Extending Abstract Interpretation to Dependency Analysis of Database Applications

Angshuman Jana\(^1\), Raju Halder\(^1\), K. V. Abhishekh\(^1\), S. D. Ganni\(^1\), and Agostino Cortesi\(^2\)

\(^1\)Indian Institute of Technology Patna, India
\(^2\)Università Ca’ Foscari Venezia, Italy

\{ajana.pcs13, halder, kalahasti.cs13, ganni.cs13\}@iitp.ac.in
cortesi@unive.it

Abstract—Dependency information (data- and/or control-dependencies) among program variables and program statements is playing crucial roles in a wide range of software-engineering activities, e.g. program slicing, information flow security analysis, debugging, code-optimization, code-reuse, code-understanding. Most existing dependency analyzers focus on mainstream languages and they do not support database applications embedding queries and data-manipulation commands. The first extension to the languages for relational database management systems, proposed by Willmor et al. in 2004, suffers from the lack of precision in the analysis primarily due to its syntax-based computation and flow insensitivity. Since then no significant contribution is found in this research direction. This paper extends the Abstract Interpretation framework for static dependency analysis of database applications, providing a semantics-based computation tunable with respect to precision. More specifically, we instantiate dependency computation by using various relational and non-relational abstract domains, yielding to a detailed comparative analysis with respect to precision and efficiency. Finally, we present a prototype semDDA, a semantics-based Database Dependency Analyzer integrated with various abstract domains, and we present experimental evaluation results to establish the effectiveness of our approach. We show an improvement of the precision on an average of 6% in the interval, 11% in the octagon, 21% in the polyhedra and 7% in the powerset of intervals abstract domains, as compared to their syntax-based counterpart, for the chosen set of Java Server Page (JSP)-based open-source database-driven web applications as part of the GotoCode project.

Index Terms—Dependency Graphs, Static Analysis, Relational Databases, Structured Query Languages.

1 INTRODUCTION

Static analysis is recognized as a fundamental approach to collect information about the behavior of computer programs for all possible inputs, without performing any actual execution [1]. Over the past several decades, continuous and concerted research efforts in this direction make them powerful enough to solve many non-trivial questions about program’s behavior, although they are undecidable in practice [1], [2]. Some notable and widely used static analysis techniques include Data-flow analysis [3], [4], Control-flow analysis [5], Type-based Theory [6], [7], [8], Abstract Interpretation [9], [10].

Observably most of the existing static analysis techniques in the literature make use, implicitly or explicitly, of dependency information among program statements and variables, solving a large number of software engineering tasks. Examples include information-flow security analysis [11], taint analysis [12], program slicing [13], optimization [14], [15], code-reuse [16], code-understanding [17]. A most common representation of these dependencies is Dependency Graph [18], [19], an intermediate form of programs which consists of both data- and control-dependencies among program components. Since the pioneer work by Ottenstein and Ottenstein [18], a number of variants of dependency graph for various programming languages are proposed by tuning them towards their suitable application domains. They are Program Dependency Graph (PDG) for intra-procedural programs [18], System Dependency Graph (SDG) for inter-procedural programs [20], Class Dependency Graph (CIDG) for object-oriented programs [21], Database-Oriented Program Dependency Graph (DOPDG) for database programs [22].

Although static analysis has been longly studied over the last several decades, researchers have not paid much attention to the case of database applications embedding database languages. In order to exploit the power of dependency graph in solving problems related to database applications, Willmor et al. [22] first introduced the notion of Database-Oriented Program Dependency Graph (DOPDG), considering the following two additional data dependencies due to the presence of database statements: (i) Program-Database dependency (PD-dependency) which represents dependency between an imperative statement and a database statement, and (ii) Database-Database dependency (DD-dependency) which represents a dependency between two database statements. However, since then no such notable contribution is found in this research direction. Some of the problems among many others which can effectively be addressed by using DOPDGs are:

(a) Slicing of Database Applications. Program slicing [23] is a well-known static analysis technique to address many software-engineering problems, including code understanding, debugging, maintenance, testing, parallelization, integration, software measurement [17], [24], [25], [26]. Existing program-slicing approaches have not considered external database states and therefore they are inapplicable to data-intensive programs in information system.
scenarios. It is imperative to say that slicing of database applications [27] based on their dependency information definitely serves as a powerful technique to solve the above-mentioned software-engineering problems relating query languages and underlying databases. In this context, preciseness of DOPDGs (hence slices) and their efficient computations are two prime factors which may affect the above-mentioned solutions to a great extent. This is yet to receive enough attention from the scientific community.

(b) **Database Leakage Analysis.** Language-based information-flow security analysis [28] has longly been studied during past decades to control illegitimate information leakage in software products. Needless to say, the confidentiality of sensitive database information can also possibly be compromised during their flow along database-applications accessing and processing them legitimately [29], [30]. The dependency information in the form of DOPDG can effectively capture any interference (if it exists) between sensitive and non-sensitive data. Of course, preciseness of dependency information highly matters to guarantee the absence of false security alarms in software products.

(c) **Data provenance.** Data provenance [31] is a static analysis technique which aids understanding and troubleshooting database queries by explaining the results in terms of input databases. Its intention is to show how (part of) the output of a query depended on (part of) its input. Precise dependency information among queries and identification of all parts of database information flowing along the program code are the basis of effective computations of data provenance.

(d) **Materialization View Creation.** Attribute dependencies are one of the prime factors for creating materialized views of databases [32]. The computation of precise static dependency information of database queries issued on a database over a certain period of time leads to a more precise materialized view creation.

A common challenge in all the above-mentioned application scenarios is to address the susceptibility of static dependency analysis to false positives, a main drawback of static analysis, which reduces development speed significantly. The best way to reduce false-positives is to allow tuning the analysis behavior towards specific needs. Our contribution in this paper on semantics-driven database dependency analyzer meets this challenge by facilitating precision control under various levels of abstractions.

To exemplify our motivation briefly, let us consider a small database code snippet, depicted in Figure 1, which increases salary of all employees by a common bonus amount $C_{bonus}$ and by an additional special bonus amount $Bonus$ only for aged employees.

A common challenge in all the above-mentioned application scenarios is to address the susceptibility of static dependency analysis to false positives, a main drawback of static analysis, which reduces development speed significantly. The best way to reduce false-positives is to allow tuning the analysis behavior towards specific needs. Our contribution in this paper on semantics-driven database dependency analyzer meets this challenge by facilitating precision control under various levels of abstractions.

Exemplified our motivation briefly, let us consider a small database code snippet, depicted in Figure 1, which increases salary of all employees by a common bonus amount $C_{bonus}$ and by an additional special bonus amount $Bonus$ only for aged employees. Observe that the syntactic presence of 'sal' as the defined-variable in $Q_1$ and as the used-variable in $Q_3$ makes $Q_3$ syntactically dependent on $Q_1$. However, a careful observation reveals that syntactic presence of variables as a way of dependency computation may often result in false positives, and thus fails to compute optimal set of dependencies. For instance, it is clear from the code that the values of 'sal' referred in the "WHERE" clauses of $Q_3$ and $Q_5$ do not overlap with each other and this results in an independency between $Q_3$ and $Q_5$. This triggers a semantics-based approach to compute dependency where values instead of variables are considered. In this context, the following research question arises: Are the values defined by one statement being used by another statement? The problem to compute semantics-based dependency among statements in concrete domain is in general undecidable [2], [33]. This is also true in the case of database applications when the input database instance is unknown. Addressing similar problems in imperative languages, Mastroeni and Zanardini [34] introduced the notion of abstract semantics-based data dependency in the Abstract Interpretation framework. Abstract Interpretation [9], [10] is a widely used formal method which offers a sound approximation of the program’s semantics to answer about the program’s runtime behavior including undecidable ones. The intuition of Abstract Interpretation is to lift the concrete semantics to an abstract domain, by replacing concrete values by suitable properties of interests and simulating the operations in the abstract domain w.r.t. its concrete counterparts, in order to ensure sound semantic approximation.

