The Pennsylvania State University
The Graduate School

EXTENDING PARALLEL DATALOG WITH LATTICE

A Thesis in
Computer Science and Engineering
by
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

List of Tables

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 in our experiments, including the scalability of lattice scheme on extended Soufflé. The results of the experiments show that the lattice scheme on extended Soufflé significantly outperforms the traditional powerset scheme on normal input programs with branches. The current experiments are still on two dataflow analysis: Sign Analysis and Constant Propagation Analysis. In the future, the lattice framework can be used in other static analysis whose lattice is large size or infinite, such as point-to analysis. These static analyses should benefit from the lattice framework and gain higher efficiency.

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:

$$\forall X_1 \ldots \forall X_m (A \vee \neg L_1 \vee \ldots \vee \neg L_n) \tag{2.1}$$

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:

$$\forall X_1 \ldots \forall X_m (A \leftarrow L_1 \wedge \ldots \wedge L_n) \tag{2.2}$$

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 QRGT (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].

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.

$$\frac{true \leftarrow G_1 \wedge \ldots \wedge G_m \qquad G_i \leftarrow L_1 \wedge \ldots \wedge L_n}{true \leftarrow G_1 \wedge \ldots G_{i-1} \wedge L_1 \wedge \ldots \wedge L_n \wedge G_{i+1} \ldots \wedge G_m} \tag{2.3}$$

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: true ← G_1 ∧ ... ∧ G_m
2: repeat
3:     select G_i ∈ 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.

$$\begin{aligned} path(x,y) &\leftarrow edge(x,y). \\ path(x,y) &\leftarrow edge(x,t), path(t,y). \\ edge('a','b'). & \\ edge('b','c'). & \\ query(y) &\leftarrow path('a',y). \end{aligned} \tag{2.4}$$

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 $path(x,y) \leftarrow edge(x,y).$, 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: $path^{bf}$, 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: $edge^{bf}$. 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).

$$path^{bf}(x,y) \leftarrow edge(x,y). \tag{2.5}$$

$$
\begin{aligned}
path^{bf}(x,y) &\leftarrow edge(x,t), path^{bf}(t,y). \\
path^{bf}(x,y) &\leftarrow path^{ff}(t,y), edge(x,t).
\end{aligned} \tag{2.6}
$$

The *adornments* for *edge* in Eqs. (2.6) are bf and 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 $path^{ff}$ in Eqs. (2.6)leads to two new adornments of *path* as shown in Eq. (2.7) and Eqs. (2.8), which forms the third layer of SLD resolution.

$$
path^{ff}(x,y) \leftarrow edge(x,y). \tag{2.7}
$$

$$
\begin{aligned}
path^{ff}(x,y) &\leftarrow edge(x,t), path^{bf}(t,y). \\
path^{ff}(x,y) &\leftarrow path^{ff}(t,y), edge(x,t).
\end{aligned} \tag{2.8}
$$

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

This particular program can be called *PATH query* since the query is on the predicate *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 *query* in its head. The format of *query* is trivial. What's important is to express the *subquery* during the SLD resolution. We use another format of pair (R^γ, J) to define a subquery, where R is an IDB predicate, γ is the *adornment* of R, and J is a set of possible values for the bound variables by γ. Here, $path^{ff}(x,y)$ 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 $(query^f, \{\})$ in the first layer of SLD resolution, since there is no bound variables. The second layer subquery is $(path^{bf}, \{\langle 'a' \rangle\})$, with rules Eq. (2.5) and Eqs. (2.6). The third layer subquery can be expressed as $(path^{ff}, \{\})$, 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)

$$path^{bf}(x,y) \leftarrow [sup_0^1(x)]edge(x,y)[sup_1^1(x,y)]. \tag{2.9}$$

$$\begin{aligned} path^{bf}(x,y) &\leftarrow [sup_0^2(x)]edge(x,t),[sup_1^2(x,t)]path^{bf}(t,y)[sup_2^2(x,y)]. \\ path^{bf}(x,y) &\leftarrow [sup_0^3(x)]path^{ff}(t,y),[sup_1^3(x,t,y)]edge(x,t)[sup_2^3(x,y)]. \end{aligned} \tag{2.10}$$

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, $input\_path(x)$ and $ans\_path(x,y)$ are created for $path^{bf}(x,y)$. Intuitively, $input\_path(x)$ 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 $G_1, G_2 \dots G_m$ be the IDB predicates of program P. For each IDB predicate G_i, let Γ_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 Γ_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

$k := 0$
repeat
    $k := k + 1$
    for every i **do**
        $G_i^{(k)} := \Gamma_i(G_1^{(k-1)}, G_2^{(k-1)} \dots G_m^{(k-1)})$
    end for
until for every i, $G_i^{(k)} = G_i^{(k-1)}$
output $G_1^{(k)}, G_2^{(k)} \dots G_m^{(k)}$

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).

$$\begin{aligned} \Delta G_1^{(k+1)} &\leftarrow L_1, L_2, \Delta G_1^{(k)}, G_2^{(k-1)}. \\ \Delta G_1^{(k+1)} &\leftarrow L_1, L_2, G_1^{(k)}, \Delta G_2^{(k)}. \end{aligned} \tag{2.11}$$

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 has 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

```
k := 0
for every i do
    ΔG_i^(0) := G_i^(0)
end for
repeat
    k := k + 1
    for every i do
        ΔG_i^(k) := ΔΓ_i(G_1^(k−1), G_2^(k−1) ... G_m^(k−1), ΔG_1^(k−1), ΔG_2^(k−1) ... ΔG_m^(k−1)) − G_i^(k−1)
    end for
    for every i do
        G_i^(k) := G_i^(k−1) ∪ ΔG_i^(k)
    end for
until for every i, ΔG_i^(k) = ∅
output G_1^(k), G_2^(k) ... G_m^(k)
```

2.1.4.1 Magic Sets

As we can see from above, the top-down evaluation starts from the query and ignore 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).

$$\begin{aligned} &path('a', y) \leftarrow edge('a', y). \\ &path('a', y) \leftarrow edge('a', t), path(t, y). \end{aligned} \tag{2.12}$$

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 `bddbddb` 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 are 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 are four classic data-flow analyses 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 2.1: Data-Flow Analysis

	Must	**May**
Forward	Available Expression	Reaching Definition
Backward	Very Busy Expressions	Live Variables

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 $\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$. It is anti-symmetric when $\forall x, y \in S, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$.

