EXTENDING PARALLEL DATALOG WITH LATTICE

A Thesis in
Computer Science and Engineering
by
Qing Gong

© 2020 Qing Gong

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2020
The thesis of Qing Gong was reviewed and approved by the following:

Gang Tan  
James F. Will Career Development Associate Professor of EECS  
Thesis Advisor

Danfeng Zhang  
Assistant Professor of EECS

Chitaranjan Das  
Department Head of Computer Science and Engineering  
Department Head and Distinguished Professor
Abstract

There have been more and more studies on Datalog, a logic programming language which is originally a query language for deductive databases. The application of Datalog in static program analysis has been noticed and explored in the last two decades. Datalog provides a general way to implement static analysis using Semi-Naïve Evaluation. We can use facts and rules to represent a specific static analysis and send them to a Datalog evaluator to solve it, without manually writing the static analyzer and corresponding debugging work. However, some static analyses with large or infinite lattices cannot be efficiently solved in the traditional powerset scheme. To overcome this drawback, we need the natural lattice scheme in such static analysis to extend the range of application of Datalog, more specifically, Soufflé.

This thesis discusses some backgrounds including the history and evaluation of Datalog, static analysis and lattice, the architecture of two Datalog variants: Soufflé and FLIX. We describe the framework of lattice including conditional operator, unary and binary case functions, lattice declaration, lattice association, and the modifications in the Semi-Naïve Evaluation algorithms in RAM program. The details of extending Soufflé with the designed lattice framework are described. The implementation has been put on Github: https://github.com/QXG2/souffle. We evaluate the functionality and scalability of the lattice scheme on extending Soufflé, and the results show that extended Soufflé is very efficient on Sign Analysis and Constant Propagation Analysis.
# Table of Contents

List of Figures vii

List of Tables viii

Acknowledgments x

Chapter 1
   Introduction 1

Chapter 2
   Background 3
     2.1 Datalog Language ........................................... 3
     2.1.1 History .................................................... 3
     2.1.2 Evaluation .................................................. 5
     2.1.3 Top-Down Evaluation ........................................ 6
     2.1.4 Bottom-Up Evaluation ....................................... 10
     2.1.4.1 Magic Sets ............................................... 11
     2.2 Static Analysis and Lattice ................................... 12
     2.2.1 Static Analysis ............................................ 12
     2.2.2 Lattice ..................................................... 14
     2.3 Soufflé Programming Language ................................. 15
     2.4 Flix Programming Language ................................... 21

Chapter 3
   Framework for Lattice 23
     3.1 Minimal Model considering Lattice .......................... 23
     3.2 Conditional Operator .......................................... 24
     3.3 Unary and Binary Case Function ............................... 24
     3.4 Lattice Declaration ........................................... 26
     3.4.1 Enum Type .................................................. 26
     3.4.2 Lattice Association ......................................... 27
     3.4.3 Lattice Relation ........................................... 27
     3.5 Modification in RAM .......................................... 28
Appendix E

Raw Data in Experiments 95
E.1 Sign Analysis without Branches ................................. 95
E.2 Sign Analysis with Branches ..................................... 98
E.3 Constant Propagation Analysis without Branches ............. 101
E.4 Constant Propagation Analysis with Branches ................ 104

Bibliography 107
## List of Figures

2.1 Powerset Lattice for Live Variables analysis ........................................... 15
2.2 Reverse Powerset Lattice for Available Expression analysis ......................... 15
2.3 Soufflé architecture ....................................................................................... 16
2.4 Precedence and SCC Graph for the example program ...................................... 18
3.1 Simple lattice ................................................................................................. 24
4.1 Class Hierarchy of Updated AstNode ............................................................... 32
4.2 Class Hierarchy of Updated RamNode ............................................................. 34
5.1 Lattice of Sign Analysis .................................................................................. 38
5.2 Lattice of Constant Propagation Analysis ....................................................... 40
5.3 Performances on Sign Analysis without Branches ........................................... 49
5.4 Performances on Sign Analysis with Branches ................................................ 49
5.5 Performances on Constant Propagation Analysis without Branches .............. 51
5.6 Performances on Constant Propagation Analysis with Branches ................. 52
List of Tables

2.1 Data-Flow Analysis ............................................................. 13

5.1 Evaluation of Operators in Sign Analysis ................................. 38
5.2 Evaluation of Operators in Constant Propagation Analysis ........... 41

E.1 Sign Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé ................................. 95
E.2 Sign Analysis without Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé ......................................................... 96
E.3 Sign Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX ............................................................. 97
E.4 Sign Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé ................................................. 98
E.5 Sign Analysis with Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé ........................................................... 99
E.6 Sign Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX ............................................................. 100
E.7 Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé ................. 101
E.8 Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé ............................ 102
E.9 Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX ......................................... 103
E.10 Constant Propagation Analysis with Branches: Runtime(s) of Tests for
the Lattice Scheme in Extended Soufflé . . . . . . . . . . . . . . . . . . . . 104

E.11 Constant Propagation Analysis with Branches: Runtime(s) of Tests for
the Powerset Scheme in Soufflé . . . . . . . . . . . . . . . . . . . . . . . . 105

E.12 Constant Propagation Analysis with Branches: Runtime(s) of Tests for
the Lattice Scheme in FLIX . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
Acknowledgments

I would like to thank everyone that helped me in the completion of this thesis. I greatly appreciate all of my family members, friends, advisors who guided me throughout this process. There are a lot of people I want to thank in this long journey since 2013, when my first master’s program in underground structure and tunneling began.

First of all, I would like to thank my family, especially my parents for all of their support: Mr. Qiuming Gong and Mrs. Yi Zhang. Without all of the time and effort that they put into raising and caring for me, especially the tuition and life expenses during my master’s program in computer science, I would not be here graduating and living the dream. It is not a small amount of money for my parents, I would never forget their trust and generosity.

I would like to thank my wonderful advisor Professor Gang Tan for all of his guidance and mentoring in this thesis. He is wise and insightful. I got a lot of inspiration from the invaluable advice from him over the thesis. His suggestions are always very substantial and helpful.

Professor Patrick Fox, Department Head of Civil and Environmental Engineering, is my previous advisor in CEE department. A huge thanks to him for the guidance of my two years’ experience in the Ph.D. program in CEE. He is always patient and encouraging. I won’t forget what he said to me: “Don’t say ‘never’."

My friends at Penn State have always been supportive and thoughtful in my life. I’d like to thank them for being in my life: Chu Wang, Yanan Li, Saharnaz Nazari, William Wang, Kun Zeng, Yawei Gao, Hao Wang. Special thanks to my friend Chu Wang, for all of his help when I first moved to State College.

My advisors at Tongji University also cares for my development: Professor Hehua Zhu and Associate Professor Xiaoying Zhuang. Their help is essential for my application to the Ph.D. program in CEE at Penn State.

I also want to thank my friend Jiaming He at Tongji University. His success in switching to computer science encouraged me to change my major and pursuing what I really like.

Last but not least, I would like to thank my girlfriend, Xixi(Hao) Fan. She has been supportive when I was looking for a job and writing this thesis. Thanks to her accompany and caring.
Chapter 1  |  Introduction

The rise of popularity of programming languages which are variants of Datalog is accompanied by the growing research on static analysis on programs. There exist quite a few static analyses with different tasks such as optimization in compilers, program soundness, and security checking. Traditionally, such static analyzer is implemented manually for a specific programming language, and can hardly be reused in other environments. Another issue is that the debugging of such analyzers is usually difficult since there could be a lot of test cases considering the combination and sequence of codes in the program to be analyzed. Datalog provides a general way to implement a static analyzer with given relations and rules.

A Datalog program is a set of facts and rules, based on relations or predicates. During the evaluation of a Datalog program, the rules in it will generate more facts for relations until the evaluator reaches a fixpoint or to say, minimal model. In nature, Datalog treats each fact as a single tuple, and it generates more tuples during the evaluation. Since every static analysis can be described via lattice in mathematical theory, the traditional way of conducting static analysis in Datalog can be called powerset scheme. This scheme works well for some static analyses such as the classical Reaching Definition analysis. However, for some large or even infinite lattice like Constant Propagation analysis, the powerset scheme cannot solve it efficiently. We need the natural lattice scheme for such static analysis, which is the motivation of this thesis.

The major contribution of this thesis is the implementation of lattice scheme in Soufflé, an open-source parallel variant of Datalog language which supports both interpreter mode and compiler mode. We implemented the lattice scheme for the interpreter mode in Soufflé. It has been put on Github: https://github.com/QXG2/souffle. The implementation involves general features that can be used anywhere in Soufflé programs, including conditional operator, and unary and binary case functions. The implementation
also includes features designed for lattice scheme including lattice declaration and lattice association, and the modifications in the Semi-Naïve Evaluation algorithms in RAM program. Each aspect of the implementation is evaluated, including the scalability of lattice scheme on extended Soufflé.

The thesis is structured as follows: Chapter 2 describes the background related to the thesis, including the history and evaluation of Datalog, static analysis and lattice, Soufflé and FLIX. Chapter 3 describes the framework of the implementation. Chapter 4 discusses the details of the implementation. Chapter 5 presents the evaluation of the implementation. Chapter 6 concludes and discusses future work.
Chapter 2  
Background

This chapter presents background on Datalog and two variants of Datalog: Soufflé and Flix. The importance and application of lattice concept in static analysis are discussed, which is the motivation of this thesis.

2.1 Datalog Language

2.1.1 History

Datalog is a declarative logic programming language mainly used as an information extraction tool for databases. Datalog was first invented as a simplified Horn logic language akin to Prolog and then used on deductive databases and recursive query processing [1]. Datalog is syntactically a subset of Prolog. They are both declarative programming languages and use facts and rules as the basic elements. There are several differences between them. The pure Datalog imposes certain stratification restrictions on the use of negation and recursion. Also, Datalog requires that every variable that appears in the head of a horn clause also appears in a non-negative literal in the body. Another requirement is that every variable appearing in a negative literal in the body of a rule, also appears in a positive literal in the body [2]. With these restrictions, Datalog queries on finite sets are guaranteed to terminate, which makes Datalog get ride of Prolog’s cut operator and makes Datalog a pure and fully declarative language. Furthermore, since Datalog is a syntactic subset of Prolog and the semantics are the same, one could create and execute Datalog programs using any Prolog evaluators since they existed around 1972 [1].

Logic programming, from the definition, is a programming paradigm that is largely based on formal logical systems. A logic program written in a pure logic programming
language should be a set of facts and rules within some domain. Since facts can be seen as special rules with only 'true' in the body, it's also common to say that a logic program is a finite set of rules [3]. Besides Prolog and Datalog, there is another major branch in logic programming family called ASP (Answer Set Programming), which aims to resolve search problems based on the answer set [4]. ASP is proposed in the research on the declarative semantics of negation in logic programming, which was motivated by the fact that the behavior of 'negation as failure' in traditional SLD resolution algorithm for logic programming does not fully match the truth tables familiar from classical propositional logic. There are many other subareas in logic programming such as Higher-order logic programming (Prolog extensions HiLog and λProlog).

Datalog is based on first-order logic as Prolog, and uses Horn clause as the logical formula for rules. Horn clause is a clause – a disjunction of literals – with at most one positive literal. The disjunction form of Datalog can be written as:

\[ ∀X_1 \ldots ∀X_m (A ∨ ¬L_1 ∨ \ldots ∨ ¬L_n) \]  

where the \( A \) is the literal for the head of a rule, the \( L_i \)’s are literals for the body of the rule, and \( X_i \)’s are all the variables occurring in a clause. In Datalog, an 'atom' is a predicate using variables or constants as arguments. A 'literal' is an atom with or without negation. The implication form of horn clause can be written as:

\[ ∀X_1 \ldots ∀X_m (A ← L_1 \land \ldots \land L_n) \]  

A Horn clause with exactly one or more negative literals is a regular rule with a body, and a horn clause with no negative literal is a fact, or to say, a rule without a body. In logic programming, it’s common to use ':-' to separate head and body, and use period to indicate the end of a rule or fact. Here is a simple example of Datalog program with 2 facts and 1 rule.

```
isPlace("State College").
hasWeather("State College", "Snow").
coldAt(v) :- isPlace(v), hasWeather(v, "Snow").
```

The first clause is a fact, whose name is `isPlace`, with one argument whose value is "State College", the second clause is also a fact with two arguments. The last clause is a rule with two literals in the body. To produce a new fact in the head, there are two conditions: the first arguments in two literals in the body should match, and the second
argument for `hasWeather` must be "Snow". The output for this Datalog program above should be `coldAt("State College")`.

A predicate is EDB (Extensional database predicates) if all of the facts associated with it already exist and won’t change. A predicate is IDB (Intensional database predicates) if this predicate appears in the head of some rules, which indicates some new facts will be generated during evaluation. During the evaluation of Datalog program, the IDB will update while more facts are generated. The evaluation will not terminate until IDB does not update anymore, or to say, reaches the "fixpoint".

### 2.1.2 Evaluation

The evaluation or execution of Datalog program is separated into two categories: bottom-up and top-down [5]. From the aspect of proof procedure in artificial intelligence, the bottom-up evaluation will always produce the entire consequence of the knowledge base. Sometime there could be an infinite number of consequences, like the example shown below. That is an obvious disadvantage if the query can be simply derived using a small portion of the knowledge base. That’s where top-down evaluation comes in. It takes in both the query and the knowledge base as input, and use backward search to prove it true or false. In proof procedure, the soundness of a proof procedure is that everything it derives follows logically from the knowledge base, or to say, is true in every model of the knowledge base. The completeness is that everything following logically from the knowledge base is derived. It can be proven that both top-down and bottom-up evaluation are sound and complete.

\[
isInteger(0).
\]
\[
isInteger(n+1) :- isInteger(n).
\]

In 1989, Ullman [6] showed that for the evaluation of any safe Datalog programs, bottom-up evaluation with optimization of Magic Sets has time complexity less than or equal to QRG (Queue-based Rule Goal Tree), which is a particular top-down strategy. But it does not mean that bottom-up is better than top-down for every Datalog programs. Ramakrishnan [7] has shown that although bottom-up evaluations are equal to or better than top-down evaluations in Prolog over a wide range of programs, they can do considerably worse for some programs that manipulate large non-ground terms. However, bottom-up methods have merit that allows a wide range of optimizations [8].

---

5
2.1.3 Top-Down Evaluation

Since the early 1970s, there have been many research works on combining SLD resolution with backtracking to develop a computational model for logic programming [9]. The well-known Prolog was the milestone of such research. Prolog was introduced to bring together the database query languages and programming language in 1981 [10] as part of the Japanese FGCS (Fifth-Generation Computer Systems). Because of the use of definite clauses, Prolog was able to base its query evaluation mechanism on SLD resolution, and apply depth-first search over the SLD tree which are built using the rules and facts.

The SLD resolution is an important technique for top-down evaluation. It works as an inference rule as shown in Eqs. (2.3). For a selected literal in a given goal clause, there is another clause where the selected literal is the head of the clause. We can replace the literal with the body of the second clause, from which we get another goal clause.

\[
\begin{align*}
\text{true} & \leftarrow G_1 \land \ldots \land G_m \\
G_i & \leftarrow L_1 \land \ldots \land L_n \\
\text{true} & \leftarrow G_1 \land \ldots \land G_{i-1} \land L_1 \land \ldots \land L_n \land G_{i+1} \ldots \land G_m
\end{align*}
\]  

(2.3)

This top-down strategy starts with an input query that is represented by one or more goal literals as shown below. Then it selects one of the literal, and searches for the selected literal in the heads of the rules. There could be several matches. We choose one match. If the match is an EDB fact, then we simply remove the literal. Otherwise, we replace the goal literal with the chosen rule’s body, possibly substituting for variables. We repeatedly and recursively do this replacement as shown below.

Algorithm 1 Top-Down Evaluation

1: Solve goal clause: \( \text{true} \leftarrow G_1 \land \ldots \land G_m \)
2: repeat
3: select \( G_i \in \text{goal clause} \)
4: choose clause \( C \) from program where \( G_i \) is the head
5: replace \( G_i \) in goal clause with the body of \( C \), possibly empty
6: until success when goal clause is empty, or fail when find clause for selected literal.

It’s possible that the "match clause chosen" fails to continue, we can go back and choose another match. If all the literals in the goal clause are eventually dispatched, then the collection of all variable substitutions used in the process can provide an answer to the original query. There could be alternative matches to the goal, and we can generate additional answers to the query.
This is just the high-level idea of top-down evaluation, which is expressed using a depth-first manner in origin research. In the later studies, some of the top-down evaluation frameworks are actually using a breadth-first search. There are various optimizations and heuristics in the practical frameworks for both depth-first and breadth-first manners. Tabling-related framework [11] was introduced in 1986 by Tamaki and Sato [12]. The recent research accomplishment of tabling approach is XSB [13], which implements SLG algorithm, a variant of tabling framework. Another well-known one is QSQ (query-subquery) framework, which is introduced in 1986 by Vieille [14]. We will discuss a bit about QSQ framework.

For QSQ framework, there are two key techniques: binding patterns or adornment, and supplementary relations [15,16]. Considering the following classical Datalog program as Eqs. (2.4). The last clause is the query, which is to find all possible destinations y starting from 'a'. It’s a simple program and we can see the result should be 'b' and 'c'. Now let’s discuss how QSQ framework runs on such a program.

\[\begin{align*}
\text{path}(x,y) & \leftarrow \text{edge}(x,y). \\
\text{path}(x,y) & \leftarrow \text{edge}(x,t), \text{path}(t,y). \\
\text{edge}('a', 'b'). \\
\text{edge}('b', 'c'). \\
\text{query}(y) & \leftarrow \text{path}('a', y). 
\end{align*}\] (2.4)

The first technique adornment is to create an adorned version of each predicate in each rule, following the left-to-right evaluation and considering the binding of variables. For the first rule \(\text{path}(x,y) \leftarrow \text{edge}(x,y)\), since we are only interested in facts for path where the first coordinate is bound to 'a', and the second coordinate is free. We can create the corresponding adorned version: \(\text{path}^{bf}\), where the superscript \(b\) (for bound) and \(f\) (for free) is called an adornment. Then for the body of this rule, all occurrences of variable \(x\) should also be bound. Since there is only one literal in the body, we can write the adorned version: \(\text{edge}^{bf}\). It should be noticed that, the adornments of IDB predicates in bodies can be used to guide evaluations of further subqueries. It is common to omit the adornment of EDB predicates in bodies [15]. Now we can rewrite the two rules in the second layer of SLD resolution as Eq. (2.5) and Eqs. (2.6).

\[\begin{align*}
\text{path}^{bf}(x,y) & \leftarrow \text{edge}(x,y). 
\end{align*}\] (2.5)
\begin{align}
\text{path}^{bf}(x, y) &\leftarrow \text{edge}(x, t), \text{path}^{bf}(t, y). \\
\text{path}^{bf}(x, y) &\leftarrow \text{path}^{ff}(t, y), \text{edge}(x, t).
\end{align}

(2.6)

The adornments for \text{edge} in Eqs. (2.6) are \text{bf} and \text{bb}. For the second rule, we can consider two orders of evaluation following the algorithm for adorning a rule. The algorithm is described here, given an adornment for the head and a selected ordering of the body, we should guarantee that: first, each occurrence of bound variables in the head are bound to the variable in the head; second, from left to right, if a variable occurs in the body, then each occurrence of the variable in subsequent literals are bound to it; and last, each occurrence of constants are bound. A different ordering of the rule body would yield different adornments. There could be multiple different orderings of a body for different adornments of a head. The occurrence of \text{path}^{ff} in Eqs. (2.6) leads to two new adornments of \text{path} as shown in Eq. (2.7) and Eqs. (2.8), which forms the third layer of SLD resolution.

\begin{align}
\text{path}^{ff}(x, y) &\leftarrow \text{edge}(x, y). \\
\text{path}^{ff}(x, y) &\leftarrow \text{edge}(x, t), \text{path}^{bf}(t, y). \\
\text{path}^{ff}(x, y) &\leftarrow \text{path}^{ff}(t, y), \text{edge}(x, t).
\end{align}

(2.7) \quad (2.8)

The adornments for \text{edge} in Eq. (2.7) is \text{ff}, for \text{edge} in Eqs. (2.8) are \text{ff} and \text{fb} respectively. It should be noticed that the predicate \text{path} has two different adornments, and they are actually treated as different relations during the computation.

This particular program can be called \textit{PATH query} since the query is on the predicate \text{path}. To generalize the context and target of a query, we use a pair \((P, q)\) to define a datalog query, where \(P\) is the whole datalog program and \(q\) is a datalog rule using relations of \(P\) in its body, and a new relation \text{query} in its head. The format of \text{query} is trivial. What’s important is to express the subquery during the SLD resolution. We use another format of pair \((R', J)\) to define a subquery, where \(R\) is an IDB predicate, \(\gamma\) is the adornment of \(R\), and \(J\) is a set of possible values for the bound variables by \(\gamma\). Here, \text{path}^{ff}(x, y) in Eqs. (2.6) calls for an evaluation as shown in Eq. (2.7) and Eqs. (2.8). For instance, the direct query of the program can be seen as a subquery \((\text{query}^{f}, \{\})\) in the first layer of SLD resolution, since there is no bound variables. The second layer subquery is \((\text{path}^{bf}, \{('a')\})\), with rules Eq. (2.5) and Eqs. (2.6). The third layer subquery can be expressed as \((\text{path}^{ff}, \{\})\), with rules Eq. (2.7) and Eqs. (2.8). Now we can see the origin
of the name of QSQ (query-subquery) framework.