Willmor’s definition for DOPDG is not fully semantics-based [22]: although they define DD-dependency in terms of defined- and used-values of databases, their definition of PD-dependency relies on the syntactic presence of variables and attributes in statements. Intuitively, the precision of DOPDG depends on how precisely one can identify the overlapping of database-parts by various database operations (INSERT, UPDATE, DELETE). Although they refer to the Condition-Action rules [35] to compute the overlapping of database-parts, this fails to capture semantic independencies when the application contains more than one database statements defining (in sequence) the same attribute which is subsequently used by another database statement. The main reason behind this is the flow-insensitivity of the Condition-Action rules. For example, in Figure 1, $Q_5$ is semantically independent on $Q_1$ as the part of sal-values defined by $Q_1$ is fully redefined by $Q_4$ and never reaches $Q_5$. Unfortunately, Condition-Action rules can not capture this independency as the approach checks every pair of database statements independently, and as a result, this finds dependency when the pair $Q_1$ and $Q_3$ is encountered.

As the values of database attributes differ from that of imperative language variables, the computation of abstract semantics (and hence semantics-based dependency) of database applications is, however, challenging and requires different treatment. The key point here is the static identification of various parts of the database information possibly accessed or manipulated by database statements at various program points. Addressing these challenges, in this paper, we aim to answer the following two main research objectives:

---

**Fig. 1: An Introductory Example**

```
Start;
Q_0: Connection c = DriverManager.getConnection(......);
Q_1: UPDATE emp SET sal = sal + Bonus WHERE age > 60;
Q_2: SELECT AVG(sal) FROM emp WHERE age > 60;
Q_3: SELECT AVG(sal) FROM emp WHERE age < 60;
Q_4: UPDATE emp SET sal = sal + Bonus;
Q_5: SELECT AVG(sal) FROM emp;
Stop;
```
• How to obtain more precise dependency information (hence more precise DOPDG)? and
• How to compute them efficiently?

To summarize, our contributions in this paper¹ are:

• Adapting the Abstract Interpretation framework to define computable abstract semantics of database applications, even in an undecidable scenario when the input database instance is unknown.
• Design of an algorithm to compute semantics-based independencies among database statements based on Abstract Semantics.
• A detailed analytical study on precision vs. efficiency when computing dependency in various well-suited non-relational and relational abstract domains, e.g. Interval, Octagon, Polyhedra, Powerset Domain.
• Development of a prototype “semDDA”, Semantics-based Database Dependency Analyzer integrated with various abstract domains, which enables users to perform precise dependency computation in various abstract domains of interest.
• Experimental evaluation on a set of open-source database-driven JSP web applications as part of the GotoCode project [37] using our semDDA tool². Experiments demonstrate the results in different abstract domains with a detailed comparison on precision and efficiency. This clearly shows that our technique improves precision w.r.t. the proposal by Willmor et al. [22].

Our preliminary theoretical proposal in [36] considered only polyhedra abstract domain. To be specific, the improvements in this paper compared to [36] are: (i) strengthening the approach by instantiating dependency computation using various non-relational and relational abstract domains, yielding a detailed comparative analysis with respect to precision and efficiency, (ii) implementation of semDDA which is tunable at various levels of abstractions, (iii) validation of our approach by performing experiments on a set of open-source database-driven JSP web applications using semDDA, establishing the effectiveness of our approach w.r.t. the literature.

The structure of the rest of paper is as follows: Section 2 illustrates a running example. In section 3 we describe the evolution of syntax-based dependency computation of database programs. Section 4 recalls some basics on syntax and concrete semantics of programs embedding SQL statements. In section 5 we describe semantics-based dependency computation in the concrete domain, whereas in section 6 we lift semantics-based dependency computation from the concrete domain to various non-relational and relational abstract domains. Section 7 illustrates the proposed framework on our running example. The soundness of our approach is proved in section 8. Section 9 describes the experimental results on a set of benchmark codes. We present in section 10 a case study on database code slicing, witnessing an improvement in the precision, by applying the proposed semantics-based dependency computation framework. We discuss in section 11 the current state-of-the-art in the literature. Finally, section 12 concludes our work.

2 A Running Example

Consider the database code snippet “Prog” depicted in Figure 2. The code implements a module which provides a set of offers on various purchases made on an online shopping system.

The main method of the class saleOffer updates the purchase amount (stored in the attribute purchase_amt) depending on various discount offers. For instance, a customer will get 5% discount if the purchase amount is between 1000 USD and 3000 USD. Similarly, a 10% of discount is offered on the purchase amount more than 3000 USD. A special offer on waiving delivery charges is also given for all customers (program point 7). Finally, the module increments the points accumulated by its customers depending on both the purchase amount and the wallet balance at program points 15 and 16.

Observing the code carefully, we can identify a number of dependencies among the statements in “Prog”. Some of them, although exist syntactically, may not be valid dependencies when we consider semantics of the program. For example, although statement 6 is syntactically DD-dependent on statement 5, but they are semantically independent as the values of the attribute purchase_amt defined by statement 5 can never be used by statement 6. In the subsequent sections, we pursue various existing approaches to refine dependency information, and finally we propose an abstract interpretation-based approach to approximate defined and used database parts by database statements (at various levels of abstractions) and hence to compute semantics-based dependencies among them based on the overlapping. We show, in section 7, how the proposed approach effectively identifies false DD-dependencies in “Prog”.

3 Revisiting Syntax-based Dependency Computation in Database Applications

This section briefly discusses the evolution of syntax-based Database-Oriented Program Dependency Graph (DOPDG) construction and its limitations w.r.t. the literature. Throughout this paper we shall use the terms “Program” and “Database Program” synonymously. Similarly, we shall use the term “Statement” which synonymously refers to either “imperative statement” or “database statement” depending on the context.

3.1 Pure Syntax-based DOPDGs

The construction of pure syntax-based Database-Oriented Program Dependency Graph (DOPDG) is straightforward. It is an extension of traditional Program Dependency Graphs (PDGs) [18] to the case of database programs, considering the following three kinds of data-dependencies: (1) Program-Program dependency (PP-dependency) which represents a dependency between two imperative statements, (2) Program-Database dependency (PD-dependency) which represents a dependency between a SQL statement

1. This work is a revised and extended version of [36].
2. Available at: https://github.com/angshumanjana/SemDDA.
and an imperative statement, and (3) Database-Database dependency (DD-dependency) which represents a dependency between two SQL statements. This is to observe that syntax-based PP-dependencies and control dependencies in DOPDGs are the same as syntax-based data-dependencies and control-dependencies in PDGs respectively. Let us define them below:

**Definition 1 (Program-Program (PP) dependency [18]).** An imperative statement \( I_2 \) is PP-dependent on another imperative statement \( I_1 \) if there exists an application variable \( x \) such that: (i) \( x \) is defined by \( I_1 \), (ii) \( x \) is used by \( I_2 \), and (iii) there is a \( x \)-definition free path from \( I_1 \) to \( I_2 \).

**Definition 2 (Program-Database (PD) dependency [22]).** A database statement \( Q \) is PD-dependent on an imperative statement \( I \) if there exists an application variable \( x \) such that: (i) \( x \) is defined by \( I \), (ii) \( x \) is used as an input to \( Q \), and (iii) there is a \( x \)-definition free path from \( I \) to \( Q \). Similarly, an imperative statement \( I \) is PD-dependency on a database statement \( Q \) if there exists an application variable \( x \) such that: (i) the execution of \( Q \) sets \( x \) to be equal to one of the output of \( Q \), (ii) \( x \) is used by \( I \), and (iii) there is a \( x \)-definition free path from \( Q \) to \( I \).

