L, \lfloor B[\bar{s}'', \bar{s}_B] \rfloor_R, BP \vdash AP, \ldots$ where AP is a set of formulae of the form $\lfloor \phi_i[\bar{s}'] \rfloor_L$, and BP a set of formulae $\lfloor \phi_j[\bar{s}''] \rfloor_R$. Since the sequent is valid by assumption, and since the underlying logic is stateless, at least one of $A[\bar{s}_A, \bar{s}'] \wedge \neg AP$ and $B[\bar{s}'', \bar{s}_B] \wedge BP$ is unsatisfiable. In the first case, the sequent can be closed with interpolant *false*, in the latter case with interpolant *true*. □

We remark that the implication ⇐ holds in all cases, whereas ⇒ needs the assumption that the logic is stateless. As a counterexample for the stateful case, consider again the interpolation problem $(\forall x, y.\ x \doteq y) \wedge (\exists x, y.\ x \not\doteq y)$ in full FOL. The abstract interpolation problem is solvable even for $Pred = \emptyset$ (with interpolant $\forall x, y.\ x \doteq y$), but no positive Boolean combination of *Pred* formulae is an interpolant.

The interpolant space can be characterised as for term interpolation abstractions (Lemma 11):

Lemma 14 (Interpolant space)**.** *Suppose the abstract interpolation problem for the relation abstraction* $(R_A^{Pred}[\bar{s}', \bar{s}], R_B^{Pred}[\bar{s}, \bar{s}''])$ *is solvable, and the underlying logic is stateless. Then there is a strongest abstract interpolant* $I_\bot[\bar{s}]$*, and a weakest abstract interpolant* $I_\top[\bar{s}]$*, each being a positive Boolean combination of predicates in Pred. A formula* $J[\bar{s}]$ *is an abstract interpolant iff the implications* $I_\bot[\bar{s}] \Rightarrow J[\bar{s}] \Rightarrow I_\top[\bar{s}]$ *hold.*

Proof. As in the proof Lemma 11, but with Boolean variables instead of $x_1, \ldots, x_n$.

□

5.3.4 Quantified Interpolation Abstractions

The previous Sections showed how interpolation abstractions are generated by finite sets of templates. A similar construction can be performed for *infinite* sets of templates, expressed schematically with the help of variables; in the verification context, this is particularly relevant if arrays or heap are encoded with the help of uninterpreted functions.

Example 25. *Suppose that the binary function H represents heap contents, with heap accesses obj.field translated to $H(obj,field)$, and is used to state an interpolation problem:*

$$(H(a,f) \doteq c \wedge H(b,g) \not\doteq null) \quad \wedge \quad (b \doteq c \wedge H(b,g) \doteq null \wedge H(H(a,f),g) \doteq null)$$

An obvious interpolant is the formula $I_1 = (H(b,g) \not\doteq null)$. Based on domain-specific knowledge, we might want to avoid interpolants with direct heap accesses $H(\cdot,g)$, and instead prefer the pattern $H(H(\cdot,f),g)$. To find alternative interpolants, we can use the templates $\{H(H(x,f),g), a, b, c\}$, the first of which contains a schematic variable x. The resulting abstraction excludes I_1, but yields the interpolant $I_2 = (b \doteq c) \rightarrow (H(H(a,f),g) \not\doteq null)$. □

Definition 17 (Schematic term abstraction)**.** *Suppose an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, and a finite set $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_n]\}$ of terms with free variables $\bar{x}_1, \ldots, \bar{x}_n$. The relation abstraction $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ defined by*

$$R_A^T[\bar{s}', \bar{s}] = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}', \bar{x}_i] \doteq t_i[\bar{s}, \bar{x}_i], \qquad R_B^T[\bar{s}, \bar{s}''] = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}, \bar{x}_i] \doteq t_i[\bar{s}'', \bar{x}_i]$$

is called schematic term interpolation abstraction *over T.*

Note that schematic term interpolation abstractions reduce to ordinary term interpolation abstractions (as in Def. 14) if none of the template terms contains free variables.

Quantified abstractions are clearly less interesting for logics that admit quantifier elimination, such as PA, but they are relevant whenever uninterpreted functions (EUF) are involved.

Lemma 15 (Solvability in EUF)**.** *Suppose* $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$ *is an interpolation problem in EUF,* $T = \{t_1[\bar{s}, \bar{x}_1], \ldots, t_n[\bar{s}, \bar{x}_n]\}$ *a finite set of schematic terms, and* $f = \langle f_1, \ldots, f_n \rangle$ *a vector of fresh functions with arities* $|\bar{x}_1|, \ldots, |\bar{x}_n|$*, respectively. The abstract interpolation problem for* $(R_A^T[\bar{s}', \bar{s}], R_B^T[\bar{s}, \bar{s}''])$ *is solvable if and only if there is a formula* $I[f_1, \ldots, f_n]$ *(without non-logical symbols other than* $\bar{f}$*) such that* $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ *is an interpolant of* $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$.

The expression $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ denotes the formula obtained by replacing each occurrence of a function f_i in $I[f_1, \ldots, f_n]$ with the template $t_i[\bar{s}, \bar{x}_i]$, substituting the arguments of f_i for the schematic variables $\bar{x}_i$.

Proof. "⇐": $I[t_1[\bar{s}, \cdot], \ldots, t_n[\bar{s}, \cdot]]$ is also an abstract interpolant, which implies that the abstract interpolation problem is solvable.

"⇒": observe that if the abstract interpolation problem is solvable, conjunction (5.5) is unsatisfiable:

$$\Big(A[\bar{s}_A, \bar{s}'] \wedge \psi_A\Big) \wedge \Big(\psi_B \wedge B[\bar{s}'', \bar{s}_B]\Big) \tag{5.5}$$

where

$$\psi_A = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ t_i[\bar{s}', \bar{x}_i] \doteq f_i(\bar{x}_i) \qquad \psi_B = \bigwedge_{i=1}^{n} \forall \bar{x}_i.\ f_i(\bar{x}_i) \doteq t_i[\bar{s}'', \bar{x}_i]$$

An interpolant $I[f_1, \ldots, f_n]$ can be computed from (5.5) using FOL interpolation techniques. □

5.4 The Algebra of Interpolation Abstractions

It is frequently useful to construct new interpolation abstractions from existing ones, for instance to combine term, inequality, and predicate interpolation abstractions. Combination is possible through several algebraic operations. For in-

stance, given two interpolation abstractions (T_A, T_B) and (T'_A, T'_B), the composition $(T_A, T_B) \circ (T'_A, T'_B)$ constructs an abstract interpolation that composes the functions in the abstractions component-wise. New interpolation abstractions can be constructed similarly from conjunction, disjunction, and complementation.

In the whole Section we fix some logic, as well as a list $\bar{s}$ of common symbols. In order to define algebraic operations on interpolation abstractions, it is first necessary to introduce a notion of equivalence:

Definition 18 (Equivalent Interpolation Abstractions)**.** *Let (T_A, T_B) and (T'_A, T'_B) be two interpolation abstractions. We say that they are* equivalent*, written $(T_A, T_B) \equiv (T'_A, T'_B)$, if for any two equivalent formulae $A \equiv A'$ it is the case that $T_A(A) \equiv T'_A(A')$, and similarly for equivalent formulae $B \equiv B'$ it holds that $T_B(B) \equiv T'_B(B')$.*

Note that $\equiv$ is *not* immediately an equivalence relation on interpolation abstractions, since an interpolation abstraction is not necessarily equivalent to itself ($\equiv$ is not reflexive): an abstraction might map equivalent, but syntactically distinct formulae to non-equivalent formulae. We therefore focus on the set L of all *self-equivalent* (or *extensive*) interpolation abstractions, for the fixed logic and symbols $\bar{s}$. In particular, relation abstractions (Def. 13) are all self-equivalent. Since $\equiv$ is an equivalence relation on L, we can in the next paragraphs consider the set $L/\equiv$ of equivalence classes.

We can observe that the set L is closed under the operations conjunction $\wedge_L$, disjunction $\vee_L$, complementation $\neg_L$, identity I_L, top $\top_L$ and composition $\circ_L$, defined as:

$$
\begin{aligned}
I_L &:= (\lambda A.A,\ \lambda B.B) \\
\top_L &:= (\lambda A.\mathit{true},\ \lambda B.\mathit{true}) \\
(T_A, T_B) \circ_L (T'_A, T'_B) &:= (\lambda A.T_A(T'_A(A)),\ \lambda B.T_B(T'_B(B))) \\
(T_A, T_B) \wedge_L (T'_A, T'_B) &:= (\lambda A.T_A(A) \wedge T'_A(A),\ \lambda B.T_B(B) \wedge T'_B(B)) \\
(T_A, T_B) \vee_L (T'_A, T'_B) &:= (\lambda A.T_A(A) \vee T'_A(A),\ \lambda B.T_B(B) \vee T'_B(B)) \\
\neg_L (T_A, T_B) &:= (\lambda A.\neg T_A(A) \vee I_L,\ \lambda B.\neg T_B(B) \vee I_L)
\end{aligned}
$$

All operations can be extended to the equivalence classes in $L/\equiv$, since $\equiv$ is a congruence relation. The resulting algebra $\mathcal{L} = \langle L/\equiv, \wedge_L, \vee_L, \neg_L, I_L, \top_L, \circ_L \rangle$ forms a *bounded distributive lattice* where I_L is the bottom element, $\top_L$ is the top element and all elements are ordered by implication: $(T_A, T_B) \Rightarrow (T'_A, T'_B)$ if $T_A(A) \Rightarrow T'_A(A)$ and $T_B(B) \Rightarrow T'_B(B)$ for all formulae A, B. Since the lattice is also complemented, it forms a Boolean algebra. $\mathcal{L}$ further has the structure of a monoid:

Lemma 16 (Monoid Algebra)**.** *$\mathcal{L}$ is a monoid under $\circ_L$ and I_L.*

Proof. Since $\circ_L$ is defined as component-wise composition of two functions, and general function composition is associative, $\circ_L$ is associative. By definition I_L a tuple of identity functions and hence is an identity element for all elements in L. □

5.5 Exploration of Interpolants

In a typical application scenario of our interpolation abstraction framework (e.g., in a model checker), we will not consider just a single fixed interpolation abstraction, but rather a whole *family* of such abstractions. Working with multiple interpolation abstractions turns out to be meaningful for several reasons: (i) for each interpolation problem we might want to compute multiple different interpolants, which can be achieved by successively applying several interpolation abstractions; (ii) by ranking interpolation abstractions, the quality of resulting interpolants can be controlled. For instance, in the Example 20, we consider interpolant I_2 constructed using templates $\{x_1 - i_1, j\}$ as "better" than interpolant I_1 for the template i_1; (iii) every individual interpolation abstraction is feasible for some interpolation problems, and infeasible for others. This necessitates the definition of a whole family of abstractions, so that some feasible abstractions can be picked for every interpolation problem.

To formalise this concept of *interpolant exploration,* we arrange families of interpolation abstractions as *abstraction lattices,* and present search algorithms on such lattices. As described, interpolation abstractions have algebraic properties that can be used when defining such abstraction lattices. Abstraction lattices are equipped with a monotonic mapping μ to interpolation abstractions (T_A, T_B), or-

dered by component-wise implication. The following paragraphs focus on the case of *finite* abstraction lattices; the handling of infinite (parametric) abstraction lattices is planned as future work.

Definition 19 (Abstraction lattice)**.** *Suppose $\bar{s}$ is a list of non-logical symbols, for some arbitrary but fixed logic. An* abstraction lattice *is a pair $(\langle L, \sqsubseteq_L \rangle, \mu)$ consisting of a complete lattice $\langle L, \sqsubseteq_L \rangle$ and a monotonic mapping μ from elements of $\langle L, \sqsubseteq_L \rangle$ to interpolation abstractions (T_A, T_B) over $\bar{s}$, with the property that $\mu(\bot) = (Id_A, Id_B)$ is the identity abstraction (i.e., $Id_A(A) = A$ and $Id_B(B) = B$ for all formulae A, B).*

Given an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, the elements of an abstraction lattice that map to *feasible* interpolation abstractions form a downward closed set; an illustration is given in Figure 5.5, where feasible elements are shaded in gray. Provided that the concrete interpolation problem is solvable, the set of feasible elements in the lattice is non-empty, due to the requirement that $\mu(\bot) = (Id_A, Id_B)$.

Particularly interesting are *maximal feasible* interpolation abstractions, i.e., the maximal elements within the set of feasible interpolation abstractions. Maximal feasible abstractions restrict interpolants in the strongest possible way, and are therefore most suitable for exploring interpolants; we refer to the set of maximal feasible elements within an abstraction lattice as *abstraction frontier*.

5.5.1 Construction of Abstraction Lattices

When working with interpolation abstractions generated by templates, abstraction lattices can naturally by constructed as the *powerset lattice* of some template base set (ordered by the superset relation); this construction applies to term, inequality, and predicate templates. Further, the operations introduced in Section 5.4 can be used to combine simple lattices into more sophisticated ones; for instance, a useful construction is to form the *product* of two lattices, defining the mapping μ as the pairwise conjunction, disjunction, or composition of the individual mappings μ_1, μ_2.

Example 26. *An abstraction lattice for the Example 20 is $(\langle \wp(T), \supseteq \rangle, \mu)$, with base templates $T = \{x_1 - i_1, i_1, j\}$ and μ mapping each element to the abstraction in*

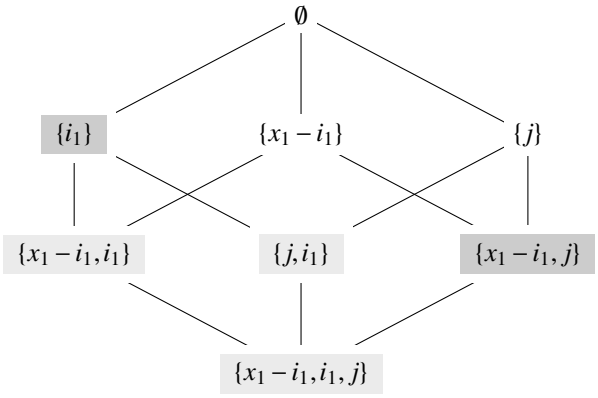


Figure 5.5: *The abstraction lattice for the running example. The light gray shaded elements are feasible, the dark gray ones maximal feasible.*

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$
Result: Set of maximal feasible interpolation abstractions
1 *Frontier* $\leftarrow \emptyset$;
2 **while** $\exists$ *feasible abs* $\in L$, *incomparable with all* $x \in$ *Frontier* **do**
3 | *Frontier* $\leftarrow$ *Frontier* $\cup \{maximise(P, abs)\}$;
4 **end**
5 **return** *Frontier*;

Algorithm 7: Exploration algorithm

Def. 14. Note that the bottom element of the lattice represents the full set T of templates (the weakest abstraction), and the top element the empty set $\emptyset$ (the strongest abstraction). Also, note that $\mu(T)$ is the identity abstraction (Id_A, Id_B), since T is a basis of the vector space of linear functions in x_1, i_1, j.

The lattice is presented in Figure 5.5, with feasible elements in light gray. The maximal feasible elements $\{i_1\}$ and $\{x_1 - i_1, j\}$ map to interpolation abstractions with the abstract interpolants I_1 and I_2, respectively, as illustrated in Figure 5.4. Smaller feasible elements (closer to $\bot$) correspond to larger sub-lattices of abstract interpolants, and therefore provide weaker guidance for a theorem prover; for instance, element $\{j, i_1\}$ can produce all abstract interpolants that $\{i_1\}$ generates, but can in addition lead to interpolants like $I_3 = (j \neq 0 \vee i_1 \leq 49)$. □

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, feasible abstraction $abs \in L$
Result: Maximal feasible abstraction
1 **while** $\exists$ *feasible immediate successor fs of abs* **do**
2 pick element *middle* such that $fs \sqsubseteq_L middle \sqsubseteq_L \top$;
3 **if** *middle is feasible* **then**
4 $abs \leftarrow middle$;
5 **else**
6 $abs \leftarrow fs$;
7 **end**
8 **end**
9 **return** *abs*;

Algorithm 8: Maximisation algorithm $maximise(P, abs)$

5.5.2 Computation of Abstraction Frontiers

In the case of abstraction lattices that are Boolean lattices, like the one in Figure 5.5, the computation of abstraction frontiers can be carried out using algorithms for the well-known problem of computing *minimal unsatisfiable subsets* (e.g., [193]). Such algorithms do not immediately carry over, however, to non-Boolean lattices, which can also be relevant abstraction lattices. We therefore present a binary search-based algorithm to compute abstraction frontiers of arbitrary finite abstraction lattices. In later Sections, this algorithm will be extended to also take *costs* into account, as a means to rank interpolation abstractions.