The second technique is called supplementary relations. It aims to maintain the information in the left-to-right evaluation. For a rule with \( n \) literals in the body, there are \( n + 1 \) supplementary relations. Intuitively, the body of a rule can be viewed as a process that takes as input tuples over the bound variables of the head, and produces as output tuples over all the variables of the head, both bound and free. Therefore, the first supplementary relation has bound variables in the head, the last supplementary relation contains all the variables of the head. Except for the two special ones, each intermediate supplementary relation has variables that are bound at the corresponding position, and also appear in either predicate after it or the last supplementary relation. In other words, considering the rule enhanced with the first and the last supplementary relations, the variables in an intermediate supplementary must appear in the enhanced rule both before and after it. For example, we can write rules with supplementary relations for Eq. (2.5) and Eqs. (2.6), as shown in Eq. (2.9) and Eqs. (2.10)

\[
\text{path}^{bf}(x, y) \leftarrow [\text{sup}^0_0(x)\text{edge}(x, y)]\text{[sup}^1_1(x, y)]. \tag{2.9}
\]

\[
\begin{align*}
\text{path}^{bf}(x, y) & \leftarrow [\text{sup}^2_0(x)\text{edge}(x, t), [\text{sup}^2_1(x, t)]\text{path}^{bf}(t, y)]\text{[sup}^2_2(x, y)]. \\
\text{path}^{bf}(x, y) & \leftarrow [\text{sup}^3_0(x)\text{path}^{ff}(t, y), [\text{sup}^3_1(x, t, y)]\text{edge}(x, t)]\text{[sup}^3_2(x, y)]. \tag{2.10}
\end{align*}
\]

With the rules enhanced with supplementary relations, we also need to create two new relations for IDB predicates. For each adornment of every IDB predicates, we will create an input relation with same arity as the number of bound variables in the adornment, and an answer relation with same arity as the predicate. For instance, \( \text{input}_\text{path}(x) \) and \( \text{ans}_\text{path}(x, y) \) are created for \( \text{path}^{bf}(x, y) \). Intuitively, \( \text{input}_\text{path}(x) \) will be used to generate new subquery.

There are different computational models for the implementation of evaluation: iterative QSQ(QSQI) and recursive QSQ(QSQR). Both of them involve producing new answers and creating new subqueries [1]. In comparison, QSQI gives priority to new answers. QSQI will suspend working on any new subqueries until all answers that do not require those subqueries have been produced. On the other hand for QSQR, as the name implies, will suspend the processing of the current subquery when encountering a new subquery, and start to focus on the new one. The details of the algorithms will not be discussed here.
2.1.4 Bottom-Up Evaluation

Comparing to top-down evaluation, bottom-up evaluation is quite simple. The basic version of bottom-up evaluation is so-called Naïve Evaluation. For a Datalog program $P$, let $G_1, G_2 \ldots G_m$ be the IDB predicates of program $P$. For each IDB predicate $G_i$, let $\Gamma_i$ be the corresponding immediate consequence operator, which is a single application of all the rules with $G_i$ in the head. The arguments of $\Gamma_i$ only consider IDB predicates since tuples of EDB predicates won’t change during evaluation. Then we construct a loop that in every iteration, the tuples of each IDB predicate $G_i$ are re-calculated. This loop terminates when no new IDB facts are produced. The Naïve Evaluation is shown below.

**Algorithm 2 Naïve Evaluation**

1: $k := 0$
2: repeat
3: $k := k + 1$
4: for every $i$ do
5: $G_i^{(k)} := \Gamma_i(G_1^{(k-1)}, G_2^{(k-1)} \ldots G_m^{(k-1)})$
6: end for
7: until for every $i$, $G_i^{(k)} = G_i^{(k-1)}$
8: output $G_1^{(k)}, G_2^{(k)} \ldots G_m^{(k)}$

Clearly, there is redundancy in the Naïve Evaluation since the produced facts in previous iterations will be produced again in later iterations. To get rid of this redundancy, there is another bottom-up evaluation called Semi-Naïve Evaluation, which is adopted in almost every implementation of bottom-up Datalog evaluator. The key idea of it is that, to produce a fresh tuple in an iteration, we must use the new tuples that were produced in the last iteration. For example, let’s consider a rule $G_1 \leftarrow L_1, L_2, G_1, G_2$. in a Datalog program. In this program, $L_1$ and $L_2$ are EDB predicates, $G_1$ and $G_2$ are IDB predicates. To produce fresh tuples for $G_1$, we only need to create two new rules in application as shown in Eqs. (2.11).

$$
\Delta G_1^{(k+1)} \leftarrow L_1, L_2, G_1^{(k)}, G_2^{(k-1)}.
$$

$$
\Delta G_1^{(k+1)} \leftarrow L_1, L_2, G_1^{(k)}, \Delta G_2^{(k)}.
$$

(2.11)

For each created rule, we consider new tuples for one occurrence of IDB predicates. If there are multiple occurrences of one IDB predicate, there would be several rules created for each occurrence. It should be noticed that in Eqs. (2.11), we use a different set of tuples for non-incremental IDB predicate in two rules. This is to make it a complete
operator because the second rule is actually a combination of two rules: $\Delta G_1^{(k+1)} \leftarrow L_1, L_2, G_1^{(k-1)}, \Delta G_2^{(k)}$ and $\Delta G_1^{(k+1)} \leftarrow L_1, L_2, \Delta G_1^{(k)}, \Delta G_2^{(k)}$. The latter rule seems to have much less computation cost than other rules, so the two rules are merged.

In practice, to maintain a copy of an IDB predicate at previous iteration $k - 1$ might cost too much space. Therefore, we can simply use superscript $k$ for all non-incremental IDB predicates. The Semi-Naïve Evaluation is shown below, in which the operator $\Delta \Gamma_i$ is a complete operator for incremental evaluation.

**Algorithm 3 Semi-Naïve Evaluation**

1: $k := 0$
2: for every $i$ do
3:   $\Delta G_i^{(0)} := G_i^{(0)}$
4: end for
5: repeat
6:   $k := k + 1$
7:   for every $i$ do
8:     $\Delta G_i^{(k)} := \Delta \Gamma_i(G_1^{(k-1)}, G_2^{(k-1)} \ldots G_m^{(k-1)}, \Delta G_1^{(k-1)}, \Delta G_2^{(k-1)} \ldots \Delta G_m^{(k-1)}) - G_i^{(k-1)}$
9:   end for
10: for every $i$ do
11:   $G_i^{(k)} := G_i^{(k-1)} \cup \Delta G_i^{(k)}$
12: end for
13: until for every $i$, $\Delta G_i^{(k)} = \emptyset$
14: output $G_1^{(k)}, G_2^{(k)} \ldots G_m^{(k)}$

2.1.4.1 Magic Sets

As we can see from above, the top-down evaluation starts from the query and ignores irrelevant tuples during the evaluation, but the implementation is complicated and hard to optimize. Bottom-up evaluation is straightforward but may produce lots of irrelevant tuples during evaluation. To combine the advantages of both evaluations, there is a technique called Magic Sets. It was inspired by an earlier technique called 'selection push down'. For example, consider the rules in Eqs. (2.4), we are only concerned about IDB predicate path with the first coordinate be 'a'. Therefore, we can push it into rules in the program, as shown in Eqs. (2.12).

\[
\text{path}(a', y) \leftarrow \text{edge}(a', y).
\]
\[
\text{path}(a', y) \leftarrow \text{edge}(a', t), \text{path}(t, y).
\] (2.12)

Magic sets aim to avoid derivation of irrelevant facts by limiting bindings based on
constants in the final goal [1,17]. The key idea of magic sets transformation is to use the adornments and supplementary relations to simulate "selection push down" [16]. The full magic sets technique has more details than the example above. It has to track every path of binding propagation in a program. It will create copies of the predicate for each adornment, and each adorned relation may be constrained differently. And there are many variants of magic sets [18]. Soufflé [19] has also incorporated magic sets that can be enabled as an option, which follows the work by Balbin, Isaac, et al [20].

However, magic sets and the variants have a serious issue that it may create numerous copies of an original predicate. In fact, one predicate may introduce $2^k$ copies, where $k$ is the arity of the predicate for possible sequences of $b$'s and $f$'s. One fact may be stored redundantly in multiple copies of the predicate. Tekle and Liu [21] proposed a variant called demand transformation, which avoids the blow-up of predicates and redundant storage of facts by storing facts for the same predicate together. The performance of bottom-up evaluation with demand transformation has the same time complexity as the top-down evaluation with variant tabling [1].

2.2 Static Analysis and Lattice

2.2.1 Static Analysis

There has been more and more research work on the application of Datalog on static analysis [22] in the last two decades. Whaley and Lam [23] introduced a system called bddbdbdb to translate Datalog programs to BDD(binary decision diagrams) implementations, and developed several context-sensitive algorithms. Those implementations are more efficient than those that were hand-tuned. Steven, Ramakrishnan, and Warren [24] introduced the implementation of logical formulations of two analysis: groundless analysis of logic programs and strictness analysis of functional programs. Another more recent example is that Alpuente, Feliú, et al [25] used two formalisms for transforming definite Datalog clauses into two efficient implementations, namely BES(Boolean Equation Systems) and RWL(Rewriting Logic). RWL is a general logical and semantical framework which is implemented in an efficient high-level executable specification language Maude. There are more examples. As Ben Greenman put it [26], "Datalog has been successful because it lies at a confluence between logic programming and static analysis."

Static analysis is the program analysis without actual execution of programs, in contrast with dynamic analysis, which is an analysis performed during the execution.
Static analysis can be performed on either source code or binary code. The static analysis of source code has been used since the early 1960s in the optimization of compilers [27]. After that, it has proven useful also for finding bugs and verification tools, and to support program development in IDEs (Integrated Development Environment). With regard to compiler optimization, there is a lot of applications. For example, it can detect and remove dead code, or at some program point check if a variable is constant, check if a complicated expression has been calculated earlier in the program and is still available.

Data-flow analysis can be dynamic [28], or static. Static data-flow analysis is common in compiler optimization, in which multiple different data-flow analyses on the CFG (Control Flow Graph) can be performed to improve the efficiency of the program. There is four classic data-flow analysis as described here. Available Expression analysis is to determine the set of expressions for a program point so that the expression does not need to be recalculated, and it can be used to eliminate common sub-expressions. Reaching Definition analysis is to determine which definitions may reach a given program point, and is often used to compute use-def chains and optimize for constant and variable propagation. Very Busy Expressions analysis is to determine the set of expressions whose value will be evaluated before it changes on every path from the given program point, and it can optimize the program by hoisting these expressions from all paths. Live Variables analysis is to determine the set of live variables at some program point that may be needed in the future, and it can be used in register allocation and to eliminate dead codes. The four data-flow analysis can be categorized as shown in Table 2.1.

<table>
<thead>
<tr>
<th></th>
<th>Must</th>
<th>May</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>Available Expression</td>
<td>Reaching Definition</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>Very Busy Expressions</td>
<td>Live Variables</td>
</tr>
</tbody>
</table>

To explain the Table 2.1, let’s consider the classical data-flow analysis: Reaching Definition analysis. For this analysis, we firstly declare that a definition $D$ reaches a point $P$ if there is a path from $D$ to $P$, in which $D$ is not redefined, or to say, not killed. We are passing information from the previous program point to some future program point, so it’s called **Forward**. And it is **May** analysis because we will consider a definition $D$ to reach a point $P$ as long as there is some path satisfying the requirements. Thus, we take union $\cup$ on sets of facts at each join point of CFG. Contrarily, for **Must** analysis we take intersection $\cap$ on sets of facts at join point.
2.2.2 Lattice

Here are some basic concepts before we use lattice theory to explain the static analysis. The first one is partial ordered set. It is represented by a pair \((S, \sqsubseteq)\), where \(S\) is a set, finite or infinite, in a specific domain and \(\sqsubseteq\) is a binary relation or partial order on \(S\). And the binary relation \(\sqsubseteq\) must be: reflexive, transitive and anti-symmetric. It’s reflexive when \(\forall x \in S, x \sqsubseteq x\). It is transitive when \(\forall x, y, z \in S, x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z\). It is anti-symmetric when \(\forall x, y \in S, x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y\).

Consider a subset \(X \subseteq L\), we say \(y \in L\) is an upper bound for \(X\) (written \(X \sqsubseteq y\)) when \(\forall x \in X, x \sqsubseteq y\). Also, we say \(y \in L\) is an lower bound for \(X\) (written \(y \sqsubseteq X\)) when \(\forall x \in X, y \sqsubseteq x\). Then, a least upper bound (also called supremum, or join) for \(X\), written \(\bigvee X\), is defined that, \(\bigvee X\) is an upper bound for \(X\) and, for any upper bound \(u\) of \(X\), \(\bigvee X \sqsubseteq u\). Similarly, a greatest lower bound (also called infimum, or meet) for \(X\), written \(\bigwedge X\), is defined that, \(\bigwedge X\) is an lower bound for \(X\) and, for any lower bound \(b\) of \(X\), \(b \sqsubseteq \bigwedge X\). Particularly, the least upper bound of the whole set \(L\) (written \(\bigvee L\)) is denoted \(\top\) (top), and the greatest lower bound of \(L\) (written \(\bigwedge L\)) is denoted \(\bot\) (bottom).

Finally, we can define the complete lattice, which is a partial ordered set \((S, \sqsubseteq)\) if every subset \(X\) of it has a least upper bound \(\bigvee X\) and a greatest lower bound \(\bigwedge X\). An operator \(f\) on ordered set \(L\) is monotone when \(\forall x, y \in L, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)\). Knaster-Tarski theorem, or known as Tarski’s Fixpoint Theorem [29] shows that, for a monotone operator \(f\) on a complete lattice \(L\), there exists a least fixpoint.

For the four classical data-flow analyses mentioned above, we can use powerset lattice and reverse powerset lattice to represent them. These lattices are also parameterized lattice since they depend on the specific program analyzed [27]. Let’s consider the sample program below.

```plaintext
int x, y, z;
x = 3;
if (x < 6) {
    y = x*2;
    z = x-y;
}
```

For May analyses including Reaching Definition analysis and Live Variables analysis, the binary operator or lattice order \(\sqsubseteq\) is \(\subseteq\), and we use union (\(\cup\)) to combine information as mentioned above. For them, we use a powerset lattice where the set \(S\) of elements are the variables or definitions occurring in the given program, and \(\bot = \emptyset, \top = S\). Consider
Live Variables analysis of the sample program above, the lattice can be represented as $(2^\{x,y,z\}, \subseteq)$, which is shown in Fig. 2.1.

![Figure 2.1: Powerset Lattice for Live Variables analysis](image)

On the other hand, for Must analyses including Available Expression analysis and Very Busy Expressions analysis, the binary operator $\subseteq$ is $\supseteq$, and we use intersection ($\cap$) to combine information. For them, we use a reverse powerset lattice with set of elements $S$ equals all existing expressions, where $\bot = S$, $\top = \emptyset$. Consider Available Expression analysis of the sample program above, the lattice can be represented as $(2^\{x*2, x-y\}, \subseteq)$, which is shown in Fig. 2.2.

![Figure 2.2: Reverse Powerset Lattice for Available Expression analysis](image)

### 2.3 Soufflé Programming Language

Soufflé is an open-source high-performance Datalog evaluator that originated from the research work by Scholz, Vorobyov, et al in Oracle Labs in 2015 [22]. In the 2015 paper, the idea of "source-to-source translator for static program analysis" is brought up which includes Semi-Naïve Evaluation and Stratified Negations, but the name 'Soufflé' was not decided. Soufflé is written with C++, specifically designed for Datalog programs over large data sets, especially encountered in the context of static program analysis. The
initial release of Soufflé is in 2016. There have been many releases with more features and optimizations since then. In this thesis, we focus on version 1.5.1 which was released on January 20, 2019.

Soufflé has two modes: interpreter mode and compiler mode. The interpreter mode is the default mode when invoking the command `souffle`. In this mode of Soufflé, the parser translates the Datalog program to a RAM (Relational Algebra Machine) program (i.e. SQL queries in 2015 paper) using the bottom-up Semi-Naïve Evaluation [22], then executes the RAM program on-the-fly. The compiler mode is more sophisticated. After the translation to a RAM program, it will continue to translate the RAM program to a C++ program and achieve further optimization. For computationally intensive Datalog programs, the interpreter mode could be slower than compiler mode. However for smaller programs, the interpreter mode can be faster since it has no costs for compiling a RAM program to C++ program and invoking the C++ compiler. The compilation could be in the order of minutes, which is for larger programs. The architecture of Soufflé is shown in Fig. 2.3.

![Soufflé architecture](image)

Figure 2.3: Soufflé architecture

To illustrate the translations in Soufflé, we use an example program as shown in Lst. 2.1. In the syntax of Soufflé, we use `.decl` to declare a relation, use `.input` and `.output` to specify relation whose facts are written from or written to external file. Clauses end with a period, e.g., `a(1).` is a fact clause for relation `a`, and `a(x) :- b(x), d(x).` is a rule clause.
Listing 2.1: Example of Soufflé program

1 .decl a(x:number)
2 .decl b(x:number)
3 .decl c(x:number)
4 .decl d(x:number)
5
6 .output a, b, c
7
8 a(1).
9 b(1).
10 d(1).
11
12 a(x) :- b(x), d(x).
13 b(x) :- a(x), d(x).
14 c(x) :- a(x), b(x).

To translate from Soufflé program to AST structure, Soufflé uses flex and bison to generate a parser and a scanner. Then there will be semantic check (including cyclic negations) and some optimizations [19] on the AST structure, such as: alias elimination, i.e., the unification of variables according to equality constraints imposed in the input program; rule elimination, i.e., the elimination of rules that do not contribute when their body contains empty relations; and the aforementioned magic sets, and so on.

During the translation from AST to RAM in Soufflé, the key is adapting Semi-Naïve Evaluation. From a high level, this is to exploit the structure of rules and find a general way to rephrase it and make it efficient through a better algorithm or data structure. Another interpretation of such process is called Futamura projection or Partial Evaluation [30]. With Partial Evaluation, the output program should run faster than the original one. Strictly, the translation from AST to RAM is not a "Partial Evaluation" since the AST structure is not a complete program with an evaluation algorithm in it.

In a Datalog program, we are only concerned about IDB relations, which could be non-recursive relations and recursive relations. To separate them into different fixpoint evaluation processes, Soufflé will compute the data dependencies among the relations through a precedence graph. Then, using the precedence graph, it will compute a strongly connected component (SCC) graph. After that, Soufflé can build stratum for the output RAM program in sequence, for either a cluster of recursive relations in an SCC component, or a cluster of non-recursive relations represented by a group of SCC
components at the same level.

![Precedence Graph](image1)
![SCC Graph](image2)

Figure 2.4: Precedence and SCC Graph for the example program

The precedence graph and SCC graph for the example program are shown in Fig. 2.4. We can see that d and c are non-recursive relations, while a and b are mutual dependent recursive relations. The RAM program generated by Soufflé from the example program above is shown in Appendix A. In the RAM program, besides the stratum for declaration, Soufflé generates a stratum for \(a, b\), and another stratum for c. There is no stratum for d since it is an EDB relation. If there are facts for the relation in an external file (.input), Soufflé will also generate a stratum to load facts from the file. The generated RAM program can be outlined as Lst. 2.2.

Listing 2.2: Pseudo Code for RAM of example program

```plaintext
1 d = {1};
2 a = {1}; \(\Delta a = a\);
3 b = {1}; \(\Delta b = b\);
4 while (\(\Delta a \cup \Delta b \neq \emptyset\)) {
5     new_a = (\(\Delta b \cap d\)) \ a;
6     new_b = (\(\Delta a \cap d\)) \ b;
7     a = a \cup new_a;
8     b = b \cup new_b;
9     \(\Delta a = new_a\);
```

18
\[ \Delta b = \text{new}_b; \]
\[ c = \{ \}; \]
\[ c = a \cap b; \]

The syntax of RAM program in Soufflé are listed in Eqs. (2.13) in BNF notation. Some auxiliary statements are not shown here such as \texttt{debug}, \texttt{load} and \texttt{store}. In the context-free grammars below, \( S \) stands for RAM statement, \( O \) is RAM operation, \( R \) is relation, \( C \) is condition, \( V \) is RAM value, and \( t_{id} \) is the variable for iteration. The statement \texttt{insert} stands for a relational algebra statement performing the evaluation of a clause. In the newer version (\( \geq 1.6.0 \)) of Soufflé the terminology \texttt{insert} is modified to \texttt{query}.

\[
\begin{align*}
S & \rightarrow \texttt{insert} \ O \\
S & \rightarrow S_1; \cdots; S_2 \\
S & \rightarrow \texttt{loop} \ S_1 \ \texttt{end\_loop} \\
S & \rightarrow \texttt{merge} \ R_1 \ \texttt{with} \ R_2 \\
S & \rightarrow \texttt{parallel} \ S_1; \cdots; S_k \ \texttt{end\_parallel} \\
S & \rightarrow \texttt{begin\_stratum\_i} \ S_1 \ \texttt{end\_stratum\_i} \\
S & \rightarrow \texttt{swap} \ (R_1, R_2) \\
S & \rightarrow \texttt{create} \ R_1 (a_1, \cdots, a_n) \\
S & \rightarrow \texttt{clear} \ R_1 \\
S & \rightarrow \texttt{drop} \ R_1 \\
S & \rightarrow \texttt{insert} \ (V_1, \cdots, V_k) \ \texttt{into} \ R_1 \\
S & \rightarrow \texttt{exit} \ C \\
O & \rightarrow \texttt{project} \ (V_1, \cdots, V_k) \ \texttt{into} \ R_1 \\
O & \rightarrow \texttt{if} \ C \ \{O\} \\
O & \rightarrow \texttt{for} \ t_{id} \ \texttt{in} \ R_i \ \{O\}
\end{align*}
\]

There are two benefits in the translation from AST (Abstract Syntax Tree) to RAM (Relational Algebra Machine). The first is that the RAM program or SQL queries can be very general and have effective use of storage in any existing database system. The second benefit, in the author’s point of view, is "modulus". We can apply various optimizations in this translation, which can take effect in both interpreter mode and compiler mode.
without loss of generality. With this translation, it’s easier to debug by checking the generated RAM program without running the whole interpreter or compiler process.

The most important optimization on the RAM program is Automatic Index Selection [31] by Subotić, Pavle, et al. in 2018. In a relational database, indexing is very common since it makes columns more efficient to search by creating pointers to where data is stored within a database. In traditional schemes, indexing either requires manual index selection or result in insufficient performance on large scale computation. In their paper, they define the minimum index selection problem (MISP), i.e., to find the minimum number of indexes to cover all primitive searches. They introduce a polynomial-time algorithm to solve MISP optimally through computing search chains. And they propose an automatic scheme to select and create a minimum number of indexes to speed up all the searches in a given Datalog program. The scheme is implemented in Soufflé and is measured to provide fast speed and low memory.