**Definition 3 (Database-Database (DD) dependency).** A database statement \( Q_2 \) is DD-dependent on another database statement \( Q_1 \) for an attribute \( a \) (denoted \( Q_1 \rightarrow_a Q_2 \)) if the following conditions hold: (i) \( a \) is defined by \( Q_1 \), (ii) \( a \) is used by \( Q_2 \), and (iii) there is no rollback operation in between them, which undoes the effect of \( Q_1 \) on \( a \).

The syntax-based dependency computation depends on the syntactic presence of one variable in the definition of another variable or on the control structure of the program. Let \( C \), \( V_a \) and \( V_d \) be the sets of statements, application-variables and database-attributes in database programs. Let

\[
V = V_a \cup V_d \quad \text{where} \quad V_a \cap V_d = \emptyset.
\]

The construction of syntax-based DOPDG can be formalized based on the two following functions:

\[
\begin{align*}
\text{USE} & : C \rightarrow \wp(\wp(\wp(V))) \\
\text{DEF} & : C \rightarrow \wp(\wp(\wp(V)))
\end{align*}
\]

which extract the set of variables (either application-variables or database-attributes) used and defined in a statement \( c \in C \).

The following example illustrates the construction of pure syntax-based DOPDG using the above functions.

\[\begin{array}{c}
0. \text{public class saleOffer}
\end{array}\]

\[\begin{array}{c}
1. \text{public static void main(String[] args) throws SQLException}
\end{array}\]

\[\begin{array}{c}
2. \text{float x = 0.1;}
\end{array}\]

\[\begin{array}{c}
3. \text{float y = 0.05;}
\end{array}\]

\[\begin{array}{c}
4. \text{try (Statement con = DriverManager.getConnection("jdbc mysql: . . . ", "scott", "tiger");}
\end{array}\]

\[\begin{array}{c}
5. \text{con.createStatement();}
\end{array}\]

\[\begin{array}{c}
6. \text{con.executeUpdate("UPDATE Sales SET purchase_amt = purchase_amt − x • purchase_amt WHERE purchase_amt < 3000 ");}
\end{array}\]

\[\begin{array}{c}
7. \text{catch (Exception e) { [ ] }}
\end{array}\]

\[\begin{array}{c}
8. \text{con.executeUpdate("UPDATE Sales SET purchase_amt = purchase_amt − delivery_charge ");}
\end{array}\]

\[\begin{array}{c}
9. \text{...}
\end{array}\]

**Fig. 2:** Database Code Snippet “Prog”

\[\begin{array}{c}
10. \text{ResultSet rs = con.executeQuery("SELECT cust_name, purchase_amt FROM Sales WHERE purchase_amt > 10000 ");}
\end{array}\]

\[\begin{array}{c}
11. \text{...}
\end{array}\]

**Fig. 3:** Pure Syntax-based DOPDG (★ denotes attribute \( \text{purchase_amt} \) of “Prog”)

**Example 1.** Consider our running example “Prog” depicted in Figures 2. The control dependencies 1→2, 1→3, 1→4, etc. are computed in similar way as in the case of traditional PDG. The used and defined variables at each program point of “Prog” are computed as follows:

3. In the rest of the paper, we represent DD-, PD-, control-dependencies by blue dashed-line, red dotted-line and black line respectively.
DEF(2) = {x}  DEF(3) = {y}  
DEF(4) = [{purchase_amt, delivery_charge, cust_name, 
          wallet_bal, point}]  
DEF(5) = [{purchase_amt}]  USE(5) = [{purchase_amt, y}]  
DEF(6) = [{purchase_amt}]  USE(6) = [{purchase_amt, x}]  
DEF(7) = [{purchase_amt}]  USE(7) = [{purchase_amt, delivery_charge}]  
USE(11) = [{purchase_amt, cust_name}]  
DEF(15) = [{point}]  
USE(15) = [{purchase_amt, wallet_bal, point}]  
DEF(16) = [{point}]  
USE(16) = [{purchase_amt, wallet_bal, point}] 

Observe that statement 4 defines all database attributes as it connects to the database, resulting DEF(4) to contain all attributes. From the above information, the following data dependencies are identified:

- DD-dependencies for purchase_amt: 4 → 5 , 4 → 6 , 4 → 7 , 4 → 11 , 4 → 15 , 4 → 16 , 5 → 6 , 5 → 7 , 5 → 11 , 5 → 15 , 6 → 16 , 6 → 7 , 6 → 11 , 6 → 15 , 7 → 16 ,

- DD-dependencies for other attributes: 4 → 7 , 4 → 11 , 4 → 15 , 4 → 16 , 15 → 16

- PD-dependencies for x and y: 2 → 6 , 3 → 5

The syntax-based DOPDG of “Prog” is depicted in Figure 3.

Limitations. Syntax-based dependency computation often introduces false dependencies, leading to an imprecise analysis. For instance, in Example 1, although the statement 6 is syntactically DD-dependent on statement 5, however one can observe that the values of the attribute purchase_amt defined by statement 5 can never be used by statement 6. This is also true for 15 → 16. Similarly observe that the redefinition of all values of purchase_amt at program point 7 makes the statements 11, 15 and 16 data-independent on statements 4, 5 and 6 for purchase_amt, which is not captured here.

3.2 An Improved Syntax-driven Construction of DOPDGs

We proposed in [38] an improvement over the syntax-driven DOPDG construction algorithm by tagging variables with labels which indicate whether a variable is fully-defined or partially-defined. This enables us to (partially) identify a number of false dependencies.

The modified definitions of USE and DEF functions are as follows:

\[
\text{USE} : C \rightarrow \varphi(V \times L) \\
\text{DEF} : C \rightarrow \varphi(V \times L)
\]  

where \( L = \{\bullet, \circ\} \) is a set of labels. The label \( \bullet \) associated with an attribute \( a \) indicates that \( a \) is fully-defined – which means all values of \( a \) in the database are defined by the database statement. On the other hand, the label \( \circ \) associated with \( a \) indicates that \( a \) is partially-defined – which means only a subset of the values of \( a \) in the database are defined. Observe that these fully- and partially-defined distinctions are also applicable to program variables representing collections, such as arrays, lists, etc. For ordinary variable holding single value, the label is by default \( \bullet \) (i.e., fully-defined). Let us illustrate this on our running example.

Example 2. Applying equations 3 and 4 on all statements in “Prog” of the running example, we get the following information:

\[
\begin{align*}
\text{DEF(2)} &= \{(x, \bullet)\} & \text{DEF(3)} &= \{(y, \bullet)\} \\
\text{DEF(4)} &= \{(\text{purchase}_\text{amt}, \bullet), (\text{cust}_\text{name}, \bullet), (\text{point}, \bullet), (\text{wallet}_\text{bal}, \bullet), (\text{delivery}_\text{charge}, \bullet)\} \\
\text{DEF(5)} &= \{(\text{purchase}_\text{amt}, \circ)\} \\
\text{USE(5)} &= \{(\text{purchase}_\text{amt}, y)\} \\
\text{DEF(6)} &= \{(\text{purchase}_\text{amt}, \bullet)\} \\
\text{USE(6)} &= \{(\text{purchase}_\text{amt}, y, \bullet)\} \\
\text{DEF(7)} &= \{(\text{purchase}_\text{amt}, \bullet)\} \\
\text{USE(7)} &= \{(\text{purchase}_\text{amt}, \bullet), (\text{delivery}_\text{charge}, \bullet)\} \\
\text{USE(11)} &= \{(\text{purchase}_\text{amt}, \bullet), (\text{cust}_\text{name}, \bullet)\} \\
\text{DEF(15)} &= \{(\text{point}, \bullet)\} \\
\text{USE(15)} &= \{(\text{purchase}_\text{amt}, \bullet), (\text{wallet}_\text{bal}, \bullet), (\text{point}, \bullet)\} \\
\text{DEF(16)} &= \{(\text{point}, \bullet)\} \\
\text{USE(16)} &= \{(\text{purchase}_\text{amt}, \bullet), (\text{wallet}_\text{bal}, \bullet), (\text{point}, \bullet)\}
\end{align*}
\]

The above information results in the following refined set of data dependencies:

- DD-dependencies for purchase_amt: 4 → 5 , 4 → 6 , 4 → 7 , 5 → 6 , 5 → 7 , 6 → 7 , 7 → 11 , 7 → 15 , 7 → 16 ,

- DD-dependencies for other attributes: 4 → 7 , 4 → 11 , 4 → 15 , 4 → 16 , 15 → 16

- PD-dependencies for x and y: 2 → 6 , 3 → 5

The label \( \bullet \) associated with purchase_amt in DEF(7) indicates that all values of purchase_amt are defined at program point 7. This means that all definitions of purchase_amt before 7 does not reach any of its use after 7, identifying false DD-dependencies 4 → 11 , 5 → 11 , 6 → 11 , 4 → 15 , 5 → 15 , 6 → 15 , 4 → 16 , 5 → 16 and 6 → 16 for purchase_amt. Observe that the DD-dependency 4 → 11 exists for cust_name and dependencies 4 → 15 , 4 → 16 exist for both wallet_bal and point. The improved syntax-based DOPDG of “Prog” is depicted in Figure 4.