The search is described in Algorithms 7 and 8. Algorithm 7 describes the top-level procedure for finding maximal elements in an abstraction lattice. The algorithm repeatedly checks whether feasible abstractions $abs \in L$ exist that are incomparable with the maximum feasible abstractions found so far, i.e., such that no $x \in Frontier$ with $abs \sqsubseteq_L x$ or $x \sqsubseteq_L abs$ exists (line 2). Suitable methods for computing such incomparable elements can be defined based on the shape of the chosen abstraction lattice; for instance, if the abstraction lattice is a Boolean lattice, finding incomparable abstractions amounts to solving the well-known problem of finding minimal *hitting sets* for the *Frontier* [194] (a hitting set is a set that has elements in common with every set in the *Frontier*). As long as incomparable elements can be found, they are maximised by calling the *maximise* function (described in Algorithm 8), and added to the frontier.

In Algorithm 8 we describe the procedure for finding a maximal feasible abstraction *mfa* with the property that $abs \sqsubseteq_L mfa$. In each iteration of the maximisation loop, it is checked whether *abs* has any feasible parents (line 1); if this is not the case, *abs* has to be maximal feasible and is returned. Otherwise, in the loop body the algorithm executes a binary search on the set of elements in between *abs* and $\top$. The algorithm depends on the ability to efficiently compute (random) middle elements between two elements $a \sqsubset b$ of the lattice (line 2); again, this functionality can best be implemented specifically for an individual lattice, and is not shown here.

It should be noted that checking the feasibility of an interpolation abstraction (T_A, T_B), for an interpolation problem $A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, can be done by a simple check whether the conjunction $T_A(A[\bar{s}_A, \bar{s}]) \wedge T_B(B[\bar{s}, \bar{s}_B])$ is unsatisfiable (assuming a logic with the interpolation property). Repeating this check for a large number of abstractions can be optimised with the help of *incremental SMT:* typically, only a small part of the formula $T_A(A[\bar{s}_A, \bar{s}]) \wedge T_B(B[\bar{s}, \bar{s}_B])$ will actually depend on the abstraction (T_A, T_B), in particular for relation abstractions. Common conjuncts can therefore be factored out and handed over to an SMT solver upfront.

Lemma 17 (Correctness of exploration algorithm)**.** *When applied to a finite abstraction lattice, Algorithm 7 terminates and returns the set of maximal feasible elements.*

Proof. To see that the returned *Frontier* only contains maximal feasible abstractions, note that algorithm $maximise(P, abs)$ only returns abstractions that are feasible, and only abstractions without feasible successors (i.e., maximal feasible ones). The returned *Frontier* contains *all* maximal feasible abstractions, since any missing maximal feasible abstractions $mfa \notin Frontier$ would have to be incomparable with the elements in *Frontier* (due to maximality), and thus the loop condition in Algorithm 7, line 2 holds.

Algorithm 7 terminates, since the considered abstraction lattice is finite, and the set *Frontier* grows by one element in every iteration of the while loop. Namely, assume that in some iteration an abstraction $maximise(P, abs)$ is produced that is already an element of *Frontier*; in this case, $abs \sqsubseteq_L maximise(P, abs)$ cannot have

been incomparable with *Frontier*. Algorithm 8 terminates since finite lattices have finite height, and *abs* grows strictly in every iteration of the while loop. □

A useful refinement of the exploration algorithm is to *canonise* lattice elements during search. Elements $a, b \in L$ are considered equivalent if they are mapped to (logically) equivalent abstraction relations by μ. Canonisation can select a representative for every equivalence class of lattice elements, and search be carried out only on such canonical elements.

5.5.3 Guiding Interpolant Exploration with Costs

Given an abstraction frontier, it is possible to compute a range of interpolants solving the original interpolation problem. However, for large abstraction frontiers this may be neither feasible nor necessary. It is more useful to define a measure for the quality of interpolation abstractions, again exploiting domain-specific knowledge, and only use the best abstractions for interpolation.

To select good maximal feasible interpolation abstractions, we define an *anti-monotonic* cost function $cost : L \to \mathbb{N}$ that maps elements of an abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$ to a natural number, with lower values indicating that an interpolation abstraction is considered better. The anti-monotonicity property ($\forall a, b \in L.\ a \sqsubseteq_L b \Rightarrow cost(a) \geq cost(b)$) encompasses that coarser abstractions (higher up in the lattice) have lower cost. In the case of abstractions constructed using a powerset lattice over templates ($L = \wp(T)$), it is natural to assign a cost to every element in T ($cost : T \to \mathbb{N}$), and to define the cost of a lattice element $A \in L$ as $cost(A) = \sum_{t \in A} cost(t)$. Similarly, for product lattices the cost function can be computed as the sum of the costs of the components.

Our abstraction lattice in Figure 5.5 has two maximal feasible abstractions, $\{i_1\}$ and $\{x_1 - i_1, j\}$, which result in computing the interpolants I_1 and I_2, respectively. We can define a cost function that assigns a high cost to $\{i_1\}$ and a low cost to $\{x_1 - i_1, j\}$, expressing the fact that we prefer to not talk about the loop counter i_1 in absolute terms. More generally, assigning a high cost to variables representing loop counters is a reasonable strategy for obtaining general interpolants (a similar observation is made in [131], and implemented with the help of "term abstraction").

Once a cost function has been defined, the goal is to compute those abstractions from the *Frontier* set that have minimal cost. Naively, this can be done by first computing the whole *Frontier* set, using Algorithms 7 and 8, and then removing those elements that are too costly; however, for realistic abstraction lattices this procedure tends to be slow. Instead, it is possible to exploit costs already during search, eagerly pruning away those parts of the search space that cannot contain abstractions with low cost. We describe an optimisation to the exploration algorithms that uses costs to this effect in Algorithms 9 and 10.

Besides *Frontier*, in Algorithm 9 an additional set of *costly* abstractions (*CostlyAbs*) is maintained. A costly abstraction c is one whose cost $cost(c)$ has been identified as being greater than the minimal cost of feasible abstractions, and that has the property that none of its successors is feasible; as a consequence, the part of the abstraction lattice above c cannot contain low cost frontier elements.

The generalised maximisation function (*boundedMaximise*, Algorithm 10) returns either a maximal feasible abstraction m of minimal cost, or it returns a costly abstraction c (which may or may not be feasible). Feasible abstractions of minimal cost are added to the *Frontier*, while costly abstractions are added to the *CostlyAbs* set. If a returned maximal feasible abstraction improves upon the current cost bound (defined by the *minCost* variable), then the *minCost* variable is updated with the new minimal cost, and all previous frontier abstractions are moved to *CostlyAbs*.

Like Algorithm 8, Algorithm 10 proceeds by increasing the abstraction *abs* until an abstraction is reached whose successors are all infeasible. To this end, the `for` loop (line 4) iterates over the immediate successors of *abs*; if a feasible successor is found, the loop is left, while knowledge about infeasible successors is used to improve the *upperBound* variable.

The algorithm maintains the invariant that *abs*, and all of its feasible successors are below *upperBound*. If it is detected that $cost(upperBound) > minCost$, it follows (thanks to anti-monotonicity of *cost*) that no feasible abstractions with low cost can exist above *abs*, and the algorithm can return immediately. In this way, the search space can be pruned significantly.

In order to update the variable *upperBound* (line 8), the algorithm exploits the fact that a feasible abstraction *abs* with an infeasible successor s has been found. Given the pair abs, s, we call an element $b \in L$ a *feasibility bound* if the following properties are satisfied:

$$feasibilityBound(abs, s, b) \equiv \begin{cases} abs \text{ is feasible and } s \text{ is infeasible,} \\ abs = s \sqcap b, \text{ and} \\ \text{for every feasible abstraction } x \text{ with } abs \sqsubseteq x \\ \qquad \text{it holds that } x \sqsubseteq b\ . \end{cases}$$

In other words, given a feasible abstraction *abs* with infeasible successor s of *abs*, the predicate *feasibilityBound* provides an upper bound b for every feasible successor of *abs*. This implies that subsequent maximisation can ignore parts of the lattice that are not underneath b.

The existence of upper bounds b is determined by the considered lattice. In the special case that the abstraction lattice is a *distributive lattice* (e.g., a powerset lattice), a simpler definition of feasibility bounds can be used:

$$feasibilityBound_{dist}(abs, s, b) \equiv \begin{cases} abs \text{ is feasible and } s \text{ is infeasible,} \\ abs = s \sqcap b, \text{ and} \\ b \text{ is a direct predecessor of } \top\ . \end{cases}$$

Since it can be observed that $feasibilityBound_{dist}(abs, s, b)$ implies the previous predicate $feasibilityBound(abs, s, b)$, for distributive lattices, the former can be used as a more effective and sufficient condition.

Lemma 18 (Correctness of optimised exploration algorithm)**.** *When applied to a finite abstraction lattice, Algorithm 9 terminates and returns the set of minimal cost, maximal feasible abstractions.*

Proof. Note that the outer loops of the algorithms have the following loop invari-

ants:

$$Inv_{alg9} = \forall x \in Frontier.\ (cost(x) = minCost \wedge x \text{ is maximal feasible }) \\ \wedge \forall x \in CostlyAbs.\ (cost(x) > minCost \wedge \text{ all successors of } x \text{ are infeasible })$$

$$Inv_{alg10} = \forall x \in L.\ (abs \sqsubseteq_L x \wedge x \text{ is feasible } \Rightarrow x \sqsubseteq_L upperBound) \\ \wedge abs \text{ is feasible}$$

It follows directly that *Frontier* in Algorithm 9 can only contain maximal feasible abstractions. Further, upon termination the *Frontier* contains *all* maximal feasible abstractions with minimal cost. Namely, assume that there is a maximal feasible abstraction $mfa \notin Frontier$ with $cost(mfa) \leq minCost$. As in the proof of Lemma 17, it follows that *mfa* is incomparable with *Frontier*. Further, *mfa* cannot be above any element in *CostlyAbs*, since successors of *CostlyAbs* are infeasible; *mfa* cannot be below any element in *CostlyAbs* due to anti-monotonicity of *cost*. Therefore the loop condition must be satisfied, contradicting the assumption that Algorithm 9 had terminated.

Termination of Algorithm 9 can be shown like in the proof of Lemma 17.

Partial correctness of Algorithm 10 follows from its loop invariant. Termination is guaranteed since finite lattices have finite height, and *abs* grows strictly in every iteration of the `while` loop while *upperBound* may only decrease strictly with every iteration. Further, since the invariant holds that $abs \sqsubseteq_L upperBound$ Algorithm 10 terminates.

□

5.6 Experiments

5.6.1 Experimental Setup

5.6.1.1 Platform

The C program experiments below were done on an Intel Core i7 Duo 2.9 GHz with 8GB of RAM. The Petri net experiments with Eldarica were done on an Intel Core i5 2-core machine with 3.2GHz; Fast was run on an Intel Core i7 2-core machine with

1.7GHz.

5.6.1.2 Benchmarks

To evaluate our technique we have integrated our technique into the Eldarica Horn Clause model checker that utilises interpolants in its model checking algorithm. We describe the two use cases for the two benchmarks below.

C Programs. We first investigate our approach using a fixed template on a set of C program benchmarks. The C programs are converted into horn clauses. Our technique can be applied whenever interpolation is used by a model checker to eliminate spurious counterexamples. To this end, it is necessary to select one or multiple *abstraction points* in the constructed interpolation problem (which might concern an inductive sequence of interpolants, tree interpolants, etc.), and then to define an abstraction lattice for each abstraction point. For instance, when computing an inductive sequence $I_0, I_1, \ldots, I_{10}$ for the conjunction $P_1 \wedge \cdots \wedge P_{10}$, we might select interpolants I_3 and I_5 as abstraction points, choose a pair of abstraction lattices, and add abstraction relations to the conjuncts P_3, P_4, P_5, P_6. We then use Algorithm 7 to search for maximal feasible interpolation abstractions in the Cartesian product of the chosen abstraction lattices. With the help of cost functions, the best maximal feasible abstractions can be determined, and subsequently be used to compute abstract interpolants.

We have integrated our technique into the predicate abstraction-based model checker Eldarica [189], which uses Horn clauses to represent different kinds of verification problems [119], and solves recursion-free Horn constraints to synthesise new predicates for abstraction [127]. As abstraction points, recurrent control locations in counterexamples are chosen (corresponding to recurrent relation symbols of Horn clauses), which represent loops in a program. Abstraction lattices are powerset lattices over the template terms

$$\begin{aligned} &\{z \mid z \text{ a variable in the program}\} \\ \cup &\{x+y,\ x-y \mid x, y \text{ variables assigned in the loop body}\} \end{aligned}$$

Petri Nets. We accommodate the analysis of Petri nets by using the CEGAR approach (Counter Example Guided Abstract Refinement) [191, 129] of Eldarica, which provides a general framework for automatically computing inductive invariants. In this approach, a finite set of formulas in a decidable logic, called predicates, are used to transform a concrete system into an abstract one. Informally, the abstract system is a finite graph; states are labeled by Boolean combinations of predicates; actions are labelled by actions of the Petri net in such a way the finite graph simulates the Petri net. For Petri nets, Presburger arithmetic is a good candidate for denoting predicates.

We apply the CEGAR loop exploration as previously presented, but interpolants are not computed directly from sequences of actions $a_1, \ldots, a_k$. In fact, as previously mentioned, the quality of predicates generated by interpolation during the execution of the CEGAR loop algorithm must be improved for analysing Petri nets. Our approach to overcome this problem is based on different heuristics combining linear algebra and acceleration techniques [195]. In the sequel, we present three different heuristics.

Global-orthogonal-space heuristic (ABS (1)).
The computation of place invariants is a classical way for efficiently computing invariants of Petri nets. Place invariants are obtained by observing that if a vector $\bar{t}$ is orthogonal to $\bar{v} - \bar{u}$ for every action $a = (\bar{u}, \bar{v})$ of the Petri net, then $\bar{t}$ is orthogonal to $\bar{y} - \bar{x}$ for every marking $\bar{y}$ reachable from $\bar{x}$. That means the dot product of $\bar{t}$ with any reachable marking is a constant. Our first heuristic is based on the observation that orthogonal vectors t are suitable templates to be used in combination with term or inequality interpolation abstractions (Section 5.3.1 and 5.3.2). We first compute a basis of the vector space orthogonal to all vectors $\bar{v}_j - \bar{u}_j$ where $a_j = (\bar{u}_j, \bar{v}_j)$. This basis is then completed as an orthogonal basis B of the whole vector space generated by the markings. Such a computation is performed with Gauss elimination in polynomial time.

We then define an abstraction lattice using the powerset lattice for B (Def. 19), with each node in the lattice mapping to an inequality interpolation abstraction for

some subset of B. The abstraction lattice is equipped with a *cost* function (as in Section 5.5.3) that maps orthogonal vectors $\bar{t}$ to a small cost, and all other basis vectors to a large cost. As a result, the search procedures from Section 5.5 are able to systematically search for interpolation abstractions, and consequently interpolants, that are defined using orthogonal vectors; such interpolants are likely to be invariant under all or many actions of a Petri net.

Acceleration of individual recurring actions (ABS (2)).