The last translation in Soufflé – from RAM to C++ program – is for the compiler mode. It can be seen as a true "Partial Evaluation" since it translates a complete program with all algorithms and data structures declared, into another program. This translation is implemented using template-based meta-programming techniques introduced in 1998 [32]. With this technique, data structures and algorithms are specialized by static information, to hoist computations from run-time to compile-time [33]. For example, in the generated C++ program, data structure interfaces are realized in the form of C++ concepts rather than polymorphic C++ base classes to eliminate virtual-call dispatches and run-time type checks. These generated data structures are highly specialized towards the use of the corresponding relations in the input Datalog program. Furthermore, primary and secondary index [19] support is provided for efficient operations on the represented relation in the generated program. Besides that, Soufflé can switch between B-tree and Trie which store the relation directly, based on the number of attributes and indices [19]. This technique is also known as indexed-organized tables.

The parallel execution of Soufflé in interpreter mode and compiler mode are both using OpenMP/C++ for PARALLEL statement in RAM program. And the degree of parallelism can be automatic or controlled by the user. It seems the earlier version of Soufflé tried to use MPI for further parallelization. However it has been abandoned in the newer version (1.7.0).

Soufflé is still under development and there are more features that have been added or in progress. In 2019, Zhao, Subotić, and Scholz [34] introduce a provenance evaluation strategy for Datalog specifications. The provenance evaluation strategy extends tuples in
the IDB with proof annotations. With the help of proof annotations, provenance queries can construct minimal proof trees incrementally.

And there are more studies on the use of Soufflé. Ashley Huhman [35] used Soufflé for type inference analysis against SPEC2006 benchmarks on the binary level. Corey Capooci [36] compared three Datalog engines, Soufflé, PA-Datalog, and Datalog Educational System (DES). And the results show that Soufflé performs the best for most algorithms including connectivity algorithm, strongly connected components algorithm, and weakly connected components algorithm.

2.4 Flix Programming Language

Similar to Soufflé, Flix is also a variant of Datalog and it adopts Semi-Naïve Evaluation and Stratified Negations. Unlike Soufflé, Flix is written with another language called Scala which supports both functional programming and object-oriented programming. Flix itself also supports functional programming. The initial release of Flix is in June 2016. In this thesis, we use version 0.9.1, which was released in December 2019. There are multiple interesting features in Flix, such as first-class constraint, polymorphic data types, concurrency with processes and channels, and high-order functions, which is the basic block of functional programming. Among these features, we focused on the support of lattice semantics in Flix.

Traditionally, Datalog can only use powersets of tuples to represent facts in static analyses. For example, let’s consider the powerset lattice as shown in Fig. 2.1. For a given program point, we can use three tuples: \( \text{Live}(x), \text{Live}(y) \) and \( \text{Live}(z) \) to represent the top lattice element \( \{x, y, z\} \). This could come in handy for such powerset lattice. However, some lattices could bring high computational cost if embedded in powersets, such as constant propagation [37].

Flix introduced semantics for lattice declaration, monotone filter functions and monotone functions to explicitly represent a static analysis program [38] using lattice. In Flix, a monotone filter function is a function mapping from one or more lattice elements to \texttt{true} or \texttt{false}, and is monotone when the booleans are ordered \texttt{false} < \texttt{true}. It can be used like a relation in the body of a rule. A monotone function is a function mapping from one or more lattice elements to one lattice element, and it is also order-preserving. It can be used as an argument of a relation in the head.

In comparison, Soufflé also has some similar features. In Soufflé, standard binary operations (\( >, <, =, \neq, \geq \) and \( \leq \)) can act as a relation in the body of a rule, e.g.
A(x+1) :- A(x), x<=99. Also, there are many special types that can be used as arguments in Soufflé, such as: aggregate functions (min, max, sum, and count); standard arithmetic operations (+, -, *, /, ^ and %); and user-defined functors that are implemented in the form of a shared library.

Flix uses Map and Array in Scala to store indexes and facts in relations. In Flix, the parallelization of rule evaluation and datastore update is supported with invokeAll in Scala. Since Flix is specifically designed for static analysis, it can even examine the input Datalog for safety and soundness [39] so that the input static analysis program is guaranteed to converge and the computed result over-approximates the concrete behavior. Like Soufflé, Flix is under development and there are more features planned to add into it.
3.1 Minimal Model considering Lattice

In 2016, Madsen, Yee, and Lhoták [37] presented the idea of extending Herbrand structure with lattice, which is adopted in Flix. In this thesis, we apply their idea on Soufflé. Now we present the simplified version of that idea. We assume that if there is a lattice argument in a predicate, it must only be the last argument. We also define a cell \( S \) for predicate symbols that, two ground atoms are in the same cell if they have the same predicate symbol, and all ground terms in them are equal except the last one, which is the lattice element type. (A ground term is a term that appears in Datalog program \( P \) and does not contain any variables. A ground atom is a predicate symbol with only ground terms as its arguments.)

A model \( M \) of a Datalog program \( P \) is a set of ground atoms that makes every input atoms and rules in \( P \) true. Considering the lattice, a model is compact when every cell of \( M \) has at most one unique tuple, i.e., one lattice element for one cell. A model is minimal if it is compact and for every cell \( S \), the lattice element is the least one. For example, given a simple Datalog program and corresponding lattice below,

\[
\begin{align*}
A(6, \text{yes}). \\
B(6, \text{no}). \\
A(x, y) & \leftarrow B(x, y). \\
C(x, y) & \leftarrow A(x, y), B(x, y).
\end{align*}
\]

With the first rule, we should generate \( A(6, \text{no}) \) and add it into IDB. Since there is \( A(6, \text{yes}) \) already, we should add \( A(6, \top) \) into IDB also. Here we use least upper bound function in insertion. Then for the last rule, although we have \( A(6, \top) \), we can only generate \( C(6, \text{no}) \) since there is only \( B(6, \text{no}) \). In this occasion, we apply greatest
lower bound function on variable $y$. Here are several correct models for this program. However, $M_1$ is not compact, $M_2$ is compact but not minimal, and only $M_3$ is the minimal model that we want.

$$M_1 = \{A(6, \top), \ B(6, \text{no}), \ C(6, \text{yes}), \ C(6, \text{no})\}$$
$$M_2 = \{A(6, \top), \ B(6, \text{no}), \ C(6, \top)\}$$
$$M_3 = \{A(6, \top), \ B(6, \text{no}), \ C(6, \text{no})\}$$

### 3.2 Conditional Operator

Conditional ternary operator, or question mark operator is a syntactic sugar we add to make some codes in Soufflé simpler to write and read. It works just like the classical conditional ternary operator as shown below. If the condition is evaluated to be true in Soufflé, the expression `exprIfTrue` will be evaluated, otherwise `exprIfFalse` will be evaluated. In comparison, Flix use if-else statement for similar function [37]. The usage and benefit of conditional ternary operator will be shown in the following sections.

`condition ? exprIfTrue : exprIfFalse`

### 3.3 Unary and Binary Case Function

Case function is a specialized function we add for static analysis using lattice. It is like a Switch statement in a function style. It’s similar to function definitions for lattice in Flix [37]. They are different in that it is able to map any type to any type, instead of being restricted to only lattice elements in Flix. See below for an example of a unary case function.
// without conditional operator
.def nonZero(x: number): number {
    case (0) => 0,
    case (_) => 1,
}

In the example above, we define a unary function called nonZero, which maps type number to type number. In Soufflé, the underscore _ can match any value. With a conditional operator, it can be simplified into another version as shown below.

// with conditional operator
.def nonZero(x: number): number {
    case (_) => x=0 ? 0 : 1
}

The case function can be called in a rule in the extended Soufflé. To separate it from the user-defined functor which has a prefix symbol @ (e.g. A(@f(i)) :- A(i), @f(i)<100.), the call to a case function has another prefix symbol &. For example:

A(&nonZero(i)) :- A(i), 0!=&nonZero(i).

Similarly, we can define a binary case function glb (greatest lower bound) and simplify it with conditional operators as shown below. The data type Simple is an instantiation of enum type, which will be explained in the following section. It can be seen that the application of conditional operators significantly reduces the number of lines in code. It could be very useful on some occasions.

// without conditional operator
.def glb(x: Simple, y:Simple): Simple {
    case ("Top", _) => y,
    case (_, "Top") => x,
    case ("yes", "yes") => "yes",
    case ("no", "no") => "no",
    case (_, _) => "Bot"
}

// with conditional operator
.def glb(x: Simple, y:Simple): Simple {
  case ("Top", _) => y,
  case (_, "Top") => x,
  case (_, _) => x=y ? x : "Bot"
}

3.4 Lattice Declaration

3.4.1 Enum Type

To declare the lattice, we add a data type called enum to Soufflé, as the syntax for lattice in Flix. See below for an example of enum definition, which is for the simple lattice as shown in Fig. 3.1. There are some differences between the two enum types. In Flix, an enum declaration can contain other data types explicitly, such as case Single(Str) [37]. However, the enum type we add into Soufflé can only contain elements of symbol type such as case "Top" as shown below, and cannot contain any declaration of other data types in it. It is not a traditional 'enum' type which covers every case for it, instead, it declares part of the possible symbol cases for a lattice type. Now we discuss the reason for such property.

.enum Simple = {
  case "Top",
  case "yes", case "no",
  case "Bot"
}

In Soufflé, there are two primitive data types: symbol and number. After the translation to abstract syntax tree as shown in Fig. 2.3, the data value and data type are stored separately. To store the data value, Soufflé uses a fixed 32-bit space for each value, for either symbol or number. For symbol type, Soufflé will use symbol table to store every existence of symbol, and translate it into a 32-bit integer. Therefore, it is not able to distinguish symbol from number during computation. However, for a constant propagation lattice element, which can be symbol (eg: 'Top') or number (eg: any number) in different tuples, it will be confusing to the interpreter. Another example is ConstSign
lattices [39]. There would be a misunderstanding in functions involving such a lattice element as input. For example, if the Simple lattice in Fig. 3.1 contains a number 1, and 'Top' also maps to 1, then the function \texttt{glb} above will generate incorrect output.

In this environment, the enum declaration we add to Soufflé has two uses: first is to notify the possible symbol values for the lattice to symbol table so it can correctly translate them into 32-bit integer; second is to offset the mapping in the symbol table for these symbols declared in enum. For the second usage, we offset all the mapping to a reserved region near the maximum 32-integer. This region does not cover the maximum value which is common to appear. Still, the user should be careful to avoid the conflict between symbol and number if the lattice element contains both of them since we haven’t added the range check during computation, because such conflict is very rare and such check would add computational cost.

### 3.4.2 Lattice Association

The lattice definition associates a lattice type with 4-tuple \( (\perp, \top, \sqcup, \sqcap) \) as shown below, where \( \perp \) is the bottom element, \( \top \) is the top element, \( \sqcup \) is the least upper bound function, and \( \sqcap \) is the greatest lower bound function. Comparing to the lattice definition in Flix which is 5-tuple, the lattice definition we add in Soufflé does not contain the declaration of partial order \( \sqsubseteq \), which has been embedded in \( \sqcup \) and \( \sqcap \) function.

\[
\texttt{.let Simple<> = ("Bot", "Top", lub, glb)}
\]

### 3.4.3 Lattice Relation

The declaration of a relation with a lattice element (as the last argument) is referred to as lattice relation. We use \texttt{.lat} instead of \texttt{.decl} for such declaration. The last and only the last argument of lattice relation must be lattice type. It can read from and/or write to external files just like ordinary relations in Soufflé. Here is an example of a lattice relation declaration as shown below.

\[
\texttt{.lat varSimple(n:number, v: Simple)}
\]
\[
\texttt{.input varSimple}
\]
\[
\texttt{.output varSimple}
\]
3.5 Modification in RAM

To obtain a compact model for the whole fix-point problem considering lattice, in our implementation, we force the lattice relation to be compact at the end of each stratum. A significant benefit of such implementation is the following computation can gain more efficiency from the compactness. We also need to adjust the Semi-Naïve Evaluation for the corresponding stratum to make it correct. There are three major modifications in translation to RAM considering the lattice data type: 1. apply glb (greatest lower bound) to bound lattice variables during the update of lattice relation; 2. apply LATCLEAN (clean lattice relation) to each new lattice relation inside each Semi-Naïve evaluation loop; 3. apply LATNORM (normalize lattice relation) to updated lattice relation at the end of each stratum. Now we explain each modification using the example program as shown in Lst. 3.1, in which the Simple lattice has been described before. The corresponding pseudo code for RAM is shown in Lst. 3.2.

The first modification is the application of glb function on bound lattice variables. In comparison, for normal data types, all bound variables must be the equal, as expressed by the intersection symbol \( \cap \) in Lst. 2.1. (In the corresponding RAM of original Soufflé, it is enclosed by IF filter, as shown at line 33, 46, 76 in Appendix. A.0.2.) However for a variable of lattice type, instead of restricting them to be equal, we should compute the greatest lower bound among all bound lattice variables. Here we use the modified intersection \( \cap^* \) to indicate this computation at lines 5, 7 and 17 in Lst. 3.2. (In the corresponding RAM of extended Soufflé, it can be seen at line 66, 77, 119 in Appendix. A.0.3.) This modification is necessary. For example, consider line 17 in Lst. 3.2 which is the evaluation of the rule \( c(x) \ :- \ a(x), \ b(x) \). At this line the lattice relation \( a \) has only one tuple \( a("Top") \), relation \( b \) has only one tuple \( b("no") \). It should generate \( c("no") \) at line 17.

Listing 3.1: Example of Soufflé program with Lattice

```plaintext
1. lat a(x:Simple)
2. lat b(x:Simple)
3. lat c(x:Simple)
4. lat d(x:Simple)
5
6. output a, b, c
7
8. a("yes").
```
The second modification is to apply a new statement LATCLEAN to each new lattice relation inside each semi-Naïve evaluation loop. Here the LATCLEAN statement is used at lines 6 and 8 in Lst. 3.2. The goal of LATCLEAN is to compute the real new tuple for each cell in lattice relation. We first apply the least upper bound $\sqcup$ to each cell to compute the least upper lattice element for each cell. Then we check if the computed tuple exists in the original lattice relation. If it exists, we remove the computed "fake" new tuple, represented by operator $\ominus$. For example, if the lattice relation a contains one tuple a("no"), and the new_a at line 5 contains one tuple new_a("yes"). After the application of LATCLEAN at line 6, the new_a will contain only one tuple new_a("Top").

The third modification is to apply LATNORM to each updated lattice relation at the end of each stratum. Here the LATNORM statement is used at lines 14, 15 and 18 in Lst. 3.2. It can be seen that after the update of lattice relation, either recursive or non-recursive, it may not be compact. For example, at the start of line 14 in Lst. 3.2, the lattice relation a will contain two tuples: a("no") and a("Top"). We need to remove the tuple a("no"). This normalization process is not carried out within the loop because, for parallel operation on a B-tree database which is adopted in Soufflé, it is easy to insert a new tuple, while not efficient to remove or update a tuple.

Listing 3.2: Modified Pseudo Code for RAM

```plaintext
1 d = {"yes"};
2 a = {"no"}; ∆a = a;
3 b = {"no"}; ∆b = b;
4 while (∆a ∪ ∆b ≠ ∅) {
5    new_a = (∆b \* d) \ a; // glb
6    new_a = (new_a ∪ a) \ a; // LATCLEAN
7    new_b = (∆a \* d) \ b; // glb
8    new_b = (new_b ∪ b) \ b; // LATCLEAN
9    a = a ∪ new_a;
10   b = b ∪ new_b;
```
\[ a = \text{new}_a; \]
\[ b = \text{new}_b; \]
\} \]
\[ a = \text{Norm}(a); \quad // \text{LATNORM} \]
\[ b = \text{Norm}(b); \quad // \text{LATNORM} \]
\[ c = \{}; \]
\[ c = a \cap^* b; \quad // \text{glb} \]
\[ c = \text{Norm}(c); \quad // \text{LATNORM} \]
Chapter 4  |  Implementation

According to the structure of Soufflé as shown in Fig. 2.3, we can separate the implementation into several segments: parsing to AST, translation from AST to RAM, RAM interpreter, translation from RAM to C++ code. The last segment is for compiler mode in Soufflé, and has yet to complete. There could be some optimization in translation to C++, especially on the unary and binary case functions since the formats of them are highly specialized. Here we discuss the details in the implementation of the first three segments, which achieved all functionalities mentioned in Chapter 3.

4.1 Parsing to AST

As mentioned before, Soufflé uses flex and bison to generate scanner (scanner.cc) and parser (parser.hh, parser.cc). Thus we add all of the new syntax mentioned in Chapter 3 into the spec file scanner.ll for flex and spec file parser.yy for bison, including conditional operator (c ? e1 : e2), unary and binary case function (.def), lattice enum type (.enum), lattice association (.let) and lattice relation (.lat). Moreover, we add corresponding CFG rules in parser.yy.

Soufflé has a class ParserDriver which will utilize the generated scanner and parser to parse the input Datalog file (.dl), and generate an object AstTranslationUnit from it, which contains all data that can be updated in the later optimization on AST node and used for the translation to RAM nodes. It contains AstProgram, a set of AstAnalysis, SymbolTable, ErrorReport and DebugReport. The class AstProgram is an intermediate representation of a Datalog program that consists relations, clauses and types. The class AstAnalysis is the base class for many core analyses on AST such as collecting recursive clause, creating adornment for magic sets and checking type environment. The class SymbolTable, as its name suggests, stores the mapping from string to number and number
To store all of the new syntax in AST, we add several classes under the base class \textit{AstNode} as shown in Fig. 4.1. The name of these new classes speak for themselves. The class \textit{AstLatticeFunctor} are calls to unary or binary case function. The class \textit{AstLatticeFunction} stores the name of the function, the type of input and output arguments,

![Diagram of Class Hierarchy of Updated AstNode]

Figure 4.1: Class Hierarchy of Updated AstNode
and a vector of struct \textit{PairMap}, each of which represents a case in the function such as \texttt{case \{"Bot", \_\}} => 1. The class \textit{AstQuestionMark} is representation of the conditional operator. The class \textit{AstLatticeAssociation} stores the name of top element, bottom element, \texttt{glb} and \texttt{lub} function for a lattice. These names will be used in the later translation to RAM program. The class \textit{AstEnumType} stores all possible symbols for the created lattice.

In order to parse the input Datalog file into these new AST classes, we add several functions in \textit{ParseDriver} in addition to the updates in spec file for flex and bison. These functions include \texttt{ParserDriver::addLatticeFunction}, \texttt{ParserDriver::addLatticeAssociation} and \texttt{ParserDriver::addFunctorDeclaration}. Then the added CFG rules in \textit{parser.yy} will call corresponding functions. For example, CFG rules involving \texttt{.let} will call \texttt{ParserDriver::addLatticeAssociation} to update \textit{AstProgram}.

4.2 Translation from AST to RAM

The updates in this translation include two major aspects. The first is to translate those new AST nodes mentioned in the last section into RAM nodes, which should be more concise since there are many built-in analyses and optimizations on AST structure in Soufflé. The second is to modify the algorithms in RAM program with respect to lattice relations, as described in Section 3.5.

In Soufflé main program \texttt{main.cpp}, the call to function \texttt{AstTranslator::translateUnit} processes translation from \texttt{AstTranslationUnit} to \texttt{RamTranslationUnit}. Similar to the parsing to AST, there are some optimizations on the translated \texttt{RamTranslationUnit}. The class \texttt{RamTranslationUnit} contains \texttt{RamProgram}, a set of \texttt{RamAnalysis}, \texttt{SymbolTable}, \texttt{ErrorReport}, \texttt{DebugReport} and a mutex lock \texttt{analysisLock}. The class \texttt{RamProgram} stores all essential data for the RAM program, including \texttt{RamRelation}, \texttt{RamStatement}, \texttt{RamLatticeAssociation} and mapping from string to \texttt{RamLatticeUnaryFunction} and \texttt{RamLatticeBinaryFunction}.

Similar to AST, in order to store the new data about lattice in RAM program, we add several classes under the base class \texttt{RamNode} as shown in Fig. 4.2. Some of them are just the RAM version of AST nodes for new syntax, including \texttt{RamLatticeAssociation}, \texttt{RamLatticeFunction}, \texttt{RamLatticeFunctor} and \texttt{RamQuestionMark}. Both \texttt{RamLatticeFunction} and \texttt{RamQuestionMark} use \texttt{RamCondition} to represent the condition in them. In translation to \texttt{RamLatticeFunction}, the underscore \_ which can match anything, is removed from the condition, so that the process of condition checking for interpreter can
Figure 4.2: Class Hierarchy of Updated RamNode
be simplified.

Besides, there are three new classes involving algorithms in RAM as mentioned in Section 3.5: RamLatClean, RamLatNorm and RamLatticeGLB. The statement LATCLEAN will compute the "real" new tuple for each cell in lattice relation. The class RamLatClean contains pointers to three relations: the origin and new relation for input, and another new relation for output. In the generated RAM program, a statement RamLatClean will always be followed by a SWAP and a CLEAR, so as to replace the input new relation with the output new relation, e.g., line 83 to 85 in Appendix. A.0.3. The class RamLatNorm will contain two pointers to the input origin relation and the output origin relation. It will also be followed by a SWAP to swap them, e.g., line 97 to 98 in Appendix. A.0.3. The class RamLatticeGLB contains a vector of references to the bound lattice variables. Such reference is stored in a struct Ref_st. An example of RamLatticeGLB is at line 119 in Appendix. A.0.3.

4.3 RAM interpreter

There are several updates in RAM interpreter, which is written in file Interpreter.h and Interpreter.cpp. For interpreter, we do not need to add anything on static information such as RamLatticeAssociation and RamLatticeFunction, including RamLatticeUnaryFunction and RamLatticeBinaryFunction. We need to add evaluation in interpreter for other new RAM classes as mentioned in the last section.

The evaluation of RamQuestionMark is quite straightforward since we adopted RamCondition which exists in Soufflé. We reuse the interpreter context InterpreterContext from outer scope, so that it can access any other variables around the conditional operator(c ? e1 : e2). Furthermore, the type of return value is class RamValue as shown in Fig. 4.2, so that we can utilize the existed evaluation for all possible types of values in Soufflé.