Limitations. This improved DOPDG construction approach also fails to compute optimal dependency results, because of its syntactic bound. For example, the false DD-dependencies 5 → 6 for purchase_amt and 15 → 16 for point still remain unidentified.

3.3 DOPDG Construction on Condition-Action Rules

Although Willmor et al. [22] defined PD-dependency (Definition 2) in terms of syntax, however interestingly they...
defined DD-dependency in terms of defined and used values (see Definition 4). This leads to an improvement in the precision of DD-dependency computation. However, the preciseness depends on how precisely one can identify the overlapping of database-parts by various database operations.

**Definition 4 (Database-Database (DD) dependency [22]).**

Let \( Q.SEL, Q.INS, Q.UPD \) and \( Q.DEL \) denote the parts of database state which are selected, inserted, updated, and deleted respectively by \( Q \). A database statement \( Q_1 \) is DD-dependent on another database statement \( Q_2 \) if (i) the database-part defined by \( Q_2 \) overlaps the database-part used by \( Q_1 \), i.e., \( Q_1.SEL \cap (Q_2.INS \cup Q_2.UPD \cup Q_2.DEL) \neq \emptyset \), and (ii) there is no roll-back operation in the execution path \( p \) between \( Q_2 \) and \( Q_1 \) (exclusive) which reverses back the effect of \( Q_2 \).

As a solution to compute this overlapping, Willmor et al. refer to the propagation algorithm in [35] designed for the static analysis of Condition-Action rules in expert database systems. The Condition-Action rules defined in an expert database system enable it to react automatically in some situations without the need of user access. These rules are, in general, expressed in the form \( E_{\text{cond}} \rightarrow E_{\text{act}} \), where \( E_{\text{act}} \) represents an action as data modification operation (e.g., INSERT, UPDATE and DELETE) and \( E_{\text{cond}} \) represents a condition. Formally, [35] considers an extended version of the relational algebra by introducing an additional operator \( \epsilon \), known as attribute extension operator, in case of database update. This operator is defined as \( \epsilon[x = expr]_\epsilon \), where the expression \( expr \) is evaluated over each tuple \( t \) of \( \epsilon \) and the resulting value is entered into the new attribute \( x \) for \( t \) under the new schema \( \text{schema}(\epsilon) \cup \{x\} \). Let us illustrate this with running example.

**Example 3.** Consider our running example in Section 2. Following the extended relational algebra, we get the following Condition-Action rules at program points 5 and 6:

- \( E_{\text{cond}}^5 \rightarrow \pi_{\text{purchase}_amt}(\sigma_{\text{purchase}_amt} > 1000 \land \text{purchase}_amt < 3000) \text{ Sales} \)
- \( E_{\text{act}}^5 \rightarrow \epsilon[\text{purchase}_amt' = \text{purchase}_amt - 0.05 \times \text{purchase}_amt](\sigma_{\text{purchase}_amt} > 1000 \lor \text{purchase}_amt < 3000) \text{ Sales} \)
- \( E_{\text{cond}}^6 \rightarrow \pi_{\text{purchase}_amt}(\sigma_{\text{purchase}_amt} > 3000) \text{ Sales} \)
- \( E_{\text{act}}^6 \rightarrow \epsilon[\text{purchase}_amt' = \text{purchase}_amt - 0.1 \times \text{purchase}_amt](\sigma_{\text{purchase}_amt} > 3000) \text{ Sales} \)

where \( \pi \) and \( \sigma \) are basic relational algebra operators for attribute projection and attribute selection respectively.

The propagation algorithm predicts how the action of one rule can affect the condition of another. In other words, the analysis checks whether a condition in one rule sees any data inserted or deleted or modified due to an action in another. This considers following three possibilities: (i) both the pre-defined part (i.e., database-part before performing the action \( E_{\text{act}} \)) and the post-defined part (i.e., database-part obtained after performing the action \( E_{\text{act}} \)) are in use by the condition \( E_{\text{cond}} \); (ii) the pre-defined part is not in use by \( E_{\text{cond}} \) whereas the post-defined part is in use by \( E_{\text{cond}} \); (iii) the pre-defined part is in use by \( E_{\text{cond}} \) whereas the post-defined part is not in use by \( E_{\text{cond}} \). Let us illustrate this by recalling the rules already defined in Example 3. This is worthwhile to note here that this kind of conditions verification makes the computational complexity exponential w.r.t. the number of defining statements.

**Example 4.** Consider the Condition-Action rules at program points 5 and 6 of our running example expressed in Example 3. Observe that the predicates \( (1000 \leq \text{purchase}_amt \leq 3000) \) in \( E_{\text{act}}^5 \) and \( (\text{purchase}_amt > 3000) \) in \( E_{\text{cond}}^6 \) are contradictory – meaning that \( E_{\text{act}}^5 \) operates on a part of data which is not accessed by \( E_{\text{cond}}^6 \). In other words, the action \( E_{\text{act}}^5 \) does not affect the condition \( E_{\text{cond}}^6 \). Therefore, DD-dependency \( 5 \rightarrow 6 \) is false. Similarly, we can also identify another false DD-dependency \( 15 \rightarrow 16 \). The refined set of data dependencies are:

- DD-dependencies for \( \text{purchase}_amt \): \( 4 \rightarrow 5, \ 4 \rightarrow 6, \ 4 \rightarrow 7, \ 4 \rightarrow 11, \ 4 \rightarrow 15, \ 4 \rightarrow 16, \ 5 \rightarrow 7, \ 5 \rightarrow 11, \ 5 \rightarrow 15, \ 5 \rightarrow 16, \ 6 \rightarrow 11, \ 6 \rightarrow 15, \ 6 \rightarrow 16, \ 7 \rightarrow 11, \ 7 \rightarrow 15, \ 7 \rightarrow 16 \)
- DD-dependencies for other attributes: \( 4 \rightarrow 7, \ 4 \rightarrow 11, \ 4 \rightarrow 15, \ 4 \rightarrow 16 \)
- PD-dependencies for \( x \) and \( y \): \( 2 \rightarrow 6, \ 3 \rightarrow 5 \)

Figure 5 depicts the refined DOPDG based on the above result.

**Limitations.** The Condition-Action rules can be applied only on a single def-use pair at a time. This fails to capture semantic independencies when a code contains more than one defining database statements (in sequence) for an attribute which is subsequently used by another database statement. The main reason behind this is the flow-insensitivity of this approach. For instance, the approach fails to identify false DD-dependencies \( 4 \rightarrow 11, \ 4 \rightarrow 15, \ 4 \rightarrow 16, \)
It is worthwhile to mention that our defined abstract syntax, which we recall from our previous work [39], has limitation in the sense that it considers only numerical attributes. However, the syntax is consistent with the SQL definition given by ANSI [40]. In fact, we have shown its equivalence with relational algebra and its extension to support nested queries in sections 8 and 10 of [39] respectively. Therefore, our formalism supports different RDBMS implementations, like Oracle, MySQL or IBM DB2.