Acceleration techniques compute reachability sets thanks to the exact effect of iterating some sequences of actions. For instance, let us consider an action $a = (\bar{u}, \bar{v})$, and observe that for every natural number $n \geq 1$, we have $\bar{x} \xrightarrow{a^n} \bar{y}$ if and only if $\bar{x} \geq \bar{u}$, $\bar{y} \geq \bar{v}$, and $\bar{y} + n.\bar{u} = \bar{x} + n.\bar{v}$. Our second heuristic is based on acceleration techniques. Basically, rather than computing interpolants directly from a sequence of actions $a_1, \ldots, a_k$, we compute interpolants $I_j[s_j]$ thanks to the following formula, where $\phi_j^{acc}[\bar{s}_{j-1}, n_j, \bar{s}_j]$ is the formula $\bar{s}_{j-1} \geq \bar{u}_j \wedge \bar{s}_j \geq \bar{v}_j \wedge \bar{s}_j + n_j.\bar{u}_j = \bar{s}_{j-1} + n_j.\bar{v}_j$ encoding the effect of iterating n_j times the action $a_j = (\bar{u}_j, \bar{v}_j)$.

$$(\bar{s}_0 = \bar{x} \wedge \phi_1^{acc}[\bar{s}_0, n_1, \bar{s}_1] \wedge \ldots \wedge \phi_j^{acc}[\bar{s}_{j-1}, n_j, \bar{s}_j])$$
$$\wedge(\phi_{j+1}^{acc}[\bar{s}_j, n_{j+1}, \bar{s}_{j+1}] \wedge \ldots \wedge \phi_k^{acc}[\bar{s}_{k-1}, n_k, \bar{s}_k] \wedge \bar{s}_k = \bar{y})$$

Note that $\exists n_j \geq 1.\ \phi_j^{acc}[\bar{s}_{j-1}, n_j, \bar{s}_j]$ is an over-approximation of $\phi_j[\bar{s}_{j-1}, \bar{s}_j]$, and can in fact be mapped to an inequality interpolation abstraction by means of quantifier elimination. As before, costs can be used to steer interpolant exploration towards interpolants that are invariant under recurrence of the accelerated action.

Detection of increasing sequences (ABS (3)).

In our last heuristics, we abstract away the sequence $a_1, \ldots, a_k$ of actions as a multiset. This abstraction basically extracts the Parikh image by counting the number of times an action occurs. Informally, thanks to linear algebra methods, sub-multisets of actions are computed in such a way that the effect of these actions is a non-negative vector. More formally, we consider for each action $a = (\bar{u}_a, \bar{v}_a)$ a natural

number n_a in such a way the following vector $\bar{v}$ satisfies $\bar{v} \geq \bar{0}$:

$$\bar{v} = \sum_{a \in A} n_a(\bar{v}_a - \bar{u}_a)$$

The computation of vectors $\bar{v}$ satisfying $\bar{v} \geq \bar{0}$ is motivated by the framework of acceleration as previously mentioned. In fact, an action $a = (\bar{u}, \bar{v})$ with a non-negative effect $\bar{v} - \bar{u}$ can be iterated an arbitrary number of times. Following this observation, we compute non-zero terms that maps a maximal (for the inclusion) set of vectors $\bar{v}$ as previously presented. This heuristics is a kind of mix of the two previously given heuristics. It provides sets of terms that are used for computing inequality interpolation abstraction.

5.6.2 Experimental Results

5.6.2.1 C Program Results

In Table 5.1 we evaluate the performance of our approach compared to Eldarica without interpolation abstraction, the acceleration-based tool Flata [189], and the Horn engine of Z3 [197] (v4.3.2). Benchmarks are taken from [196], and from a recent collection of Horn problems in SMT-LIB format.[2] They tend to be small ($10-750$ Horn clauses each), but challenging for model checkers. We focused on benchmarks on which Eldarica without interpolation abstraction diverges; since interpolation abstraction gives no advantages when constructing long counterexamples, we mainly used correct benchmarks (programs not containing errors). Lattice sizes in interpolation abstraction are typically $2^{15} - 2^{300}$; we used a timeout of 1s for exploring abstraction lattices. The letter after the model name distinguishes **C**orrect benchmarks from benchmarks with a reachable **E**rror state. For Eldarica, we give the number of required CEGAR iterations (N), and the runtime in seconds; for Flata and Z3, the runtime is given. Items with "*" indicate a timeout (set to 10 minutes), while - indicates inability to run the benchmark due to lack of support for some operators in the problems.

Overall, interpolation abstraction only incurs a reasonable runtime overhead.

[2] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/`

The biggest (relative) overhead could be observed for the rate_limiter example, where some of the feasibility checks for abstraction take long time. FLATA is able to handle a number of the benchmarks on which ELDARICA times out, but can overall solve fewer problems than ELDARICA. Z3 is able to solve many of the benchmarks very quickly, but overall times out on a larger number of benchmarks than ELDARICA with interpolation abstraction.

The results demonstrate the feasibility of our technique and its ability to avoid divergence, in particular on problems from [196].

5.6.2.2 Petri Net Benchmarks

In order to evaluate the efficacy of the different interpolation abstractions, we implemented a Petri net checker on the basis of the model checker ELDARICA and integrated the three forms of abstraction defined above, i.e., **ABS (1)** is the global-orthogonal-space heuristic, **ABS (2)** accelerates individual actions, **ABS (3)** detects increasing sequences, **ABS-all** combines all abstraction methods.

For each benchmark, "U" denotes that the considered configuration is unreachable, while "R" represents reachable configurations. Items with "*" indicate a time-out (set to 1 hour).

Experiments were done using a set of (bounded and unbounded) Petri net benchmarks taken from the literature.

The results are given in Table 5.2, in terms of runtime and the required number of CEGAR iterations. As can be seen, ELDARICA without interpolation abstraction performs poorly on Petri nets, and times out in many cases. The three interpolation abstractions show complementary performance, and each of our benchmarks could be solved using at least one of the heuristics. A combination of the interpolation abstractions (**ABS-all**) is also able to solve all benchmarks, although not always with the best runtime.

Finally, we compared to the acceleration-based model checker FAST [198]. FAST checks reachability queries by first computing a closed-form representation of the complete reachability set, and therefore has the same runtime for reachable as for unreachable cases. FAST is able to solve all bounded Petri nets in very short time,

but times out for a number of the unbounded ones. In particular, FAST fails for the "Exponential" example, and has a reachability set that cannot be defined in Presburger arithmetic.

5.7 Discussion

Compared to Chapter 3 the approach presented here takes a different approach to specialisation. Instead of generating specialised machinery for computing an interpolant we instead guide the theorem prover by instrumenting the original problem in order to guide the theorem prover to limit the interpolants that it can derive.

Our approach is very much related to a body of research that attempts to derive better interpolants.

Syntactic restrictions of considered interpolants [188, 199], for instance limiting the magnitude of literal constants in interpolants, can be used to enforce convergence and completeness of model checkers. This method is theoretically appealing, and has been the main inspiration for the work presented in this paper. In practice, syntactic restrictions tend to be difficult to implement, since they require deep modifications of an interpolating theorem prover; in addition, completeness does not guarantee convergence within an acceptable amount of time. We present an approach that is semantic and more pragmatic in nature; while not providing any theoretic convergence guarantees, the use of domain-specific knowledge can lead to performance advantages in practice.

It has been proposed to use term abstraction to improve the quality of interpolants [131, 200]: intuitively, the occurrence of individual symbols in an interpolant can be prevented through renaming. Our approach is highly related to this technique, but is more general since it enables fine-grained control over symbolic occurrences in an interpolant. For instance, in Example 20 arbitrary occurrence of the variable i_1 is forbidden, but occurrence in the context $x_1 - i_1$ is allowed.

The strength of interpolants can be controlled by choosing different interpolation calculi [48, 201], applied to the same propositional resolution proof. To the best of our knowledge, no conclusive results are available relating interpolant strength

with model checking performance. In addition, the extraction of different interpolants *from the same proof* is less flexible than imposing conditions already on the level of proof construction; if a proof does not leverage the right arguments why a program path is infeasible, it is unlikely that good interpolants can be extracted using any method.

In a similar fashion, proofs and interpolants can be minimised by means of proof transformations [202, 203]. The same comments as in the previous paragraph apply.

Divergence of model checkers can be prevented by combining interpolation with acceleration, which computes precise loop summaries for restricted classes of programs [204, 205, 196]. Again, our approach is more pragmatic, can incorporate domain knowledge, but is not restricted to any particular class of programs. Our experiments show that our method is similarly effective as acceleration for preventing divergence when verifying error-free programs. However, in contrast to acceleration, our method does not support the construction of long counterexamples spanning many loop iterations.

Templates have been used to synthesise program invariants in various contexts, for instance [206, 207, 208], and typically search for invariants within a rigidly defined set of constraints (e.g., with predefined Boolean or quantifier structure). Our approach can be used similarly, with complex building blocks for invariants specified by the user, but leaves the construction of interpolants from templates entirely to the theorem prover.

A number of systems compute interpolants by means of constraint-based interpolation, including CLP-Prover [209] and CSIsat [210]. This approach is similar in spirit to the template methods discussed in the previous paragraph, and imposes strong restrictions on the shape of considered interpolants. To the best of our knowledge, no attempts have been made to exploit domain-specific knowledge to guide constraint-based interpolation tools. Since our abstraction techniques are agnostic to the underlying interpolation engine, they can also be used in the context of constraint-based interpolation.

The proposes of generating beautiful has interpolants has been proposed by Albarghouthi etal. [47]. Here interpolants with particularly simple shape and Boolean structure are sought; empirically, interpolants of this kind were found to be beneficial for the convergence of model checkers. Domain-specific knowledge is not explicitly used when computing beautiful interpolants, but it is possible to use the procedure in [47] in combination with our abstraction framework.

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, abstraction lattice $(\langle L, \sqsubseteq_L \rangle, \mu)$
Result: Set of *all* maximal feasible interpolation abstractions of minimal cost

```
CostlyAbs ← ∅;
Frontier ← ∅;
minCost ← ∞;
while ∃ feasible abs ∈ L, incomparable with Frontier and CostlyAbs do
    m or c ← boundedMaximise(P, abs, minCost);
    if m was returned, and cost(m) < minCost then
        CostlyAbs ← CostlyAbs ∪ Frontier;
        Frontier ← {m};
        minCost ← cost(m);
    else
        Frontier ← Frontier ∪ {m} or CostlyAbs ← CostlyAbs ∪ {c};
    end
end
return Frontier;
```

Algorithm 9: Optimised Exploration Algorithm

Input: Interpolation problem $P = A[\bar{s}_A, \bar{s}] \wedge B[\bar{s}, \bar{s}_B]$, feasible abstraction $abs \in L$, minimal cost bound *minCost*

Result:

$m \in L$ s.t. $abs \sqsubseteq_L m$, m is maximal feasible, and $cost(m) \leq minCost$ **or**

$c \in L$ s.t. $abs \sqsubseteq_L c$, $cost(c) > minCost$, and all successors of c are infeasible

```
1  upperBound ← ⊤;
2  while true do
3      fs ← undef;
4      for all immediate successors s of abs, while fs is undefined do
5          if s ⊑_L upperBound then
6              if s is feasible then
7                  fs ← s;
8              else if ∃b. feasibilityBound(abs, s, b) then
9                  upperBound ← upperBound ⊓ b;
10                 if cost(upperBound) > minCost then
11                     return m ← upperBound;
12                 end
13                 if upperBound is feasible then
14                     return c ← upperBound;
15                 end
16             end
17         end
18     end
19     if fs is defined then
20         pick abstraction middle such that fs ⊑_L middle ⊑_L upperBound;
21         if middle is feasible then
22             abs ← middle;
23         else
24             abs ← fs;
25         end
26     else
27         if cost(abs) > minCost then
28             return c ← abs;
29         else
30             return m ← abs;
31         end
32     end
33 end
```

Algorithm 10: Optimised maximisation algorithm *boundedMaximise*$(P, abs, minCost)$

Benchmark	Eldarica		Eldarica-ABS		Flata	Z3
	N	sec	N	sec	sec	sec
C programs from [196]						
boustrophedon (C)	*	*	10	10.7	*	**0.1**
boustrophedon_expansed (C)	*	*	11	7.7	*	**0.1**
halbwachs (C)	*	*	53	2.4	*	**0.1**
gopan (C)	17	22.2	62	57.0	**0.4**	349.5
rate_limiter (C)	11	2.7	11	19.1	1.0	**0.1**
anubhav (C)	1	1.7	1	1.6	**0.9**	*
cousot (C)	*	*	3	7.7	**0.7**	*
bubblesort (E)	1	2.8	1	2.3	77.6	**0.3**
insdel (C)	1	0.9	1	0.9	0.7	**0.0**
insertsort (E)	1	1.8	1	1.7	1.3	**0.1**
listcounter (C)	*	*	8	2.0	**0.2**	*
listcounter (E)	1	0.9	1	0.9	0.2	**0.0**
listreversal (C)	1	1.9	1	1.9	4.9	*
mergesort (E)	1	2.9	1	2.6	1.1	**0.2**
selectionsort (E)	1	2.4	1	2.4	1.2	**0.2**
rotation_vc.1 (C)	7	2.0	7	0.3	1.9	**0.2**
rotation_vc.2 (C)	8	2.7	8	**0.2**	2.2	0.3
rotation_vc.3 (C)	0	2.3	0	0.2	2.3	**0.0**
rotation.1 (E)	3	1.8	3	1.8	0.5	**0.1**
split_vc.1 (C)	18	3.9	17	3.2	*	**1.1**
split_vc.2 (C)	*	*	18	1.1	*	**0.2**
split_vc.3 (C)	0	2.8	0	1.5	*	**0.0**
Recursive Horn SMT-LIB Benchmarks						
addition (C)	1	0.7	1	0.8	0.4	**0.0**
bfprt (C)	*	*	5	8.3	-	**0.0**
binarysearch (C)	1	0.9	1	0.9	-	**0.0**
buildheap (C)	*	*	*	*	-	*
countZero (C)	2	2.0	2	2.0	-	**0.0**
disjunctive (C)	10	2.4	5	5.0	**0.2**	0.3
floodfill (C)	*	*	*	*	41.2	**0.1**
gcd (C)	4	**1.2**	4	2.0	-	*
identity (C)	2	1.1	2	2.1	-	**0.1**
mccarthy91 (C)	4	1.4	3	2.4	0.2	**0.0**
mccarthy92 (C)	38	5.6	7	8.7	**0.1**	**0.1**
merge-leq (C)	3	1.1	7	7.0	15.7	**0.1**
merge (C)	3	1.1	4	4.5	14.7	**0.1**
mult (C)	*	*	15	**52.8**	-	*
palidrome (C)	4	1.4	2	2.1	-	**0.1**
parity (C)	4	1.6	4	2.9	**0.8**	*
remainder (C)	2	**1.1**	3	1.6	-	*
running (C)	2	0.9	2	1.7	0.2	**0.1**
triple (C)	4	2.0	4	5.1	-	**0.1**

Table 5.1: *Comparison of* ELDARICA *without interpolation abstraction,* ELDARICA *with* ABS*traction,* FLATA*, and Z3*

Benchmark		Eldarica		ABS (1)		ABS (2)		ABS (3)		ABS-all		Fast
		N	sec	N	sec	N	sec	N	sec	N	sec	sec
Bounded Petri nets												
Basic ME	U	3	1.3	3	1.55	3	1.3	3	1.3	3	1.7	**<1**
IFIP	U	12	2.3	2	1.7	12	4.3	10	4.6	2	1.8	**<1**
L6000	U	*	*	17	16.5	8	4.7	*	*	3	4.0	**<1**
Long 1	U	*	*	1	1.2	7	7.1	*	*	1	1.2	**<1**
Long 2	U	*	*	1	1.4	10	11.1	13	15.4	1	1.4	**<1**
Long 3	U	*	*	*	*	10	11.5	8	8.2	11	19.2	**<1**
Long 4	U	*	*	1	2.8	9	11.2	103	79.6	1	3.0	**<1**
Manufacturing 3	U	*	*	323	802	441	2635	675	1946	354	1588	**2.4**
Manufacturing 9	R	*	*	232	801	264	632	560	3053	295	1515	**10.8**
Unbounded Petri nets												
Alternating bit prot.	R	64	14.8	16	10.5	44	17.5	35	15.2	16	14.7	**4.5**
FMS	R	25	**20.5**	23	28.4	25	27.3	17	24.7	23	32.4	98.4
”	U	18	9.8	2	7.0	13	17.6	18	10.7	2	**6.7**	37.4
FinkelKM	R	16	5.8	15	8.9	16	11.6	17	11.6	15	22.7	**5.7**
”	U	14	5.7	3	**2.4**	6	6.5	7	3.4	3	2.5	5.7
Finkel Counterex.	R	12	2.3	10	3.5	12	2.3	12	2.6	10	3.6	**<1**
Kanban	R	28	**33.3**	19	35.8	29	70.0	22	41.5	25	67.3	*
”	U	*	*	1	3.9	*	*	*	*	1	**3.8**	*
Mesh 2x2	R	75	**52.3**	64	82.9	60	56.6	68	102	65	105	97
”	U	186	170	18	**33.7**	*	*	*	*	18	37.8	97
Multipool	U	56	423	1	5.4	*	*	*	*	1	**5.0**	*
Pingpong	U	3	1.4	2	1.5	2	1.4	2	1.3	2	1.5	**<1**
PNCSA Cover	R	32	15.0	17	16.5	32	**14.5**	26	16.4	17	17.7	*
Exponential	U	*	*	8	3.9	8	**3.4**	6	5.1	5	5.2	*
Language inclusion	U	*	*	*	*	5	3.7	2	1.7	6	9.0	**<1**

Table 5.2: *Comparison of tools for checking reachability in bounded and unbounded Petri nets, on benchmarks taken from the literature.*

Chapter 6

Industrial Applications

In this chapter, we describe several application areas of logic defined static analysis. In particular, these real world use cases describe various applications of logic defined static analysis in industrial settings. As a secondary consequence, this chapter aims to further evaluate the scalability of tools that implement the techniques of this thesis by showing their scalability in complex industrial settings.