The evaluation of RamLatticeFunctor, including RamLatticeUnaryFunctor and RamLatticeBinaryFunctor, is also simple. Each functor will have reference to its corresponding RamLatticeUnaryFunction or RamLatticeBinaryFunction. The visit in Soufflé interpreter to a functor will create a temporary InterpreterContext with only the input arguments, and iterate over the cases in the corresponding function using the context. It will check the RamCondition for each case. If the RamCondition is nullptr or evaluated to be true, the RamValue within the case will be evaluated and output as result. The evaluation of RamLatticeGLB is very similar to the RamLatticeFunctor. The only difference is
that, there could be more than two input arguments in a RamLatticeGLB, which can be resolved with traversing them and considering only two arguments at one time.

The algorithms for LATNORM and LATCLEAN are shown below. For LATNORM, the algorithm is intuitive. We traverse every cell in the origin relation $R_{\text{origin}}$, and compute the least upper bound lattice for lattices within the cell. Then we put the computed tuple into a temporary relation, which will be swapped with the origin relation immediately after the statement LATNORM. On the other hand, the algorithm for LATCLEAN has two inputs: origin relation and new relation. It will traverse every cell in the new relation $R_{\text{new}}$, and compute the least upper bound lattice considering tuples in both $R_{\text{new}}$ and $R_{\text{origin}}$. If the computed tuple does not belong to $R_{\text{origin}}$, it is a true new tuple, otherwise it is a "fake" new tuple and will be ignored.

**Algorithm 4 LATNORM algorithm**

1: input: $R_{\text{origin}}$, output: $R_{\text{temp}}$ ($R$ is a lattice relation)
2: for each cell $S_i$ in $R_{\text{origin}}$ do
3:     apply the least upper bound function $\sqcup$ to all tuples in $S_i$ in $R_{\text{origin}}$, get $t_i$
4:     insert $t_i$ into $R_{\text{temp}}$
5: end for

**Algorithm 5 LATCLEAN algorithm**

1: input: $R_{\text{origin}}$, $R_{\text{new}}$, output: $R_{\text{new}_\text{temp}}$ ($R$ is a lattice relation)
2: for each cell $S_i$ in $R_{\text{new}}$ do
3:     apply the least upper bound function $\sqcup$ to all tuples in $S_i$ in $R_{\text{origin}}$ and $R_{\text{new}}$, get the result $t_i$
4:     if $t_i \notin R_{\text{origin}}$ then
5:         insert $t_i$ into $R_{\text{new}_\text{temp}}$
6:     end if
7: end for
Chapter 5 | Experiments

This chapter presents the evaluation of Soufflé extended with lattice. We firstly describe the static analysis used in our experiments. Then we illustrate the application of the greatest lower bound function and least upper bound. And we discuss the effect of while loop in a program for analysis. Last but not least, we present the scalability of extended Soufflé on two static analyses which will benefit from the lattice framework: sign analysis and constant propagation analysis. We compare the scalability of Soufflé with the lattice scheme and Soufflé with the traditional powerset scheme, as well as the FLIX lattice framework. The test environment for the comparison of scalability is the ACI-B node, which is a high-performance computing (HPC) infrastructure at Penn State University. We use Basic node (Intel Xeon E5-2650v4 2.2GHz) with 8 cores and 10 GB RAM.

5.1 Description of Experiments

5.1.1 Sign Analysis

Sign Analysis is a simple data-flow analysis to determine the sign of variables within a program. The lattice of Sign Analysis is shown in Fig. 5.1. We only consider integer values in our analysis. And we assume there is at most one arithmetic operator on two variables in one line of an input program. The evaluation of each operator is shown in Table. 5.1.

The corresponding Datalog program for the extended Soufflé is shown in Lst. 5.1. We define five symbols in the lattice \texttt{Sign}, as well as the least upper bound function \texttt{lub} and the greatest lower bound function \texttt{glb}. The evaluation of arithmetic operators are represented by other binary case functions: \texttt{lat\_sum}, \texttt{lat\_minus}, \texttt{lat\_mult} and \texttt{lat\_div}. It can be noticed that the application of conditional operators greatly simplifies...
the declaration of functions.

Listing 5.1: Soufflé program for Sign Lattice Declaration

```plaintext
1 .enum Sign = {
2     case "Top",
3     case "Neg", case "Zer", case "Pos",
4     case "Bot"
5 };
6
7 /// The least upper bound relation on the lattice elements.
8 .def lub(x: Sign, y: Sign): Sign {
9     case ("Bot", _) => y,
10     case (_ , "Bot") => x,
11 };
12 .def lub(x: Sign, y: Sign): Sign {
13     case ("Bot", _) => y,
14     case (_ , "Bot") => x,
15 };
```
case (_, _) => x=y ? x : "Top"
}

/// The greatest lower bound relation on the lattice elements.
def glb(x: Sign, y: Sign): Sign {
case ("Top", _) => y,
case (_, "Top") => x,
case (_, _) => x=y ? x : "Bot"
}

def lat_sum(x: Sign, y: Sign): Sign {
case ("Bot", _) => "Bot",
case (_, "Bot") => "Bot",
case ("Zer", _) => y,
case (_, "Zer") => x,
case (_, _) => x=y ? x : "Top"
}

def lat_minus(x: Sign, y: Sign): Sign {
case ("Bot", _) => "Bot",
case (_, "Bot") => "Bot",
case ("Top", _) => "Top",
case (_, "Top") => "Top",
case (_, "Zer") => x,
case ("Zer", "Neg") => "Pos",
case ("Zer", "Pos") => "Neg",
case (_, _) => x=y ? "Top" : x
}

def lat_mult(x: Sign, y: Sign): Sign {
case ("Bot", _) => "Bot",
case (_, "Bot") => "Bot",
case ("Zer", _) => "Zer",
case (_, "Zer") => "Zer",
case ("Top", _) => "Top",
case ("Top", _) => "Top",
}
48     case (_, "Top") => "Top",
49     case (_, _)     => x=y ? "Pos" : "Neg"
50 }

52 .def lat_div(x: Sign, y: Sign): Sign {
53     case ("Bot", _) => "Bot",
54     case (_, "Bot") => "Bot",
55     case ("Zer", _) => "Zer",
56     case ("Top", _) => "Top",
57     case (_, "Top") => "Top",
58     case (_, _)     => x=y ? "Pos" : "Neg"
59 }

60 // assert lattice association
61 .let Sign<> = ("Bot", "Top", lub, glb)

5.1.2 Constant Propagation Analysis

The constant propagation analysis is a classical static analysis involving infinite lattice as shown in Fig. 5.2. Similar to Sign Analysis we only consider integer values. Theoretically, the lattice of it covers the infinite integer domain. We also assume there is at most one arithmetic operator in one line of a program. The evaluation of each operator is shown in Table. 5.2. The corresponding Datalog program for the extended Soufflé is shown in Lst. 5.2. For the division operator, the requirement that the divisor must be non-zero will be restricted in the Datalog rule for division operation, instead of being handled in the lat_div function.

Figure 5.2: Lattice of Constant Propagation Analysis

Listing 5.2: Soufflé program for Constant Propagation Lattice Declaration
Table 5.2: Evaluation of Operators in Constant Propagation Analysis

<table>
<thead>
<tr>
<th>Operator</th>
<th>⊥</th>
<th>x</th>
<th>⊤</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition(+)</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td></td>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td>Minus(-)</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td></td>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td>Multiplication(*)</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td></td>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td>Division(/)</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>⊥</td>
<td>⊤</td>
</tr>
<tr>
<td></td>
<td>⊤</td>
<td>⊥</td>
<td>⊤</td>
</tr>
</tbody>
</table>

```scala
def lub(x: Constant, y: Constant): Constant {
  case "Bot", _ => y,
  case _, "Bot" => x,
  case _, _ => x=y ? x : "Top"
}

def glb(x: Constant, y: Constant): Constant {
  case "Top", _ => y,
  case _, "Top" => x,
  case _, _ => x=y ? x : "Bot"
}

def lat_sum(x: Constant, y: Constant): Constant {
  case "Bot", _ => "Bot",
  case _, "Bot" => "Bot",
  case "Top", _ => "Top",
  case _, "Top" => "Top",
  case _, _ => x+y
}
26 .def lat_minus(x: Constant, y: Constant): Constant {
27     case ("Bot", _) => "Bot",
28     case (_, "Bot") => "Bot",
29     case ("Top", _) => "Top",
30     case (_, "Top") => "Top",
31     case (_, _) => x-y
32 }
33
34 .def lat_mult(x: Constant, y: Constant): Constant {
35     case ("Bot", _) => "Bot",
36     case (_, "Bot") => "Bot",
37     case ("Top", _) => "Top",
38     case (_, "Top") => "Top",
39     case (_, _) => x*y
40 }
41
42 .def lat_div(x: Constant, y: Constant): Constant {
43     case ("Bot", _) => "Bot",
44     case (_, "Bot") => "Bot",
45     case ("Top", _) => "Top",
46     case (_, "Top") => "Top",
47     case (_, _) => x/y
48 }
49
50 // assert lattice association
51 .let Const<> = ("Bot", "Top", lub, glb)

5.2 Greatest Lower Bound and Least Upper Bound

To test the functionality of Greatest Lower Bound and Least Upper Bound in a lattice, we use the Sign Lattice Declaration in Lst. 5.1, combined with relations and rules as shown in Lst. 5.3. Consider the first rule, the application of greatest lower bound function (glb) on three lattice elements – "Top", "Top", and "Pos" – will produce R("Pos"). Then, the second rule will add a new tuple T("Pos"). With the application of least upper bound function (lub) in RAM statement LATNORM, it will produce T("Top"). Thus, the
output for this program should be \( R("Pos") \), \( T("Top") \), which has been verified in our experiment.

Listing 5.3: Soufflé program for Lattice Function Test

```plaintext
1 .lat A(v: Sign)
2 .lat B(v: Sign)
3 .lat C(v: Sign)
4 .lat R(v: Sign)
5 .lat T(v: Sign)
6
7 .output R
8 .output T

10 A("Top").
11 B("Top").
12 C("Pos").
13 T("Neg").

15 R(x) :- A(x), B(x), C(x).
16 T(x) :- R(x).
```

### 5.3 While Loop

A significant advantage of using lattice is that it can handle while loop easily. Consider a simple program as shown below. We use the Constant Propagation Lattice Declaration in Lst. 5.2, combined with relations and rules as shown in Lst. 5.4. \texttt{varEntry(1, k, v)} indicates that at the start of line 1, the value of variable \( k \) is \( v \). Only after one iteration of the while loop, we will have \texttt{varEntry(4, "a", 1)} and \texttt{varEntry(4, "a", 2)}. With the application of least upper bound function (\texttt{lub}) on this cell, it will produce \texttt{varEntry(4, "a", "Top")}, and there will not be any other new tuples for the cell \texttt{varEntry(4, "a", _)} that are generated by the while loop in this program. This has been verified in our experiment.

```
0  a=1
1  b=1
2  WHILE (condition):
```
3     a=a+b
4     END WHILE

Listing 5.4: Soufflé program for Lattice While Loop Test
1 .decl setConstStm(l: number, r: symbol, c: number)    // r = c
2 .decl addStm(l: number, r: symbol, x: symbol, y: symbol) // r = x + y
3 .decl assignVar(l: number, r: symbol) // this statement assign r to a new value
4
5 .decl flow(l1: number, l2: number) // control flow from l1 to l2
6
7 .lat varEntry(l: number, k: symbol, v: Constant)
8 .output varEntry
9 .lat varExit(l: number, k: symbol, v: Constant)
10 .output varExit
11
12 setConstStm(0, "a", 1).
13 setConstStm(1, "b", 1).
14 addStm(3, "a", "a", "b").
15 flow(0, 1).
16 flow(1, 2).
17 flow(2, 3).
18 flow(3, 2).
19 flow(2, 4).
20
21 // if the statement doesn't assign to r
22 assignVar(l, r) :- setConstStm(l, r, _).
23 assignVar(l, r) :- addStm(l, r, _, _).
24
25 // varEntry of l2 is the union of {varExit(l1) | flow(l1, l2)}
26 varEntry(l2, k, v) :- varExit(l1, k, v), flow(l1, l2).
27
28 // statement: set to constant number
29 varExit(l, r, &lat_alpha(c)) :- setConstStm(l, r, c).
// addition statement r = x+y, and the value of x is v1, the
// value of y is v2
varExit(l, r, &lat_sum(v1, v2)) :- addStm(l, r, x, y),
    varEntry(l, x, v1),
    varEntry(l, y, v2).

// r is not re-assigned
varExit(l, r, v) :- varEntry(l, r, v), !assignVar(l, r).

On the other hand, Constant Propagation Analysis on the program is unpractical
with the traditional powerset scheme. In such a scheme, a new tuple for the cell
varEntry(4, "a", _) will be generated after each iteration. The evaluation of such a
Datalog program will not terminate until throwing an error on 'integer out of range'.

5.4 Scalability on Sign Analysis

To test the scalability of extended Soufflé on Sign Analysis, we use the Sign Lattice
Declaration in Lst. 5.1, combined with relations and rules for data-flow analysis as shown
in Lst. 5.5. Besides the relations and rules, we added a unary function lat_alpha to
transfer number to Sign lattice. The normal relation setConstStm, addStm, minusStm,
multStm, divStm represent 'assign constant statement', 'addition statement', 'minus
statement', "multiplication statement", "division statement" respectively. The lattice
relation varEntry and varExit represent the Sign lattice of variable at the start point
and end point of a line in the input program. It can be noticed that the requirement that
divisor must be non-zero for division operation is forced at line 47, in the rule for division
statement. That’s why zero value divisor is not handled within lat_div function.

We compare the lattice scheme in extended Soufflé with two other tools: the traditional
powerset scheme in Soufflé, and the lattice scheme in FLIX. The Soufflé program for
the former one is presented in Appendix. C, and the FLIX program is presented in
Appendix. D. With regard to the Soufflé program in the traditional powerset scheme,
the output is in the powerset style. To compare its output with other outputs, we can
simply add the Sign Lattice Declaration in Lst. 5.1, and extract the output in lattice
form with corresponding lattice relations and rules:

.lat varEntry(l:number, k: symbol, v: Sign)
.output varEntry

45
The input programs for analysis are generated by a random program generator as shown in Appendix B. The number of variables ($\text{totVar}$) is set to 5. The values in the probability vector ($\text{prob}$) are probability for selecting 'assign constant statement', 'addition statement', 'minus statement', 'multiplication statement', 'division statement', 'IF-ELSE statement', 'WHILE-LOOP statement' respectively in the process of generating next line of code. We generated two types of random programs: without and with branches. For random programs without branches, the probability vector is set to $[0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]$, where the probability for selecting "IF-ELSE statement" is 0. For random programs with branches, the probability vector is set to $[0.18, 0.18, 0.18, 0.18, 0.18, 0.18, 0.1, 0.0]$, where the probability for selecting "IF-ELSE statement" is 10%. The total lines of a generated program can be: 25, 50, 75, 100, 150 and 200. For a given total line, we generated 20 random programs as input. The mean and standard deviation of running time for the 20 programs is calculated, as shown in Fig. 5.3 and Fig. 5.4. The raw data is presented in Appendix E.

Listing 5.5: Soufflé program (lattice scheme) for Data-flow Analysis

```plaintext
1 // function to transfer number to enum type
2 .def lat_alpha(x: number): Sign {
3   case (_) => x>0 ? "Pos" : (x<0 ? "Neg" : "Zer")
4 }
5
6 .decl setConstStm(l:number, r: symbol, c: number) // r = c
7 .input setConstStm
8 .decl addStm(l:number, r: symbol, x: symbol, y: symbol) // r = x + y
9 .input addStm
10 .decl minusStm(l:number, r: symbol, x: symbol, y: symbol) // r = x - y
11 .input minusStm
12 .decl multStm(l:number, r: symbol, x: symbol, y: symbol) // r = x * y
13 .input multStm
```
.decl divStmt(l: number, r: symbol, x: symbol, y: symbol) // r = x / y
.input divStmt
// statement in line l assign r to a new value
.decl assignVar(l: number, r: symbol)

.decl flow(l1: number, l2: number) // control flow from l1 to l2
.input flow

.lat varEntry(l: number, k: symbol, v: Sign)
.output varEntry
.lat varExit(l: number, k: symbol, v: Sign)
.output varExit

// if the statement doesn’t assign to r
assignVar(l, r) :- setConstStm(l, r, _).
assignVar(l, r) :- addStm(l, r, _, _).
assignVar(l, r) :- minusStm(l, r, _, _).
assignVar(l, r) :- multStm(l, r, _, _).
assignVar(l, r) :- divStm(l, r, _, _).

// r is not re-assigned
varExit(l, r, v) :- varEntry(l, r, v), !assignVar(l, r).

// statement: set to constant number
varExit(l, r, &lat_alpha(c)) :- setConstStm(l, r, c).

// addition statement r = x+y, and the value of x is v1, the
// value of y is v2
varExit(l, r, &lat_sum(v1, v2)) :- addStm(l, r, x, y),
   varEntry(l, x, v1),
   varEntry(l, y, v2).

// minus statement: r = x - y
varExit(l, r, &lat_minus(v1, v2)) :- minusStm(l, r, x, y),
   varEntry(l, x, v1),
   varEntry(l, y, v2).
49 // multiplication statement: \( r = x \times y \)
50 varExit(l, r, &lat_mult(v1, v2)) :- multStm(l, r, x, y),
51                     varEntry(l, x, v1),
52                     varEntry(l, y, v2).
53 // division statement: \( r = x / y \)
54 varExit(l, r, &lat_div(v1, v2)) :- divStm(l, r, x, y),
55                     varEntry(l, x, v1),
56                     varEntry(l, y, v2), v2!=&lat_alpha(0).
57
58 // varEntry of 12 is the union of \{varExit(l1) \mid flow(l1, l2)\}
59 varEntry(12, k, v) :- varExit(l1, k, v), flow(l1, l2).

We present the scalability of different tools on Sign Analysis on random programs without branches in Fig. 5.3. It can be seen that the lattice scheme on Soufflé program has the exact same performance as the traditional powerset scheme on Soufflé program. The performance of FLIX program with the use of lattice is much slower than Soufflé. The running time of FLIX analysis for programs with 150 lines or more is longer than 1 hour, so these cases have not been tested. One of the reasons for such performance is the limitation of arguments for lattice relations in FLIX. In the version of FLIX we used (0.9.1), the lattice relation can only support up to 2 arguments, while there are 3 arguments in both lattice relations varEntry and varExit. Thus we concatenate the first two arguments into one string argument, and use several auxiliary functions to match the single argument with the real two arguments. Another reason is that, Soufflé uses symbol map to transfer string to number so as to improve efficiency, while FLIX does not. There are also many other optimizations in Soufflé on AST structure and RAM program that makes it faster than FLIX.

The benefit of using lattice in Sign Analysis on programs with branches is shown in Fig. 5.4. Similar to Fig. 5.3, the running time of FLIX analysis for random programs with 150 lines or more is longer than 1 hour and has not been tested. It can be noticed that for random programs with 200 lines of code, the lattice scheme on extended Soufflé is twice as fast as the traditional powerset scheme in Soufflé program. Another important observation is that the standard deviation of the traditional powerset scheme is large. That’s because for input programs with many branches, there can be many tuples within a cell, e.g., varEntry(6, "a", "Neg"), varEntry(6, "a", "Zer"), varEntry(6, "a", "Pos") and varEntry(6, "a", "Top"). This increases the size of relations. Consequently, the computational cost will grow and the running time can be
5.5 Scalability on Constant Propagation Analysis

To test the scalability of extended Soufflé on Constant Propagation Analysis, we use the Constant Propagation Lattice Declaration in Lst. 5.2, combined with relations and rules for data-flow analysis. The relations and rules in this analysis are almost the extremely long.
same as presented in Lst. 5.5, except that \texttt{Sign} is replaced by \texttt{Constant} in lattice relations \texttt{varEntry} and \texttt{varExit}, and the \texttt{lat\_alpha} function is different as shown below. The input programs for analysis are the same as the beforehand generated random program, as shown in Appendix. B. It is worth mentioning that the output of lattice scheme and traditional powerset scheme could be different. For example, given \texttt{varEntry(6, \textit{"a"}, 0)} and \texttt{varEntry(6, \textit{"a"}, 2)}, the lattice scheme will generate \texttt{varEntry(6, \textit{"a"}, \textit{"Top"})}. Hence given another tuple \texttt{varEntry(6, \textit{"b"}, 8)} and a division operation at line 6: \texttt{c=b/a}, we will get \texttt{varExit(6, \textit{"c"}, \textit{"Top"})}. However in traditional powerset scheme, the tuple \texttt{varEntry(6, \textit{"a"}, 0)} will be ignored by the rule, and we will get \texttt{varExit(6, \textit{"c"}, 4)} instead of \texttt{varExit(6, \textit{"c"}, \textit{"Top"})}. Such difference is considered acceptable here.