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.

```
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.

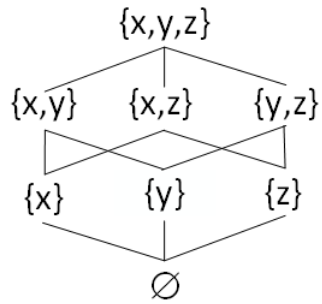


Figure 2.1: Powerset Lattice for Live Variables analysis

On the other hand, for **Must** analyses including *Available Expression* analysis and *Very Busy Expressions* analysis, the binary operator $\sqsubseteq$ 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 $(2^{\{x*2,x-y\}}, \supseteq)$, which is shown in Fig. 2.2.

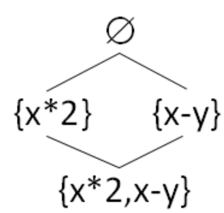


Figure 2.2: Reverse Powerset Lattice for Available Expression analysis

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.

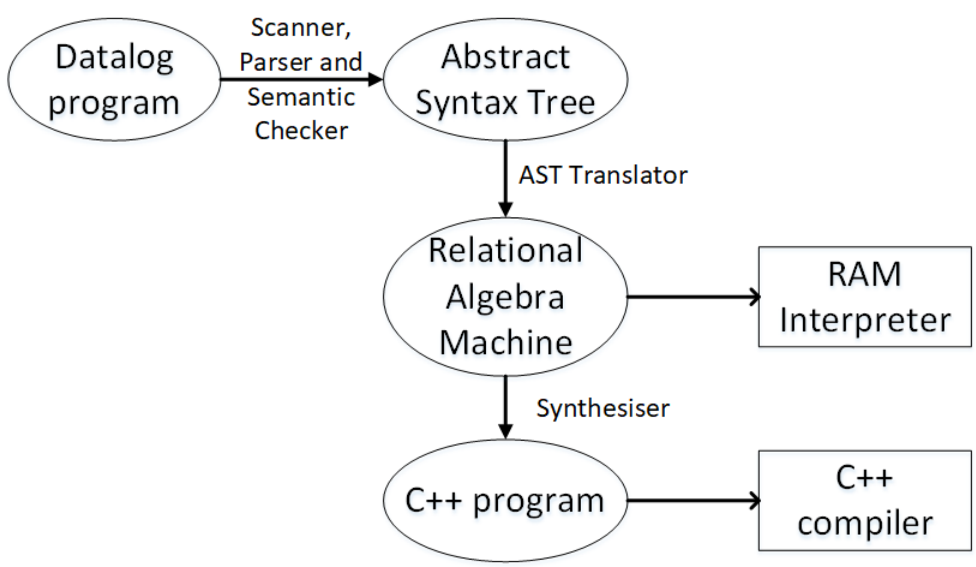


Figure 2.3: Soufflé architecture

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.

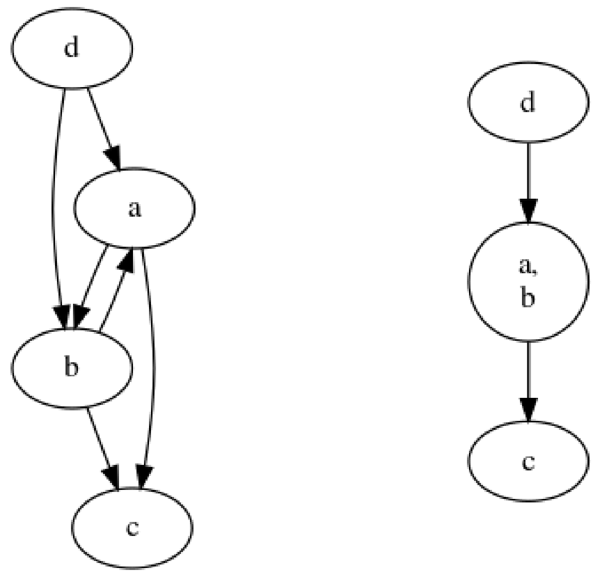
Listing 2.1: Example of Soufflé program

```
.decl a(x:number)
.decl b(x:number)
.decl c(x:number)
.decl d(x:number)

.output a, b, c

a(1).
b(1).
d(1).

a(x) :- b(x), d(x).
b(x) :- a(x), d(x).
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.



(a) Precedence Graph (b) SCC Graph

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

```
d = {1};
a = {1}; Δa = a;
b = {1}; Δb = b;
while (Δa ∪ Δb ≠ ∅) {
      new_a = (Δb ∩ d) \ a;
      new_b = (Δa ∩ d) \ b;
      a = a ∪ new_a;
      b = b ∪ new_b;
      Δa = new_a;
```

```
        Δb = new_b;
}
c = {};
c = a ∩ 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 **debug**, **load** and **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 **insert** stands for a relational algebra statement performing the evaluation of a clause. In the newer version(≥1.6.0) of Soufflé the terminology **insert** is modified to **query**.

$$
\begin{aligned}
& S \rightarrow \textbf{insert } O \\
& S \rightarrow S_1; \cdots ; S_2 \\
& S \rightarrow \textbf{loop } S_1 \textbf{ end\_loop} \\
& S \rightarrow \textbf{merge } R_1 \textbf{ with } R_2 \\
& S \rightarrow \textbf{parallel } S_1; \cdots ; S_k \textbf{ end\_parallel} \\
& S \rightarrow \textbf{begin\_stratum\_i } S_1 \textbf{ end\_stratum\_i} \\
& S \rightarrow \textbf{swap}\,(R_1,\ R_2) \\
& S \rightarrow \textbf{create } R_1\,(a_1, \cdots, a_n) \\
& S \rightarrow \textbf{clear } R_1 \\
& S \rightarrow \textbf{drop } R_1 \\
& S \rightarrow \textbf{insert }\ (V_1, \cdots, V_k)\ \textbf{ into } R_1 \\
& S \rightarrow \textbf{exit } C \\
& O \rightarrow \textbf{project }\ (V_1, \cdots, V_k)\ \textbf{ into } R_1 \\
& O \rightarrow \textbf{if } C\,\{O\} \\
& O \rightarrow \textbf{for } t_{id} \textbf{ in } R_i\ \{O\}
\end{aligned}
\tag{2.13}
$$

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 "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: `Live(x)`,`Live(y)` and `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 `true` or `false`, and is monotone when the booleans are ordered `false` $<$ `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 (`>`, `<`, `=`, `!=`, `>=` and `<=`) 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.

Chapter 3
Framework for Lattice

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 lattice as shown in Fig. 3.1 and a simple Datalog program as shown below,

```
A(6, yes).
B(6, no).
A(x, y) :- B(x, y).
C(x, y) :- A(x, y), B(x, y).
```

With the first rule, we should generate $A(6, \textit{no})$ and add it into IDB. Since there is $A(6, \textit{yes})$ 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

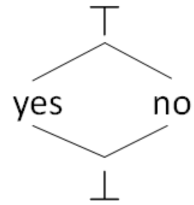


Figure 3.1: Simple lattice

only generate $C(6, no)$ since there is only $B(6, no)$. 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.

$$
\begin{aligned}
M_1 &= \{A(6, \top), \quad B(6, no), \quad C(6, yes), \quad C(6, no)\} \\
M_2 &= \{A(6, \top), \quad B(6, no), \quad C(6, \top)\} \\
M_3 &= \{A(6, \top), \quad B(6, no), \quad C(6, no)\}
\end{aligned}
$$

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, same 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. Meanwhile, an enum type can also cover the "number" type to handle lattices like constant propagation. To cover the primitive type: number in Soufflé, we need a special case: `.number_type`. Other than that, the enum type does not support other special data types. 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 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 `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 integers; 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. We added range range for the input facts, while 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 $(\bot, \top, \sqcup, \sqcap)$ as shown below, where $\bot$ is the bottom element, $\top$ is the top element, $\sqcup$ is the least upper bound function, and $\sqcap$ is the greatest lower bound function. Compared 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.

```
.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 `.lat` instead of `.decl` for such declaration. The last and only the last argument of lattice relation must be of the lattice type, which will be checked by extended Soufflé. 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.

```
.lat varSimple(n:number, v: Simple)
.input varSimple
.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 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

```
.lat a(x:Simple)
.lat b(x:Simple)
.lat c(x:Simple)
.lat d(x:Simple)

.output a, b, c

a("yes").
```

```
b("no").
d("no").

a(x) :- b(x), d(x).
b(x) :- a(x), d(x).
c(x) :- a(x), b(x).
```

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. In a `LATCLEAN` statement, 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. For example in the first iteration, the lattice relation `a` contains one tuple `a("yes")`, and the `new_a` at line 5 contains one tuple `new_a("no")`. 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. To achieve compactness, the `LATNORM` statement will traverse each cell in a relation, and keep only the least upper lattice element for each cell. For example, at the start of line 14 in Lst. 3.2, the lattice relation `a` will contain two tuples: `a("yes")` and `a("Top")`. We need to remove the tuple `a("yes")`. 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. In the interpreter of extended Soufflé, the `LATNORM` statement will traverse the whole relation in one thread and harm the parallel computing in Soufflé.

Listing 3.2: Modified Pseudo Code for RAM

```
1 d = {"no"};
2 a = {"yes"}; Δa = a;
3 b = {"no"}; Δb = b;
4 while (Δa ∪ Δb ≠ ∅) {
5       new_a = (Δb ∩* d) \ a;       // glb
6       new_a = (new_a ⊔ a) \ a;      // LATCLEAN
7       new_b = (Δa ∩* d) \ b;       // glb
```

```
        new_b = (new_b ⊔ b) \ b;        // LATCLEAN
        a = a ∪ new_a;
        b = b ∪ new_b;
        Δa = new_a;
        Δb = new_b;
}
a = Norm(a);            // LATNORM
b = Norm(b);            // LATNORM
c = {};
c = a ∩* b;             // glb
c = Norm(c);            // 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 be completed. Other than that, the implementation has been put on Github: `https://github.com/QXG2/souffle`. 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 string.