The first use case we present, evaluates Soufflé as a computational backend for verifying security properties on virtual networks. This use case was performed in collaboration with Amazon Web Services (AWS). The second use case evaluates Soufflé as a static analyser for performing points-to and security static analyses on the Oracle JDK 7 code base. This case study was performed in collaboration with Oracle Labs.

6.1 Use Case: Security Analysis of Amazon Networks

Computer networks are typically built from a variety of specialised heterogeneous devices running complex distributed protocols. Network administrators, responsible for operability of a network, must configure and deploy every protocol separately on each individual device. In an effort to simplify this task, software-defined networking (SDN) [211] has been proposed as a modern alternative. Modern platforms for cloud computing offer their users means of configuring private networks in the style of SDN. This includes Amazon Virtual Private Cloud (Amazon virtual) [212] networks, the networking layer for the Amazon Elastic Compute Cloud (Amazon EC2) infrastructure. Administrators of virtual networks use a centralised control panel or a specialised API to launch EC2 instances, set up subnets and route tables, and configure connectivity and security settings of the network.

Despite the increase of usability provided by platforms such as Amazon EC2, the virtual networks remain prone to misconfigurations. These misconfigurations are caused by the complexity of large-scale enterprise networks and might lead to downtimes and breaches of security. Discovering such misconfiguration in industrial-size networks can be both extremely laborious and computationally in-

tensive. Therefore, verifying the correctness of network configurations is an important and challenging task. In this use case we investigate the use of Soufflé as a tool for verifying various network configuration properties hold.

6.1.1 Reachability Properties for Virtual Networks

Figure 6.1 shows an example of a virtual network that consists of two subnets "Web" and "Database" with three network instances in each of them. Each of the subnets is assigned with a route table that configures the "Web" subnet to be accessible from the internet and the "Database" to not be accessible from the internet. In addition, each of the subnets is assigned with an access control list (ACL) that contains their security access rules. In particular, one of the rules forbids SSH access to the database servers directly or indirectly (via the web servers).

In a realistic setting, this network administrator may want to make sure that this network retain certain properties after each change in its configuration. For example, the network administrator may want to check the following property.

Example 27. *All network instances in the subnet "Web" can access all network instances in the subnet "Database".*

In addition, the network administrator might want to know which networking components satisfy a given property, such as the ones in the following example.

Example 28. *All network instances that have the port 22 (SSH) accessible from the internet.*

We will refer to questions that network administrators might want to answer, such as the ones in Examples 27 and 28, as *network questions*. In particular, we will refer to questions similar to Example 27 as *boolean questions*, because they expect a true or false answer, and to questions similar to Example 28 as *list questions*, because they expect a list of networking components as an answer.

6.1.2 EC2 Network Semantics

We answer network questions *statically*, that is, instead of sending packets in a network, we build a model of the network and reason about this model. Our *network model* consists of two parts, the *formal specification* and the *snapshot* of the

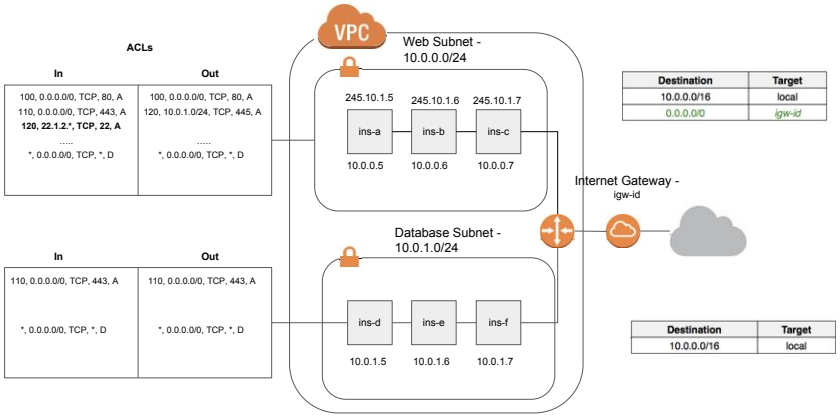


Figure 6.1: *An example virtual network*

network. The specification formalises the semantics of each of the components available in the network. For example, the formal specification describes how a route table directs network traffic in a subnet or in which order a firewall applies rules in the access control list (ACL). The snapshot describes the topology of the given network. For example, the snapshot contains the list of network instances, subnets, and their route tables. Naturally, the formal specification in the model of each particular virtual network is the same, whereas the snapshot differs. We express network questions in the language of many-sorted first-order logic. In the remainder of this section we describe syntax and semantics of network models and network questions.

6.1.2.1 Network Models

A network model is defined as a finite set of first-order Horn clauses. We disallow function symbols and allow stratified negation. We assume the plain logic programming semantics for these Horn clauses, defined in the standard way. In particular, we make the closed-world assumption and treat negation as failure. In addition, our network models use the theory of bit vectors to describe ports, IPv4 addresses, and subnet masks.

A *signature* of the network model is a triple (T, C, P), where T is a set of *types*, C is a set of *constants*, and P is a set of *predicates*. We assign each constant with a type $\tau \in T$ and each predicate with a type $\tau_1 \times \ldots \times \tau_n$ $(n \geq 0)$, where $\tau_i \in T$ for each $1 \leq i \leq n$. We assume a countable infinite set of *variables*. We assign each variable

with a type $\tau \in T$. We call a *term* of the type $\tau \in T$ a constant or a variable of that type. We call an *atom* an expression of the form $p(t_1, \ldots, t_n)$, where $n > 0$, $p \in P$ is a predicate of the type $\tau_1 \times \ldots \times \tau_n$, and each t_i, $1 \leq i \leq n$ is a term of the type τ_i. We call a *literal* an atom or its negation.

A *rule* is a Horn clause of the form $A \leftarrow L_1 \wedge \ldots \wedge L_n$ ($n \geq 0$), where A is an atom which we call the *head* of the rule and each of $B_1, \ldots, B_n$ is a literal. If $n = 0$ and all arguments of A are constants then we call such rule a fact. We call a *definition* of the predicate $p \in P$ the set of all rules in the network model that use p in their head.

We assume that the signature contains (i) types bits16 and bits32; (ii) 2^{16} constants of the type bits16; (iii) 2^{32} constants of the type bits32; (iv) predicates $bits16_{<}$, $bits16_{\leq}$, $bits16_{+1}$, $bits16_{-1}$ of the type bits16 $\times$ bits16 with a special semantics and (iv) predicate $bits32_{\wedge}$ or the type bits32 $\times$ bits32 $\times$ bits32 with a special semantics.

bits16 and bits32 represent the types of 16-bit and 32-bit vectors. The semantics of the predicates is that of the correspondent operations over bit vectors defined in the standard way.

We assume that for each type $\tau \in T$ the signature contains the equality predicate $=_\tau$ of the type $\tau \times \tau$ and the network model contains the rule $=_\tau(X, X)$.

The network specification part of the model contains types, predicates, constants, and rules that describe the semantics of the networking components in virtual networks. For example, the specification defines the semantics of SSH tunnelling. One network interface (ENI) can SSH tunnel to another ENI iff it can either connect to it over SSH directly, or through a chain of one or more intermediate ENIs. In order to express this concept, the specification contains predicates *canSshTunnel* and *canSsh*, each of the type eni $\times$ eni, and the two following rules.

$$canSshTunnel(Eni_1, Eni_2) \leftarrow canSsh(Eni_1, Eni_2).$$

$$canSshTunnel(Eni_1, Eni_2) \leftarrow canSshTunnel(Eni_1, Eni_3) \wedge canSshTunnel(Eni_3, Eni_2).$$

The specification of Amazon virtual networks that we used in this work consists of approximately 50 types, 200 predicates, and over 240 rules.

The network snapshot part of the model contains constants and facts that describe the configuration of the networking components in a given Amazon virtual network. For example, the snapshot of a network with a single instance `i-abcd1234` in a single subnet "Web" consists of the constants $\text{instance}_{\text{abcd1234}}$ and $\text{subnet}_{\text{Web}}$, and the fact

$$\mathit{instanceHasSubnet}(\text{instance}_{\text{abcd1234}}, \text{subnet}_{\text{Web}}).$$

6.1.2.2 Network Questions

We express network questions as formulas of many-sorted first-order logic with the standard logical connectives $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\oplus$, and equality. These formulas only use types, constants, and predicates from the signature of the network model. The formulas do not use any function symbols. We allow interpretation of these formulas to use empty domains and otherwise assume the standard semantics of many-sorted first-order logic.

We express boolean questions as closed formulas, that is, formulas in which all occurrences of variables are bound by a quantifier. Conversely, we express list questions as formulas with free variables. The answer to a boolean question is true iff its correspondent formula is valid. The answer to a list question is the set of variable substitutions that satisfies its correspondent formula.

The boolean question in Example 27 is expressed as the following formula.

$$\begin{aligned}
&(\forall w : \text{instance})(\forall d : \text{instance}) \\
&\quad (\mathit{instanceHasSubnet}(w, \text{subnet}_{\text{Web}}) \wedge \\
&\quad\ \mathit{instanceHasSubnet}(d, \text{subnet}_{\text{Database}}) \Rightarrow \\
&\quad\quad \mathit{instanceCanConnectToInstance}(w, d))
\end{aligned} \tag{6.1}$$

The list questions in Example 28 is expressed as the following formula with

the free variables i of the type instance and e of the type eni.

$$\begin{aligned} &\mathit{instanceHasEni}(i, e) \wedge \\ &\mathit{reachablePublicTcpUdp}(\mathrm{dir}_{\mathrm{ingress}}, \mathrm{proto}_6, e, \mathrm{port}_{22}, \\ &\qquad\qquad \mathrm{publicIp}_{8:8:8:8}, \mathrm{port}_{40000}) \end{aligned} \tag{6.2}$$

All predicates and constants used in Formulas 6.1 and 6.2 are part of the signature of the network model. Constants $\mathrm{subnet}_{\mathrm{Web}}$ and $\mathrm{subnet}_{\mathrm{Database}}$ are part of the network snapshot, and all other predicates and constants are part of the network specification.

6.1.3 Translating Virtual Networks in Datalog

6.1.3.1 Snapshot

Soufflé accepts definitions of typed relations, contains the predefined symbol and numeric types, and accepts definitions of new types. The types in Soufflé are interpreted under the open-world assumption. We model the types of the network models, interpreted as finite domains, using Datalog relations with one argument. Let τ be a type and $c_1, \ldots, c_n$ ($n \geq 0$) be constants of this type. We introduce a relation τ and add the facts $\tau(c_i)$, $1 \leq i \leq n$ to the set of Datalog rules. We use literals of the form $\tau(t)$ in every Datalog rule to guard the argument t of the type τ in the head of the rule.

6.1.3.2 Rules

Let $p(t_1, \ldots, t_n) \leftarrow L_1 \wedge \ldots \wedge L_m$ ($n \geq 0, m \geq 0$) be a rule in the network model, where p is a predicate of the type $\tau_1 \times \ldots \times \tau_n$ and each of $L_1, \ldots, L_m$ is a literal. We translate p to a Datalog relation R and translate this rule to the Datalog rule

$$R(t_1, \ldots, t_n) \text{ :- } \tau_1(t_1) \wedge \ldots \wedge \tau_n(t_n) \wedge L_1 \wedge \ldots \wedge L_m.$$

6.1.3.3 Questions

We automatically translate a network question expressed as a first-order formula ϕ without function symbols to a Datalog query and a set of Datalog rules. We start by

converting ϕ to a prenex disjunctive normal form that is

$$(\forall x_1 : \tau_1)\ldots(\forall x_n : \tau_n)(\exists y_1 : \sigma_1)\ldots(\exists y_m : \sigma_m)D,$$

where $n \geq 0$, $m \geq 0$ and D is a disjunction $C_1 \vee \ldots \vee C_k$ ($k \geq 0$) of conjunctions of atomic formulas. Let $z_1 : \upsilon_1, \ldots, z_l : \upsilon_l$ ($l \geq 0$) be all free variables of ϕ. Recall that $l = 0$ for formulas expressing boolean network questions and $l > 0$ for formulas expressing list network question. We introduce two fresh relations R and Q of the types $\tau_1 \times \ldots \times \tau_n \times \upsilon_1 \times \ldots \times \upsilon_l$ and $\upsilon_1 \times \ldots \times \upsilon_l$, respectively. The translated set of Datalog rules consists of $n + 1$ rules: n rules of the form

$$R(x_1, \ldots, x_n, z_1, \ldots, z_l) \text{ :- } \tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n) \wedge \upsilon_1(z_1) \wedge \ldots \wedge \upsilon_l(z_l) \wedge C_i$$

for each $1 \leq i \leq k$ and the rule

$$Q(z_1, \ldots, z_l) \text{ :- } \upsilon_1(z_1) \wedge \ldots \wedge \upsilon_l(z_l) \wedge \neg R(x_1, \ldots, x_n, z_1, \ldots, z_l).$$

Note that we can use each conjunction C_i in a Datalog rule because each literal in C_i only contains variables and constants — there are no function symbols in ϕ and they do not appear during a conversion to prenex disjunctive normal form. Finally, the Datalog query is $\neg Q(z_1, \ldots, z_l)$.

We translate types bits16 and bits32 to numeric types for 32 and 16-bit integers, respectively, and translate the predicates over bit vectors into their correspondent built-in Soufflé operations.

We illustrate our translation using examples from Section 6.1.1. We translate

Formula 6.1 that expresses a boolean network question to the Datalog rules

$$
\begin{aligned}
R(w,d) \text{ :- } & \mathit{Instance}(w) \wedge \\
& \neg \mathit{InstanceHasSubnet}(w, \text{subnet}_{\text{Web}}). \\
R(w,d) \text{ :- } & \mathit{Instance}(d) \wedge \\
& \neg \mathit{InstanceHasSubnet}(w, \text{subnet}_{\text{Database}}). \\
R(w,d) \text{ :- } & \mathit{Instance}(w) \wedge \mathit{Instance}(D) \wedge \\
& \mathit{InstanceCanConnectToInstance}(w,d). \\
Q() \text{ :- } & \neg R(w,d).
\end{aligned}
$$

and the Datalog query $\neg Q()$. Note that multiple rules in the definition of R appear because of the translation to disjunctive normal form. We translate Formula 6.2 that expresses a list network question to the Datalog rules

$$
\begin{aligned}
R(i,e) \text{ :- } & \mathit{Instance}(i) \wedge \mathit{Eni}(e) \wedge \\
& \mathit{InstanceHasEni}(i,e) \wedge \\
& \mathit{ReachablePublicTcpUdp}(\text{dir}_{\text{ingress}}, \text{proto}_6, \\
& \qquad\qquad e, \text{port}_{22}, \\
& \qquad\qquad \text{publicIp}_{8:8:8:8}, \text{port}_{40000}). \\
Q(i,e) \text{ :- } & \mathit{Instance}(i) \wedge \mathit{Eni}(e) \wedge \neg R(i,e).
\end{aligned}
$$

and the Datalog query $\neg Q(i,e)$.