Application Environment. Given a set of application variables \( V_a \) and the domain of values \( \text{Val} \), let \( \mathcal{E}_a : V_a \mapsto \text{Val} \) be the set of all functions with domain \( V_a \) and range included in \( \text{Val} \). An application environment \( \rho_a \in \mathcal{E}_a \) maps application variables to their values in \( \text{Val} \).

Database Environment. A database \( d \) is a set of tables \( \{ t_i \mid i \in I_d \} \) for a given set of indexes \( I_d \). A database environment is defined as a function \( \rho_d \) whose domain is \( I_d \), such that for \( i \in I_d \), \( \rho_d(i) = t_i \).

Table Environment. Given a database table \( t \) with attributes \( \text{attr}(t)=\{a_1, a_2, \ldots, a_k\} \). So, \( t \subseteq D_1 \times D_2 \times \ldots \times D_k \) where \( a_i \) is the attribute corresponding to the typed domain \( D_i \). A table environment \( \rho_t \) for a table \( t \) is defined as a function such that for any attribute \( a_i \in \text{attr}(t) \), \( \rho_t(a_i) = (\pi_i(l_i)) \mid l_i \in t \) where \( \pi_i \) is the projection operator and \( \pi_i(l_i) \) represents the \( i \)-th element of the \( l_i \)-th row. In other words, \( \rho_t \) maps \( a_i \) to the ordered set of values over the rows of the table \( t \).

Concrete Semantics. Let \( \Sigma_{\text{dbs}} \) be the set of states for the database language under consideration, defined by \( \Sigma_{\text{dbs}} = \mathcal{E}_{\text{dbs}} \times \mathcal{E}_a \) where \( \mathcal{E}_{\text{dbs}} \) and \( \mathcal{E}_a \) denote the set of all database environments and the set of all application environments respectively. Therefore, a state \( \rho \in \Sigma_{\text{dbs}} \) is denoted by a tuple \( (\rho_d, \rho_a) \) where \( \rho_d \in \mathcal{E}_{\text{dbs}} \) and \( \rho_a \in \mathcal{E}_a \). The transition relation

\[
\mathcal{T}_{\text{dbs}} : (C \times \Sigma_{\text{dbs}}) \mapsto (\Sigma_{\text{dbs}})
\]

specifies which successor states \( (\rho_d', \rho_a') \in \Sigma_{\text{dbs}} \) can follow when a statement \( c \in C \) executes on state \( (\rho_d, \rho_a) \in \Sigma_{\text{dbs}} \). Let us illustrate the concrete semantics of an update statement.

Example 5.
Consider the database table \( t \) in Table 2(a) and the following update statement:

\[
\text{Q}_\text{upd} : \text{UPDATE } t \text{ SET } \text{sal} = \text{sal} + 100 \text{ WHERE } \text{age} \geq 35
\]

The abstract syntax is denoted by \( \langle \text{UPDATE}(\bar{v}_a, \bar{v}), \phi \rangle \) where \( \phi = (\text{age} \geq 35) \) and \( \bar{v}_a = \langle \text{sal} \rangle \) and \( \bar{v} = \langle \text{sal} + 100 \rangle \).

The table targeted by \( \text{Q}_\text{upd} \) is \( \text{target}(\text{Q}_\text{upd}) = \{t\} \). The semantics of \( \text{Q}_\text{upd} \) is:

\[
\mathcal{T}_{\text{dbs}}\llbracket \text{Q}_\text{upd} \rrbracket (\rho_d, \rho_a) = \mathcal{T}_{\text{dbs}}\llbracket \langle \text{UPDATE}(\text{sal}, (\text{sal} + 100)), (\text{age} \geq 35) \rangle \rrbracket (\rho_d, \rho_a)
\]

Since, \( \text{target}(\text{Q}_\text{upd})=\{t\} \)

\[
\mathcal{T}_{\text{dbs}}\llbracket \langle \text{UPDATE}(\text{sal}, (\text{sal} + 100)) \rangle (\rho_d, \rho_a) \cup (\rho_d, \rho_a) \mid (\rho_d, \rho_a) \models \phi = (\text{age} \geq 35)
\]

Absorbing
TABLE 2: Database before and after the update operation

<table>
<thead>
<tr>
<th></th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>table t</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>62</td>
</tr>
</tbody>
</table>

(a) table t

TABLE 2: Database before and after the update operation

\[(\rho_t', \rho_d) \cup (\rho_{1 \uparrow (age \geq 35)}, \rho_d)\]

\[(\rho_t' \cup \rho_{1 \uparrow (age \geq 35)}, \rho_d)\]

\[(\rho_t', \rho_d)\]

= (\rho_t', \rho_d) \cup (\rho_{1 \uparrow (age \geq 35)}, \rho_d)

= (\rho_t' \cup \rho_{1 \uparrow (age \geq 35)}, \rho_d)

= (\rho_t', \rho_d)

\[(\rho_t \mapsto E[\rho_t \mapsto sal + 100](\rho_{1 \uparrow (age \geq 35)} \bowtie \rho_d))\]

\[(\rho_{1 \uparrow (age \geq 35)} \mapsto [sal \mapsto (1600, 2600, 3100)])\]

The notation \((t \downarrow (age \geq 35))\) denotes the set of tuples in \(t\) for which \((age \geq 35)\) is true (denoted by red part in \(t\) of Table 2(a)). \(E[\] is the semantic function for arithmetic expression which maps “sal + 100” to a list of values \((1600, 2600, 3100)\) on the table environment \(\rho_{1 \uparrow (age \geq 35)}\). The notation \(\mapsto\) denotes a substitution by new values. Observe that the substitution of ‘sal’ by the list of values in \(\rho_{1 \uparrow (age \geq 35)}\) results in a new table environment \(\rho_{t'}\) (denoted...
by red part in Table 2(b)). Finally, the least upper bound (denoted \( \sqcup \)) which is defined over the lattice of table environments partially ordered by \( \leq \), results in a new state \((\rho_{t'}, \rho_z)\) where \( t' \) is depicted in Table 2(b).

### 5 Semantics-based Dependency: A Formalization in Concrete Domain

As witnessed in section 3, the DOPDG construction approaches based on the syntax often fail to compute optimal set of dependencies. This motivates researchers towards semantics-based dependency computation considering values rather than variables [34]. For instance, consider an arithmetic expression \( \phi = x^2 + 4w \mod 2 + z' \). Although in this expression \( x \) syntactically depends on \( w \), semantically there is no dependency as the evaluation of \( "4w \mod 2" \) is always zero.

We, in our previous work [36], formalized the notion of semantics-based dependencies of database programs. Let us first recall this and then in the subsequent sections we build a computational framework considering this as the basis.

Given a SQL statement \( Q = \langle A, \phi \rangle \) and its target table \( t \). Suppose \( \bar{x} = \text{USE}(A) \), \( \bar{y} = \text{USE}(\phi) \) and \( \bar{z} = \text{DEF}(Q) \). According to the concrete semantics, suppose \( \mathcal{T}_{\text{db}}[Q](\rho_t, \rho_z) = (\rho_{t'}, \rho_z) \).

The used and defined part of \( t \) by \( Q \) are computed according to the following equations:

\[
A_{\text{def}}(Q, t) = \Delta(\rho_{t'}(z), \rho_1(z)) \quad (6)
\]

\[
A_{\text{use}}(Q, t) = \rho_{1_{t'}}(\bar{x}) \cup \rho_{1_{t'}}(\bar{y}) \quad (7)
\]

where

- \( t \downarrow \phi \) : Set of tuples in table \( t \) which satisfies the condition-part \( \phi \).
- \( \rho_{1_{t'}}(\bar{x}) \) : Values of \( \bar{x} \) in \( t \downarrow \phi \).
- \( \rho_{1_{t'}}(\bar{y}) \) : Values of \( \bar{y} \) in \( t \downarrow \phi \).
- \( \Delta \) : Computes the difference between the original database state on which \( Q \) operates and the new database state obtained after performing the action-part \( A \).