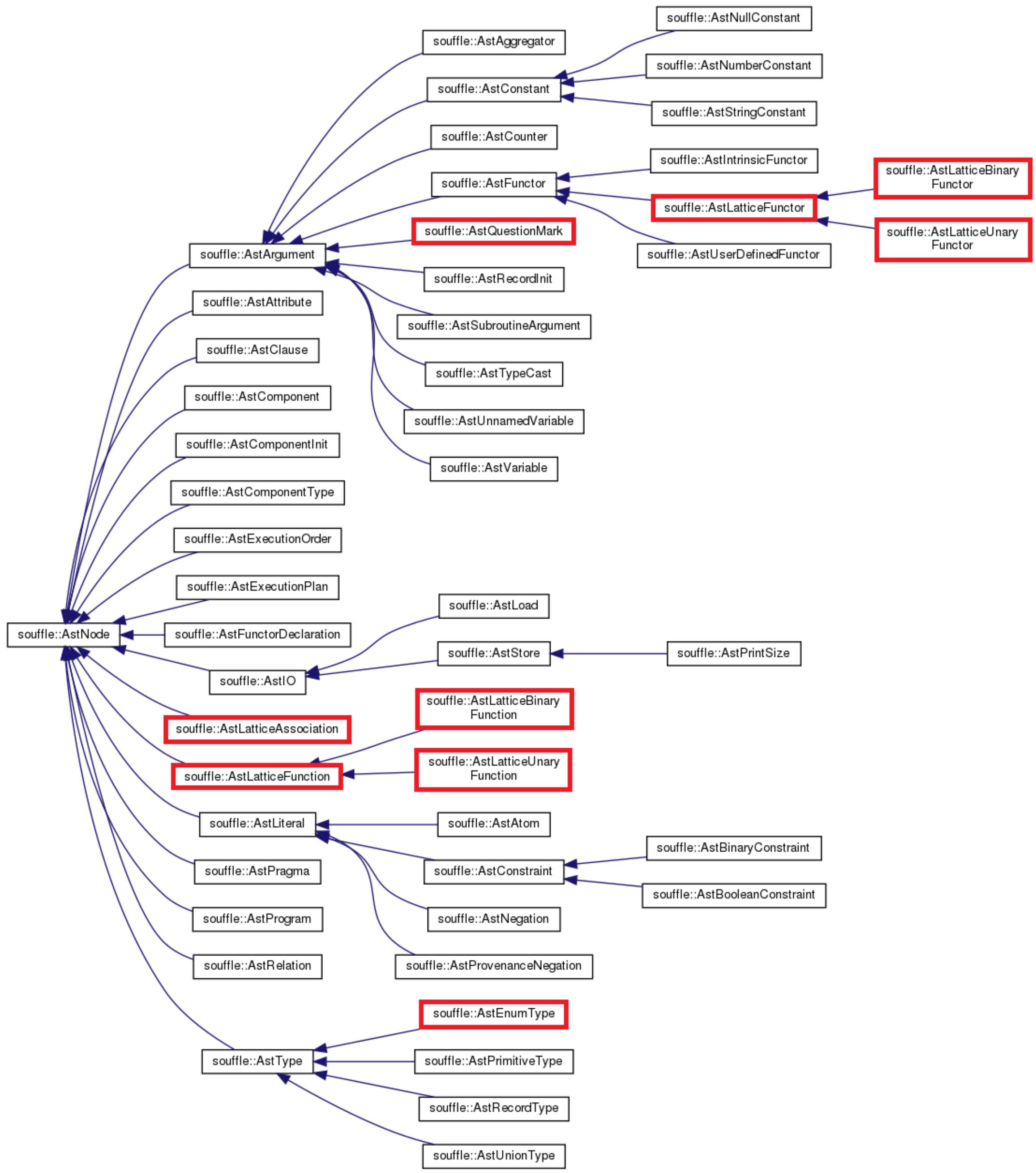


Figure 4.1: Class Hierarchy of Updated AstNode

To store all of the new syntax in AST, we add several classes under the base class *AstNode* as shown in Fig. 4.1. The name of these new classes speak for themselves. The class *AstLatticeFunctor* are calls to unary or binary case function. The class *AstLatticeFunction* stores the name of the function, the type of input and output arguments, and a vector of struct *PairMap*, each of which represents a case in the function such as `case ("Bot", _)=> 1`. The class *AstQuestionMark* is representation of the conditional operator. The class *AstLatticeAssociation* stores the name of top element, bottom element, `glb` and `lub` function for a lattice. These names will be used in the later translation to RAM program. The class *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 *ParseDriver* in addition to the updates in spec file for flex and bison. These functions include *ParserDriver::addLatticeFunction*, *ParserDriver::addLatticeAssociation* and *ParserDriver::addFunctorDeclaration*. Then the added CFG rules in *parser.yy* will call corresponding functions. For example, CFG rules involving `.let` will call *ParserDriver::addLatticeAssociation* to update *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 *main.cpp*, the call to function *AstTranslator::translateUnit* processes translation from *AstTranslationUnit* to *RamTranslationUnit*. Similar to the parsing to AST, there are some optimizations on the translated *RamTranslationUnit*. The class *RamTranslationUnit* contains *RamProgram*, a set of *RamAnalysis*, *SymbolTable*, *ErrorReport*, *DebugReport* and a mutex lock *analysisLock*. The class *RamProgram* stores all essential data for the RAM program, including *RamRelation*, *RamStatement*, *RamLatticeAssociation* and mapping from string to *RamLatticeUnaryFunction* and *RamLatticeBinaryFunction*.

Similar to AST, in order to store the new data about lattice in RAM program, we add several classes under the base class *RamNode* as shown in Fig. 4.2. Some of them are just the RAM version of AST nodes for new syntax, including *RamLatticeAssociation*, *RamLatticeFunction*, *RamLatticeFunctor* and *RamQuestionMark*. Both *RamLatticeFunction* and *RamQuestionMark* use *RamCondition* to represent the condition in them.