6.1.3.4 Optimisations

Syntactic Inlining Transformation A common bottleneck in Datalog programs is the case when very large relations are constructed only to be later constrained via additional rules. This bottleneck may be regarded as bad programming practice, however, it is often unavoidable when Datalog is used to construct libraries of relations as in this case study.

To understand this bottleneck, assume we have a relation $R(x,y)$. In addi-

tion to R, an extended interface may exist n the relation G, defined by the rule $G(w,x,y,z)$:- $R(x,y),D(w,z)$. Due to a lack of equality bindings, G will contain the product of both relations. We then project values in G that have a equality binding between the first and third, and second and fourth attributes. When evaluated we will construct a very large relation G only to constrain it to build A.

To mitigate this bottleneck, rules can be *propagated*, i.e., the occurrences of a relation can be replaced with its rule bodies. For each rule unification is necessary to equate the different variable names in an occurring relation and the body of its rule. We demonstrate this in the example below:

$$G(w,x,y,z) \text{ :- } R(x,y),D(w,z).$$
$$A(x,y) \text{ :- } G(x,y,x,y).$$

can be transformed into the rule below with the unification $\{w = y, x = z\}$:

$$A(x,y) \text{ :- } R(x,y),D(y,x).$$

In the transformed program, the large relation G, does not need to be computed before it is constrained in the proceeding rule, instead we substitute the body of the rule and constrain before it is constructed. Despite its large optimisation potential, rule propagation is limited in its use. For instance, input and output relations cannot be propagated, rules that form a cycle in their precedence graph, and which are composed entirely of propagated relations cannot be forward propagated (a rule must be selected to be ignored to break the cycle), and relations that introduces new variables in their rules but appear negated in a clause cannot be propagated.

Lemma 19 (Correctness of Inlining Transformation)**.** *The inlining transformation is preserves the correctness of the Datalog program*

Proof. Since the transformation can only be performed on sets of chained rules with no-negated links, inlining is equivalent to repeated application of one step of top-down resolution. □

Existence Elimination Transformation A bottleneck occurs when we build large relations and only need to assert the existence of an arbitrary element. For example, we may have the Datalog program:

$$A(x) \text{ :- } A(x), G(x).$$

$$A(x) \text{ :- } B(x).$$

$$Q() \text{ :- } A(x).$$

Again, in this case, A may be a very large relation and we want to avoid computing the entire relation to assert that it has an element in it. For such problems, magic set transformation is not possible and forward propagation, which in some cases may alleviate the problem, is not sufficient for recursive rules. This problem can be generalised for any relation where all of its variables are not in the head of a rule and not bound in another body relation, i.e., they can be replaced by _. In this case, we make the observation that as the relation is used purely in an existential fashion, and we can ignore recursive rules which do not need to be computed due to the monotonicity of Datalog. Assuming we are in the realm of stratified Datalog, then since a relation cannot appear negated when it is at the head of a clause, a recursive rule will only produce a result if the relation already contains a tuple. Hence this transformation, if applicable, results in the removal of all recursive rules of a where relation A is the head. Moreover, this may result in additional relations being reduces to existential form as recursive rules are removed. The example above can be rewritten as follows:

$$A() \text{ :- } B(\_).$$

$$Q() \text{ :- } A().$$

Here, we can treat A as a nullary relation which is true or false depending on the existence of an element in B. Note that, if B is an IDB relation and not used in any other way, the transformation can be applied to B.

Lemma 20 (Correctness of Existence Elimination)**.** *The existence elimination transformation preserves the semantics of the Datalog program.*

Proof. Given the monotonicity property of the semi-naïve algorithm, the cardinality of a relation can only increase at each derivation. Also note, a rule cannot derive new tuples if its body contains an empty relation. Hence, derived relation with has a cardinality of greater than 1 in its initial delta (using the base case) or will always have a cardinality of 0. □

6.1.4 Experiments

6.1.4.1 Experimental Setup

Table 6.1 provides characteristics each benchmark, namely, number of instances, security groups and ENIs. Each query was chosen to test different performance characteristics. The Query 1 is a list query with large outputs many literal bindings. Query 2 is a boolean query for which magic set transformation cannot be applied. The benchmarks are well distributed containing benchmarks of hundreds of mega bytes in size, with thousands of instances, security groups and ENIs to very small networks with a few instances. Each benchmark contains an **L**, **M** or **S** which indicates the size of the snapshot file. Benchmarks with **L** are typically 100 MB and up, **M** between 10 and 100 MB and **S** under 10 MB.

Platform. All experiments were run on an Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz with a 3072 KB cache and 4GB RAM.

Usage of Soufflé. In this use case Soufflé is used as a standalone tool for large analyses and in the context of a wider tool configuration. When used standalone, Soufflé can synthesise/compile the one off analysis and run on various virtual networks (see results in Chapters 3 and 4). However, Soufflé can also be used as a backend in a wider tool. In this case, synthesis/compilation becomes part of the overall computation (no longer static), however, even in this case we show that Soufflé can exhibit superior performance when compared to a state-of-the-art tool such as μZ. When we analyse small inputs, Soufflé uses its RAM interpreter. When the input is large then the synthesis/compilation payoff becomes apparent. In practice, once an analysis is compiled it can be cached if used again and this elevating the synthesis/compiler overhead.

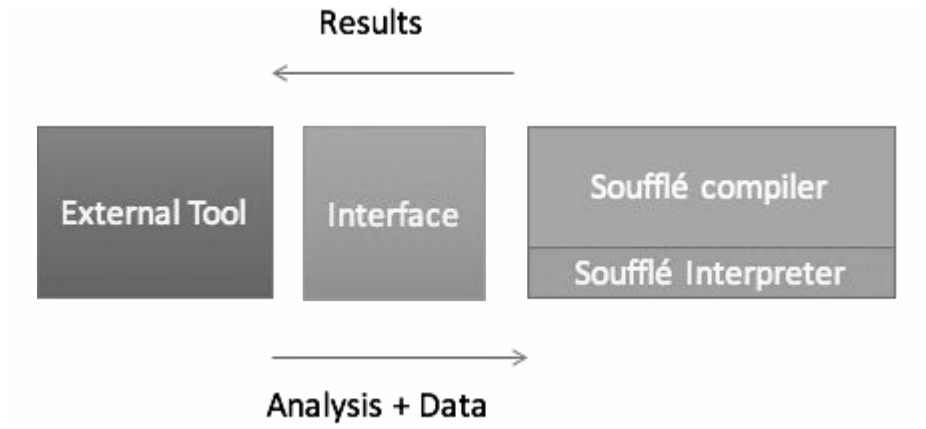


Figure 6.2: *Souffle Usage Setup*

6.1.4.2 Experimental Results

Figure 6.4 describe a comparison of Soufflé compiler and interpreter modes with μZ. Each mark on the plot represents a relative performance value categorised by the size of the snapshot. The plot is broken down into 4 regions:

- (I) This region represents values where both the Soufflé compiler and interpreter perform better than μZ
- (II) This region represents values where only the Soufflé interpreter performs better than μZ
- (III) This region represents values where only the Soufflé compiler performs better than μZ
- (IV) This region represents values where both the Soufflé compiler and interpreter are worse than μZ

As we can see in Figure 6.4, the vast majority of networks are in region (I), i.e., both compilation and interpretation is better than μZ. Region (II) contains small networks for which the compilation overhead was larger than interpretation however, these are still faster than μZ. Likewise, region (III) contains several larger networks that performed only better than μZ by compilation. Finally, region IV) contains a single small network that performed better than Soufflé. A similar pattern can be seen in the memory consumption. The majority of networks are in region (I),

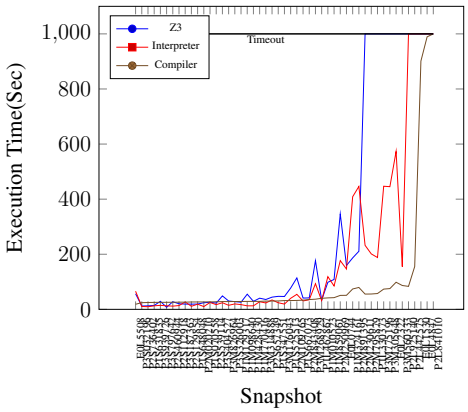
Snapshot	Inst	Eni	Sec
E0L08	599	2163	27
E0L44	990	9037	28
E0L22	2090	670	26
E0L30	7508	7577	26
E0L47	8908	1062	26
P1L330373	2116	2719	129
P1L462867	4	8	4988
P1M129317	3	38	382
P1M098640	1	1439	1
P1M859061	420	422	887
P1M428430	285	561	5
P1M010523	323	324	627
P1M324161	1196	1415	38
P1M770416	545	557	12
P1S070558	117	165	214
P1S525513	93	170	328
P1S347551	37	38	266
P1S631835	85	87	173
P1S648048	233	271	47
P1S667073	208	210	301
P2L925140	204	204	1459
P2L442575	870	873	1563
P2L841010	1268	1277	1565
P2M891395	1269	1481	39
P2M020210	795	795	1

Snapshot	Inst	Eni	Sec
P2M230611	1471	3636	23
P2M795879	1453	1458	21
P2M843408	170	170	292
P2M250969	1299	3426	22
P2M109765	510	1815	21
P3M560333	588	596	21
P3M314888	340	430	11
P3M268929	693	2348	110
P3M826964	297	585	115
P3M176043	1218	1230	327
P3M775196	1218	1439	28
P3M305648	1211	1432	1155
P3S939726	45	538	120
P3S404527	15	1045	27
P3S797642	100	112	5
P3S032196	188	188	4
P3S328003	22	23	38
P3S187363	192	217	38
P2S736402	116	133	7
P2S129854	262	308	142
P2S657349	322	358	91
P2S112918	80	89	3
P2S539114	73	127	329
P2S460944	279	321	39

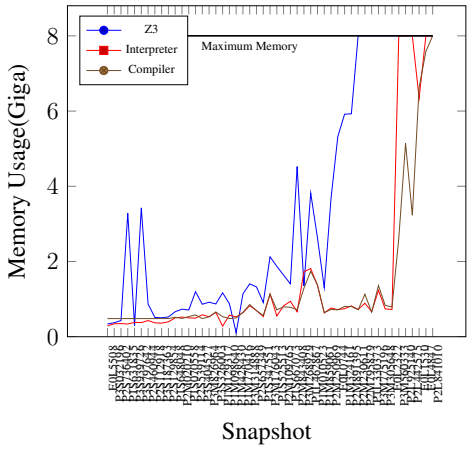
Table 6.1: *Virtual Network Benchmarks*

with a few small networks in region (II) and a large network in region (III). Again a single (the same) network can be found in region (IV). In Figure 6.3 we can see the absolute execution times. The execution times range from 10 seconds to 450 seconds. Memory consumption is between 100 MB to 7.5 GB. Both the interpreter and compiler typically perform better than μZ. Further, we can see that μZ times out (due to running out of memory) more frequently than both the interpreter and compiler, with the compiler running out of memory only once, compared to μZ 4 times and the interpreter 3 times. We can also see as the sizes of the networks get larger and more memory is consumed, the very low memory overhead of Soufflé becomes more noticeable in both modes.

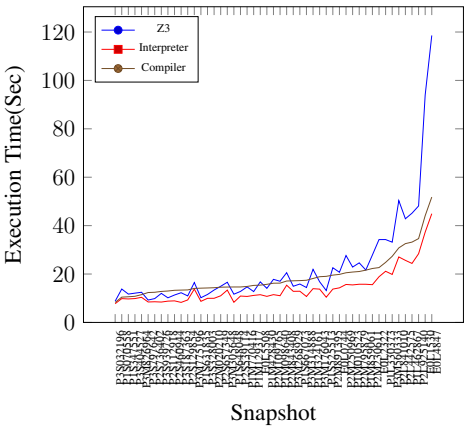
When caching is added (avoiding re-compilation) when an analysis is invoked



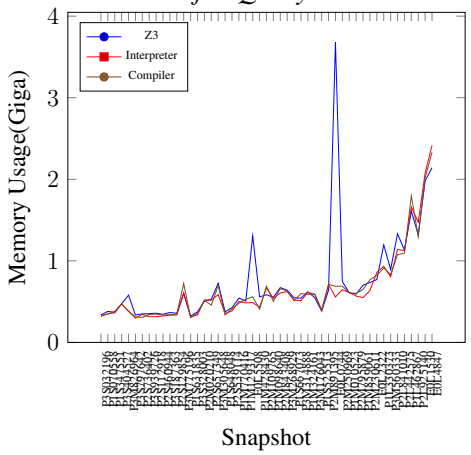
(a) Runtime on Virtual Network Benchmarks for Query 1



(b) Memory Usage on Virtual Network Benchmarks for Query 1



(c) Runtime on Virtual Network Benchmarks for Query 2



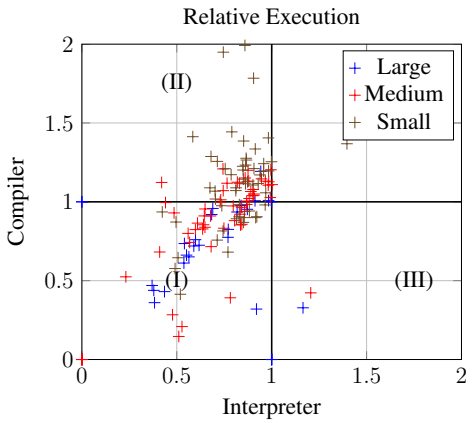
(d) Memory Usage on Virtual Network Benchmarks for Query 2

Figure 6.3: *Performance on Virtual Network Benchmarks*

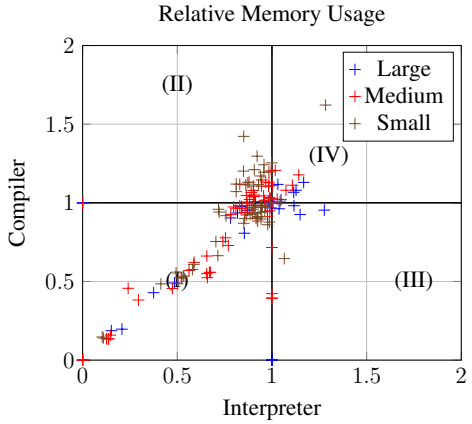
several times, we can expect the Soufflé compiler to perform without compilation overhead which on these benchmarks is approx. 10 seconds. Furthermore, as more complex and larger virtual networks are analysed at Amazon, the Soufflé approach will be come increasingly beneficial.