In other words, the function \( A_{\text{use}} \) maps a query \( Q \) to the part of the database information used by it, whereas the function \( A_{\text{def}} \) defines the changes occurred in the database states when data is updated or deleted or inserted by \( Q \). The following example illustrates this.

#### Example 6

Let us consider the concrete database table \( t \) shown in Table 2(a) and the following update statement:

\[
Q_{\text{upd}} : \text{UPDATE} \; t \; \text{SET} \; \text{sal} = \text{sal} + 100 \; \text{WHERE} \; \text{age} \geq 35
\]

where \( A = \text{UPDATE}((\text{sal}), (\text{sal} + 100)) \) and \( \phi = \text{age} \geq 35 \). According to equations 6 and 7, the used-part and defined-part are as follows:

\[
A_{\text{use}}(Q_{\text{upd}}, t) = \rho_{t}(\text{age} \geq 35)(\text{sal}) \cup \rho_{t}(\text{age} \geq 35)(\text{age})
\]

\[
A_{\text{def}}(Q_{\text{upd}}, t) = \Delta(\rho_{t'}(\text{sal}), \rho_{t}(\text{sal}))
\]

These are depicted in Tables 3(a) and 3(b) respectively where we have denoted \( A_{\text{use}}(Q_{\text{upd}}, t) \) and \( A_{\text{def}}(Q_{\text{upd}}, t) \) by red color.

### 6 Semantics-based Abstract Dependency: A Sound Approximation

In this section, we first briefly introduce the Abstract Interpretation framework [9], [10]. Then we define abstract semantics of database statements in various non-relational and relational abstract domains. Finally, we present the computation of abstract dependencies among statements identifying their approximated used and defined database-parts based on the abstract semantics.

#### 6.1 The Abstract Interpretation Framework: Preliminaries

Abstract Interpretation is a method of sound approximation of the program’s concrete semantics which enables to provide sound answers to questions about the program’s run-time behaviour. The idea is to lift concrete semantics to an abstract setting by replacing concrete values by suitable properties of interest, and simulating the concrete operations by sound abstract operations. The concrete and the abstract domains are partially ordered sets (lattices, or complete lattices, possibly), where the ordering relations describe the relative precision of the denotations including the top elements representing no information. The mapping

<table>
<thead>
<tr>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1300</td>
<td>35</td>
<td>10</td>
</tr>
<tr>
<td>800</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>2000</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>3000</td>
<td>62</td>
<td>10</td>
</tr>
</tbody>
</table>

### Table 3: The used and defined part of \( t \) by \( Q_{\text{upd}} \) (marked with red color)
between concrete and abstract semantics domains is usually established by a Galois Connection:

**Definition 6 (Galois Connections [9])**. Consider two partial orders \((D, \leq)\) and \((\overline{D}, \subseteq)\) where the first one represents a concrete domain and the second one represents an abstract domain. The Galois Connection between \(D\) and \(\overline{D}\) is denoted by \(\big\langle (D, \leq), \alpha, \gamma, (\overline{D}, \subseteq) \big\rangle\) or \((D, \leq) \iff (\overline{D}, \subseteq)\)

where \(\alpha : D \rightarrow \overline{D}\) and \(\gamma : \overline{D} \rightarrow D\) holds iff:

- \(\forall v \in D. v \subseteq \gamma \circ \alpha(v)\).
- \(\forall v \in D. \alpha \circ \gamma(v) \subseteq \overline{v}\).
- \(\alpha\) and \(\gamma\) are monotonic.

In other words, if \(\forall v \in D, \overline{v} \subseteq \alpha(v) \iff v \subseteq \gamma(\overline{v})\).

A number of abstract domains, non-relational and relational, exist in the literature [9], [10], [41], [42], [43]. Let us briefly illustrate them below:

**Non-relational Abstract Domains.**

An abstract domain is said to be non-relational if it does not preserve any relation among program variables. Non-relational abstract domains care only about the actual variables being updated, rather than having potential to change multiple values at once [10]. Some widely used non-relational abstract domains for program analysis include sign domain for sign property analysis, parity domain for parity property analysis, interval domain for division-by-zero or overflows [9]. Analyses in these domains are, although efficient, but imprecise w.r.t. relational abstract domains. Figure 6 pictorially depicts a scenario where a set of points SP (indicated by \(\bullet\)) on the xy-plane are abstracted by sign and interval properties.

**Relational Abstract Domains.**

Unlike non-relational abstract domains, the relational abstract domains preserve relations among program variables [41]. Analyses in these domains are more precise as compared to the non-relational abstract domains, in particular, for large number of relations among variables in the code. Widely used relational abstract domains are the domains of Polyhedra, Octagons, Difference-Bound Matrices (DBM), etc [41], [42], [43]. Abstractions of the same set of points in the octagon and polyhedra domains are exemplified in Figure 7.

4. Notice that for some abstract domains only a concretization function exists, like in the case of Polyhedra.

6.2 Defining Abstract Semantics of Database Statements

We are now in a position to define the abstract semantics of database statements in various abstract domains. To this aim, let us first define abstract database states and the abstract semantic transition relation in an abstract domain of interest w.r.t. its concrete counterpart (section 4).

**Definition 7 (Abstract Table).** Given a concrete table \(t \in \mathcal{O}(D)\) where \(D = D_1 \times D_2 \times \ldots \times D_k\) such that \(\text{attr}(t) = \{a_1, a_2, \ldots, a_l\}\) and \(a_i\) is the attribute corresponding to the typed domain \(D_i\). Let \(\overline{D}\) be an abstract domain which represents properties of the attributes of \(t\) establishing the Galois Connection \(\big\langle \mathcal{O}(D), \subseteq, a, \gamma, (\overline{D}, \subseteq) \big\rangle\). An element \(\overline{t} \in \overline{D}\) is said to be a sound abstraction of the concrete table \(t\) if for all tuples \(l \in t \iff l \in \gamma(\overline{t})\).

**Definition 8 (Abstract Table Environment).** Given an abstract table \(\overline{t}\), an abstract table environment \(\rho_I\) is defined as \(\rho_I(a_i) = \pi_I(l)\) for any attribute \(a_i \in \text{attr}(l)\), where \(\pi\) is the projection operator in the abstract domain and \(\pi_I(l)\) represents the projected abstract values corresponding to the \(i^{th}\) attribute in \(l\).

**Definition 9 (Abstract Database States).** An abstract database \(\overline{d}\) is a set of abstract tables \(\{\overline{t}_i \mid i \in I_x\}\) for a given set of indexes \(I_x\). An abstract database environment is defined as a function \(\rho_I\) whose domain is \(I_x\), such that for \(i \in I_x\), \(\rho_I(l) = \overline{t}_i\).

**Definition 10 (Abstract States).** An abstract state \(\overline{p} \in \Sigma_{\text{dba}}\) for database applications is defined as a tuple \((\rho_I, \rho_C)\) where \(\rho_I \in \Sigma_{\text{dba}}\) and \(\rho_C \in \Sigma_a\) are an abstract database environment and an abstract application environment.

Observe that, as any constraint defined at database level has no role in the dependency computation at application-code level, we consider database abstraction without taking these constraints into consideration.

In order to formalize the abstract semantics of database applications, we define the following sound abstract transition relation corresponding to its concrete counterpart \(\mathcal{T}_{\text{dba}}\) (defined in equation 5 of section 4):

\[
\mathcal{T}_{\text{dba}} : C \times \Sigma_{\text{dba}} \rightarrow \Sigma_{\text{dba}}
\]

which specifies the successor abstract state \((\rho_I, \rho_C) \in \Sigma_{\text{dba}}\) when a statement \(c \in C\) executes on an abstract state

5. Notice that for some abstract domain the abstraction function may not exist.
(ρ_rors, ρ_φs) ∈ Σ_dba. The soundness of the abstract semantics relies on the soundness of ℱ_dba w.r.t. 𝒪_dba (see the proof in section 8).