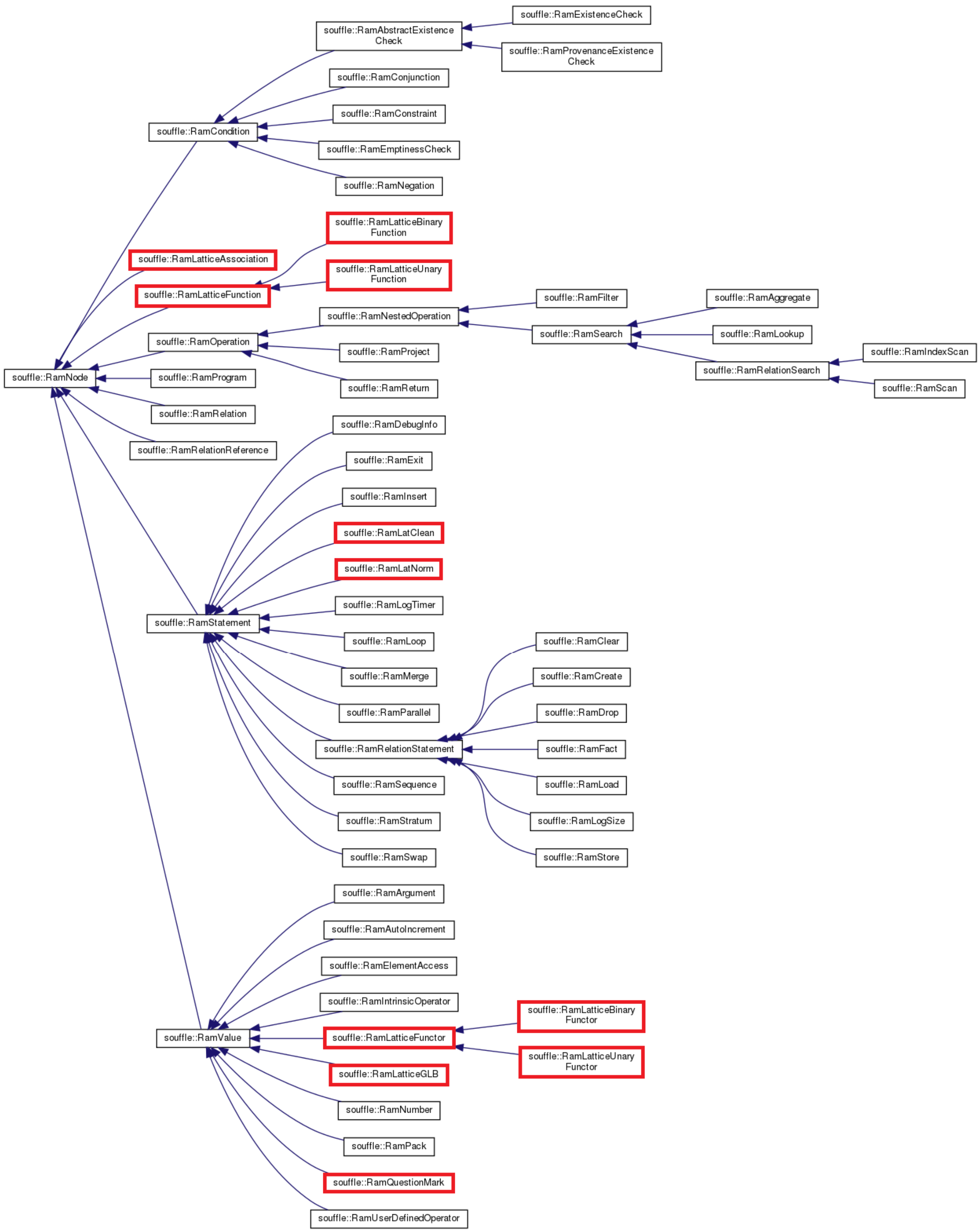


Figure 4.2: Class Hierarchy of Updated RamNode

In translation to *RamLatticeFunction*, the underscore _ which can match anything, is removed from the condition, so that the process of condition checking for interpreter can be simplified.

Besides, there are three new classes involving algorithms in RAM as mentioned in 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 existing 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_{origin}, and compute the least upper bound lattice for lattice values 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_{new}, and compute the least upper bound lattice considering tuples in both R_{new} and R_{origin}. If the computed tuple does not belong to R_{origin}, it is a true *new* tuple, otherwise it is a "fake" *new* tuple and will be ignored.

Algorithm 4 LATNORM algorithm

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

Algorithm 5 LATCLEAN algorithm

input: R_{origin}, R_{new}, output: R_{new_temp} (R is a lattice relation)
for each cell S_i **in** R_{new} **do**
    apply the least upper bound function $\sqcup$ to all tuples in S_i in R_{origin} and R_{new}, get the result t_i
    if $t_i \notin R_{origin}$ **then**
        insert t_i **into** R_{new_temp}
    end if
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 and least upper bound functions. And we discuss the effect of while loops 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. It's worth mentioning that the output sign for division operation on two positive values are $\top$, because the output could be zero or positive.

The corresponding Datalog program for the extended Soufflé is shown in Lst. 5.1. We define five symbols in the lattice `Sign`, as well as the least upper bound function `lub` and the greatest lower bound function `glb`. The evaluation of arithmetic operators are represented by other binary case functions: `lat_sum`, `lat_minus`, `lat_mult` and

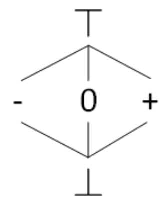


Figure 5.1: Lattice of Sign Analysis

Table 5.1: Evaluation of Operators in Sign Analysis

Addition(+)

	⊥	**0**	−	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	-	+	⊤
−	⊥	-	-	⊤	⊤
+	⊥	+	⊤	+	⊤
⊤	⊥	⊤	⊤	⊤	⊤

Minus(-)

	⊥	**0**	−	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	+	-	⊤
−	⊥	-	⊤	-	⊤
+	⊥	+	+	⊤	⊤
⊤	⊥	⊤	⊤	⊤	⊤

Multiplication(*)

	⊥	**0**	−	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	0	0	0	0
−	⊥	0	+	-	⊤
+	⊥	0	-	+	⊤
⊤	⊥	0	⊤	⊤	⊤

Division(/)

	⊥	**0**	−	+	⊤
⊥	⊥	⊥	⊥	⊥	⊥
0	⊥	⊥	0	0	⊤
−	⊥	⊥	⊤	⊤	⊤
+	⊥	⊥	⊤	⊤	⊤
⊤	⊥	⊥	⊤	⊤	⊤

`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

```
.enum Sign = {
              case "Top",

    case "Neg", case "Zer", case "Pos",

              case "Bot"
}

/// The least upper bound relation on the lattice elements.
.def lub(x: Sign, y: Sign): Sign {
    case ("Bot", _)   => y,
```

```
    case (_, "Bot")   => x,
    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",
    case (_, _)          => x=y ? "Pos" : "Neg"
}

.def lat_div(x: Sign, y: Sign): Sign {
    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)
```

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.

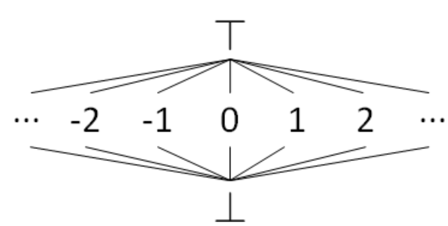


Figure 5.2: Lattice of Constant Propagation Analysis

Listing 5.2: Soufflé program for Constant Propagation Lattice Declaration

Table 5.2: Evaluation of Operators in Constant Propagation Analysis

Addition(+)

	$\bot$	$\boldsymbol{y}$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
$\boldsymbol{x}$	$\bot$	$x + y$	$\top$
$\top$	$\bot$	$\top$	$\top$

Minus(-)

	$\bot$	$\boldsymbol{y}$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
$\boldsymbol{x}$	$\bot$	$x - y$	$\top$
$\top$	$\bot$	$\top$	$\top$

Multiplication(*)

	$\bot$	$\boldsymbol{y}$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
$\boldsymbol{x}$	$\bot$	$x * y$	$\top$
$\top$	$\bot$	$\top$	$\top$

Division(/)

	$\bot$	$\boldsymbol{y}$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
$\boldsymbol{x}$	$\bot$	x/y	$\top$
$\top$	$\bot$	$\top$	$\top$

```
// number type is included as a special case
.enum Constant = {
    case "Top",
    case .number_type,
    case "Bot"
}

.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
}

.def lat_minus(x: Constant, y: Constant): Constant {
    case ("Bot", _)   => "Bot",
    case (_, "Bot")   => "Bot",
    case ("Top", _)   => "Top",
    case (_, "Top")   => "Top",
    case (_, _)          => x-y
}

.def lat_mult(x: Constant, y: Constant): Constant {
    case ("Bot", _)   => "Bot",
    case (_, "Bot")   => "Bot",
    case ("Top", _)   => "Top",
    case (_, "Top")   => "Top",
    case (_, _)          => x*y
}

.def lat_div(x: Constant, y: Constant): Constant {
    case ("Bot", _)   => "Bot",
    case (_, "Bot")   => "Bot",
    case ("Top", _)   => "Top",
    case (_, "Top")   => "Top",
    case (_, _)          => x/y
}

// assert lattice association
.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

```
.lat A(v: Sign)
.lat B(v: Sign)
.lat C(v: Sign)
.lat R(v: Sign)
.lat T(v: Sign)

.output R
.output T

A("Top").
B("Top").
C("Pos").
T("Neg").