6.1.5 Discussion

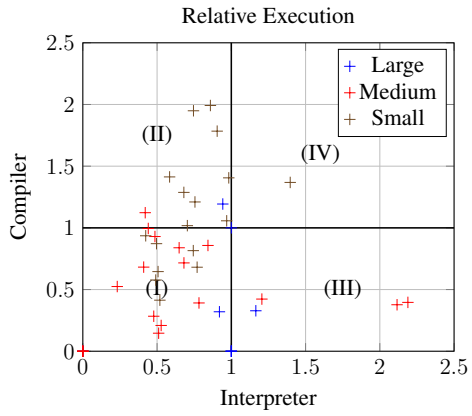
Several tools have been developed [73, 213, 214, 215, 216, 217, 218, 219] in an effort to verify various SDN components. These tools employ specialised algorithms [216] as well as general purpose reasoning engines such as Datalog [37, 219], BDDs [217], SMT [213, 214], and SAT [218, 215]. The most related of these to our work is that of NoD/μZ [72, 37] and batfish [73]. NoD/μZ checks packet reach-



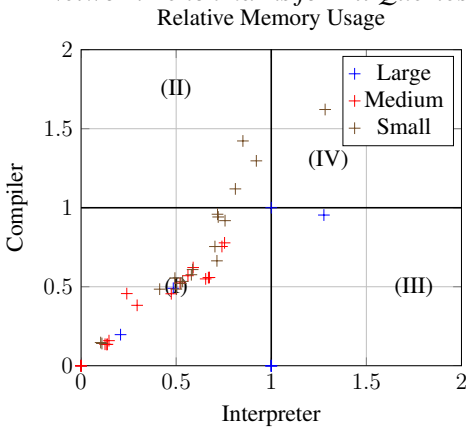
(a) Relative Runtime on Virtual Network Benchmarks for All Queries



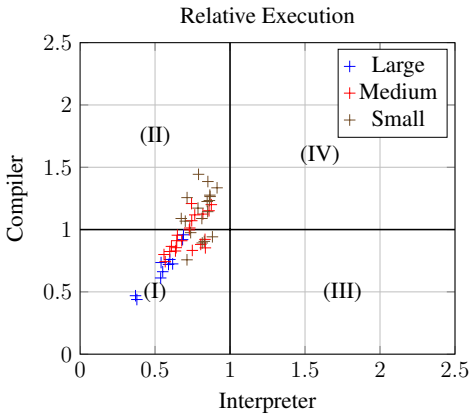
(b) Relative Memory Usage on Virtual Network Benchmarks for All Queries



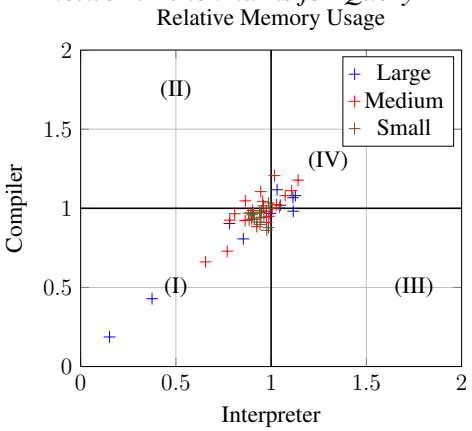
(c) Relative Runtime on Virtual Network Benchmarks for Query 1



(d) Relative Memory Usage on Virtual Network Benchmarks for Query 1



(e) Relative Runtime on Virtual Network Benchmarks for Query 2



(f) Relative Memory Usage on Virtual Network Benchmarks for Query 2

Figure 6.4: *Relative Performance on Virtual Network Benchmarks*

ability and this the network is topology is encoded as part of the rules. We on the other hand, encode the network as a set of input relations. Batfish employs a similar encoding to NoD/μZ (and uses μZ) as an underlying engine and uses the SMT capabilities of Z3 to find error traces. This approach has been further expanded to synthesising EDB relations in Datalog using SMT [219]. This approach could be used in conjuction with our approach to repair networks once they are found to be incorrect. As our experiments demonstrate we scale significantly better to μZ by combining the Soufflé interpreter and compiler. Incorporating error traces and synthesis into Soufflé is left to future work.

Less related work includes SecGuru, which is a tool for the verification of network connectivity policies at Microsoft Azure using Z3 [213]. Here two sets of ACL rules from firewalls and calculate their semantic difference. Compared to [213] which only talks about ACLs, they extended their work [214] with verification of routing tables and Border gateway Protocol (BGP). VeriCon [215] is a verification tool for SDN controllers. VeriFlow [216] is a tool that checks network-wide invariants in real-time, using a specialised algorithm. Anteater [218] is a data plane verification using SAT. CrystalNet [220] is an emulation based tool for large production networks such as those observed at AWS.

6.2 Use Case: Program Analysis of the Java Development Kit

Java is a leading programming language used in a variety of applications including, internet programming, smartphone applications, financial systems and many more. Java's popularity and high usage has lead to it being a target to significant security attacks [221] which cover a diverse range of attacks. Typically, a significant portion of a Java applications functionality is contained as part of the *Java library* which applications leverage to reuse core functionality.

While static analysis of Java applications have been readily documented [222], to ensure correctness associated libraries must also be included in the analysis. This however, poses several challenges including, the fact that the OpenJDK consists of a very large codebase, with millions of variables/instructions and hundreds of thousands of methods. Secondly, in a library code base application code is missing, and must be safely over approximated. Moreover, the sheer complexity of the library and the complex semantics of the Java programming language lends itself to ample opportunity to various potential attacks/exploits that are regularly reported and hence must be integrated into the static analyser used to ensure future versions of the OpenJDK are secure.

6.2.1 Points-to Analysis

A fundamental requirement for static analysis of Java code is the consider the memory configuration. Since the heap is the primary structure for global program data, pointer analysis forms the substrate of most inter-procedural static analyses.

While there exist a wide range of such static analyses [223, 7, 224], with their rightful place in the axes of precision and performance/efficiency, for large industrial code bases, techniques such as points-to analysis that exist on the scalable yet less precise portion of the axes are general though of as more appropriate for industrial scale static analysis applications. Points-to analysis limits itself to identifying which objects can point to approximate the set of program objects that a pointer variable or expression can refer to. In contrast to more precise approaches which are

undecidable [223] or intractable [7], points-to analysis has polynomial time complexity and hence can be encoded in Datalog. As a result, the use of points-to analysis is suited to use cases where there is a strong bias for modest performance cost, realistic scalability, and automated whole-program analysis efforts.

Anderson's Points-to. The best-known family of points-to analyses are based on work attributed to Andersen [224]. An Andersen-style analysis can be defined as several subset constraints.

In the simplest form, Anderson style analyses are context insensitive, flow-insensitive, field-insensitive. context-sensitivity refers to the ability of a program analysis to distinguish between analysis results based on different calling contexts. Flow-sensitivity refers to the ability to incorporate program control-flow in the analysis which can result in precision gains. Field-sensitivity refers to the ability of the analysis to distinguish different fields of the same abstract object instead of combining all fields together. Anderson analyses can incorporate the above considerations to improve precision at the cost of performance. Another optional addition to Andersen analyses is on-the-fly call-graph construction. This refers to the property that a points-to analysis also infers simultaneously which methods are called at each call-site. Incorporating call-graph construction typically improves analysis precision.

Context Sensitivity. Points-to analysis is polynomial time decidable, however within this complexity class, there is large manoeuvrability which can impact the practicality of an analysis. Typically, precision can be tuned at the cost of modest performance costs by increasing the context sensitivity of an analysis. Context sensitivity may be perform in several ways: two aspects of context sensitivity is usually distinguished when selecting appropriate heuristics, namely, heap-sensitivity and object-sensitivity. Both these are forms of context-sensitivity in the sense that context refers only to a restriction of analysis results based on some program factor such as call sites. Heap-sensitivity attempts to reduce pollution of the analyses abstract heap by differentiating between memory objects allocated by the same instruction. Object sensitivity is another form of context sensitivity that has emerged

as the dominant flavor of context sensitivity for object-oriented languages such as Java. It is similar to the call site context-sensitivity, though context is derived from a calls receiver object, instead of the call instruction. Using receiver objects can be beneficial, particularly for languages like Java which make heavy use of virtual dispatch.

Through various experimentation a "sweet spot" of 2-object and 1-heap sensitivity (2o2h) has been deemed as a good mix between precision and scalability . Compared to a context-insensitive points-to a 2o2h removes 97false positives in points-to on the OpenJDK and removes 49Soufflé and yet only results in modest performance costs.

Flow Sensitivity. Another consideration in a large scale analysis is flow-sensitivity. This type of analysis is relevant where analysis results encode temporal relationships between results. For simplicity, it is common to develop an analysis which does not track the order of instruction executions, in which case the program semantics are to execute any instruction at any time. Flow-insensitivity is an over-approximation which may conclude, for example, that data is shared between given variables even if one did not hold the data at the time it was copied to the other. A popular way to cheaply provide partial flow-sensitivity is to rely static single-assignment(SSA) conversions. This partial flow sensitivity is assumed in the use case industrial analyses.

6.2.2 Encoding Points-to in Datalog

We describe a Datalog encoding of a flow-insensitive, context-insensitive analysis based on Anderson's analysis.

The Datalog points-to analysis has two relations for computing a points-to analysis, namely, a `vP(v,h)` relation which asserts that a variable `v` may point to a heap object `h` and `hP(`h_1`, f, `h_2`)` which asserts that the field `f` of h_1 may point to h_2.

To define Andersen analysis we encode four constructs into Datalog as follows:

Allocations are modelled by an initial allocation into a points-to relation *vP*. The next rule models the effect of store instructions on the heap. Given a statement

	Java Code	Datalog Encoding
Allocations	`h: v = new C()`	`vP(v,h) :- "h: v = new C()".`
Store	`v1.f=v2`	`hP(h1,f, h2) :- "v1.f=v2", vP(v1, h1), vP(v2, h2).`
Load	`v2=v1.f`	`vP(v2, h2) :- "v2=v1.f", hP(h1,f,h2), vP(v1,h1).`
Moves, Args	$v_2=v_1$	`vP(v2, h) :- "v2=v1", vP(v1,h).`

Table 6.2: *Points-to analysis Datalog Constraints*

v_1.f = v_2, if v_1 can point to h_1 and v_2 can point to h_2, then h_1.f can point to h_2. The next rule resolves load instructions. Given a statement v_2 = v_1.f, if v_1 can point to h_1 and h_1.f can point to h_2, then v_2 can point to h_2. The last rule computes the transitive closure over inclusion edges. If variable v_2 can point to object h and v_1 includes v_2, then v_1 can also point to h.

The first step in a Datalog analysis is to convert the Java source code into input relations. For this simple example we assume four types of language constructs as listed in Table 6.2. Allocation instructions are converted into a input relation `new(var: V, obj: O)` which takes a variable and an object denoted by a instruction location. Stores are converted to an input relation `store(dest: V, field: F, src: V)` which contains a destination variable, a field name and source variable. Loads are converted to an input relation `load(dest: V, src: V, field: F)` which contains a destination variable, a source variable and its field name. Assigns are converted to an input relation `assign(dest: V, src: V)` containing a destination and source variable. For variables, fields and objects we declare respective types and encode the Datalog rules such that they reflect the constraints in Table 6.2. The Datalog analysis is shown in Figure 6.5.

If the code in program in Figure 6.6a is converted into input relations and the analysis in Figure 6.5 is executed the *vP* and *hP* relations compute the points-to graph in Figure 6.6b where the dotted lines denote the *vP* and the full lines denote *hP*.

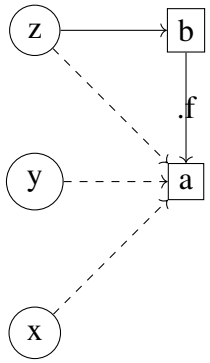
This analysis can be extended by improving the precision of the analysis and by expanding the constructs to include real world language constructs such as Arrays,

```
.type V
.type F
.type O

new(var: V, obj: O)
.input
assign(dest: V, src: V)
.input
load(dest: V, src: V, field: F)
.input
store(dest: V, field: F, src: V)
.input

vP(var: V, heap: O)
.output

hP(base: O, field: F, target: O)
.output

vP(v, h) :- new(v, h).

vP(v1, h) :- assign(v1, v2), vP(v2, h).

vP(v2, h2) :- load(v1, v2, f),
              vP(v1, v2), hP(v2, f, h2).

hP(h1, f, h2) :- store(v1, f, v2),
                 vP(v1, h1), vP(v2, h2).
```

Figure 6.5: *Context-Insensitive, Flow-Insensitive Points-To Analysis*

```
a: x = new Foo();
y = x;
if(cond) {
  x = y;
} else {
  b: z = new G();
  z.f = y;
}
```

(a) *Example Program*



(b) *Points-To Graph*

Reflection etc. A comprehensive study of modelling points-to with Datalog is not in the scope of this thesis and we refer the reader to the tutorial Pointer Analysis [225] by Smaragdakis and Balatsouras.

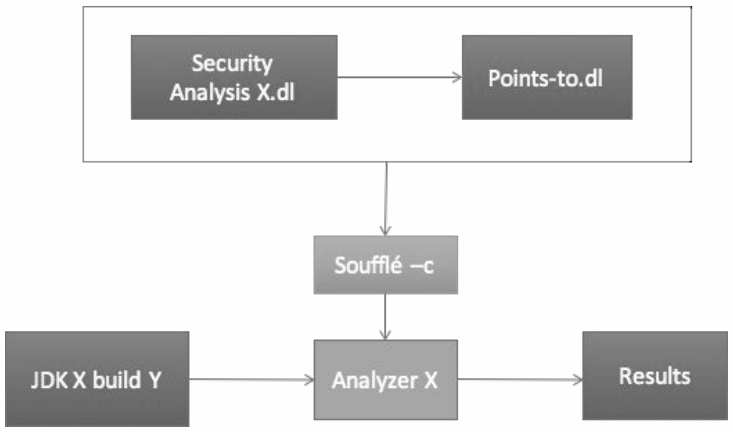
6.2.3 Experiments

6.2.3.1 Experimental Setup

To evaluate SOUFFLÉ's capability of handling large, industrial-scale analysis, we have evaluated the application of a context-insensitive version of the points-to analysis on the entire code base of the OpenJDK library containing 1.4M variables, 350K heap objects, 160K methods, 590K invocations and 17K types. Table 6.3 summarises the performance characteristics of various state-of-the-art tools for Datalog based program analysis for the given problem statement. All of them have been processing the same Datalog program comprising several dozen relations and rules producing $\approx 840M$ resulting tuples when being applied to the OpenJDK7 input set.

For μZ the size of the resulting Datalog query has been too large to obtain results within a reasonable time (DNF = did not finish). Also, the SQL engine based Datalog solver required a significant amount of computation time, rendering it practically unsuitable for the development of real-world, large scale analysis. BDDBDDB could handle the query within much more reasonable time scales. For this real-world benchmark SOUFFLÉ is capable of computing the desired result more than 34x faster then the best state-of-the-art solver – a factor that moves the development of more sophisticated large-scale static programming analysis from the realm of academic exercises into practical reality.

Platform. The evaluations of the BDDBDDB, μZ and SQLite [79] and based analysis have been conducted on a 8 core Intel Xeon E5-2690 v2 @ 3.0GHz, 128GB RAM server system due to resource and licensing constraints, while the SOUFFLÉ experiments have been conducted on a 4 core Intel i7-4790 CPU @ 3.6GHz, 32GB RAM desktop system. However, the huge performance gap between the various approaches are far beyond what can arise from the performance discrepancy of the hardware.



Usage of Soufflé. This use case demonstrates Soufflé's usage as a development framework for static analysis. Security analysis rely on points-to analyses that compute an approximation of the memory configuration in which the potential heap objects that a given reference variable may point to at runtime are discovered. The points to analysis is included into a security analysis using the include directive and instantiated for use in the security analysis. The security analysis is then synthesised into an analyser that can be used to perform the given security analysis on a number of versions of the JDK.

6.2.3.2 Experimental Results

Our last experiment evaluates Soufflé's capability of providing the computational framework exceeding the practical capabilities of state-of-the-art solvers when processing even more sophisticated large-scale analysis. To that end, we have been processing a *context-sensitive* points-to analysis on OpenJDK7 build 147. The context-sensitive analysis is a 2-Object-1-Heap points-to analysis [226] using an open-world abstraction [227]. The evaluation has been conducted on a 8 core Intel Xeon E5-2690 v2 @ 3.0GHz server system. Table 6.4 summarises the obtained results.

Only Soufflé has been able to cope with the large-scale program analysis problem for analysing context-sensitive points-to on the OpenJDK 7. In addition Soufflé was able to compute a security analysis based on [24] in 17 seconds using 2.77 gigabytes of memory for the Java package of the OpenJDK 7 in 14 hours 27 minutes using 75.3 gigabytes of memory for the entire JDK 7.