Since our objective is to compute semantics-based DD-independencies, it is important to identify database-parts (identified by the condition φ) before and after performing the action A. With this objective, unlike equation 8 which results in a single abstract state ρ, we define a variant of the abstract transition relation as follows:

$$\overline{ℱ}_{dep} : C × Σ_dba \mapsto (Σ_dba × Σ_dba × Σ_dba)$$

which results in a three-tuple (ρ_∆, ρ_φs, ρ_φf) of abstract database states, where ρ_∆ represents an abstract database state which does not satisfy φ, whereas the second component ρ_φs represents an abstract database state which satisfies φ. Observe that an abstract database-part which may or may not satisfy φ (due to abstraction) will be included in both ρ_∆ and ρ_φs. The third component ρ_φf is obtained after performing an action A on ρ_φs. These are depicted in Figure 8.

The abstract semantics of database statements in various abstract domains following equations 8 and 9 are defined in the subsequent sections.

### 6.2.1 Domain of Intervals

Let L = ⟨φ(R), ⊑, θ, R, ∩, ∪⟩ be a concrete lattice of the powerset of numerical values R. Let I = {|[l, h]| l ∈ R∪{−∞}, h ∈ R∪{+∞}, l ≤ h} ⊆ I be the abstract domain of intervals forming an abstract lattice L = (L, ⊑, θ, {[−∞, +∞], ∪, ⊆}), such that:

- [l₁, h₁] ⊆ [l₂, h₂] ⇔ l₂ ≤ l₁ ∧ h₁ ≥ h₂
- [l₁, h₁] ∩ [l₂, h₂] = [max(l₁, l₂), min(h₁, h₂)]
- [l₁, h₁] ∪ [l₂, h₂] = [min(l₁, l₂), max(h₁, h₂)]

The correspondence between L and L_φ is formalized as the Galois connection (L, α, γ, L_φ) where ∀S ∈ φ(R) and ∀p ∈ Σ.

The pictorial representation of the Galois connections (L_c, α_c, γ_c, L_φ) is shown in Figure 9.

### Abstract Semantics of Imperative Language in Interval: A Quick Tour [44].

Consider the set of concrete states Σ : V_a ↦ R representing the mapping of imperative program variables to their semantic domain values. Given the set of arithmetic expressions E, boolean expressions B and commands C. The concrete denotational semantics functions ⟨ℱ_c⟩ : (E ∪ B) ↦ (Σ ↦ R ∪ {true, false}) for expressions evaluation, ⟨ℱ_c⟩ : (E ∪ B) ↦ (φ(Σ) ↦ φ(Σ)) for state-filtering based on boolean satisfiability and ⟨ℱ_c⟩ : C ↦ φ(Σ) specifying effects of commands on states, are defined in Figure 10.

Given an abstract domain I of intervals, the set of abstract states is defined as Σ : V_a ↦ I which respects the Galois Connection, i.e. ∀p ∈ Σ, ∀p ∈ Σ : α(p) ⊆ p ⇔ p ∈ γ(p).

The corresponding sound abstract semantics function ⟨ℱ⟩ : (E ∪ B) ↦ (Σ ↦ I ∪ {true, false}, τ_B) for expression evaluation where τ_B denotes “may be true or may be false”, is defined as:

$$⟨ℱ⟩[k] = (p, [k]) | p ∈ Σ$$

$$⟨ℱ⟩[x] = (p, p(x)) | p ∈ Σ$$

Examples of sound abstract arithmetic and relational operations ∇ and ⊆ respectively in the domain of intervals are:

- [l₁, h₁] ∇ [l₂, h₂] = [l₁ × l₂, h₁ × h₂]
- [l₁, h₁] ⊆ [l₂, h₂] = [min(l₁ × l₂, l₁ × h₂, l₂ × h₁, h₃ × h₂), max(l₁ × l₁, l₂ × h₂, l₁ × h₂, h₁ × h₂)]

Abstract versions of other arithmetic and relational operations are also defined this way, ensuring the soundness in I.

Similarly, the abstract semantics functions ⟨ℱ⟩ : B ↦ (Σ ↦ Σ) for abstract state-filtering and ⟨ℱ⟩ : C ↦ (Σ ↦ Σ) for commands are:

$$⟨ℱ⟩[x < k] = (p, p(x) ≤ [l, min(h,k)]) | p ∈ Σ, p(x) = [l, h], l ≤ k$$

$$⟨ℱ⟩[x ≤ k] = (p, p(x) ≤ [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l ≤ k$$

$$⟨ℱ⟩[x ≥ k] = (p, p(x) ≥ [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l ≥ k$$

$$⟨ℱ⟩[x > k] = (p, p(x) > [l, min(h,k)]) | p ∈ Σ, p(x) = [l, h], l > k$$

$$⟨ℱ⟩[x > k] = (p, p(x) > [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l ≥ k$$

$$⟨ℱ⟩[x = k] = (p, p(x) = [l, min(h,k)]) | p ∈ Σ, p(x) = [l, h], l = k$$

$$⟨ℱ⟩[x = k] = (p, p(x) = [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l ≥ k$$

$$⟨ℱ⟩[x = k] = (p, p(x) = [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l > k$$

$$⟨ℱ⟩[x ≥ k] = (p, p(x) ≥ [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l ≥ k$$

$$⟨ℱ⟩[x > k] = (p, p(x) > [l, max(h,k)]) | p ∈ Σ, p(x) = [l, h], l > k$$
Given a database statement $\text{dba}$ is defined below:

\begin{equation}
\text{dba} = \langle a, \phi \rangle
\end{equation}

where $a$ and $\phi$ are the attributes and conditions, respectively.

### Defining Abstract Semantics of Database Language in Interval Domain

Let us recall the semantic function $\mathcal{F}_{\text{dbs}}$ defined in equation 8 which specifies the successor abstract state $(\rho_T, \rho_S) \in \Sigma_{\text{dbs}}$ when a statement $c \in C$ executes on an abstract state $(\rho_T, \rho_S) \in \Sigma_{\text{dbs}}$.

Given a database statement $Q = \langle A, \phi \rangle$ and an abstract database state $\overline{\rho} = (\rho_T, \rho_S)$, the abstract semantics of $Q$ w.r.t. $\overline{\rho}$ is defined below:

\begin{equation}
\mathcal{F}_{\text{dbs}}(\langle A, \phi \rangle, \overline{\rho})
\end{equation}

where $t = \text{target}(\langle A, \phi \rangle)$ and $\exists \overline{a} : t \in \gamma(\overline{a})$.

### Example 7

Consider the statement $c := \text{if } x > 5 \text{ then } x = x + y \text{ else } x = x - y$. Consider an abstract state in the domain of intervals $\overline{\rho} = \langle x \mapsto [2,10], y \mapsto [1,1]\rangle$. The abstract semantics of $c$ w.r.t. $\overline{\rho}$ is illustrated below:

\begin{align*}
\mathcal{F}_{\text{dbs}}(\text{if } x > 5 \text{ then } x = x + y \text{ else } x = x - y, \overline{\rho}) &= \overline{\rho}_3 = (\overline{\rho}_1 \cup \overline{\rho}_4) \cap [100,100]\rangle \\
\mathcal{F}_{\text{dbs}}(\text{if } x > 5 \text{ then } x = x + y \text{ else } x = x - y, \overline{\rho}) &= \overline{\rho}_4 = (\overline{\rho}_1 \cap \overline{\rho}_3) \cap [100,100]\rangle \\
\mathcal{F}_{\text{dbs}}(\text{if } x > 5 \text{ then } x = x + y \text{ else } x = x - y, \overline{\rho}) &= \overline{\rho}_5 = (\overline{\rho}_1 \cap \overline{\rho}_2) \cap [100,100]\rangle
\end{align*}