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

5.3 While Loop

A significant advantage of using lattice is that it can handle while loops 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. varEntry(l, k, v) indicates that at the start of line l, the value of variable k is v. Only after one iteration of the while loop, we will have varEntry(3, "a", 1) and varEntry(3, "a", 2). With the application of least upper bound function (lub) on this cell, it will produce varEntry(3, "a", "Top"), and there will not be any other new tuples for the cell varEntry(3, "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

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

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

setConstStm(0, "a", 1).
setConstStm(1, "b", 1).
addStm(3, "a", "a", "b").
flow(0, 1).
flow(1, 2).
flow(2, 3).
flow(3, 2).
flow(2, 4).

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

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

```

```
// 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).

// 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 impractical with the traditional powerset scheme. In such a scheme, a new tuple for the cell `varEntry(3, "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
.lat varExit(l:number, k: symbol, v: Sign)
.output varExit

// extract lattice from number
varEntry(l, k, &lat_symbol(v)) :- varEntry_symbol(l, k, v).
varExit(l, k, &lat_symbol(v)) :- varExit_symbol(l, k, v).
```

The input programs for analysis are generated by a random program generator as shown in Appendix. B. The number of variables (`totVar`) is set to 5. The values in the probability vector (`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 $[0.2, 0.2, 0.2, 0.2, 0.2, 0, 0]$, where the probability for selecting "IF-ELSE statement" is 0. For random programs with branches, the probability vector is set to $[0.18, 0.18, 0.18, 0.18, 0.18, 0.1, 0]$, 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 seed of the 20 programs are $9527 + i * 17$, where i ranges from 0 to 19. 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

```
// function to transfer number to enum type
.def lat_alpha(x: number): Sign {
    case (_)          => x>0 ? "Pos" : (x<0 ? "Neg" : "Zer")
}

.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
```

```
.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
// 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).
// multiplication statement: r = x * y
varExit(l, r, &lat_mult(v1, v2)) :- multStm(l, r, x, y),
                                varEntry(l, x, v1),
                                varEntry(l, y, v2).
// division statement: r = x / y
varExit(l, r, &lat_div(v1, v2)) :- divStm(l, r, x, y),
                                varEntry(l, x, v1),
                                varEntry(l, y, v2), v2!=&lat_alpha(0).

// varEntry of l2 is the union of {varExit(l1) | flow(l1,l2)}
varEntry(l2, 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 programs has the exact same performance as the traditional powerset scheme on Soufflé programs. 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. Furthermore, there are 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

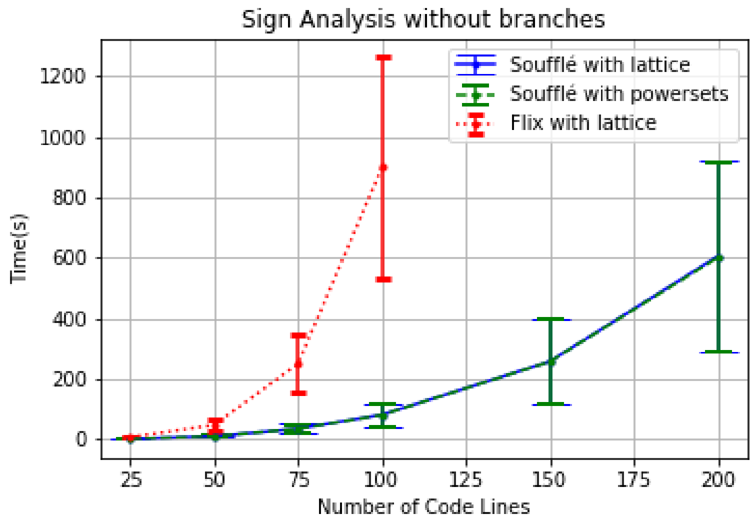


Figure 5.3: Performances on Sign Analysis without Branches

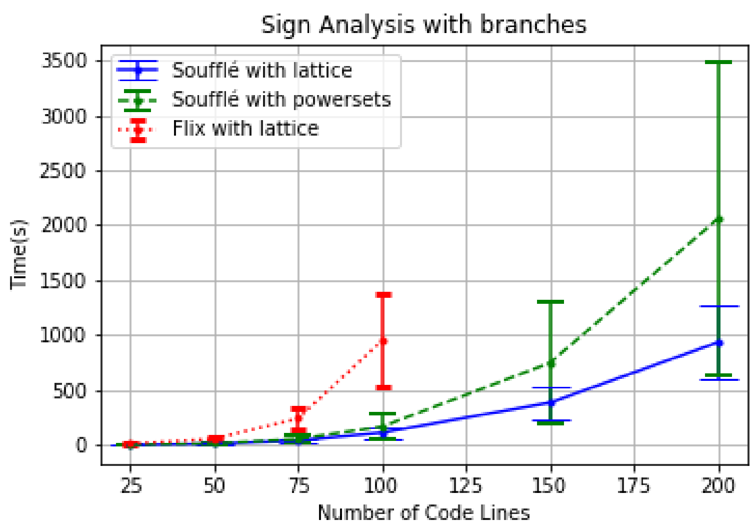


Figure 5.4: Performances on Sign Analysis with Branches

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 extremely long.

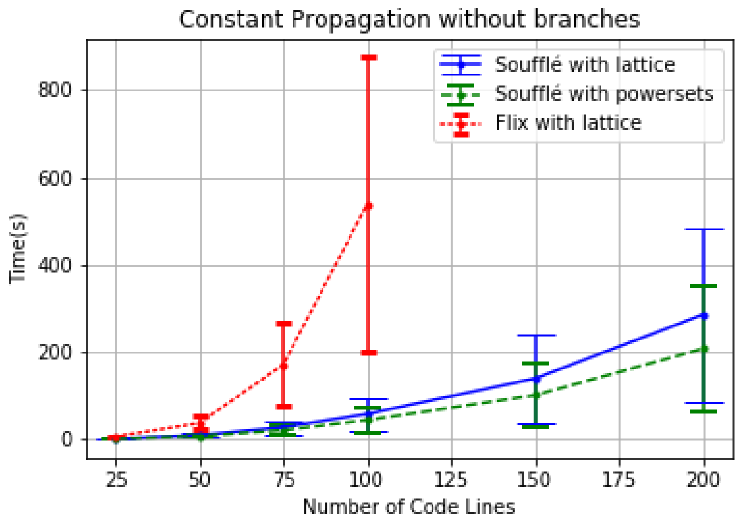
5.5 Scalability on Constant Propagation Analysis

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 same as presented in Lst. 5.5, except that `Sign` is replaced by `Constant` in lattice relations `varEntry` and `varExit`, and the `lat_alpha` function is different as shown below. The input programs for analysis are the same as the beforementioned 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 `varEntry(6, "a", 0)` and `varEntry(6, "a", 2)`, the lattice scheme will generate `varEntry(6, "a", "Top")`. Hence given another tuple `varEntry(6, "b", 8)` and a division operation at line 6: `c=b/a`, we will get `varExit(6, "c", "Top")`. However in traditional powerset scheme, the tuple `varEntry(6, "a", 0)` will be ignored by the rule, and we will get `varExit(6, "c", 4)` instead of `varExit(6, "c", "Top")`. Such difference is considered acceptable here.