```python
.def lat\_alpha(x: number): Constant {
    case (_) => x
}
```

Similar to Sign Analysis, we compare the lattice scheme in extended Soufflé with two other tools: the traditional powerset scheme in Soufflé (Appendix. C), and the lattice scheme in FLIX (Appendix. D). The performances of different tools on Constant Propagation Analysis on random programs \textbf{without branches} is shown in Fig. 5.5 (with raw data in Appendix. E), in both linear scale and log-linear scale. Similarly, the running time of FLIX analysis for random programs with 150 lines or more is longer than 1 hour and has not been tested. It can be seen that the lattice scheme on Soufflé \texttt{program} is slightly slower than the traditional powerset scheme on Soufflé \texttt{program}, due to the extra computational cost of \texttt{LATCLEAN} and \texttt{LATNORM}. For random programs with 200 lines of code, this extra cost is 37\% on running time, which is relatively small and acceptable.

The performances of different tools on Constant Propagation Analysis on random programs \textbf{with branches} is shown in Fig. 5.5 (with raw data in Appendix. E), in both linear scale and log-linear scale. The running time of FLIX analyses for input programs with 150 and 200 lines is longer than 1 hour and has not been tested. Similarly, the traditional powerset analyses on extended Soufflé for programs with 100 or more lines also exceed 1 hour running time and have not been tested. From this figure, it can be noticed that comparing the lattice scheme on extended Soufflé with the powerset scheme on Soufflé, the former one runs faster for 25 lines of input programs, while the latter one is more efficient for 50 or more lines of input programs. The lattice scheme on extended Soufflé scales well with the size of input programs, while the powerset scheme does not
scale well. For random programs with 75 lines of code, the running time of the lattice scheme on extended Soufflé is 887% of that of the powerset scheme on extended Soufflé, and 532% of that of lattice scheme on FLIX. It’s worth mentioning that the lattice scheme on FLIX also scales better than the powerset scheme on Soufflé and outperforms it for 75 lines of input programs, which proves the advantage of lattice scheme for Constant Propagation Analysis.

Figure 5.5: Performances on Constant Propagation Analysis without Branches
Figure 5.6: Performances on Constant Propagation Analysis with Branches

(a) Linear Scale

(b) Log-linear scale
Chapter 6  
Summary and Future Works

In this thesis, we compare two variants of Datalog called Soufflé and FLIX. We present the framework of lattice we used to extend Soufflé with lattice, as well as the implementation of this extension, which has been put on Github: [https://github.com/QXG2/souffle](https://github.com/QXG2/souffle). We also present and discuss the experiments to show the functionality of the lattice feature, and the scalability of the lattice scheme of extended Soufflé on Sign Analysis and Constant Propagation Analysis.

For some large or infinite lattice such as constant propagation lattice, the traditional powerset scheme could bring high computational cost, which can be significantly reduced by utilizing the lattice scheme. We design a lattice scheme for Soufflé, following the lattice scheme by Madsen, Yee and Lhoták [37] in 2016 with some modification. We use unary and binary case functions for the specialized use of the lattice scheme. And we add conditional operator which is a syntactic sugar and greatly reduces the codes for case functions. We add lattice declaration and lattice association, as well as two statements within the RAM program for lattice: `LATCLEAN` and `LATNORM`.

We use two static analyses in our experiments: Sign Analysis and Constant Propagation Analysis. We present the use of the greatest lower bound function and least upper bound function for Soufflé programs with lattice. We discuss the advantage of the lattice scheme for input programs with While-loop. We test the scalability of the two static analyses on three tools: lattice scheme on extended Soufflé, transitional powerset scheme on Soufflé and lattice scheme on FLIX. The results of the experiment have shown that the lattice scheme on extended Soufflé, while brings small extra computational cost, significantly outperforms the traditional powerset scheme on normal input programs with branches.

There could be more work to do with the lattice scheme on extended Soufflé. The first is to add the lattice scheme to compiler mode using template-based meta-programming
techniques [32] which is adopted in Soufflé. There is a lot of work to do on the translation from RAM program to C++ program, especially on the lattice declaration and unary and binary case functions. Also, Soufflé used to apply MPI for large scale computation in C++ program in previous versions. Although it has been abandoned in the later version (1.7.0), the use of MPI would be a great feature and improve the scalability of Soufflé. Since the lattice scheme has not brought extra database operations such as update or delete, it is feasible to include MPI and lattice scheme in Soufflé compiler mode without conflict.

The second is to reduce the extra computational cost brought by the lattice scheme. Instead of use \texttt{LATCLEAN} on new relations in each iteration of Semi-Naïve evaluation, we could reduce the usage of \texttt{LATCLEAN}. For example, we can use \texttt{LATCLEAN} once every three iterations. The evaluation can still converge and the computation cost for each iteration will be decreased. However, the number of iterations to reach a fixpoint may be increased. So there should be a trade-off. Another possible optimization is to insert \texttt{LATNORM} within the loop of Semi-Naïve evaluation. The benefit is that the size of the relation can be reduced during the evaluation. There could be several tuples within one cell for a lattice relation (especially after the reduction of usage of \texttt{LATCLEAN}). The insertion of \texttt{LATNORM} operation will reduce the size of lattice relation and avoid redundant computation. Similarly, there is a trade-off since \texttt{LATNORM} will bring extra computational cost and harm the concurrency.
Appendix A
RAM Program by Soufflé

A.0.1 Example Program

\[
a(x) :- b(x), d(x).
b(x) :- a(x), d(x).
c(x) :- a(x), b(x).
\]

A.0.2 RAM for Example Program without Lattice

1 DECLARATION
2 @delta_a(x) @delta_b(x) @lat_temp_a(x) @lat_temp_b(x) @lat_temp_c(x)
   @lat_temp_d(x) @new_a(x) @new_b(x) a(x) b(x) c(x) d(x)
3 END DECLARATION
4 PROGRAM
5 BEGIN_STRATUM_0
6   CREATE d(x) ;
7   BEGIN_DEBUG "d(1). in file /home/gq/Programming/souffle/dataflowTest/
                  Example_test/example.dl [10:1-10:6]"
8     INSERT (number(1)) INTO d
9   END_DEBUG
10  END_STRATUM_0;
11 BEGIN_STRATUM_1
12   CREATE a(x) ;
13   CREATE @delta_a(x) ;
14   CREATE @new_a(x) ;
15   CREATE b(x) ;
CREATE @delta_b(x) ;
CREATE @new_b(x) ;
BEGIN_DEBUG "a(1). in file /home/gq/Programming/souffle/dataflowTest
/Example_test/example.dl [8:1-8:6]"
INSERT (number(1)) INTO a
END_DEBUG;
MERGE @delta_a WITH a;
BEGIN_DEBUG "b(1). in file /home/gq/Programming/souffle/dataflowTest
/Example_test/example.dl [9:1-9:6]"
INSERT (number(1)) INTO b
END_DEBUG;
MERGE @delta_b WITH b;
LOOP
  PARALLEL
  BEGIN_DEBUG "a(x) :- b(x), d(x). in file /home/gq/
  Programming/souffle/dataflowTest/Example_test/example.dl
  [12:1-12:20]"
  INSERT
    for t0 in @delta_b {
      for t1 in d {
        IF (not (t0.x) ∈ a) {
          IF (t0.x = t1.x) {
            PROJECT (t0.x) INTO @new_a
          }
        }
      }
    }
  END_DEBUG;
BEGIN_DEBUG "b(x) :- a(x), d(x).in file /home/gq/
Programming/souffle/dataflowTest/Example_test/example.dl
INSERT
  for t0 in @delta_a {

for t1 in d {
    IF (not (t0.x ∈ b)) {
        IF (t0.x = t1.x) {
            PROJECT (t0.x) INTO
                @new_b
                
        }
    }
}

END_DEBUG END PARALLEL;
EXIT (@new_a = ∅) and (@new_b = ∅));
MERGE a WITH @new_a;
SWAP (@delta_a, @new_a);
CLEAR @new_a;
MERGE b WITH @new_b;
SWAP (@delta_b, @new_b);
CLEAR @new_b

END LOOP;
DROP @delta_a;
DROP @new_a;
DROP @delta_b;
DROP @new_b;
STORE DATA FOR a TO {{{"IO","file"},{"attributeNames","x"},{"filename","./a.csv"},{"name","a"}}};
STORE DATA FOR b TO {{{"IO","file"},{"attributeNames","x"},{"filename","./b.csv"},{"name","b"}}};

DROP d
END_STRATUM_1;
BEGIN_STRATUM_2
CREATE c(x) ;
BEGIN_DEBUG "c(x) :- a(x), b(x). in file /home/gq/Programming/souffle/dataflowTest/Example_test/example.dl [14:1-14:20]"
INSERT
    for t0 in a {

for t1 in b {
    IF (t0.x = t1.x) {
        PROJECT (t0.x) INTO c
    }
}

END_DEBUG;
STORE DATA FOR c TO {{{"IO","file"},{"attributeNames","x"},{"filename","./c.csv"},{"name","c"}}};
DROP a;
DROP b
END_STRATUM_2
END_PROGRAM

A.0.3 RAM for Example Program with Lattice

DECLARATION
@delta_a(x) @delta_b(x) @lat_temp_a(x) @lat_temp_b(x) @lat_temp_c(x)
    @lat_temp_d(x) @new_a(x) @new_b(x) @new_lat_a(x) @new_lat_b(x)
    @org_lat_a(x) @org_lat_b(x) a(x) b(x) c(x) d(x)

LATTICE ASSOCIATION DEFINITION.
lub:
LATTICE BINARY FUNCTION
size: 5
match:(argument(0) = number(2147418114)), output:argument(1)
match:(argument(1) = number(2147418114)), output:argument(0)
match:((argument(0) = number(2147418112)) and (argument(1) = number (2147418112))), output:number(2147418112)
match:((argument(0) = number(2147418113)) and (argument(1) = number (2147418113))), output:number(2147418113)
match:True, output:number(2147418111)

egl:
LATTICE BINARY FUNCTION
size: 5
match:(argument(0) = number(2147418111)), output:argument(1)
match:(argument(1) = number(2147418111)), output:argument(0)
match:((argument(0) = number(2147418112)) and (argument(1) = number(2147418112))), output:number(2147418112)
match:((argument(0) = number(2147418113)) and (argument(1) = number(2147418113))), output:number(2147418113)
match:True, output:number(2147418114)
END DECLARATION
PROGRAM
BEGIN_STRATUM_0
CREATE d(x) ;
BEGIN_DEBUG "d("no")\n. in file /home/gq/Programming/souffle/dataflowTest/Example_lat_test/example.dl [37:1-37:9]"
    INSERT (number(2147418113)) INTO d
END_DEBUG;
CREATE @lat_temp_d(x) ;
LATNORM d INTO @lat_temp_d;
SWAP (d, @lat_temp_d);
DROP @lat_temp_d
END_STRATUM_0;
BEGIN_STRATUM_1
CREATE a(x) ;
CREATE @delta_a(x) ;
CREATE @new_a(x) ;
CREATE @org_lat_a(x) ;
CREATE @new_lat_a(x) ;
CREATE b(x) ;
CREATE @delta_b(x) ;
CREATE @new_b(x) ;
CREATE @org_lat_b(x) ;
CREATE @new_lat_b(x) ;
BEGIN_DEBUG "a("yes")\n. in file /home/gq/Programming/souffle/dataflowTest/Example_lat_test/example.dl [35:1-35:10]"
    INSERT (number(2147418112)) INTO a
END_DEBUG;
CREATE @lat_temp_a(x) ;
LATNORM a INTO @lat_temp_a;
SWAP (a, @lat_temp_a);
DROP @lat_temp_a;
MERGE @delta_a WITH a;
BEGIN_DEBUG "b("no"). in file /home/gq/Programming/souffle/dataflowTest/Example_lat_test/example.dl [36:1-36:9]"
    INSERT (number(2147418113)) INTO b
END_DEBUG;
CREATE @lat_temp_b(x) ;
LATNORM b INTO @lat_temp_b;
SWAP (b, @lat_temp_b);
DROP @lat_temp_b;
MERGE @delta_b WITH b;
LOOP
    PARALLEL
        BEGIN_DEBUG "a(x) :- b(x), d(x). in file /home/gq/Programming/souffle/dataflowTest/Example_lat_test/example.dl [39:1-39:20]"
            INSERT
                for t0 in @delta_b {
                    for t1 in d {
                        IF (not (glb( t0.x, t1.x )) ∈ a) {
                            PROJECT (glb( t0.x, t1.x ))
                                INTO @new_a
                        }
                    }
                }
        END_DEBUG;
        BEGIN_DEBUG "b(x) :- a(x), d(x). in file /home/gq/Programming/souffle/dataflowTest/Example_lat_test/example.dl [40:1-40:20]"
            INSERT
                for t0 in @delta_a {
                    for t1 in d {

IF (not (glb( t0.x, t1.x )) ∈ b) {
    PROJECT (glb( t0.x, t1.x ))
    INTO @new_b
}
END_DEBUG END PARALLEL;
LATCLEAN @new_a INTO @new_lat_a USING a;
SWAP (@new_a, @new_lat_a);
CLEAR @new_lat_a;
LATCLEAN @new_b INTO @new_lat_b USING b;
SWAP (@new_b, @new_lat_b);
CLEAR @new_lat_b;
EXIT ((@new_a = ∅) and (@new_b = ∅));
MERGE a WITH @new_a;
SWAP (@delta_a, @new_a);
CLEAR @new_a;
MERGE b WITH @new_b;
SWAP (@delta_b, @new_b);
CLEAR @new_b
END LOOP;
LATNORM a INTO @org_lat_a;
SWAP (a, @org_lat_a);
DROP @delta_a;
DROP @new_a;
DROP @org_lat_a;
DROP @new_lat_a;
LATNORM b INTO @org_lat_b;
SWAP (b, @org_lat_b);
DROP @delta_b;
DROP @new_b;
DROP @org_lat_b;
DROP @new_lat_b;
STORE DATA FOR a TO {{{"IO","file"},"attributeNames","x"},"
filename", ".a.csv"],{"name","a"}}};

STORE DATA FOR b TO {{{"IO","file"},{"attributeNames","x"},{"filename", ".b.csv"},{"name","b"}}};

DROP d

END_STRATUM_1;

BEGIN_STRATUM_2

CREATE c(x);

BEGIN_DEBUG "c(x) :- a(x), b(x). in file /home/gq/Programming/souffle/dataflowTest/Example_lat_test/example.dl [41:1-41:20]"

INSERT

for t0 in a {
    for t1 in b {
        PROJECT (glb( t0.x, t1.x )) INTO c
    }
}

END_DEBUG;

CREATE @lat_temp_c(x);
LATNORM c INTO @lat_temp_c;
SWAP (c, @lat_temp_c);
DROP @lat_temp_c;
STORE DATA FOR c TO {{{"IO","file"},{"attributeNames","x"},{"filename", ".c.csv"},{"name","c"}}};

DROP a;
DROP b

END_STRATUM_2

END_PROGRAM
Appendix B
Python Script for Random While-Program Generator and Sample Codes

B.1 Python Script

```python
import numpy as np
import random

VarMap='abcdefghijklmnopqrstuvwxyz'

class Program:
    # pi is probability of a statement, p0:x=c, p1:x=y+z,
    # p2:x=y-z, p3:x=y*z, p4:x=y/z
    # p5: if [cond] then S1 else S2, p6: while [cond] do S
    def __init__(self, seed, prob, totVar, minLines):
        for p in prob:
            assert p>=0
            assert abs(sum(prob)-1.0)<1e-6

        self.totVar = totVar
        assert self.totVar>0
        assert self.totVar<1000 # size of VarMap
```
self.minLines = minLines
# Must have more lines than initialization of variables
assert self.minLines>self.totVar

self.curLine = 0

self.thres=[sum(prob[: i+1]) for i in range(len(prob))]

self.setFacts=[] # x=c
self.AddFacts=[] # x=y+z
self.MinusFacts=[] # x=y−z
self.MultFacts=[] # x=y*z
self.DivFacts=[] # x=y/z
self.Flow=[] # control flow from l1 to l2

self.Code = "" # Code of the program

self.seed = seed
return

# initialize all variables in the generated program
def initVars(self):
    for i in range(self.totVar):
        random.seed(self.seed)
        self.increSeed()
        value = random.randint(-2, 2)

        self.setFacts.append((i, VarMap[i], value))
        self.insertLineWithLable(VarMap[i] + '=' + str(value),0)
        self.addFlow((i, i+1))
        self.increLine() # start a new line

def increLine(self):
    self.curLine += 1
```python
    self.Code += '\n'
    return

    # randomly select 3 variables and put in a tuple
    def randThreeVars(self):
        random.seed(self.seed)
        self.increSeed()
        s1 = VarMap[random.randint(0, self.totVar-1)]
        random.seed(self.seed)
        self.increSeed()
        s2 = VarMap[random.randint(0, self.totVar-1)]
        random.seed(self.seed)
        self.increSeed()
        s3 = VarMap[random.randint(0, self.totVar-1)]
        return (s1, s2, s3)

    def addSet(self, level):
        random.seed(self.seed)
        self.increSeed()
        varInd = random.randint(0, self.totVar-1)
        random.seed(self.seed)
        self.increSeed()
        value = random.randint(-2, 2)  # -20, 20
        self.setFacts.append((self.curLine, VarMap[varInd], value))
        self.insertLineWithLable(VarMap[varInd]+"="+str(value), level)
        return

    def addAdd(self, level):
        varTuple = self.randThreeVars()
        t = (self.curLine,) + varTuple
        self.AddFacts.append(t)
```
self.insertLineWithLable(varTuple[0]+"="+varTuple[1]"
"++"+varTuple[2], level)

return

def addMinus(self, level):
    varTuple = self.randThreeVars()
    t = (self.curLine, ) + varTuple
    self.MinusFacts.append(t)
    self.insertLineWithLable(varTuple[0]+"="+varTuple[1]"
    + "−" + varTuple[2], level )
    return

def addMult(self, level):
    varTuple = self.randThreeVars()
    t = (self.curLine, ) + varTuple
    self.MultFacts.append(t)
    self.insertLineWithLable(varTuple[0]+"="+varTuple[1]"
    + "*" + varTuple[2], level )
    return

def addDiv(self, level):
    varTuple = self.randThreeVars()
    t = (self.curLine, ) + varTuple
    self.DivFacts.append(t)
    self.insertLineWithLable(varTuple[0]+"="+varTuple[1]"
    + "/" + varTuple[2], level )
    return

def addFlow(self, fl):
    self.Flow.append(fl)
    return

def insertLineWithLable(self, string, level):
    self.Code += str(self.curLine)+
(5-len(str(self.curLine)))*" " + "\t"*level
self.Code += string

def insertLineWithoutLabel(self, string, level):
    self.Code += " "*5 + '\t'*level
    self.Code += string

def increSeed(self):
    random.seed(self.seed)
    self.seed = random.randint(-9999999999, 9999999999)

def printAll(self):
    print('total number of Variables: ', self.totVar)
    print('minimum lines: ', self.minLines)
    print('current line: ', self.curLine)
    print('setFacts: ', self.setFacts)
    print('AddFacts: ', self.AddFacts)
    print('MinusFacts: ', self.MinusFacts)
    print('MultFacts: ', self.MultFacts)
    print('Flow: ', self.Flow)
    print('Code:')
    print(self.Code)

# totVar: total number of variables
# minL: least lines required
# level: scope level of current statement
def buildIfElse(prog, minL, level):
    random.seed(prog.seed)
    prog.increSeed()
    S1_L = random.randint(0, minL-2)
    S2_L = minL-2-S1_L
    if_label = prog.curLine
    prog.insertLineWithLabel('IF ( condition): ', level)
prog.increLine()
S1_start = prog.curLine
if (S1_L!=0):
    count1 = buildStatement(prog, S1_L, level+1)
else:
    count1 = 1
    prog.insertLineWithLable("", level+1)
S1_end = prog.curLine

prog.increLine()
prog.insertLineWithoutLable('ELSE:\n', level)
S2_start = prog.curLine
if (S2_L!=0):
    count2 = buildStatement(prog, S2_L, level+1)
else:
    count2 = 1
    prog.insertLineWithLable("", level+1)
S2_end = prog.curLine

# start a new line
prog.increLine()
prog.insertLineWithLable('End IF', level)

# connect IF to starts of S1 and S2
prog.addFlow((if_label, S1_start))
prog.addFlow((if_label, S2_start))

# connect ends of S1 and S2 to new line
prog.addFlow((S1_end, prog.curLine))
prog.addFlow((S2_end, prog.curLine))

return 2+count1+count2

def buildWhile(prog, minL, level):
while_label = prog.curLine
prog.insertLineWithLable("While (condition): ", level)

prog.increLine()
S_start = prog.curLine
count = buildStatement(prog, minL-2, level+1)
S_end = prog.curLine

# start a new line
prog.increLine()
prog.insertLineWithLable('End While', level)

prog.addFlow((while_label, S_start))
prog.addFlow((S_end, while_label))

prog.addFlow((while_label, prog.curLine))

return 2+count

def buildStatement(prog, minL, level):
    count = 0
    if (count < minL):
        # print("count: ", count, ", minL: ", minL)
        random.seed(prog.seed)
        prog.increSeed()
        r = random.random()
        if (r<prog.thres[0]):
            prog.addSet(level)
            count += 1
        elif (r<prog.thres[1]):
            prog.addAdd(level)
            count += 1
        elif (r<prog.thres[2]):
            prog.addMinus(level)
count += 1

elif (r<prog.thres[3]):
    prog.addMult(level)
    count += 1

elif (r<prog.thres[4]):
    prog.addDiv(level)
    count += 1

elif (r<prog.thres[5]):
    random.seed(prog.seed)
    prog.increSeed()
    n = random.randint(4, max(4, min(30, minL)))
    count += buildIfElse(prog, n, level)

else:
    random.seed(prog.seed)
    prog.increSeed()
    n = random.randint(3, max(3, min(30, minL)))
    count += buildWhile(prog, n, level)

    # start a new line
    if (count<minL):
        prog.addFlow((prog.curLine, prog.curLine+1))
        prog.increLine()
        count+=buildStatement(prog,minL−count, level)

return count

# p0:x=c, p1:x=y+z, p2:x=y−z, p3:x=y*z, p4:x=y/z
# p5:if [cond] then S1 else S2, p6:while [cond] do S
# Notice: to test no-lattice souffle, must p6 to zero
myProg = Program(9527, [0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2,0.2], 5, 50)
myProg.initVars()
buildStatement(myProg, myProg.minLines−myProg.curLine, 0)
myProg.printAll()
B.2 Sample Codes Generated

B.2.1 Sample Code without Branches

0  a=0
1  b=-1
2  c=1
3  d=0
4  e=-2
5  b=e+c
6  c=a+a
7  c=d-a
8  e=a+b
9  c=b*a
10 e=e/d
11 e=0
12 e=c-d
13 d=b+b
14 a=e/e
15 d=d/d
16 e=c+c
17 e=c/b
18 b=a-b
19 c=a+b
20 a=a*e
21 b=b-d
22 c=0
23 d=e+e
24 e=d/e
25 c=d-c
26 d=d*e
27 b=a-b
28 d=c/a
29 b=a+c
30 d=b/a
B.2.2 Sample Code with Branches
14 a=e/e
15 IF (condition):
16 b=e-c
17 e=e*c
18 d=b*a
19 b=c+a
20 e=a+a
21 e=b/b
22 b=c-c
23 d=e+e
24 e=d/e
25 c=d*c
26 d=d*e
27 b=a-b
28 d=c/a
ELSE:
29 b=a+c
30 IF (condition):
31 ELSE:
32 d=-1
33 c=b+d
34 End IF
35 End IF
36 c=c+d
37 b=a*c
38 d=0
39 a=e+d
40 c=a*c
41 c=d*d
42 a=d+e
43 c=b/b
44 d=e+a
45 e=a/a
46 d=2
47  c = -2
48  a = e + a
49  e = d - d
Appendix C
Soufflé Powerset Programs in Experiments

C.1 Sign Analysis