Tool	Time [hh:mm::ss]	Memory [GB]
bddbddb	0:30:00	5.7
μZ	DNF	DNF
SQLite	6:20:00	40.2
`PA-Datalog`	0:20:00	100
Soufflé (sequential)	0:01:15	7.5
Soufflé (parallel)	0:00:35	8.5

Table 6.3: *Comparison of Datalog evaluation tools for a context-insensitive points-to analysis on the OpenJDK7 library.*

Tool	Time [hh:mm::ss]	Memory [GB]
bddbddb	DNF	DNF
μZ	DNF	DNF
SQLite	DNF	DNF
`PA-Datalog`	15:30:00	450
Soufflé (parallel, 8 cores)	6:44:08	186GB

Table 6.4: *Comparison of Datalog evaluation tools for a context-sensitive points-to analysis on the OpenJDK7 library.*

Comparison to Manual Implementation. Within recent work, the points-to problem over the OpenJDK library has been investigated in detail and a specialised, graph based algorithm for its efficient computation has been devised [50] in Java. In particular, the proposed solution comprises of specialised data structures to effectively represent and compute the points-to relation. The work has the groundbreaking capability of obtaining the analysis results for the OpenJDK library in under a minute. With our Datalog engine the same result on the same dataset can be obtained utilising a general purpose analysis infrastructure within 35s on a commodity desktop system. This result provides an indication on the competitiveness of Soufflé in regards to manually encoded static program analysis.

6.2.4 Discussion

Verifying OpenJDK using other than Datalog has been performed in [50], as explained above with similar times to Soufflé despite using a high specialised handcrafted graph algorithm. At Oracle Labs there is on-going research verifying large code bases such as the OpenJDK using Soufflé, we point the user to the following publications [228, 229].

Chapter 7

Conclusion and Future Work

This thesis has explored the use of several techniques to improve the performance of logic defined static analysis.

In Chapter 3 we introduced a framework that avoids evaluation of a logic language but instead treats it as a logical specification to synthesise a parallel C++ analyser. They key insight into this approach is that through a staged partial evaluation, new optimisations can be performed as the fidelity of the analyser representation increases. The approach has resulted in considerable performance gains compared to state-of-the-art tools and is able to scale to complex analyses on code bases typically deemed to large for logic defined static analyses.

In Chapter 4 we investigated several automatic indexing techniques for use in query engines aimed at large scale computations such as SOUFFLÉ. In particular, we presented an automated indexing approach that results in the minimal sized index sets required to speedup data lookups. The approach presents a balance between speed and memory usage and does not require manual user annotations as is typically needed in other state-of-the-art engines.

In Chapter 5 we investigated symbolic reasoning of horn clause programs as is typically required when non-close world assumptions are present in the analysis. A large body of algorithms that solve recursive Horn clauses symbolically rely on Craig interpolants for improved performance, e.g., CEGAR based predicate abstraction algorithms. In this chapter we identify a significant performance bottleneck: interpolation engines find interpolants agnostic to the larger Horn clause problem. Since the space of interpolants is often large (sometimes infinite) interpolation engine have significant degrees of freedom for choosing an interpolant, resulting in choices of interpolants for the given problem at hand. In Chapter 5 we have proposed a mechanism and theory for guiding interpolation engines to find the right interpolant, based on domain specific knowledge obtained from the problem. This approach has resulted in the ability of algorithms that use interpolation to solve recursive Horn clause problems with faster convergence and therefore speedups.

In Chapter 6 we evaluated SOUFFLÉ on two industrial use cases, namely program analysis of the JDK 7 and security verification of Amazon virtual networks. The

purpose of this evaluation was to demonstrate the scalability of our approach to industrial sized use cases.

7.1 Future Work

The work presented in this thesis has several interesting branches for further investigation. In this section, a few key areas of future work are presented:

7.1.1 Further Language Extensions in Soufflé

In Chapter 3 we presented several non-standard language constructs not present in other logic based languages. Among these are constructors which can be used to create data structures such as lists and trees. It would be interesting to explore extending these constructs to create lattice based analyses, as in logic tools such as Flix [27]. The plan for this joint work with Tamas Szabo and Itemis AG is to link up Soufflé to the IncA [29] tool and investigate both incremental analysis and encoding abstract domains in Soufflé.

7.1.2 Partial Evaluation of Algorithms for constrained Horn Clauses

The interpolation exploration technique of Chapter 5 was implemented in Eldarica. Ideally, solving symbolic Horn clauses can be integrated into the Framework presented in Chapter 3. Since techniques from model checking and constraint logic programming use interpolation for improved performance, the technique of Chapter 5 can be used on both algorithms. Moreover, work relating to partial evaluation of Prolog programs [230] provides a good foundation for this extension.

Another area to explore is combining different algorithms for solving a set of Horn clauses. At times, the precision and efficiency of the Datalog algorithm is useful, however other times symbolic techniques are much more appropriate. While some use cases contain characteristics that make the choice of algorithm obvious, several use cases such as networking could benefit from a hybrid approach where a portion of the problem is solved using Datalog algorithms and another portion by symbolic techniques.

7.1.3 Indexing with R+ Trees

The technique in Chapter 4 imposes syntactical limits on range searches. For example, to support multiple inequalities, we may need to resort to other forms of indexes, e.g., the multi-dimensional index R-tree. Assume the general primitive search $\sigma_{1\leq x\leq 3,2\leq y\leq 4}$. It can be translated into a range search in a multi-dimensional space. However, as discussed in Section 4.2.1, multi-dimensional indexes are generally more expensive to build and also more expensive to query. Therefore, we leave the study of extending our techniques to multiple inequalities, and also to richer variants of Datalog [57], as our future work.

7.1.4 Interaction with Literal Scheduling and Index Selection

As stated in Chapter 4 some Datalog engines such as `Logicblox` version 4 [136] use a leapfrog join that, while requiring users to specify indexes manually, alleviates users from specifying join order. Integrating our technique into such an engine is not obvious as we assume a fixed literal order before our technique is applied. Typically, this can be identified using a profiler like Soufflé profiler, or alternatively, loop schedules can be automated using heuristic techniques [142]. Our technique then can compute the optimal index assignment for the given loop schedule. During performance tuning of large Datalog programs, only a few rules require manual loop scheduling. Therefore, our preference is to fix loop orders rather than indexes for a better user experience. Soufflé's auto-scheduler typically resolves this automatically for the user. Nevertheless, there is ongoing work to integrate automatic loop scheduling and automatic index selection.

Just as the Soufflé framework enables automatic and static index selection (as shown in Chapter 4), a similar benefit can be performed for literal scheduling. Due to the fact that Soufflé synthesises rules, at the RAM level by using cardinality approximations [143], we can derive, not one but several candidate literal schedules and synthesise them all. Since cardinality estimations have been show to produce significant error [144], giving a runtime option to apply several schedules with real cardinality data (obviously available at runtime) can reduce this error. Furthermore, since there is a dependence with the literal schedule and the minimal index sets size,

we can incorporate our minimal index selection algorithm as an additional cost metric for choosing a schedule. This is ongoing work, however not developed enough for inclusion as a chapter in this thesis. The implementation requires, cardinality guards to integrated into RAM, and modifications to the scheduling framework in SOUFFLÉ.

7.1.5 Bottom-Up Guided, Top-Down Evaluation

We use Lemma 1 for a basis for additional work for developing an approach where we annotate the domain of the Datalog computation to help guide Top-down performance to discover minimal height proof derivations. For example, current work with David Zhao and Bernhard Scholz proposes a data provenance technique using this technique. However, this approach may be extended to a general evaluation approach.

7.1.6 Synthesis, Abduction and Provenance

It would be beneficial to synthesise EDB data from a set of Datalog. The approach in [219] proposes encoded Datalog and the Clark's completion [231] into an SMT solver to perform the synthesis. This approach, has limited scalability. It would be interesting to investigate alternative approaches to this. This is very related to Abductive Logic Programming, which requires significant increase in Datalog semantic expressiveness.

Bibliography

[1] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Operational semantics for declarative networking. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings*, pages 76–90, 2009.

[2] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[3] Ryan Chuang, Benjamin A. Hall, David Benque, Byron Cook, Samin Ishtiaq, Nir Piterman, Alex Taylor, Moshe Vardi, Steffen Koschmieder, Berthold Gottgens, and Jasmin Fisher. Drug target optimization in chronic myeloid leukemia using innovative computational platform. February 2015.

[4] S. C. Johnson. Lint, a c program checker. In *COMP. SCI. TECH. REP*, pages 78–1273, 1978.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[6] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Collo-*

quium on Automata, Languages and Programming, pages 169–181, London, UK, UK, 1980. Springer-Verlag.

[7] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, May 2002.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.

[9] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.

[10] Fausto Spoto. Precise null-pointer analysis. *Softw. Syst. Model.*, 10(2):219–252, May 2011.

[11] Colin Barker. The top 10 it disasters of all time. https://www.zdnet.com/article/the-top-10-it-disasters-of-all-time/, 2007.

[12] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[13] Food and Drug Administration. Infusion Pump Software Safety Research at FDA. `https://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm`, 2010. Online; accessed 30/4/18.

[14] Office Nuclear Regulation. Computer based safety systems - technical guidance for assessing software aspects of digital computer based protection systems. `http://webarchive.nationalarchives.gov.uk/`

20130104193206/http://www.hse.gov.uk/nuclear/operational/tech_asst_guides/tast046.pdf, 2008. Online; accessed 29/4/18.

[15] Checkmarx. Checkmarx. https://www.checkmarx.com/, 2018. Online; accessed 29/4/18.

[16] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[17] Microfocus. Fortify. https://www.microfocus.com/documentation/fortify-static-code/1720/, 2018. Online; accessed 29/4/18.

[18] Facebook. Infer. https://github.com/facebook/infer, 2018. Online; accessed 29/4/18.

[19] PMD. PMD. https://pmd.github.io/pmd/pmd_projectdocs_trivia_news.html, 2018. Online; accessed 29/4/18.

[20] Evgeny Lebanidze. The Need for Fourth Generation Static Analysis Tools for Security From Bugs to Flaws. https://www.owasp.org/images/4/41/OWASP-AppSecEU08-Lebanidze.pdf, 2008. Online; accessed 29/4/18.

[21] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[22] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of javascript web applications in the wild. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 61–70, New York, NY, USA, 2016. ACM.

[23] Sam Weber, Paul A. Karger, and Amit Paradkar. A software flaw taxonomy: Aiming tools at security. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[24] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: a class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, pages 7–12, 2015.

[25] Cristina Cifuentes, Nathan Keynes, Lian Li, Nathan Hawes, Manuel Valdiviezo, Andrew Browne, Jacob Zimmermann, Andrew Craik, Douglas Teoh, and Christian Hoermann. Static deep error checking in large system applications using parfait. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 432–435, 2011.

[26] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[27] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. *SIGPLAN Not.*, 51(6):194–208, June 2016.

[28] Yannis Smaragdaiks, Martin Bravenboer, and George Kastrinis. Doop: A framework for java pointer analysis. `http://doop.program-analysis.org/`.

[29] T. Szab, S. Erdweg, and M. Voelter. Inca: A dsl for the definition of incremental program analyses. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 320–331, Sept 2016.

[30] Todd J. Green, Molham Aref, and Grigoris Karvounarakis. LogicBlox, platform and language: A tutorial. In *Datalog in Academia and Industry - Second*

International Workshop, volume 7494 of *Lecture Notes in Computer Science*, pages 1–8. Springer, September 2012.

[31] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 422–430. Springer, 2016.

[32] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 462–473. ACM, 2015.

[33] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, pages 24–51, 2015.

[34] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 245–251, Berlin, Heidelberg, 2011. Springer-Verlag.

[35] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. *SIGPLAN Not.*, 49(1):113–126, January 2014.

[36] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144. ACM, June 2004.

[37] Krystof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. *Z*- an efficient engine for fixed points with constraints. In *Proceedings of the International Conference on Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 457–462. Springer, July 2011.

[38] Thomas W. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the 5th International Conference on Compiler Construction*, CC '94, pages 389–403, London, UK, UK, 1994. Springer-Verlag.

[39] Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems – a case study. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 117–126, New York, NY, USA, 1996. ACM.

[40] Bernhard Scholz, Herbert Jordan, Pavle Subotic, and Till Westmann. On fast large-scale program analysis in datalog. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 196–206. ACM, 2016.

[41] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 97–118, Berlin, Heidelberg, 2005. Springer-Verlag.

[42] Damien Sereni, Pavel Avgustinov, and Oege de Moor. Adding magic to an optimising datalog compiler. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 553–566, 2008.

[43] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *Proceedings of the 2015*

ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pages 1371–1382, New York, NY, USA, 2015. ACM.

[44] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[45] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013.

[46] William Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(3), 1957.

[47] Aws Albarghouthi and Kenneth L. McMillan. Beautiful interpolants. In *CAV*, pages 313–329, 2013.

[48] Vijay D'Silva, Daniel Kroening, Mitra Purandare, and Georg Weissenbacher. Interpolant strength. In *VMCAI*, pages 129–145, 2010.

[49] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, 2012.

[50] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 535–551, New York, NY, USA, 2015. ACM.

[51] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Two concurrent data structures for efficient datalog query processing. In *Proceedings of the 23rd*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018, pages 399–400, 2018.

[52] Pavle Subotic, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *PVLDB*, 12(2):141–153, 2018.

[53] Philipp Rümmer and Pavle Subotic. Exploring interpolants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 69–76, 2013.

[54] Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. Guiding craig interpolation with domain-specific abstractions. *Acta Inf.*, 53(4):387–424, 2016.

[55] Jeffrey D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 140–149, New York, NY, USA, 1989. ACM.

[56] Raghu Ramakrishnan and S. Sudarshan. Top-down versus bottom-up revisited. In *Proceedings of the International Symposium on Logic Programming*, pages 321–336. MIT Press, October 1991.

[57] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 58–73, London, UK, UK, 2003. Springer-Verlag.

[58] P.G. Kolaitis and M.Y. Vardi. On the expressive power of datalog: Tools and a case study. *Journal of Computer and System Sciences*, 51(1):110 – 134, 1995.

[59] Kave Eshghi and Robert A. Kowalski. Abduction compared with negation by failure. In *Logic Programming, Proceedings of the Sixth International Conference, Lisbon, Portugal, June 19-23, 1989*, pages 234–254, 1989.

[60] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the coral deductive database system. *SIGMOD Rec.*, 22(2):167–176, June 1993.

[61] Günther Specht. LOLA, LDL, NAIL!, RDL, Aditi and Starburst: A comparison of deductive database systems, 1991.

[62] Burkhard Freitag, Heribert Schütz, and Günther Specht. LOLA - a logic language for deductive databases and its implementation. In *Proceedings of the International Symposium on Database Systems for Advanced Applications*, volume 2 of *Advanced Database Research and Development Series*, pages 216–225. World Scientific, April 1991.

[63] Joseph M. Hellerstein and Michael Stonebraker. *Readings in Database Systems: Fourth Edition*. The MIT Press, 2005.

[64] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5(2):105–195, 2013.

[65] Maurizio Lenzerini. Data integration: A theoretical perspective, 2002.

[66] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, May 2005.

[67] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization, 2006.

[68] William R. Marczak, Shan Shan Huang, Martin Bravenboer, Micah Sherr, Boon Thau Loo, and Molham Aref. Secureblox: Customizable secure distributed data processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 723–734, New York, NY, USA, 2010. ACM.

[69] Peter C. Chapin, Christian Skalka, and X. Sean Wang. Authorization in trust management: Features and foundations. *ACM Comput. Surv.*, 40(3):9:1–9:48, August 2008.

[70] Herbert Jordan, Bernhard Scholz, Pavle Subotic, and Till Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 196–206, 2016.

[71] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris De Volder. Codequest: querying source code with datalog. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 102–103, 2005.

[72] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 499–512. USENIX Association, 2015.

[73] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 469–483, Berkeley, CA, USA, 2015. USENIX Association.

[74] William C. Benton and Charles N. Fischer. Interactive, scalable, declarative program analysis: From prototype to implementation. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '07, pages 13–24, New York, NY, USA, 2007. ACM.

[75] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: a class of access control vulnerabilities in the java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, pages 7–12, 2015.

[76] Peter Z. Revesz. The constraint database approach to software verification. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, pages 329–345, 2007.