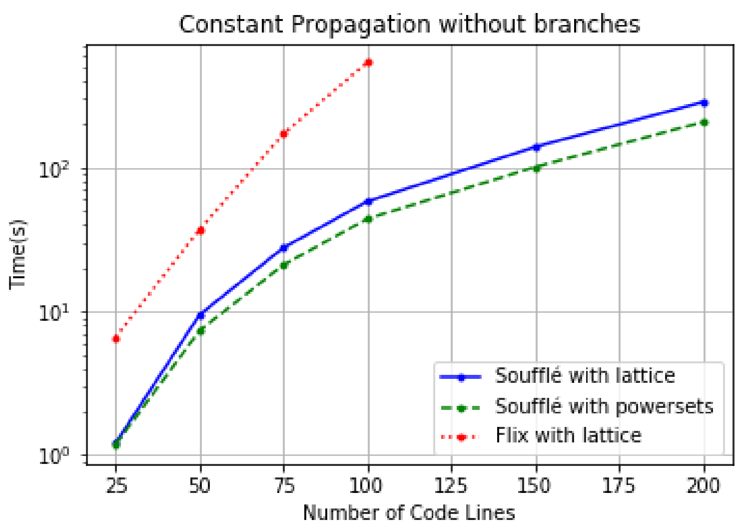
```
.def lat_alpha(x: number): Constant {
    case (_)          => x
}
```

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 **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 programs is slightly slower than the traditional powerset scheme on Soufflé programs, due to the extra computational cost of `LATCLEAN` and `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 **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



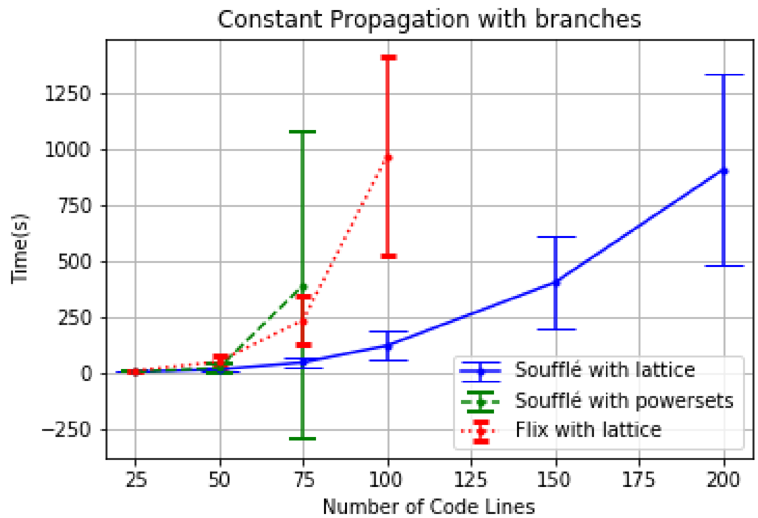
(a) Linear Scale



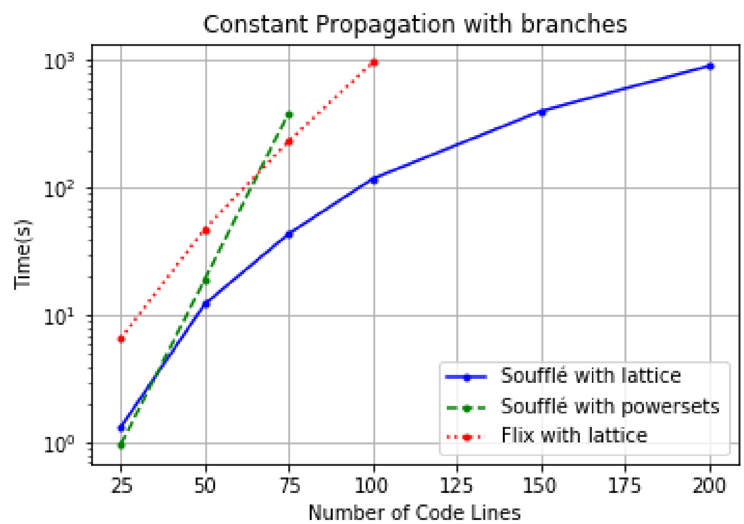
(b) Log-linear scale

Figure 5.5: Performances on Constant Propagation Analysis without Branches

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 efficiency of the lattice scheme on extended Soufflé is 887% of that of the powerset scheme on extended Soufflé, and



(a) Linear Scale



(b) Log-linear scale

Figure 5.6: Performances on Constant Propagation Analysis with Branches

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.

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`. 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. Moreover, the lattice framework can be used in other static analysis whose lattice is large size or infinite, such as point-to analysis.

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++ programs 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 `LATCLEAN` on *new* relations in each iteration of Semi-Naïve evaluation, we could reduce the usage of `LATCLEAN`. For example, we can use `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 `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 `LATCLEAN`). The insertion of `LATNORM` operation will reduce the size of lattice relation and avoid redundant computation. Similarly, there is a trade-off since `LATNORM` will bring extra computational cost and harm the concurrency. The last possible optimization is to run `LATCLEAN` and `LATNORM` in parallel, if we can find a way to separate cells evenly and distribute the computations to multiple threads.

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

```
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) a(x) b(x) c(x) d(x)
END DECLARATION
PROGRAM
BEGIN_STRATUM_0
    CREATE d(x) ;
    BEGIN_DEBUG "d(1). in file /home/gq/Programming/souffle/dataflowTest/Example_test/example.dl [10:1-10:6]"
        INSERT (number(1)) INTO d
    END_DEBUG
END_STRATUM_0;
BEGIN_STRATUM_1
    CREATE a(x) ;
    CREATE @delta_a(x) ;
    CREATE @new_a(x) ;
    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
           [13:1-13:20]"
            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)
glb:
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\"). 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\"). 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

```
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
```

```
        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 insertLineWithoutLable(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("DivFacts: ", self.DivFacts)
        print("Flow: ", self.Flow)
        print("Code:")
        print(self.Code)


# totVar: total number of variables
# minL: least lines requried
# 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.insertLineWithLable("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, [0.2,0.2,0.2,0.2,0.2,0,0], 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=b-c
d=0
c=d+e
a=c+a
c=c-a
d=e*c
c=2
d=d+b
d=2
c=b/b
d=e+a
e=a*a
d=2
c=-2
a=e+a
e=d-d
b=c/c
e=d/c
e=d*b
```

B.2.2 Sample Code with Branches

```
a=0
b=-1
c=1
d=0
e=-2
b=e+c
c=a-a
c=d-a
e=a+b
c=b*a
e=e/d
e=0
e=c-d
d=b+b
```

```
a=e/e
IF (condition):
 b=e-c
 e=e*c
 d=b*a
 b=c+a
 e=a+a
 e=b/b
 b=c-c
 d=e+e
 e=d/e
 c=d*c
 d=d*e
 b=a-b
 d=c/a
ELSE:
 b=a+c
 IF (condition):
 ELSE:
 d=-1
 c=b+d
 End IF
End IF
c=c+d
b=a*c
d=0
a=e+d
c=a*c
c=d*d
a=d+e
c=b/b
d=e+a
e=a/a
d=2
```

```
c=-2
a=e+a
e=d-d
```

Appendix C Soufflé Powerset Programs in Experiments

C.1 Sign Analysis

```
// 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"
}

.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
.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).
// minus statement: r = x - y
varExit_symbol(l, r, &nolat_minus(v1, v2)) :- minusStm(l, r, x, y),
                            varEntry_symbol(l, x, v1),
```

```
                            varEntry_symbol(l, y, v2).
// multiplication statement: r = x * y
varExit_symbol(l, r, &nolat_mult(v1, v2)) :- multStm(l, r, x, y),
                            varEntry_symbol(l, x, v1),
                            varEntry_symbol(l, y, v2).
// division statement: r = x / y
varExit_symbol(l, r, &nolat_div(v1, v2)) :- divStm(l, r, x, y),
                            varEntry_symbol(l, x, v1),
                            varEntry_symbol(l, y, v2),
                            v2!="Zer".