```plaintext
// use function to transfer number to symbol
.def nolat_alpha(x: number): symbol {
    case (_) => x>0 ? "Pos" : (x<0 ? "Neg" : "Zer")
}

// sum
.def nolat_sum(x: symbol, y: symbol): symbol {
    case ("Bot", _) => "Bot",
    case (_, "Bot") => "Bot",
    case ("Zer", _) => y,
    case (_, "Zer") => x,
    case (_, _) => x=y ? x : "Top"
}

// minus
.def nolat_minus(x: symbol, y: symbol): symbol {
    case ("Bot", _) => "Bot",
    case (_, "Bot") => "Bot",
    case ("Top", _) => "Top",
    case (_, "Top") => "Top",
```
case (_, "Zer") => x,
case ("Zer", "Neg") => "Pos",
case ("Zer", "Pos") => "Neg",
case (_, _) => x=y ? "Top" : x
)

// multiplication
.def nolat_mult(x: symbol, y: symbol): symbol {
case ("Bot", _) => "Bot",
case (_, "Bot") => "Bot",
case ("Zer", _) => "Zer",
case (_, "Zer") => "Zer",
case ("Top", _) => "Top",
case (_, "Top") => "Top",
case (_, _) => x=y ? "Pos" : "Neg"
}

// division
.def nolat_div(x: symbol, y: symbol): symbol {
case ("Bot", _) => "Bot",
case (_, "Bot") => "Bot",
case ("Zer", _) => "Zer",
case ("Top", _) => "Top",
case (_, "Top") => "Top",
case (_, _) => x=y ? "Pos" : "Neg"
}

// assert lattice association
.let Sign<> = ("Bot", "Top", lub, glb)

decl setConstStm(l: number, r: symbol, c: number) // r = c
.input setConstStm
decl addStm(l: number, r: symbol, x: symbol, y: symbol) // r = x + y
.input addStm
decl minusStm(l: number, r: symbol, x: symbol, y: symbol) // r = x - y

76
.input minusStm
.decl multStm(l:number, r: symbol, x: symbol, y: symbol) // r = x * y
.input multStm
.decl divStm(l:number, r: symbol, x: symbol, y: symbol) // r = x / y
.input divStm
.decl assignVar(l:number, r: symbol) // this statement assign r to a new value

.decl flow(l1: number, l2: number) // control flow from l1 to l2
.input flow

// intermediate relations for all possible values of each variable
.decl varEntry_symbol(l:number, k: symbol, v: symbol)
.output varEntry_symbol
.decl varExit_symbol(l:number, k: symbol, v: symbol)
.output varExit_symbol

// if the statement doesn’t assign to r
assignVar(l, r) :- setConstStm(l, r, _).
assignVar(l, r) :- addStm(l, r, _, _).
assignVar(l, r) :- minusStm(l, r, _, _).
assignVar(l, r) :- multStm(l, r, _, _).
assignVar(l, r) :- divStm(l, r, _, _).

// varEntry of l2 is the union of {varExit(l1) | flow(l1,l2)}
varEntry_symbol(l2, k, v) :- varExit_symbol(l1, k, v), flow(l1, l2).

// statement: set to constant number
varExit_symbol(l, r, &nolat_alpha(c)) :- setConstStm(l, r, c).

// addition statement r = x+y, and the value of x is v1, the value of y is v2
varExit_symbol(l, r, &nolat_sum(v1, v2)) :- addStm(l, r, x, y),
            varEntry_symbol(l, x, v1),
            varEntry_symbol(l, y, v2).
90  // minus statement: r = x - y
91  varExit_symbol(l, r, &nolat_minus(v1, v2)) :- minusStm(l, r, x, y),
92                      varEntry_symbol(l, x, v1),
93                      varEntry_symbol(l, y, v2).
94  // multiplication statement: r = x * y
95  varExit_symbol(l, r, &nolat_mult(v1, v2)) :- multStm(l, r, x, y),
96                      varEntry_symbol(l, x, v1),
97                      varEntry_symbol(l, y, v2).
98  // division statement: r = x / y
99  varExit_symbol(l, r, &nolat_div(v1, v2)) :- divStm(l, r, x, y),
100                       varEntry_symbol(l, x, v1),
101                       varEntry_symbol(l, y, v2),
102                       v2!="Zer".
103
104  // r is not re-assigned
105  varExit_symbol(l, r, v) :- varEntry_symbol(l, r, v), !assignVar(l, r).

C.2 Constant Propagation Analysis

1  // input relations of statements
2  .decl setConstStm(l:number, r: symbol, c: number)      // r = c
3  .input setConstStm
4  .decl increStm(l:number, r: symbol)                  // r++
5  .input increStm
6  .decl addStm(l:number, r: symbol, x: symbol, y: symbol) // r = x + y
7  .input addStm
8  .decl minusStm(l:number, r: symbol, x: symbol, y: symbol) // r = x - y
9  .input minusStm
10  .decl multStm(l:number, r: symbol, x: symbol, y: symbol) // r = x * y
11  .input multStm
12  .decl divStm(l:number, r: symbol, x: symbol, y: symbol) // r = x / y
13  .input divStm
14  .decl assignVar(l:number, r: symbol) // this statement assign r to a new
15  value
// control flow from l1 to l2
.decl flow(l1: number, l2: number)
.input flow

// intermediate relations for all possible values of each variable
.decl varEntry_num(l:number, k: symbol, v: number)
.output varEntry_num
.decl varExit_num(l:number, k: symbol, v: number)
.output varExit_num

// all assignment statements
assignVar(l, r) :- setConstStm(l, r, _).
assignVar(l, r) :- addStm(l, r, _, _).
assignVar(l, r) :- minusStm(l, r, _, _).
assignVar(l, r) :- multStm(l, r, _, _).
assignVar(l, r) :- divStm(l, r, _, _).

// varEntry of l2 is the union of {varExit(l1) | flow(l1,l2)}
varEntry_num(l2, k, v) :- varExit_num(l1, k, v), flow(l1, l2).

// statement: set to constant number
varExit_num(l, r, c) :- setConstStm(l, r, c).

// addition statement r = x+y, and the value of x is v1, the
// value of y is v2
varExit_num(l, r, v1+v2) :- addStm(l, r, x, y),
                                 varEntry_num(l, x, v1),
                                 varEntry_num(l, y, v2).

// minus statement: r = x - y
varExit_num(l, r, v1-v2) :- minusStm(l, r, x, y),
                          varEntry_num(l, x, v1),
                          varEntry_num(l, y, v2).

// multiplication statement: r = x * y
varExit_num(l, r, v1*v2) :- multStm(l, r, x, y),
                          varEntry_num(l, x, v1),
varEntry_num(l, y, v2).

// division statement: r = x / y
// can not handle division correctly
varExit_num(l, r, v1/v2) :- divStm(l, r, x, y),
varEntry_num(l, x, v1),
varEntry_num(l, y, v2),
v2!=0.

// r is not re-assigned
varExit_num(l, r, v) :- varEntry_num(l, r, v), !assignVar(l, r).
Appendix D
Flix Programs in Experiments

D.1 Sign Analysis

1 enum Sign {
2     case Top,
3
4     case Neg, case Zer, case Pos,
5
6     case Bot
7 }
8
9 // statements in the test code
10 rel SetConstStm(l: Str, r: Str, c: Int) // r = c
11 rel AddStm(l: Str, r: Str, x: Str, y: Str) // r = x + y
12 rel MinusStm(l: Str, r: Str, x: Str, y: Str) // r = x - y
13 rel MultStm(l: Str, r: Str, x: Str, y: Str) // r = x * y
14 rel DivStm(l: Str, r: Str, x: Str, y: Str) // r = x / y
15
16 rel Flow(l1: Str, l2: Str) // flow from l1 to l2
17
18 // intermediate relations
19 rel AssignVar(l: Str, r: Str) // variable r is re-assigned at lable l
20 rel NonAssign(l: Str) // there is not assignment at lable l
21
22 // result of the analysis.
lat VarEntry(key: Str, v: Sign)
lat VarExit(key: Str, v: Sign)

// Define equality of two lattice elements by structural equality.
//
def equ(e1: Sign, e2: Sign): Bool = e1 == e2

// Define the partial order on Constant.
// This is straightforward with pattern matching on the arguments.
//
def leq(e1: Sign, e2: Sign): Bool = match (e1, e2) with {
    case (Bot, _) => true
    case (Zer, Zer) => true
    case (Zer, Neg) => true
    case (Zer, Pos) => true
    case (Neg, Neg) => true
    case (Pos, Pos) => true
    case (_, Top) => true
    case _ => false
}

// Least upper bound
//
def lub(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
    case (Bot, x) => x
    case (x, Bot) => x
    case (Zer, Zer) => Zer
    case (Zer, Neg) => Neg
    case (Neg, Zer) => Neg
    case (Zer, Pos) => Pos
    case (Pos, Zer) => Pos
case (Neg, Neg) => Neg
case (Pos, Pos) => Pos
  case _ => Top
}

// Greatest lower bound

// def glb(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
  case (Top, x) => x
  case (x, Top) => x
  case (Zer, Zer) => Zer
  case (Neg, Neg) => Neg
  case (Pos, Pos) => Pos
  case (Zer, Neg) => Zer
  case (Neg, Zer) => Zer
  case (Zer, Pos) => Zer
  case (Pos, Zer) => Zer
  case (Neg, Pos) => Zer
  case (Pos, Neg) => Zer
  case (x, Top) => x
  case (Top, x) => x
  case _ => Bot
}

// This complete the specification of the lattice. Associate the
// lattice components with the Constant type in the following way

let Sign<> = (Bot, Top, equ, leq, lub, glb)

/// Abstracts a concrete number into the sign domain.

///
pub def alpha(i: Int32): Sign = switch {
    case i < 0 => Neg
    case i > 0 => Pos
    case true => Zer
}

/// Over-approximates integer ‘addition‘.

pub def plus(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
    case (Bot, _) => Bot
    case (_, Bot) => Bot
    case (Neg, Neg) => Neg
    case (Neg, Zer) => Neg
    case (Neg, Pos) => Top
    case (Zer, Neg) => Neg
    case (Zer, Zer) => Zer
    case (Zer, Pos) => Pos
    case (Pos, Neg) => Top
    case (Pos, Zer) => Pos
    case (Pos, Pos) => Pos
    case _ => Top
}

/// Over-approximates integer ‘subtraction‘.

pub def minus(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
    case (Bot, _) => Bot
    case (_, Bot) => Bot
    case (Neg, Neg) => Top
    case (Neg, Zer) => Neg
    case (Neg, Pos) => Neg
    case (Zer, Neg) => Pos
    case (Zer, Zer) => Zer
}

84
case (Zer, Pos) => Neg
case (Pos, Neg) => Pos
case (Pos, Zer) => Pos
case (Pos, Pos) => Top
case _ => Top
}

/// Over-approximates integer ‘multiplication’.

pub def times(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
case (Bot, _) => Bot
case (_, Bot) => Bot
case (Neg, Neg) => Pos
case (Neg, Zer) => Zer
case (Neg, Pos) => Neg
case (Zer, Neg) => Zer
case (Zer, Zer) => Zer
case (Zer, Pos) => Zer
case (Pos, Neg) => Neg
case (Pos, Zer) => Zer
case (Pos, Pos) => Pos
case _ => Top
}

// the div transfer function

pub def div(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
case (_, Bot) => Bot
case (Bot, _) => Bot
case (_, Zer) => Bot
case (Neg, Neg) => Pos
case (Neg, Pos) => Neg
case (Neg, Zer) => Zer
case (Zer, Neg) => Zer
case (Zer, Zer) => Zer
case (Zer, Pos) => Zer
case (Pos, Neg) => Neg
case (Pos, Pos) => Pos
case _ => Top
}

// match key "123 xx", with label "123" but not variable "xx"
def matchLabelNotVar(key: Str, l: Str, r: Str): Bool = {
  let lst = String.split(key, " ");
  l==lst[0] && !(r==lst[1])
}

// match key "123 xx", with label "123" and variable "xx"
def matchLabelAndVar(key: Str, l: Str, r: Str): Bool = {
  let lst = String.split(key, " ");
  l==lst[0] && r==lst[1]
}

// match key with label
def matchLabel(key: Str, l: Str): Bool = String.split(key, " ")[0]==l

// The main entry point.
def main(): #{ SetConstStm, AddStm, MinusStm, MultStm, DivStm,
AssignVar, NonAssign, Flow, VarEntry, VarExit } = {

let facts = #{

// Here is a simple example of input facts
  SetConstStm("0", "a", -2).
  SetConstStm("1", "b", 0).
  AddStm("2", "a", "a", "b").
  MinusStm("3", "b", "a", "b").
  MultStm("4", "a", "b", "d").
  DivStm("5", "a", "a", "b").
  Flow("0", "1").
}
Flow("1", "2").
Flow("2", "3").
Flow("3", "4").
Flow("4", "5").
}

let rules_1 = #{
  AssignVar(l, r) :- SetConstStm(l, r, c).
  AssignVar(l, r) :- AddStm(l, r, x, y).
  AssignVar(l, r) :- MinusStm(l, r, x, y).
  AssignVar(l, r) :- MultStm(l, r, x, y).
  AssignVar(l, r) :- DivStm(l, r, x, y).

  NonAssign(l) :- Flow(_, l), !AssignVar(l, _).
  NonAssign(l) :- Flow(l, _), !AssignVar(l, _).
};

let rules_2 = #{
  VarExit(k, v) :- VarEntry(k, v), AssignVar(l, r),
    matchLabelNotVar(k, l, r).
  VarExit(k1, v) :- VarEntry(k1, v), NonAssign(l1),
    matchLabel(k1, l1).

  VarExit(String.concat(l, String.concat(" ", r)), alpha(c)) :-
    SetConstStm(l, r, c).
  VarExit(String.concat(l, String.concat(" ", r)), plus(v1, v2 )) :-
    AddStm(l, r, x, y),
    VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
    VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
  VarExit(String.concat(l, String.concat(" ", r)), minus(v1, v2)) :-
    MinusStm(l, r, x, y),
    VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
    ...
VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
VarExit(String.concat(l, String.concat(" ", r)), div(v1, v2)) :-
   DivStm(l, r, x, y),
   VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
   VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
VarExit(String.concat(l, String.concat(" ", r)), times(v1, v2)) :-
   MultStm(l, r, x, y),
   VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
   VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
VarEntry(String.concat(l2, String.concat(" ", String.split(k1, " ")[1])), v) :-
   VarExit(k1, v), Flow(l1, l2), matchLabel(k1, l1).

};

let m1 = solve (facts <+> rules_1);
solve (m1 <+> rules_2)

D.2 Constant Propagation Analysis

//
// Define an enum with three variants: bottom and top elements, and a constructor Cst(i) for any integer i.
//
enum Constant {
   case Top,
   case Cst(Int),
   case Bot
}
// Represent the program-under-analysis using six input relations.
// Five of them are for each type of statement, and one for control flow

rel SetConstStm(l: Str, r: Str, c: Int) // r = c
rel AddStm(l: Str, r: Str, x: Str, y: Str) // r = x + y
rel MinusStm(l: Str, r: Str, x: Str, y: Str) // r = x - y
rel MultStm(l: Str, r: Str, x: Str, y: Str) // r = x * y
rel DivStm(l: Str, r: Str, x: Str, y: Str) // r = x / y

rel Flow(l1: Str, l2: Str) // flow from l1 to l2

// intermediate relations
rel AssignVar(l: Str, r: Str) // variable r is re-assigned at lable l
rel NonAssign(l: Str) // there is not assignment at lable l

// Results
lat VarEntry(key: Str, v: Constant)
lat VarExit(key: Str, v: Constant)

// Define equality of two lattice elements by structural equality.
//
def equ(e1: Constant, e2: Constant): Bool = e1 == e2

// Define the partial order on Constant.
// This is straightforward with pattern matching on the arguments

def leq(e1: Constant, e2: Constant): Bool = match (e1, e2) with {
  case (Bot, _) => true
  case (Cst(n1), Cst(n2)) => n1 == n2
  case (_, Top) => true
  case _ => false
}
// Least upper bound:

def lub(e1: Constant, e2: Constant): Constant = match (e1, e2) with {
  case (Bot, x)       => x
  case (x, Bot)       => x
  case (Cst(n1), Cst(n2)) => if (n1 == n2) e1 else Top
  case _               => Top
}

// Greatest lower bound

def glb(e1: Constant, e2: Constant): Constant = match (e1, e2) with {
  case (Top, x)       => x
  case (x, Top)       => x
  case (Cst(n1), Cst(n2)) => if (n1 == n2) e1 else Bot
  case _               => Bot
}

// Associate the lattice components with the Constant type

let Constant<> = (Bot, Top, equ, leq, lub, glb)

// NB: In the future, this syntax is subject to change.

// lift an integer into an element of the constant propagation lattice

def alpha(i: Int): Constant = Cst(i)

// Define the sum transfer function

def plus(e1: Constant, e2: Constant): Constant = match (e1, e2) with {
  case (Bot, _) => Bot
  case (_, Bot) => Bot
  case (Cst(n1), Cst(n2)) => Cst(n1 + n2)
  case _ => Top
}

// Minus transfer function
def minus(e1: Constant, e2: Constant): Constant = match (e1, e2) with {
  case (Bot, _) => Bot
  case (_, Bot) => Bot
  case (Cst(n1), Cst(n2)) => Cst(n1 - n2)
  case _ => Top
}

// Div transfer function, consider the case divisor is 0
//
def div(e1: Constant, e2: Constant): Constant = match (e1, e2) with {
  case (_, Bot) => Bot
  case (Bot, _) => Bot
  case (Cst(n1), Cst(n2)) => if (n2 == 0) Bot else Cst(n1 / n2)
  case _ => Top
}

// Multiplication transfer function
def times(e1: Constant, e2: Constant): Constant = match (e1, e2) with {
  case (_, Bot) => Bot
  case (Bot, _) => Bot
  case (Cst(n1), Cst(n2)) => Cst(n1 * n2)
  case _ => Top
}

// match key "123 xx", with label "123" but not variable "xx"
def matchLabelNotVar(key: Str, l: Str, r: Str): Bool = {
    let lst = String.split(key, " ");
    l==lst[0] && !(r==lst[1])
}

// match key "123 xx", with label "123" and variable "xx"
def matchLabelAndVar(key: Str, l: Str, r: Str): Bool = {
    let lst = String.split(key, " ");
    l==lst[0] && r==lst[1]
}

// match key with label
def matchLabel(key: Str, l: Str): Bool = String.split(key, " ")[0]==l