[77] Mathieu Verbaere, Elnar Hajiyev, and Oege de Moor. Improve software quality with semmlecode: an eclipse plugin for semantic code search. In *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 880–881, 2007.

[78] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 2:1–2:25, 2016.

[79] B.Scholz, K.Vorobyov, P.Krishnan, and T.Westmann. A datalog source-to-source translator for static program analysis: An experience report. *Australasian Software Engineering Conference*, 2015.

[80] Ulrich Zukowski and Burkhard Freitag. The deductive database system lola. In *International Conference on Logic Programming and Nonmonotonic Rea-*

soning, volume 1265 of *Lecture Notes in Computer Science*, pages 376–387. Springer, July 1997.

[81] Werner Kießling, Helmut Schmidt, Werner Strauß, and Gerhard Dünzinger. DECLARE and SDS: Early efforts to commercialize deductive database technology. *Very Large Data Bases*, 3(2):211–243, 1994.

[82] Filippo Cacace, Stefano Ceri, Stefano Crespi-Reghizzi, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. The LOGRES prototype. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 550–551. ACM Press, May 1993.

[83] Barry Bishop and Florian Fischer. IRIS - integrated rule inference system. In *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense*. CEUR-WS.org, June 2008.

[84] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive datalog system DLV. In *Datalog Reloaded - First International Workshop*, volume 6702 of *Lecture Notes in Computer Science*, pages 282–301. Springer, March 2010.

[85] Fernando Sáenz-Pérez. Outer joins in a deductive database system. *Electronic Notes in Theoretical Computer Science*, 282:73–88, 2012.

[86] Jiwon Seo, Stephen Guo, and Monica S. Lam. Socialite: An efficient graph query language based on datalog. *IEEE Trans. Knowl. Data Eng.*, 27(7):1824–1837, 2015.

[87] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1135–1149, New York, NY, USA, 2016. ACM.

[88] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.

[89] C. A. Martínez-Angeles, I. Dutra, V. S. Costa, and J Buenabad-Chávez. A datalog engine for GPUs. In *Declarative Programming and Knowledge Management*, LNCS 8439, pages 152–168. Springer, 2014.

[90] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[91] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[92] Konstantinos Sagonas, Terrance Swift, and David Scott Warren. XSB as an efficient deductive database engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.*, pages 442–453, 1994.

[93] Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhao. Datalog and recursive query processing. *Foundations and Trends in Databases*, 5:106–187, 2012.

[94] Stefan Brass. Implementation alternatives for bottom-up evaluation. In *ICLP (Technical Communications)*, volume 7 of *LIPIcs*, pages 44–53. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[96] Neil D. Jones. The essence of program transformation by partial evaluation and driving. In Dines Bjøner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics*, pages 62–79, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[97] S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[98] Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[99] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications, First International Conference, RTA-85, Dijon, France, May 20-22, 1985, Proceedings*, pages 124–140, 1985.

[100] Michael Sperber and Peter Thiemann. Generation of lr parsers by partial evaluation. *ACM Trans. Program. Lang. Syst.*, 22:224–264, 2000.

[101] Michael Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, 26(3):413–463, May 2004.

[102] Jacob M. Howe and Andy King. Specialising fininte domain programs using polyhedra. In *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR'99, Venezia, Italy, September 22-24, 1999, Selected Papers*, pages 118–135, 1999.

[103] Ranjeet Singh and Andy King. Partial evaluation for java malware detection. In *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, pages 133–147, 2014.

[104] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.

[105] William R. Cook and Ralf Lämmel. Tutorial on online partial evaluation. In *Proceedings IFIP Working Conference on Domain-Specific Languages, DSL 2011, Bordeaux, France, 6-8th September 2011.*, pages 168–180, 2011.

[106] A. Razavi and K. Kontogiannis. Partial evaluation of model transformations. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 562–572, June 2012.

[107] Stefan Brass and Heike Stephan. A variant of earley deduction with partial evaluation. In Wolfgang Faber and Domenico Lembo, editors, *Web Reasoning and Rule Systems*, pages 35–49, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[108] Yoshihiko Futamura. Partial evaluation of computation process. *Systems, Computers, Controls*, 12(4):377–380, December 1971.

[109] Pär Emanuelson and Anders Haraldsson. On compiling embedded languages in lisp. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP '80, pages 208–215, New York, NY, USA, 1980. ACM.

[110] Arun Lakhotia and Leon Sterling. How to control unfolding when specializing interpreters. *New Generation Computing*, 8(1):61–70, Jun 1990.

[111] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[112] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[113] Andrea Blass and Yuri Gurevich. Existential fixed-point logic. pages 20–36, August 1987.

[114] Edmund Melson Clarke, Jr. Programming language constructs for which it is impossible to obtain good hoare-like axiom systems. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 10–20, New York, NY, USA, 1977. ACM.

[115] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings, 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, New York, NY, USA, 1987. ACM.

[116] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. TRACER: A symbolic execution tool for verification. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 758–766, 2012.

[117] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 157–171, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[118] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 17–34, Cham, 2014. Springer International Publishing.

[119] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[120] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *CAV*, 2013.

[121] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[122] Gustavo Pelaitay and Aldo Figallo. Lukasiewicz implication prealgebras. *Annals of the University of Craiova*, 44, 07 2017.

[123] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[124] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. *J. Symb. Comput.*, 45(11):1212–1233, 2010.

[125] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.

[126] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. On recursion-free horn clauses and craig interpolation. *Form. Methods Syst. Des.*, 47(1):1–25, August 2015.

[127] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.

[128] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[129] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *31st POPL*, 2004.

[130] Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu. An interpolation method for clp traversal. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 454–469, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[131] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In *LPAR*, 2012.

[132] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.

[133] Bernhard Scholz and Pavle Subotić. Souffle tutorial. `https://tinyurl.com/y99mr6tl`.

[134] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM.

[135] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. Graph queries in a next-generation datalog system. *Proc. VLDB Endow.*, 6(12):1258–1261, August 2013.

[136] Inc. LogicBlox. Declartive cloud platform for applications that combine transactions & analytics. `http://www.logicblox.com`.

[137] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[138] Todd L. Veldhuizen. C++ templates as partial evaluation. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, USA, January 22-23, 1999. Technical report BRICS-NS-99-1*, pages 13–18, 1999.

[139] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[140] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. Understanding caller-sensitive method vulnerabilities: a class of access control vulnerabilities in the java platform. In Møller and Naik [232], pages 7–12.

[141] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[142] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference*

on Management of Data, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[143] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *PVLDB*, 11(4):499–512, 2017.

[144] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015.

[145] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Two concurrent data structures for efficient datalog query processing. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 399–400, 2018.

[146] Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu. Optimizing large-scale semi-naïve datalog evaluation in hadoop. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, pages 165–176, Berlin, Heidelberg, 2012. Springer-Verlag.

[147] G. Hulin. Parallel processing of recursive queries in distributed architectures. In *Proceedings of the 15th International Conference on Very Large Data Bases*, VLDB '89, pages 87–96, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[148] S. Cohen and O. Wolfson. Why a single parallelization strategy is not enough in knowledge bases. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, pages 200–216, New York, NY, USA, 1989. ACM.

[149] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. A framework for the parallel processing of datalog queries. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 143–152, New York, NY, USA, 1990. ACM.

[150] Jürgen Seib and Georg Lausen. Parallelizing datalog programs by generalized pivoting. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '91, pages 241–251, New York, NY, USA, 1991. ACM.

[151] Ouri Wolfson and Avi Silberschatz. Distributed processing of logic programs. *SIGMOD Rec.*, 17(3):329–336, June 1988.

[152] Ouri Wolfson and Aya Ozeri. A new paradigm for parallel and distributed rule-processing. *SIGMOD Rec.*, 19(2):133–142, May 1990.

[153] Sajindra Jayasena and Sharad Ganesh. Conversion of NSP DAGs to SP DAGs, 2003.

[154] Arturo Gonzalez Escribano, Arjan J.C. van Gemund, and Facultad De Ciencias. An algorithm for transforming NSP to SP graphs, 1996.

[155] Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson. Parallel algorithms for the execution of relational database operations. *ACM Trans. Database Syst.*, 8(3):324–353, September 1983.

[156] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[157] Stefan Brass and Heike Stephan. Bottom-up evaluation of datalog: Preliminary report. In *WLP / WFLP*, 2017.

[158] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental partial evaluation: The key to high performance, modularity and portability in operating systems. In *PEPM*, 1993.

[159] Madhubanti Mukherjee and Ranga Vemuri. A novel synthesis strategy driven by partial evaluation based circuit reduction for application specific dsp circuits. In *Proceedings 21st International Conference on Computer Design*, pages 435–440, Oct 2003.

[160] Christoph Kreitz. Program synthesis. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III, chapter III.2.5, pages 105–134. Kluwer, 1998.

[161] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer, editors, *Approaches and Applications of Inductive Programming*, volume 5812 of *Lecture Notes in Computer Science*, pages 50–73. Springer Berlin Heidelberg, 2010.

[162] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 405–416, New York, NY, USA, 2012. ACM.

[163] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. Horn clauses for communicating timed systems. In *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014.*, pages 39–52, 2014.

[164] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In *Computer Aided Verification (CAV '97)*. Springer-Verlag, 1997.

[165] Stefan Brass and Heike Stephan. A variant of earley deduction with partial evaluation. In Wolfgang Faber and Domenico Lembo, editors, *Web Reasoning and Rule Systems*, volume 7994 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2013.

[166] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. ACM.

[167] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 396–407, New York, NY, USA, 2014. ACM.

[168] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98. Prentice-Hall, 1972.

[169] Thanh Truong. *Main-Memory Query Processing Utilizing External Indexes*. PhD thesis, Uppsala University, Sweden, 2016.

[170] Vítor Santos Costa, Konstantinos Sagonas, and Ricardo Lopes. Demand-driven indexing of prolog clauses. In *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, pages 395–409, 2007.

[171] Douglas Comer. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3(4):440–445, December 1978.

[172] M.Y.L. Ip, L.V. Saxton, and V.V. Raghavan. On the selection of an optimal set of indexes. *Software Engineering, IEEE Transactions on*, SE-9(2):135–143, March 1983.

[173] Jozef Kratica, Ivana Ljubic, and Dušan Tošic. A genetic algorithm for the index selection problem. In *Proceedings of the 2003 International Conference on Applications of Evolutionary Computing*, EvoWorkshops'03, pages 280–290, Berlin, Heidelberg, 2003. Springer-Verlag.

[174] Mario Schkolnick. The optimal selection of secondary indices for files. *Information Systems*, 1(4):141 – 146, 1975.

[175] Gregory Piatetsky-Shapiro. The Optimal Selection of Secondary Indices is NP-complete. *SIGMOD Rec.*, 13(2):72–75, January 1983.

[176] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. pages 146–155, Athens, Greece, 1997. Morgan Kaufmann.

[177] Emmanuel Bruno, Nicolas Faessel, Hervé Glotin, Jacques Le Maitre, and Michel Scholl. Indexing and querying segmented web pages: the blockweb model. *World Wide Web*, 14(5-6):623–649, 2011.

[178] Microsoft. Clustered and nonclustered indexes described. `TinyURL.com/yawm9ds9`.

[179] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Ann. Math. (2)*, 51:161–166, 1950.

[180] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 535–551. ACM, 2015.

[181] D. R. Fulkerson. Note on dilworth's decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4):pp. 701–702, 1956.

[182] Wim Pijls and Rob Potharst. Another note on dilworth's decomposition theorem. *Journal of Discrete Mathematics*, 2013:4, 2013.

[183] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[184] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[185] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In Patrick Dewilde and Joos Vandewalle, editors, *Computer Systems and Software Engineering*, pages 287–324. Springer US, 1992.

[186] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[187] Kotagiri Ramamohanarao and James Harland. An introduction to deductive database languages and systems. *Journal of Very Large Data Bases*, 3(2):107–122, 1994.

[188] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473, 2006.

[189] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *FM*, pages 247–251, 2012.

[190] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. An interpolating sequent calculus for quantifier-free Presburger arithmetic. In *Proceedings, IJCAR*, LNCS. Springer, 2010.

[191] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[192] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In *VMCAI*, LNCS. Springer, 2011.

[193] Joao Marques-Silva, Mikolás Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In *CAV*, pages 592–607, 2013.

[194] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[195] Laurent Fribourg. Petri nets, flat languages and linear arithmetic. In María Alpuente, editor, *Proc. of WFLP'2000*, pages 344–365, 2000.

[196] Hossein Hojjat, Radu Iosif, Filip Konecný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *ATVA*, pages 187–202, 2012.

[197] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, pages 157–171, 2012.

[198] Sbastien Bardin, Alain Finkel, Jrme Leroux, and Laure Petrucci. Fast: acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(5):401–424, 2008.

[199] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, pages 413–427, 2008.

[200] Nishant Totla and Thomas Wies. Complete instantiation-based interpolation. In *POPL*, pages 537–548, 2013.

[201] Simone Fulvio Rollini, Ondrej Sery, and Natasha Sharygina. Leveraging interpolant strength in model checking. In *CAV*, pages 193–209, 2012.

[202] Simone Rollini, Roberto Bruttomesso, and Natasha Sharygina. An efficient and flexible approach to resolution proof reduction. In *HVC*, pages 182–196, 2010.

[203] Krystof Hoder, Laura Kovács, and Andrei Voronkov. Playing in the grey area of proofs. In *POPL*, pages 259–272, 2012.

[204] Nicolas Caniart, Emmanuel Fleury, Jérôme Leroux, and Marc Zeitoun. Accelerating interpolation-based model-checking. In *TACAS*, pages 428–442, 2008.

[205] Mohamed Nassim Seghir. A lightweight approach for loop summarization. In *ATVA*, pages 351–365, 2011.

[206] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *FME*, pages 500–517, 2001.

[207] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*. Springer, 2007.

[208] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.

[209] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *Proceedings, VMCAI*, volume 4349 of *LNCS*, pages 346–362. Springer, 2007.

[210] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. CSIsat: Interpolation for LA+EUF. In *CAV*, volume 5123 of *LNCS*, pages 304–308. Springer, 2008.

[211] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[212] Amazon Virtual Private Cloud. `https://aws.amazon.com/vpc/`. Accessed: March 2018.

[213] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. *Microsoft Research*, pages 1–11, 2014.

[214] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in microsoft azure. In *Distributed Computing and Internet Technology - 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings*, pages 21–32, 2015.

[215] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: towards verifying controller programs in software-defined networks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, UK — June 9–11, 2014*, pages 282–293, 2014.

[216] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and Philip Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 15–27, 2013.

[217] Ehab Al-Shaer, Wilfredo Marrero, Adel El-Atawy, and Khalid Elbadawi. Network configuration in A box: Towards end-to-end verification of network reachability and security. In *Proceedings of the 17th annual IEEE International Conference on Network Protocols, 2009. ICNP 2009, Princeton, NJ, USA, 13-16 October 2009*, pages 123–132, 2009.

[218] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*, pages 290–301, 2011.

[219] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 261–281, 2017.

[220] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 599–613, 2017.

[221] Daniel Somerfield. *Professional Java Security*. Wrox Press Ltd., Birmingham, UK, UK, 2001.

[222] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[223] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[224] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, 1994.

[225] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015.

[226] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the ACM SIGPLAN 2013 conference on Programming language design and implementation*, PLDI '13, New York, NY, USA, 2013. ACM.

[227] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 13–18, New York, NY, USA, 2015. ACM.

[228] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In *SOAP@PLDI*, pages 13–18. ACM, 2017.

[229] Yi Lu, Sora Bae, Padmanabhan Krishnan, and K. R. Raghavendra. Inference of security-sensitive entities in libraries. In *IEEE Symposium on Security and Privacy Workshops*, pages 102–109. IEEE Computer Society, 2017.

[230] Stephen-John Craig. *Practicable Prolog specialisation*. PhD thesis, University of Southampton, UK, 2005.

[231] Alex Simpson. Lecture notes on clarks completion. `http://www.inf.ed.ac.uk/teaching/courses/lp/2012/slides/lpTheory8.pdf`.

[232] Anders Møller and Mayur Naik, editors. *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*. ACM, 2015.