// r is not re-assigned
varExit_symbol(l, r, v) :- varEntry_symbol(l, r, v), !assignVar(l, r).
```

C.2 Constant Propagation Analysis

```
// input relations of statements
.decl setConstStm(l:number, r: symbol, c: number)         // r = c
.input setConstStm
.decl increStm(l:number, r: symbol)         // r++
.input increStm
.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
.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

// 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 assginment 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

```
enum Sign {
        case Top,

case Neg, case Zer, case Pos,

        case Bot
}

// statements in the test code
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

// 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
```

```
    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 (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

#Lines	25	50	75	100	150	200
0	1.419	12.244	45.379	111.606	383.119	907.821
1	1.482	12.068	30.121	47.659	117.653	309.894
2	1.186	11.774	41.316	85.243	152.974	348.848
3	1.504	13.167	44.842	89.987	167.016	432.359
4	0.917	10.746	42.209	103.968	353.264	875.228
5	1.472	13.088	44.279	103.224	338.271	732.925
6	0.953	3.822	10.007	15.987	52.435	122.758
7	1.419	13.949	50.198	117.179	383.197	872.203
8	0.539	2.048	3.572	6.501	17.344	82.093
9	1.459	13.054	38.702	103.029	356.861	746.531
10	1.537	14.031	47.707	115.875	399.984	962.825
11	0.338	3.349	21.238	66.676	278.914	756.136
12	0.866	2.333	4.107	6.68	25.978	129.162
13	0.783	4.424	23.856	68.698	278.695	740.145
14	1.452	12.217	45.808	111.641	372.462	882.843
15	1.457	12.981	43.48	111.873	386.874	922.538
16	1.335	13.357	46.416	109.661	263.497	459.81
17	1.521	10.87	22.79	33	73.188	144.092
18	0.846	8.76	36.017	91.124	342.42	859.862
19	1.692	15.003	46.828	115.884	395.023	806.43
avg	**1.209**	**10.164**	**34.444**	**80.775**	**256.958**	**604.725**
std	**0.380**	**4.360**	**15.036**	**38.506**	**137.733**	**314.027**

Table E.2: Sign Analysis without Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

#Lines	25	50	75	100	150	200
0	1.431	12.219	45.082	110.37	380.771	907.101
1	1.499	11.957	29.916	47.708	116.57	303.958
2	1.189	11.656	41.116	84.798	151.315	348.068
3	1.514	13.275	45.151	90.293	166.251	434.573
4	0.934	10.813	42.193	103.369	353.226	872.719
5	1.445	12.88	43.697	102.992	338.918	729.923
6	0.924	3.748	9.965	16.009	52.284	122.969
7	1.477	13.883	49.646	117.708	386.384	861.855
8	0.533	2.049	3.546	6.416	17.404	81.912
9	1.489	12.846	38.34	101.916	354.321	734.404
10	1.553	13.846	47.005	113.536	396.474	952.887
11	0.345	3.321	20.775	65.626	277.242	753.055
12	0.833	2.309	3.964	6.549	26.048	128.732
13	0.781	4.403	23.812	68.065	279.909	739.466
14	1.45	12.008	44.964	107.702	374.467	870.333
15	1.464	13.042	43.848	113.67	390.691	919.868
16	1.333	13.185	46.375	108.358	263.59	456.459
17	1.489	10.44	22.549	32.619	73.695	144.11
18	0.846	8.293	36.057	90.286	342.113	843.878
19	1.673	13.581	47.641	114.219	391.068	796.854
avg	**1.210**	**9.988**	**34.282**	**80.110**	**256.637**	**600.156**
std	**0.384**	**4.267**	**15.017**	**38.166**	**137.688**	**311.043**

Table E.3: Sign Analysis without Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX

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

E.2 Sign Analysis with Branches

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

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

Table E.5: Sign Analysis with Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

#Lines	25	50	75	100	150	200
0	1.432	15.942	53.024	140.099	574.589	1328.049
1	1.696	19.849	91.014	356.902	2018.934	4474.172
2	1.074	29.602	106.539	204.819	456.932	1164.379
3	1.511	10.867	38.446	119.845	747.659	2081.517
4	2.112	21.376	168.905	483.338	2143.539	5932.033
5	1.251	16.783	74.302	351.002	1388.203	3151.252
6	0.994	5.389	23.082	77.073	566.65	1607.079
7	1.125	11.313	35.495	108.14	420.352	917.46
8	0.713	3.693	8.068	18.817	60.913	133.858
9	1.626	23.967	59.541	164.263	772.903	2172.253
10	1.38	11.642	44.471	157.126	956.777	2901.884
11	0.421	3.205	13.236	99.093	576.94	1585.181
12	1.247	15.206	60.679	134.234	450.225	985.683
13	0.642	2.973	7.825	12.811	101.638	404.033
14	1.412	20.948	75.676	295.145	1164.436	3354.204
15	1.423	15.904	56.06	157.696	565.689	2050.406
16	1.356	15.695	36.608	92.615	578.682	2933.376
17	1.205	21.539	76.56	148.874	286.307	593.064
18	0.567	7.715	31.576	94.354	484.998	1869.327
19	3.497	14.791	48.907	118.579	612.645	1489.572
avg	**1.334**	**14.420**	**55.501**	**166.741**	**746.451**	**2056.439**
std	**0.653**	**7.376**	**37.785**	**118.639**	**553.250**	**1420.990**

Table E.6: Sign Analysis with Branches: Runtime(s) of Tests for the Lattice Scheme in FLIX

#Lines	25	50	75	100
0	14.719	67.912	302.923	1191.527
1	14.258	70.945	393.684	1336.273
2	9.445	56.459	245.607	804.947
3	12.569	60.059	285.518	1140.637
4	11.756	48.132	372.256	1324.018
5	12.587	59.022	248.231	1374.854
6	11.67	56.017	270.816	981.215
7	12.188	60.764	205.902	882.76
8	11.163	24.213	54.952	157.959
9	12.606	98.619	313.904	1219.709
10	13.158	47.806	254.649	1035.029
11	10.874	26.204	76.853	396.257
12	12.239	52.268	239.909	864.45
13	12.321	22.41	71.419	147.928
14	12.018	82.748	347.924	1868.991
15	12.686	69.945	324.078	1272.354
16	13.122	68.437	205.957	822.5
17	10.667	61.787	201.155	559.58
18	11.459	36.51	178.041	720.795
19	16.154	52.267	212.094	869.525
avg	**12.383**	**56.126**	**240.294**	**948.565**
std	**1.484**	**19.045**	**94.677**	**424.976**

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

#Lines	25	50	75	100	150	200
0	1.757	12.61	46.385	84.066	147.355	223.111
1	1.618	12.149	27.839	44.111	99.014	270.972
2	1.246	10.358	32.918	70.977	120.138	253.914
3	0.747	1.721	7.813	22.343	50.512	178.85
4	0.9	11.669	46.079	114.168	232.572	390.145
5	1.535	14.092	48.604	112.839	310.43	530.766
6	1.023	4.111	10.838	17.415	48.965	114.024
7	1.353	13.94	52.002	124.029	368.973	834.463
8	0.537	2.056	3.627	6.792	16.466	77.635
9	1.537	13.155	27.527	68.481	193.605	394.519
10	1.514	14.002	34.494	63.112	121.281	209.807
11	0.353	2.723	6.803	13.719	39.144	79.436
12	0.874	2.393	4.113	6.7	22.983	104.359
13	0.798	4.619	25.089	57.952	154.059	233.567
14	1.463	12.56	30.674	48.925	89.482	188.233
15	1.547	10.27	19.134	44.446	93.069	234.388
16	1.315	12.561	28.986	63.75	158.006	288.373
17	1.489	11.435	24.037	34.698	69.005	135.889
18	0.936	8.884	28.44	45.033	111.736	270.352
19	1.656	13.773	47.885	118.536	333.83	687.641
avg	**1.210**	**9.454**	**27.664**	**58.105**	**139.031**	**285.022**
std	**0.407**	**4.608**	**15.479**	**37.351**	**102.466**	**199.149**

Table E.8: Constant Propagation Analysis without Branches: Runtime(s) of Tests for the Powerset Scheme in Soufflé

#Lines	25	50	75	100	150	200
0	1.452	9.806	34.572	62.991	105.385	160.497
1	1.496	9.403	21.417	34.027	72.787	197.338
2	1.212	8.09	25.11	53.289	84.182	183.556
3	0.828	1.547	6.241	16.562	32.507	130.982
4	0.967	8.983	34.837	85.836	167.566	282.765
5	1.423	10.889	36.926	86.341	224.483	380.527
6	1.066	3.395	8.383	13.182	35.408	81.991
7	1.281	10.591	38.999	93.16	268.083	607.093
8	0.691	1.776	2.934	5.262	12.037	56.396
9	1.429	10.125	20.755	52.014	141.079	283.906