def main(): #{ SetConstStm, AddStm, MinusStm, MultStm, DivStm, AssignVar, NonAssign, Flow, VarEntry, VarExit } = {
    let facts = #{
        // a simple example
        SetConstStm("0", "a", -2).
        SetConstStm("1", "b", 0).
        AddStm("2", "a", "a", "b").
        MinusStm("3", "b", "a", "b").
        MultStm("4", "a", "b", "d").
        DivStm("5", "a", "a", "b").
        Flow("0", "1").
        Flow("1", "2").
        Flow("2", "3").
        Flow("3", "4").
        Flow("4", "5").
    };

    let rules_1 = #{
        AssignVar(l, r) :- SetConstStm(l, r, c).
        AssignVar(l, r) :- AddStm(l, r, x, y).
    }
AssignVar(l, r) :- MinusStm(l, r, x, y).
AssignVar(l, r) :- MultStm(l, r, x, y).
AssignVar(l, r) :- DivStm(l, r, x, y).

NonAssign(l) :- Flow(_, l), !AssignVar(l, _).
NonAssign(l) :- Flow(l, _), !AssignVar(l, _).

let rules_2 = #{
    VarExit(k, v) :- VarEntry(k, v), AssignVar(l, r),
                    matchLabelNotVar(k, l, r).
    VarExit(k1, v) :- VarEntry(k1, v), NonAssign(l1),
                     matchLabel(k1, l1).
    VarExit(String.concat(l, String.concat(" ", r)), alpha(c)) :-
                    SetConstStm(l, r, c).
    VarExit(String.concat(l, String.concat(" ", r)), plus(v1, v2 )) :-
                    AddStm(l, r, x, y),
                    VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
                    VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
    VarExit(String.concat(l, String.concat(" ", r)), minus(v1, v2)) :-
                    MinusStm(l, r, x, y),
                    VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
                    VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
    VarExit(String.concat(l, String.concat(" ", r)), div(v1, v2) ) :-
                    DivStm(l, r, x, y),
                    VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
                    VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
    VarExit(String.concat(l, String.concat(" ", r)), times(v1, v2)) :-
                    MultStm(l, r, x, y),
}
VarEntry(k1, v1), matchLabelAndVar(k1, l, x),
VarEntry(k2, v2), matchLabelAndVar(k2, l, y).
VarEntry(String.concat(l2,
    String.concat(" ", String.split(k1, " ")[1])), v) :-
    VarExit(k1, v), Flow(l1, l2), matchLabel(k1, l1)
};

let m1 = solve (facts <+> rules_1);
solve (m1 <+> rules_2)
## Appendix E

### Raw Data in Experiments

#### E.1 Sign Analysis without Branches

Table E.1: Sign Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.419</td>
<td>12.244</td>
<td>45.379</td>
<td>111.606</td>
<td>383.119</td>
<td>907.821</td>
</tr>
<tr>
<td>1</td>
<td>1.482</td>
<td>12.068</td>
<td>30.121</td>
<td>47.659</td>
<td>117.653</td>
<td>309.894</td>
</tr>
<tr>
<td>2</td>
<td>1.186</td>
<td>11.774</td>
<td>41.316</td>
<td>85.243</td>
<td>152.974</td>
<td>348.848</td>
</tr>
<tr>
<td>3</td>
<td>1.504</td>
<td>13.167</td>
<td>44.842</td>
<td>89.987</td>
<td>167.016</td>
<td>432.359</td>
</tr>
<tr>
<td>4</td>
<td>0.917</td>
<td>10.746</td>
<td>42.209</td>
<td>103.968</td>
<td>353.264</td>
<td>875.228</td>
</tr>
<tr>
<td>5</td>
<td>1.472</td>
<td>13.088</td>
<td>44.279</td>
<td>103.224</td>
<td>338.271</td>
<td>732.925</td>
</tr>
<tr>
<td>6</td>
<td>0.953</td>
<td>3.822</td>
<td>10.007</td>
<td>15.987</td>
<td>52.435</td>
<td>122.758</td>
</tr>
<tr>
<td>7</td>
<td>1.419</td>
<td>13.949</td>
<td>50.198</td>
<td>117.179</td>
<td>383.197</td>
<td>872.203</td>
</tr>
<tr>
<td>8</td>
<td>0.539</td>
<td>2.048</td>
<td>3.572</td>
<td>6.501</td>
<td>17.344</td>
<td>82.093</td>
</tr>
<tr>
<td>9</td>
<td>1.459</td>
<td>13.054</td>
<td>38.702</td>
<td>103.029</td>
<td>356.861</td>
<td>746.531</td>
</tr>
<tr>
<td>10</td>
<td>1.537</td>
<td>14.031</td>
<td>47.707</td>
<td>115.875</td>
<td>399.984</td>
<td>962.825</td>
</tr>
<tr>
<td>11</td>
<td>0.338</td>
<td>3.349</td>
<td>21.238</td>
<td>66.676</td>
<td>278.914</td>
<td>756.136</td>
</tr>
<tr>
<td>12</td>
<td>0.866</td>
<td>2.333</td>
<td>4.107</td>
<td>6.68</td>
<td>25.978</td>
<td>129.162</td>
</tr>
<tr>
<td>13</td>
<td>0.783</td>
<td>4.424</td>
<td>23.856</td>
<td>68.698</td>
<td>278.695</td>
<td>740.145</td>
</tr>
<tr>
<td>14</td>
<td>1.452</td>
<td>12.217</td>
<td>45.808</td>
<td>111.641</td>
<td>372.462</td>
<td>882.843</td>
</tr>
<tr>
<td>15</td>
<td>1.457</td>
<td>12.981</td>
<td>43.48</td>
<td>111.873</td>
<td>386.874</td>
<td>922.538</td>
</tr>
<tr>
<td>16</td>
<td>1.335</td>
<td>13.357</td>
<td>46.416</td>
<td>109.661</td>
<td>263.497</td>
<td>459.81</td>
</tr>
<tr>
<td>17</td>
<td>1.521</td>
<td>10.87</td>
<td>22.79</td>
<td>33</td>
<td>73.188</td>
<td>144.092</td>
</tr>
<tr>
<td>18</td>
<td>0.846</td>
<td>8.76</td>
<td>36.017</td>
<td>91.124</td>
<td>342.42</td>
<td>859.862</td>
</tr>
<tr>
<td>19</td>
<td>1.692</td>
<td>15.003</td>
<td>46.828</td>
<td>115.884</td>
<td>395.023</td>
<td>806.43</td>
</tr>
<tr>
<td>avg</td>
<td>1.209</td>
<td>10.164</td>
<td>34.444</td>
<td>80.775</td>
<td>256.958</td>
<td>604.725</td>
</tr>
<tr>
<td>std</td>
<td>0.380</td>
<td>4.360</td>
<td>15.036</td>
<td>38.506</td>
<td>137.733</td>
<td>314.027</td>
</tr>
</tbody>
</table>
Table E.2: Sign Analysis without Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.431</td>
<td>12.219</td>
<td>45.082</td>
<td>110.37</td>
<td>380.771</td>
<td>907.101</td>
</tr>
<tr>
<td>1</td>
<td>1.499</td>
<td>11.957</td>
<td>29.916</td>
<td>47.708</td>
<td>116.57</td>
<td>303.958</td>
</tr>
<tr>
<td>2</td>
<td>1.189</td>
<td>11.656</td>
<td>41.116</td>
<td>84.798</td>
<td>151.315</td>
<td>348.068</td>
</tr>
<tr>
<td>3</td>
<td>1.514</td>
<td>13.275</td>
<td>45.151</td>
<td>90.293</td>
<td>166.251</td>
<td>434.573</td>
</tr>
<tr>
<td>4</td>
<td>0.934</td>
<td>10.813</td>
<td>42.193</td>
<td>103.369</td>
<td>353.226</td>
<td>872.719</td>
</tr>
<tr>
<td>5</td>
<td>1.445</td>
<td>12.88</td>
<td>43.697</td>
<td>102.992</td>
<td>338.918</td>
<td>729.923</td>
</tr>
<tr>
<td>6</td>
<td>0.924</td>
<td>3.748</td>
<td>9.965</td>
<td>16.009</td>
<td>52.284</td>
<td>122.969</td>
</tr>
<tr>
<td>7</td>
<td>1.477</td>
<td>13.883</td>
<td>49.646</td>
<td>117.708</td>
<td>386.384</td>
<td>861.855</td>
</tr>
<tr>
<td>8</td>
<td>0.533</td>
<td>2.049</td>
<td>3.546</td>
<td>6.416</td>
<td>17.404</td>
<td>81.912</td>
</tr>
<tr>
<td>9</td>
<td>1.489</td>
<td>12.846</td>
<td>38.34</td>
<td>101.916</td>
<td>354.321</td>
<td>734.404</td>
</tr>
<tr>
<td>10</td>
<td>1.553</td>
<td>13.846</td>
<td>47.005</td>
<td>113.536</td>
<td>396.474</td>
<td>952.887</td>
</tr>
<tr>
<td>11</td>
<td>0.345</td>
<td>3.321</td>
<td>20.775</td>
<td>65.626</td>
<td>277.242</td>
<td>753.055</td>
</tr>
<tr>
<td>12</td>
<td>0.833</td>
<td>2.309</td>
<td>3.964</td>
<td>6.549</td>
<td>26.048</td>
<td>128.732</td>
</tr>
<tr>
<td>13</td>
<td>0.781</td>
<td>4.403</td>
<td>23.812</td>
<td>68.065</td>
<td>279.909</td>
<td>739.466</td>
</tr>
<tr>
<td>14</td>
<td>1.45</td>
<td>12.008</td>
<td>44.964</td>
<td>107.702</td>
<td>374.467</td>
<td>870.333</td>
</tr>
<tr>
<td>15</td>
<td>1.464</td>
<td>13.042</td>
<td>43.848</td>
<td>113.67</td>
<td>390.691</td>
<td>919.868</td>
</tr>
<tr>
<td>16</td>
<td>1.333</td>
<td>13.185</td>
<td>46.375</td>
<td>108.358</td>
<td>263.59</td>
<td>456.459</td>
</tr>
<tr>
<td>17</td>
<td>1.489</td>
<td>10.44</td>
<td>22.549</td>
<td>32.619</td>
<td>73.695</td>
<td>144.11</td>
</tr>
<tr>
<td>18</td>
<td>0.846</td>
<td>8.293</td>
<td>36.057</td>
<td>90.286</td>
<td>342.113</td>
<td>843.878</td>
</tr>
<tr>
<td>19</td>
<td>1.673</td>
<td>13.581</td>
<td>47.641</td>
<td>114.219</td>
<td>391.068</td>
<td>796.854</td>
</tr>
<tr>
<td>avg</td>
<td>1.210</td>
<td>9.988</td>
<td>34.282</td>
<td>80.110</td>
<td>256.637</td>
<td>600.156</td>
</tr>
<tr>
<td>std</td>
<td>0.384</td>
<td>4.267</td>
<td>15.017</td>
<td>38.166</td>
<td>137.688</td>
<td>311.043</td>
</tr>
</tbody>
</table>
Table E.3: Sign Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9.826</td>
<td>64.048</td>
<td>291.317</td>
<td>1073.449</td>
</tr>
<tr>
<td>1</td>
<td>8.2</td>
<td>59.2</td>
<td>301.614</td>
<td>654.309</td>
</tr>
<tr>
<td>2</td>
<td>7.265</td>
<td>46.991</td>
<td>280.449</td>
<td>1006.306</td>
</tr>
<tr>
<td>3</td>
<td>7.964</td>
<td>57.811</td>
<td>276.928</td>
<td>943.32</td>
</tr>
<tr>
<td>4</td>
<td>6.375</td>
<td>54.29</td>
<td>293.975</td>
<td>1027.235</td>
</tr>
<tr>
<td>5</td>
<td>7.994</td>
<td>54.321</td>
<td>320.6</td>
<td>1112.094</td>
</tr>
<tr>
<td>6</td>
<td>7.813</td>
<td>55.452</td>
<td>316.876</td>
<td>1180.73</td>
</tr>
<tr>
<td>7</td>
<td>7.397</td>
<td>57.733</td>
<td>298.775</td>
<td>1159.196</td>
</tr>
<tr>
<td>8</td>
<td>5.49</td>
<td>11.477</td>
<td>24.701</td>
<td>70.329</td>
</tr>
<tr>
<td>9</td>
<td>7.955</td>
<td>53.939</td>
<td>312.94</td>
<td>1154.041</td>
</tr>
<tr>
<td>10</td>
<td>7.679</td>
<td>67.162</td>
<td>325.842</td>
<td>1184.078</td>
</tr>
<tr>
<td>11</td>
<td>5.735</td>
<td>16.562</td>
<td>161.968</td>
<td>695.637</td>
</tr>
<tr>
<td>12</td>
<td>6.495</td>
<td>14.835</td>
<td>29.339</td>
<td>72.02</td>
</tr>
<tr>
<td>13</td>
<td>6.371</td>
<td>25.22</td>
<td>142.167</td>
<td>733.906</td>
</tr>
<tr>
<td>14</td>
<td>7.511</td>
<td>42.87</td>
<td>279.823</td>
<td>1180.294</td>
</tr>
<tr>
<td>15</td>
<td>7.781</td>
<td>53.145</td>
<td>324.958</td>
<td>1282.404</td>
</tr>
<tr>
<td>16</td>
<td>6.926</td>
<td>53.101</td>
<td>265.524</td>
<td>1037.277</td>
</tr>
<tr>
<td>17</td>
<td>7.602</td>
<td>44.259</td>
<td>154.8</td>
<td>331.842</td>
</tr>
<tr>
<td>18</td>
<td>6.967</td>
<td>40.496</td>
<td>247.147</td>
<td>942.68</td>
</tr>
<tr>
<td>19</td>
<td>8.183</td>
<td>56.544</td>
<td>340.263</td>
<td>1127.877</td>
</tr>
<tr>
<td>avg</td>
<td>7.376</td>
<td>46.473</td>
<td>249.500</td>
<td>898.451</td>
</tr>
<tr>
<td>std</td>
<td>0.982</td>
<td>16.590</td>
<td>95.228</td>
<td>364.145</td>
</tr>
</tbody>
</table>
### E.2 Sign Analysis with Branches

Table E.4: Sign Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.581</td>
<td>16.1</td>
<td>52.633</td>
<td>137.603</td>
<td>446.161</td>
<td>1050.23</td>
</tr>
<tr>
<td>1</td>
<td>1.633</td>
<td>17.034</td>
<td>63.113</td>
<td>166.967</td>
<td>548.193</td>
<td>1223.587</td>
</tr>
<tr>
<td>2</td>
<td>1.034</td>
<td>17.251</td>
<td>53.356</td>
<td>116.051</td>
<td>288.978</td>
<td>848.81</td>
</tr>
<tr>
<td>3</td>
<td>1.431</td>
<td>9.721</td>
<td>29.962</td>
<td>87.836</td>
<td>329.11</td>
<td>824.976</td>
</tr>
<tr>
<td>4</td>
<td>1.796</td>
<td>13.003</td>
<td>76.454</td>
<td>185.379</td>
<td>537.692</td>
<td>1211.808</td>
</tr>
<tr>
<td>5</td>
<td>1.285</td>
<td>14.328</td>
<td>49.853</td>
<td>161.395</td>
<td>473.027</td>
<td>1055.497</td>
</tr>
<tr>
<td>6</td>
<td>0.98</td>
<td>5.148</td>
<td>22.562</td>
<td>76.535</td>
<td>467.417</td>
<td>1253.421</td>
</tr>
<tr>
<td>7</td>
<td>1.131</td>
<td>11.47</td>
<td>35.532</td>
<td>103.422</td>
<td>396.243</td>
<td>887.17</td>
</tr>
<tr>
<td>8</td>
<td>0.715</td>
<td>3.818</td>
<td>8.283</td>
<td>19.451</td>
<td>61.75</td>
<td>136.202</td>
</tr>
<tr>
<td>9</td>
<td>1.651</td>
<td>24.044</td>
<td>59.167</td>
<td>157.921</td>
<td>503.482</td>
<td>1190.398</td>
</tr>
<tr>
<td>10</td>
<td>1.424</td>
<td>10.715</td>
<td>42.895</td>
<td>122.812</td>
<td>386.396</td>
<td>891.639</td>
</tr>
<tr>
<td>11</td>
<td>0.431</td>
<td>3.248</td>
<td>13.81</td>
<td>57.374</td>
<td>271.504</td>
<td>730.574</td>
</tr>
<tr>
<td>12</td>
<td>1.244</td>
<td>11.784</td>
<td>40.889</td>
<td>100.158</td>
<td>371.533</td>
<td>882.576</td>
</tr>
<tr>
<td>13</td>
<td>0.656</td>
<td>2.993</td>
<td>7.916</td>
<td>12.942</td>
<td>98.664</td>
<td>403.024</td>
</tr>
<tr>
<td>14</td>
<td>1.423</td>
<td>20.607</td>
<td>76.409</td>
<td>238.946</td>
<td>633.172</td>
<td>1469.986</td>
</tr>
<tr>
<td>15</td>
<td>1.42</td>
<td>16.18</td>
<td>54.62</td>
<td>135.076</td>
<td>462.661</td>
<td>1115.498</td>
</tr>
<tr>
<td>16</td>
<td>1.349</td>
<td>15.749</td>
<td>36.938</td>
<td>92.377</td>
<td>448.17</td>
<td>1124.439</td>
</tr>
<tr>
<td>17</td>
<td>1.219</td>
<td>14.158</td>
<td>39.103</td>
<td>74.159</td>
<td>152.087</td>
<td>360.22</td>
</tr>
<tr>
<td>18</td>
<td>0.577</td>
<td>6.583</td>
<td>28.86</td>
<td>79.731</td>
<td>333.733</td>
<td>811.395</td>
</tr>
<tr>
<td>19</td>
<td>3.439</td>
<td>14.287</td>
<td>47.093</td>
<td>114.363</td>
<td>485.657</td>
<td>1191.08</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td><strong>1.321</strong></td>
<td><strong>12.411</strong></td>
<td><strong>41.972</strong></td>
<td><strong>112.025</strong></td>
<td><strong>384.782</strong></td>
<td><strong>933.127</strong></td>
</tr>
<tr>
<td><strong>std</strong></td>
<td><strong>0.626</strong></td>
<td><strong>5.810</strong></td>
<td><strong>19.824</strong></td>
<td><strong>54.615</strong></td>
<td><strong>150.944</strong></td>
<td><strong>333.204</strong></td>
</tr>
</tbody>
</table>
Table E.5: Sign Analysis with Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.432</td>
<td>15.942</td>
<td>53.024</td>
<td>140.099</td>
<td>574.589</td>
<td>1328.049</td>
</tr>
<tr>
<td>1</td>
<td>1.696</td>
<td>19.849</td>
<td>91.014</td>
<td>356.902</td>
<td>2018.934</td>
<td>4474.172</td>
</tr>
<tr>
<td>2</td>
<td>1.074</td>
<td>29.602</td>
<td>106.539</td>
<td>204.819</td>
<td>456.932</td>
<td>1164.379</td>
</tr>
<tr>
<td>3</td>
<td>1.511</td>
<td>10.867</td>
<td>38.446</td>
<td>119.845</td>
<td>747.659</td>
<td>2081.517</td>
</tr>
<tr>
<td>4</td>
<td>2.112</td>
<td>21.376</td>
<td>168.905</td>
<td>483.338</td>
<td>2143.539</td>
<td>5932.033</td>
</tr>
<tr>
<td>5</td>
<td>1.251</td>
<td>16.783</td>
<td>74.302</td>
<td>351.002</td>
<td>1388.203</td>
<td>3151.252</td>
</tr>
<tr>
<td>6</td>
<td>0.994</td>
<td>5.389</td>
<td>23.082</td>
<td>77.073</td>
<td>566.65</td>
<td>1607.079</td>
</tr>
<tr>
<td>7</td>
<td>1.125</td>
<td>11.313</td>
<td>35.495</td>
<td>108.14</td>
<td>420.352</td>
<td>917.46</td>
</tr>
<tr>
<td>8</td>
<td>0.713</td>
<td>3.693</td>
<td>8.068</td>
<td>18.817</td>
<td>60.913</td>
<td>133.858</td>
</tr>
<tr>
<td>9</td>
<td>1.626</td>
<td>23.967</td>
<td>59.541</td>
<td>164.263</td>
<td>772.903</td>
<td>2172.253</td>
</tr>
<tr>
<td>10</td>
<td>1.38</td>
<td>11.642</td>
<td>44.471</td>
<td>157.126</td>
<td>956.777</td>
<td>2901.884</td>
</tr>
<tr>
<td>11</td>
<td>0.421</td>
<td>3.205</td>
<td>13.236</td>
<td>99.093</td>
<td>576.94</td>
<td>1585.181</td>
</tr>
<tr>
<td>12</td>
<td>1.247</td>
<td>15.206</td>
<td>60.679</td>
<td>134.234</td>
<td>450.225</td>
<td>985.683</td>
</tr>
<tr>
<td>13</td>
<td>0.642</td>
<td>2.973</td>
<td>7.825</td>
<td>12.811</td>
<td>101.638</td>
<td>404.033</td>
</tr>
<tr>
<td>14</td>
<td>1.412</td>
<td>20.948</td>
<td>75.676</td>
<td>295.145</td>
<td>1164.436</td>
<td>3354.204</td>
</tr>
<tr>
<td>15</td>
<td>1.423</td>
<td>15.904</td>
<td>56.06</td>
<td>157.696</td>
<td>565.689</td>
<td>2050.406</td>
</tr>
<tr>
<td>16</td>
<td>1.356</td>
<td>15.695</td>
<td>36.608</td>
<td>92.615</td>
<td>578.682</td>
<td>2933.376</td>
</tr>
<tr>
<td>17</td>
<td>1.205</td>
<td>21.539</td>
<td>76.56</td>
<td>148.874</td>
<td>286.307</td>
<td>593.064</td>
</tr>
<tr>
<td>18</td>
<td>0.567</td>
<td>7.715</td>
<td>31.576</td>
<td>94.354</td>
<td>484.998</td>
<td>1869.327</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td><strong>1.334</strong></td>
<td><strong>14.420</strong></td>
<td><strong>55.501</strong></td>
<td><strong>166.741</strong></td>
<td><strong>746.451</strong></td>
<td><strong>2056.439</strong></td>
</tr>
<tr>
<td><strong>std</strong></td>
<td><strong>0.653</strong></td>
<td><strong>7.376</strong></td>
<td><strong>37.785</strong></td>
<td><strong>118.639</strong></td>
<td><strong>553.250</strong></td>
<td><strong>1420.990</strong></td>
</tr>
</tbody>
</table>
Table E.6: Sign Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14.719</td>
<td>67.912</td>
<td>302.923</td>
<td>1191.527</td>
</tr>
<tr>
<td>1</td>