10	1.416	10.693	26.535	48.132	88.54	153.632
11	0.531	2.233	5.291	10.54	28.874	57.383
12	0.949	1.988	3.372	5.281	16.702	75.403
13	0.883	3.672	18.869	43.989	112.893	169.724
14	1.364	9.638	23.343	36.874	65.985	135
15	1.432	7.971	14.496	33.369	68.026	171.367
16	1.265	9.667	22.068	47.888	114.45	207.546
17	1.415	8.736	18.477	26.184	50.388	99.402
18	0.992	7.015	21.771	34.315	81.739	197.335
19	1.531	10.538	35.8	89.171	244.282	508.881
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

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

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

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

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

#Lines	25	50	75
0	0.974	7.342	28.19
1	1.52	18.624	110.615
2	0.943	84.386	445.107
3	0.851	5.802	25.686
4	1.422	23.843	845.822
5	0.544	4.478	23.82
6	0.748	4.003	15.315
7	0.691	2.873	6.184
8	0.509	2.526	5.388
9	2.15	28.218	341.789
10	0.962	9.021	50.5
11	0.308	2.256	8.642
12	0.801	10.741	245.702
13	0.507	2.166	5.48
14	1.027	30.874	260.405
15	1.091	16.966	2154.466
16	0.899	60.155	744.749
17	0.671	43.569	2360.17
18	0.449	5.398	15.757
19	2.331	19.719	58.043
avg	**0.970**	**19.148**	**387.592**
std	**0.531**	**21.789**	**686.011**

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

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

Bibliography

[1] MAIER, D., K. T. TEKLE, M. KIFER, and D. S. WARREN (2018) "Datalog: concepts, history, and outlook," in *Declarative Logic Programming: Theory, Systems, and Applications*, pp. 3–100.

[2] WIKIPEDIA (2020), "Datalog — Wikipedia, The Free Encyclopedia," `http://en.wikipedia.org/w/index.php?title=Datalog&oldid=938469317`, [Online; accessed 06-February-2020].

[3] GRECO, S. and C. MOLINARO (2015) "Datalog and logic databases," *Synthesis Lectures on Data Management*, **7**(2), pp. 1–169.

[4] DIMOPOULOS, Y., B. NEBEL, and J. KOEHLER (1997) "Encoding planning problems in nonmonotonic logic programs," in *European Conference on Planning*, Springer, pp. 169–181.

[5] ZHAO, D. (2017) *Large-Scale Provenance for Soufflé*, Master's thesis, The University of Sydney, Australia.

[6] ULLMAN, J. D. (1989) "Bottom-up beats top-down for datalog," in *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 140–149.

[7] RAMAKRISHNAN, R. and S. SUDARSHAN (1991) "Top-down vs. bottom-up revisited," in *Proceedings of the International Logic Programming Symposium*, pp. 321–336.

[8] RAMAKRISHNAN, R., D. SRIVASTAVA, and S. SUDARSHAN (1992) "Efficient bottom-up evaluation of logic programs," in *Computer Systems and Software Engineering*, Springer, pp. 287–324.

[9] KOWALSKI, R. (1974) "Predicate logic as programming language," in *IFIP congress*, vol. 74, pp. 569–544.

[10] FUCHI, K. (1981) "Aiming for knowledge information processing systems," in *Fifth Generation Computer Systems*.

[11] TEKLE, K. T. and Y. A. LIU (2011) "More efficient datalog queries: subsumptive tabling beats magic sets," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pp. 661–672.

[12] Tamaki, H. and T. Sato (1986) "OLD resolution with tabulation," in *International Conference on Logic Programming*, Springer, pp. 84–98.

[13] Sagonas, K., T. Swift, and D. S. Warren (1994) "XSB as an efficient deductive database engine," *ACM SIGMOD Record*, **23**(2), pp. 442–453.

[14] Vieille, L. (1986) "Recursive axioms in deductive databases: The query/subquery approach," in *Proc. First Intl. Conf. on Expert Database Systems, Charleston, 1986*, pp. 179–193.

[15] Abiteboul, S., R. Hull, and V. Vianu (1995) *Foundations of databases*, vol. 8, Addison-Wesley Reading.

[16] Koutris, P. (2016), "Lecture 8: Datalog: Evaluation," `http://pages.cs.wisc.edu/~paris/cs838-s16/lecture-notes/lecture8.pdf`.

[17] Bancilhon, F. (1986) "0. Maier, Y. Sagiv, and JD I'll man." Magic sets and Other Strange Ways to Implement Logic Program," in *Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Data-base Systems*.

[18] Beeri, C. and R. Ramakrishnan (1991) "On the power of magic," *The journal of logic programming*, **10**(3-4), pp. 255–299.

[19] Scholz, B., H. Jordan, P. Subotić, and T. Westmann (2016) "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, pp. 196–206.

[20] Balbin, I., G. S. Port, K. Ramamohanarao, and K. Meenakshi (1991) "Efficient bottom-up computation of queries on stratified databases," *The Journal of logic programming*, **11**(3-4), pp. 295–344.

[21] Tekle, K. T. and Y. A. Liu (2010) "Precise complexity analysis for efficient datalog queries," in *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 35–44.

[22] Scholz, B., K. Vorobyov, P. Krishnan, and T. Westmann (2015) "A datalog source-to-source translator for static program analysis: An experience report," in *2015 24th Australasian Software Engineering Conference*, IEEE, pp. 28–37.

[23] Whaley, J. and M. S. Lam (2004) "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 131–144.

[24] Dawson, S., C. R. Ramakrishnan, and D. S. Warren (1996) "Practical program analysis using general purpose logic programming systemsâĂŤa case study," in *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pp. 117–126.

[25] Alpuente, M., M. A. Feliú, C. Joubert, and A. Villanueva (2010) "Datalog-based program analysis with BES and RWL," in *International Datalog 2.0 Workshop*, Springer, pp. 1–20.

[26] Greenman, B. "Datalog for Static Analysis May 9, 2017," .

[27] Møller, A. and M. I. Schwartzbach (2012) "Static program analysis," *Notes. Feb.*

[28] Cain, A. A., T. Y. Chen, D. Grant, and J.-G. Schneider (2005) *Dynamic data flow analysis for object oriented programs*, Swinburne University of Technology, Faculty of Information & Communication âĂę.

[29] Tarski, A. et al. (1955) "A lattice-theoretical fixpoint theorem and its applications." *Pacific journal of Mathematics*, **5**(2), pp. 285–309.

[30] Stuart, T. (2013), "Compilers for Free," .
URL `https://codon.com/compilers-for-free`

[31] Subotić, P., H. Jordan, L. Chang, A. Fekete, and B. Scholz (2018) "Automatic index selection for large-scale datalog computation," *Proceedings of the VLDB Endowment*, **12**(2), pp. 141–153.

[32] Veldhuizen, T. L. (1998) "C++ templates as partial evaluation," *arXiv preprint cs/9810010*.

[33] Jordan, H., B. Scholz, and P. Subotić (2016) "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*, Springer, pp. 422–430.

[34] Zhao, D., P. Subotic, and B. Scholz (2019) "Provenance for Large-scale Datalog," *arXiv preprint arXiv:1907.05045*.

[35] Huhman, A. B. (2018) *Binary-Level Type Inference using Datalog*, Master's thesis.

[36] Capooci, C. (2018) *Performance Comparison of Three Datalog Engines*, Master's thesis, Pennsylvania State University.

[37] Madsen, M., M.-H. Yee, and O. Lhoták (2016) "From Datalog to flix: a declarative language for fixed points on lattices," *ACM SIGPLAN Notices*, **51**(6), pp. 194–208.

[38] ——— (2016) "Programming a Dataflow Analysis in Flix," *Tools for Automatic Program Analysis (TAPAS)*.

[39] Madsen, M. and O. Lhoták (2018) "Safe and sound program analysis with Flix," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 38–48.