<td>14.258</td>
<td>70.945</td>
<td>393.684</td>
<td>1336.273</td>
</tr>
<tr>
<td>2</td>
<td>9.445</td>
<td>56.459</td>
<td>245.607</td>
<td>804.947</td>
</tr>
<tr>
<td>3</td>
<td>12.569</td>
<td>60.059</td>
<td>285.518</td>
<td>1140.637</td>
</tr>
<tr>
<td>4</td>
<td>11.756</td>
<td>48.132</td>
<td>372.256</td>
<td>1324.018</td>
</tr>
<tr>
<td>5</td>
<td>12.587</td>
<td>59.022</td>
<td>248.231</td>
<td>1374.854</td>
</tr>
<tr>
<td>6</td>
<td>11.67</td>
<td>56.017</td>
<td>270.816</td>
<td>981.215</td>
</tr>
<tr>
<td>7</td>
<td>12.188</td>
<td>60.764</td>
<td>205.902</td>
<td>882.76</td>
</tr>
<tr>
<td>8</td>
<td>11.163</td>
<td>24.213</td>
<td>54.952</td>
<td>157.959</td>
</tr>
<tr>
<td>9</td>
<td>12.606</td>
<td>98.619</td>
<td>313.904</td>
<td>1219.709</td>
</tr>
<tr>
<td>10</td>
<td>13.158</td>
<td>47.806</td>
<td>254.649</td>
<td>1035.029</td>
</tr>
<tr>
<td>11</td>
<td>10.874</td>
<td>26.204</td>
<td>76.853</td>
<td>396.257</td>
</tr>
<tr>
<td>12</td>
<td>12.239</td>
<td>52.268</td>
<td>239.909</td>
<td>864.45</td>
</tr>
<tr>
<td>13</td>
<td>12.321</td>
<td>22.41</td>
<td>71.419</td>
<td>147.928</td>
</tr>
<tr>
<td>14</td>
<td>12.018</td>
<td>82.748</td>
<td>347.924</td>
<td>1868.991</td>
</tr>
<tr>
<td>15</td>
<td>12.686</td>
<td>69.945</td>
<td>324.078</td>
<td>1272.354</td>
</tr>
<tr>
<td>16</td>
<td>13.122</td>
<td>68.437</td>
<td>205.957</td>
<td>822.5</td>
</tr>
<tr>
<td>17</td>
<td>10.667</td>
<td>61.787</td>
<td>201.155</td>
<td>559.58</td>
</tr>
<tr>
<td>18</td>
<td>11.459</td>
<td>36.51</td>
<td>178.041</td>
<td>720.795</td>
</tr>
<tr>
<td>19</td>
<td>16.154</td>
<td>52.267</td>
<td>212.094</td>
<td>869.525</td>
</tr>
<tr>
<td>avg</td>
<td>12.383</td>
<td>56.126</td>
<td>240.294</td>
<td>948.565</td>
</tr>
<tr>
<td>std</td>
<td>1.484</td>
<td>19.045</td>
<td>94.677</td>
<td>424.976</td>
</tr>
</tbody>
</table>
### E.3 Constant Propagation Analysis without Branches

Table E.7: Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.757</td>
<td>12.61</td>
<td>46.385</td>
<td>84.066</td>
<td>147.355</td>
<td>223.111</td>
</tr>
<tr>
<td>1</td>
<td>1.618</td>
<td>12.149</td>
<td>27.839</td>
<td>44.111</td>
<td>99.014</td>
<td>270.972</td>
</tr>
<tr>
<td>2</td>
<td>1.246</td>
<td>10.358</td>
<td>32.918</td>
<td>70.977</td>
<td>120.138</td>
<td>253.914</td>
</tr>
<tr>
<td>3</td>
<td>0.747</td>
<td>1.721</td>
<td>7.813</td>
<td>22.343</td>
<td>50.512</td>
<td>178.85</td>
</tr>
<tr>
<td>4</td>
<td>0.9</td>
<td>11.669</td>
<td>46.079</td>
<td>114.168</td>
<td>232.572</td>
<td>390.145</td>
</tr>
<tr>
<td>5</td>
<td>1.535</td>
<td>14.092</td>
<td>48.604</td>
<td>112.839</td>
<td>310.43</td>
<td>530.766</td>
</tr>
<tr>
<td>6</td>
<td>1.023</td>
<td>4.111</td>
<td>10.838</td>
<td>17.415</td>
<td>48.965</td>
<td>114.024</td>
</tr>
<tr>
<td>7</td>
<td>1.353</td>
<td>13.94</td>
<td>52.002</td>
<td>124.029</td>
<td>368.973</td>
<td>834.463</td>
</tr>
<tr>
<td>8</td>
<td>0.537</td>
<td>2.056</td>
<td>3.627</td>
<td>6.792</td>
<td>16.466</td>
<td>77.635</td>
</tr>
<tr>
<td>9</td>
<td>1.537</td>
<td>13.155</td>
<td>27.527</td>
<td>68.481</td>
<td>193.605</td>
<td>394.519</td>
</tr>
<tr>
<td>10</td>
<td>1.514</td>
<td>14.002</td>
<td>34.494</td>
<td>63.112</td>
<td>121.281</td>
<td>209.807</td>
</tr>
<tr>
<td>11</td>
<td>0.353</td>
<td>2.723</td>
<td>6.803</td>
<td>13.719</td>
<td>39.144</td>
<td>79.436</td>
</tr>
<tr>
<td>12</td>
<td>0.874</td>
<td>2.393</td>
<td>4.113</td>
<td>6.7</td>
<td>22.983</td>
<td>104.359</td>
</tr>
<tr>
<td>13</td>
<td>0.798</td>
<td>4.619</td>
<td>25.089</td>
<td>57.952</td>
<td>154.059</td>
<td>233.567</td>
</tr>
<tr>
<td>14</td>
<td>1.463</td>
<td>12.56</td>
<td>30.674</td>
<td>48.925</td>
<td>89.482</td>
<td>188.233</td>
</tr>
<tr>
<td>15</td>
<td>1.547</td>
<td>10.27</td>
<td>19.134</td>
<td>44.446</td>
<td>93.069</td>
<td>234.388</td>
</tr>
<tr>
<td>16</td>
<td>1.315</td>
<td>12.561</td>
<td>28.986</td>
<td>63.75</td>
<td>158.006</td>
<td>288.373</td>
</tr>
<tr>
<td>17</td>
<td>1.489</td>
<td>11.435</td>
<td>24.037</td>
<td>34.698</td>
<td>69.005</td>
<td>135.889</td>
</tr>
<tr>
<td>18</td>
<td>0.936</td>
<td>8.884</td>
<td>28.44</td>
<td>45.033</td>
<td>111.736</td>
<td>270.352</td>
</tr>
<tr>
<td>19</td>
<td>1.656</td>
<td>13.773</td>
<td>47.885</td>
<td>118.536</td>
<td>333.83</td>
<td>687.641</td>
</tr>
<tr>
<td>avg</td>
<td>1.210</td>
<td>9.454</td>
<td>27.664</td>
<td>58.105</td>
<td>139.031</td>
<td>285.022</td>
</tr>
<tr>
<td>std</td>
<td>0.407</td>
<td>4.608</td>
<td>15.479</td>
<td>37.351</td>
<td>102.466</td>
<td>199.149</td>
</tr>
</tbody>
</table>
Table E.8: Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.452</td>
<td>9.806</td>
<td>34.572</td>
<td>62.991</td>
<td>105.385</td>
<td>160.497</td>
</tr>
<tr>
<td>1</td>
<td>1.496</td>
<td>9.403</td>
<td>21.417</td>
<td>34.027</td>
<td>72.787</td>
<td>197.338</td>
</tr>
<tr>
<td>2</td>
<td>1.212</td>
<td>8.09</td>
<td>25.11</td>
<td>53.289</td>
<td>84.182</td>
<td>183.556</td>
</tr>
<tr>
<td>3</td>
<td>0.828</td>
<td>1.547</td>
<td>6.241</td>
<td>16.562</td>
<td>32.507</td>
<td>130.982</td>
</tr>
<tr>
<td>4</td>
<td>0.967</td>
<td>8.983</td>
<td>34.837</td>
<td>85.836</td>
<td>167.566</td>
<td>282.765</td>
</tr>
<tr>
<td>5</td>
<td>1.423</td>
<td>10.889</td>
<td>36.926</td>
<td>86.341</td>
<td>224.483</td>
<td>380.527</td>
</tr>
<tr>
<td>6</td>
<td>1.066</td>
<td>3.395</td>
<td>8.383</td>
<td>13.182</td>
<td>35.408</td>
<td>81.991</td>
</tr>
<tr>
<td>7</td>
<td>1.281</td>
<td>10.591</td>
<td>38.999</td>
<td>93.16</td>
<td>268.083</td>
<td>607.093</td>
</tr>
<tr>
<td>8</td>
<td>0.691</td>
<td>1.776</td>
<td>2.934</td>
<td>5.262</td>
<td>12.037</td>
<td>56.396</td>
</tr>
<tr>
<td>9</td>
<td>1.429</td>
<td>10.125</td>
<td>20.755</td>
<td>52.014</td>
<td>141.079</td>
<td>283.906</td>
</tr>
<tr>
<td>10</td>
<td>1.416</td>
<td>10.693</td>
<td>26.535</td>
<td>48.132</td>
<td>88.54</td>
<td>153.632</td>
</tr>
<tr>
<td>11</td>
<td>0.531</td>
<td>2.233</td>
<td>5.291</td>
<td>10.54</td>
<td>28.874</td>
<td>57.383</td>
</tr>
<tr>
<td>12</td>
<td>0.949</td>
<td>1.988</td>
<td>3.372</td>
<td>5.281</td>
<td>16.702</td>
<td>75.403</td>
</tr>
<tr>
<td>13</td>
<td>0.883</td>
<td>3.672</td>
<td>18.869</td>
<td>43.989</td>
<td>112.893</td>
<td>169.724</td>
</tr>
<tr>
<td>14</td>
<td>1.364</td>
<td>9.638</td>
<td>23.343</td>
<td>36.874</td>
<td>65.985</td>
<td>135</td>
</tr>
<tr>
<td>15</td>
<td>1.432</td>
<td>7.971</td>
<td>14.496</td>
<td>33.369</td>
<td>68.026</td>
<td>171.367</td>
</tr>
<tr>
<td>16</td>
<td>1.265</td>
<td>9.667</td>
<td>22.068</td>
<td>47.888</td>
<td>114.45</td>
<td>207.546</td>
</tr>
<tr>
<td>18</td>
<td>0.992</td>
<td>7.015</td>
<td>21.771</td>
<td>34.315</td>
<td>81.739</td>
<td>197.335</td>
</tr>
<tr>
<td>19</td>
<td>1.531</td>
<td>10.538</td>
<td>35.8</td>
<td>89.171</td>
<td>244.282</td>
<td>508.881</td>
</tr>
</tbody>
</table>

avg 1.181 | 7.338 | 21.010 | 43.920 | 100.770 | 207.036

std 0.296 | 3.460 | 11.565 | 28.135 | 74.611 | 145.445
Table E.9: Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8.126</td>
<td>43.219</td>
<td>292.667</td>
<td>773.707</td>
</tr>
<tr>
<td>1</td>
<td>7.359</td>
<td>45.349</td>
<td>180.068</td>
<td>400.88</td>
</tr>
<tr>
<td>2</td>
<td>6.467</td>
<td>35.957</td>
<td>181.44</td>
<td>663.203</td>
</tr>
<tr>
<td>3</td>
<td>5.698</td>
<td>10.953</td>
<td>55.722</td>
<td>211.65</td>
</tr>
<tr>
<td>4</td>
<td>5.985</td>
<td>48.468</td>
<td>276.799</td>
<td>1043.721</td>
</tr>
<tr>
<td>5</td>
<td>7.102</td>
<td>55.093</td>
<td>309.67</td>
<td>1005.417</td>
</tr>
<tr>
<td>6</td>
<td>6.448</td>
<td>19.019</td>
<td>69.855</td>
<td>166.059</td>
</tr>
<tr>
<td>7</td>
<td>6.699</td>
<td>53.11</td>
<td>294.201</td>
<td>1102.137</td>
</tr>
<tr>
<td>8</td>
<td>5.137</td>
<td>11.632</td>
<td>27.75</td>
<td>68.662</td>
</tr>
<tr>
<td>9</td>
<td>7.229</td>
<td>50.809</td>
<td>183.223</td>
<td>644.714</td>
</tr>
<tr>
<td>10</td>
<td>7.124</td>
<td>51.308</td>
<td>212.734</td>
<td>577.809</td>
</tr>
<tr>
<td>11</td>
<td>5.086</td>
<td>14.669</td>
<td>45.752</td>
<td>135.515</td>
</tr>
<tr>
<td>12</td>
<td>6.027</td>
<td>13.926</td>
<td>31.663</td>
<td>68.489</td>
</tr>
<tr>
<td>13</td>
<td>5.705</td>
<td>21.413</td>
<td>165.343</td>
<td>605.177</td>
</tr>
<tr>
<td>14</td>
<td>6.896</td>
<td>46.686</td>
<td>173.668</td>
<td>433.88</td>
</tr>
<tr>
<td>15</td>
<td>7.314</td>
<td>37.018</td>
<td>105.078</td>
<td>423.475</td>
</tr>
<tr>
<td>16</td>
<td>6.599</td>
<td>43.759</td>
<td>182.772</td>
<td>585.452</td>
</tr>
<tr>
<td>17</td>
<td>7.304</td>
<td>44.298</td>
<td>137.126</td>
<td>336.494</td>
</tr>
<tr>
<td>18</td>
<td>6.18</td>
<td>36.928</td>
<td>180.17</td>
<td>411.655</td>
</tr>
<tr>
<td>19</td>
<td>7.44</td>
<td>55.715</td>
<td>316.734</td>
<td>1101.452</td>
</tr>
<tr>
<td>avg</td>
<td>6.596</td>
<td>36.966</td>
<td>171.122</td>
<td>537.977</td>
</tr>
<tr>
<td>std</td>
<td>0.815</td>
<td>15.704</td>
<td>94.206</td>
<td>336.219</td>
</tr>
</tbody>
</table>
### E.4 Constant Propagation Analysis with Branches

Table E.10: Constant Propagation Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in Extended Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>150</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.52</td>
<td>10.096</td>
<td>39.462</td>
<td>124.723</td>
<td>428.335</td>
<td>1004.246</td>
</tr>
<tr>
<td>1</td>
<td>1.957</td>
<td>16.533</td>
<td>58.632</td>
<td>168.34</td>
<td>655.291</td>
<td>1338.019</td>
</tr>
<tr>
<td>2</td>
<td>1.148</td>
<td>15.909</td>
<td>48.02</td>
<td>113.858</td>
<td>293.98</td>
<td>847.928</td>
</tr>
<tr>
<td>3</td>
<td>1.257</td>
<td>7.855</td>
<td>30.32</td>
<td>95.461</td>
<td>346.172</td>
<td>819.385</td>
</tr>
<tr>
<td>4</td>
<td>1.757</td>
<td>12.038</td>
<td>74.384</td>
<td>178.979</td>
<td>492.256</td>
<td>1082.955</td>
</tr>
<tr>
<td>5</td>
<td>0.84</td>
<td>6.191</td>
<td>28.325</td>
<td>125.824</td>
<td>466.888</td>
<td>1137.271</td>
</tr>
<tr>
<td>6</td>
<td>1.125</td>
<td>5.263</td>
<td>20.343</td>
<td>60.996</td>
<td>112.494</td>
<td>199.334</td>
</tr>
<tr>
<td>7</td>
<td>1.042</td>
<td>4.006</td>
<td>8.558</td>
<td>23.946</td>
<td>71.732</td>
<td>124.103</td>
</tr>
<tr>
<td>8</td>
<td>0.796</td>
<td>3.562</td>
<td>7.49</td>
<td>17.248</td>
<td>61.33</td>
<td>129.596</td>
</tr>
<tr>
<td>9</td>
<td>2.041</td>
<td>31.372</td>
<td>73.567</td>
<td>180.501</td>
<td>564.481</td>
<td>1235.537</td>
</tr>
<tr>
<td>10</td>
<td>1.417</td>
<td>10.17</td>
<td>48.982</td>
<td>111.892</td>
<td>375.591</td>
<td>808.436</td>
</tr>
<tr>
<td>11</td>
<td>0.535</td>
<td>3.141</td>
<td>12.469</td>
<td>61.462</td>
<td>345.267</td>
<td>868.941</td>
</tr>
<tr>
<td>12</td>
<td>1.174</td>
<td>14.553</td>
<td>72.442</td>
<td>153.517</td>
<td>496.535</td>
<td>945.617</td>
</tr>
<tr>
<td>13</td>
<td>0.754</td>
<td>3.051</td>
<td>7.559</td>
<td>12.31</td>
<td>122.478</td>
<td>515.845</td>
</tr>
<tr>
<td>14</td>
<td>1.411</td>
<td>21.57</td>
<td>77.374</td>
<td>254.267</td>
<td>706.489</td>
<td>1434.943</td>
</tr>
<tr>
<td>15</td>
<td>1.616</td>
<td>18.155</td>
<td>71.421</td>
<td>180.36</td>
<td>606.231</td>
<td>1294.182</td>
</tr>
<tr>
<td>16</td>
<td>1.343</td>
<td>23.463</td>
<td>64.13</td>
<td>188.332</td>
<td>702.503</td>
<td>1492.173</td>
</tr>
<tr>
<td>17</td>
<td>1.02</td>
<td>18.256</td>
<td>59.517</td>
<td>110.735</td>
<td>243.535</td>
<td>497.824</td>
</tr>
<tr>
<td>18</td>
<td>0.717</td>
<td>6.437</td>
<td>26.805</td>
<td>73.071</td>
<td>375.844</td>
<td>898.311</td>
</tr>
<tr>
<td>19</td>
<td>3.139</td>
<td>14.01</td>
<td>44.57</td>
<td>128.197</td>
<td>537.192</td>
<td>1392.206</td>
</tr>
<tr>
<td>avg</td>
<td>1.330</td>
<td>12.282</td>
<td>43.719</td>
<td>118.201</td>
<td>400.231</td>
<td>903.343</td>
</tr>
<tr>
<td>std</td>
<td>0.591</td>
<td>7.773</td>
<td>24.700</td>
<td>64.188</td>
<td>203.529</td>
<td>426.477</td>
</tr>
</tbody>
</table>
Table E.11: Constant Propagation Analysis with Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.974</td>
<td>7.342</td>
<td>28.19</td>
</tr>
<tr>
<td>1</td>
<td>1.52</td>
<td>18.624</td>
<td>110.615</td>
</tr>
<tr>
<td>2</td>
<td>0.943</td>
<td>84.386</td>
<td>445.107</td>
</tr>
<tr>
<td>3</td>
<td>0.851</td>
<td>5.802</td>
<td>25.686</td>
</tr>
<tr>
<td>4</td>
<td>1.422</td>
<td>23.843</td>
<td>845.822</td>
</tr>
<tr>
<td>5</td>
<td>0.544</td>
<td>4.478</td>
<td>23.82</td>
</tr>
<tr>
<td>6</td>
<td>0.748</td>
<td>4.003</td>
<td>15.315</td>
</tr>
<tr>
<td>7</td>
<td>0.691</td>
<td>2.873</td>
<td>6.184</td>
</tr>
<tr>
<td>8</td>
<td>0.509</td>
<td>2.526</td>
<td>5.388</td>
</tr>
<tr>
<td>9</td>
<td>2.15</td>
<td>28.218</td>
<td>341.789</td>
</tr>
<tr>
<td>10</td>
<td>0.962</td>
<td>9.021</td>
<td>50.5</td>
</tr>
<tr>
<td>11</td>
<td>0.308</td>
<td>2.256</td>
<td>8.642</td>
</tr>
<tr>
<td>12</td>
<td>0.801</td>
<td>10.741</td>
<td>245.702</td>
</tr>
<tr>
<td>13</td>
<td>0.507</td>
<td>2.166</td>
<td>5.48</td>
</tr>
<tr>
<td>14</td>
<td>1.027</td>
<td>30.874</td>
<td>260.405</td>
</tr>
<tr>
<td>15</td>
<td>1.091</td>
<td>16.966</td>
<td>2154.466</td>
</tr>
<tr>
<td>16</td>
<td>0.899</td>
<td>60.155</td>
<td>744.749</td>
</tr>
<tr>
<td>17</td>
<td>0.671</td>
<td>43.569</td>
<td>2360.17</td>
</tr>
<tr>
<td>18</td>
<td>0.449</td>
<td>5.398</td>
<td>15.757</td>
</tr>
<tr>
<td>19</td>
<td>2.331</td>
<td>19.719</td>
<td>58.043</td>
</tr>
<tr>
<td><strong>avg</strong></td>
<td><strong>0.970</strong></td>
<td><strong>19.148</strong></td>
<td><strong>387.592</strong></td>
</tr>
<tr>
<td><strong>std</strong></td>
<td><strong>0.531</strong></td>
<td><strong>21.789</strong></td>
<td><strong>686.011</strong></td>
</tr>
</tbody>
</table>
Table E.12: Constant Propagation Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX

<table>
<thead>
<tr>
<th>#Lines</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.002</td>
<td>40.669</td>
<td>242.711</td>
<td>1113.378</td>
</tr>
<tr>
<td>1</td>
<td>7.81</td>
<td>63.044</td>
<td>302.7</td>
<td>1222.376</td>
</tr>
<tr>
<td>2</td>
<td>5.919</td>
<td>51.232</td>
<td>235.395</td>
<td>823.564</td>
</tr>
<tr>
<td>3</td>
<td>6.729</td>
<td>35.742</td>
<td>206.003</td>
<td>867.132</td>
</tr>
<tr>
<td>4</td>
<td>7.27</td>
<td>41.875</td>
<td>335.909</td>
<td>1213.243</td>
</tr>
<tr>
<td>5</td>
<td>5.5</td>
<td>24.72</td>
<td>148.306</td>
<td>1014.991</td>
</tr>
<tr>
<td>6</td>
<td>6.478</td>
<td>24.772</td>
<td>118.436</td>
<td>600.144</td>
</tr>
<tr>
<td>7</td>
<td>5.957</td>
<td>18.904</td>
<td>57.039</td>
<td>222.969</td>
</tr>
<tr>
<td>8</td>
<td>5.388</td>
<td>24.914</td>
<td>147.222</td>
<td>598.983</td>
</tr>
<tr>
<td>9</td>
<td>7.693</td>
<td>124.522</td>
<td>362.342</td>
<td>1457.091</td>
</tr>
<tr>
<td>10</td>
<td>6.959</td>
<td>40.542</td>
<td>264.369</td>
<td>1025.524</td>
</tr>
<tr>
<td>11</td>
<td>5.025</td>
<td>14.954</td>
<td>71.356</td>
<td>492.73</td>
</tr>
<tr>
<td>12</td>
<td>6.115</td>
<td>53.613</td>
<td>340.174</td>
<td>1177.431</td>
</tr>
<tr>
<td>13</td>
<td>5.618</td>
<td>16.413</td>
<td>58.792</td>
<td>130.025</td>
</tr>
<tr>
<td>14</td>
<td>6.668</td>
<td>76.32</td>
<td>389.257</td>
<td>1868.802</td>
</tr>
<tr>
<td>15</td>
<td>6.994</td>
<td>71.586</td>
<td>421.304</td>
<td>1628.357</td>
</tr>
<tr>
<td>16</td>
<td>6.368</td>
<td>89.675</td>
<td>285.337</td>
<td>1358.728</td>
</tr>
<tr>
<td>17</td>
<td>5.792</td>
<td>64.558</td>
<td>281.961</td>
<td>794.575</td>
</tr>
<tr>
<td>18</td>
<td>5.483</td>
<td>29.663</td>
<td>162.367</td>
<td>733.926</td>
</tr>
<tr>
<td>19</td>
<td>11.032</td>
<td>47.256</td>
<td>224.513</td>
<td>946.079</td>
</tr>
<tr>
<td>avg</td>
<td>6.590</td>
<td>47.749</td>
<td>232.775</td>
<td>964.502</td>
</tr>
<tr>
<td>std</td>
<td>1.306</td>
<td>27.793</td>
<td>110.178</td>
<td>441.620</td>
</tr>
</tbody>
</table>
Bibliography


[26] **Greenman, B.** “Datalog for Static Analysis May 9, 2017,”.