### Example 8

Consider the simple code fragment $x = 1$; while($x < 100$)($x = x + 1$). Figure 11 illustrates a data flow-based analysis of the code in $\mathcal{L}$ for the absence of runtime errors. The data-flow equation for each node is mentioned on the controlling edge of the corresponding node. The fix-point solution of these equations represent the abstract collecting semantics (denoted by red color).
TABLE 4: Concrete and its corresponding Abstract Database

<table>
<thead>
<tr>
<th>cm</th>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1500</td>
<td>55</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>800</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>2500</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3000</td>
<td>62</td>
<td>10</td>
</tr>
</tbody>
</table>

(a) Concrete table t

<table>
<thead>
<tr>
<th>cm</th>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1500,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
<tr>
<td>2</td>
<td>[800,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
<tr>
<td>3</td>
<td>[1600,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
<tr>
<td>4</td>
<td>[800,1499]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
</tbody>
</table>

(b) Abstract table i in interval domain

### TABLE 5: Execution result t' by Update on t

<table>
<thead>
<tr>
<th>cm</th>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1600</td>
<td>35</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>800</td>
<td>28</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>2600</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3100</td>
<td>62</td>
<td>10</td>
</tr>
</tbody>
</table>

### TABLE 6: Execution result of Q_{upd} on i

<table>
<thead>
<tr>
<th>cm</th>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1500,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
<tr>
<td>2</td>
<td>[800,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
</tbody>
</table>

(a) Abstract table TM

<table>
<thead>
<tr>
<th>cm</th>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1600,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
<tr>
<td>2</td>
<td>[800,3000]</td>
<td>[28,62]</td>
<td>[10,20]</td>
</tr>
</tbody>
</table>

(b) Abstract table FM

Limitations. Let us illustrate the limitation of the analysis in I by the following example.

### Example 10. Consider the abstract state \( \vec{p} = (p_t, p_\varphi) \) defined in Example 9. Consider the following statement:

\[ Q_{upd} : UPDATE \ i \ SET \ sal = sal + x \ WHERE \ sal \geq 1550 \]

Here \( A = UPDATE((sal), (sal + x)) \) and \( \varphi = sal \geq 1500 \). The concrete semantics yields the resultant table t' shown in Table 5.

The abstract semantics of \( Q_{upd} \) w.r.t. \( \vec{p} = (p_t, p_\varphi) \) is

\[
\begin{align*}
\mathcal{T}_d \varphi(A, \phi) \parallel (p_t, p_\varphi) & = \mathcal{T}_d \varphi(UPDATE((sal), (sal + x)), sal \geq 1500) \parallel (p_t, p_\varphi) \\
& = \mathcal{T}_d \varphi(UPDATE((sal), (sal + x))) \parallel (p_{TM}, p_{\varphi}) \\
& = (p_{TM}, p_\varphi) \cup (p_{TM}, p_\varphi) \\
& = (p_{TM}, p_\varphi) \cup (p_{TM}, p_\varphi) \\
& = p_{TM}, p_\varphi
\end{align*}
\]

where

\[
\begin{align*}
\rho_{TM} & = \mathcal{T}_d \varphi(sal \geq 1500) \parallel (p_t) = (p_t)[sal \leftarrow [1500, 3000]] \\
\rho_{TM} & = \mathcal{T}_d \varphi(UPDATE((sal), (sal + x))) \parallel (p_{TM}, p_\varphi) \\
& = \mathcal{T}_d \varphi(sal = sal + x) \parallel (p_{TM}, p_\varphi) \\
& = \mathcal{T}_d \varphi(sal + x = [100, 100]) \parallel (p_{TM}, p_\varphi) \\
& = p_{TM} \parallel (sal \leftarrow [1600, 3001])
\end{align*}
\]

### Abstract Semantics towards Independence Computation.

In order to compute semantics-based DD-independancies, we define \( \mathcal{T}_d \varphi \) according to equations 9, in I for database statements as follows:

\[
\begin{align*}
\mathcal{T}_d \varphi(A, \phi) \parallel \vec{p} & = \mathcal{T}_d \varphi(A, \phi) \parallel (p_t, p_\varphi) \\
& = \mathcal{T}_d \varphi(A, \phi) \parallel (p_t, p_\varphi) \\
& = \mathcal{T}_d \varphi(A, \phi) \parallel (p_t, p_\varphi) \\
& \text{where } t = target((A, \phi)) \text{ and } \exists \vec{a} : t \in \gamma(\vec{a})
\end{align*}
\]

where

- \( \mathcal{T}_d \varphi(A, \phi) \parallel (p_t, p_\varphi) = (p_{TM}, p_\varphi) \)
• $\mathcal{T}_d \models \phi((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_\perp)$
• $\mathcal{T}_d \models \forall((\rho_{TM}, \rho_\perp)) = (\rho_{TM}, \rho_\perp)$

Let us now define $\mathcal{T}_{dep}$ for UPDATE, DELETE, INSERT and SELECT.

**UPDATE statement:**

$\mathcal{T}_{dep} \models \langle UPDATE(\vec{v}_d, \phi) \rangle((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_{TM}, \rho_{TM})$

where $\mathcal{T}_d \models \phi((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_\perp)$

$\mathcal{T}_d \models \forall((\rho_{TM}, \rho_\perp)) = (\rho_{TM}, \rho_\perp)$

$\mathcal{T}_{dep} \models \langle UPDATE(\vec{v}_d, \phi) \rangle((\rho_{TM}, \rho_\perp)) = (\rho_{TM}, \rho_\perp)$

**INSERT statement:**

$\mathcal{T}_{dep} \models \langle INSERT(\vec{v}_d, \phi, false) \rangle((\rho_T, \rho_\perp)) = (\rho_T, \rho_\perp, \rho_{\perp\perp})$

where $\mathcal{T}_d \models \phi((\rho_T, \rho_\perp)) = (\rho_T, \rho_\perp)$

$\mathcal{T}_d \models \forall((\rho_T, \rho_\perp)) = (\rho_T, \rho_\perp)$

$\mathcal{T}_{dep} \models \langle INSERT(\vec{v}_d, \phi, false) \rangle((\rho_T, \rho_\perp)) = (\rho_T, \rho_\perp)$

**DELETE statement:**

$\mathcal{T}_{dep} \models \langle DELETE(\vec{v}_d) \rangle((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_{TM}, \rho_{\perp})$

**SELECT statement:**

$\mathcal{T}_{dep} \models \langle SELECT(f(\vec{e}), \phi_2, \phi_3) \rangle((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_{TM}, \rho_{TM})$

Observe that the select operation does not change any information.

**Example 11.** Consider the abstract state $\tau = (\rho_T, \rho_\perp)$ and $\rho_{\perp} = \{x \mapsto [100, 100]\}$ where $\tau$ is depicted in Table 4(b), as defined in Example 9. Consider the following statements:

$Q_{upd} =$ UPDATE $t$ SET sal = sal + $x$ WHERE sal $\geq$ 1500

$Q_{ins} =$ INSERT INTO $t$ (eid, sal, age, dno)VALUES(5, 2700, 52, 20)

$Q_{del} =$ DELETE FROM $t$ WHERE age $\geq$ 61

$Q_{sel} =$ SELECT age FROM $t$ WHERE age $\leq$ 50

The abstract syntax of the statements are:

$Q_{upd} =$ (UPDATE((sal), (sal + $x$)), sal $\geq$ 1500)

$Q_{ins} =$ (INSERT((eid, sal, age, dno), (5, 2700, 52, 20)), false)

$Q_{del} =$ (DELETE((eid, sal, age, dno), age $\geq$ 61)

$Q_{sel} =$ (SELECT(age), age $\leq$ 50)

Abstract semantics of $Q_{upd}$ w.r.t. $\tau$ is

$\mathcal{T}_{dep} \models \langle UPDATE((sal), (sal + x)), sal \geq 1500 \rangle((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_{TM}, \rho_{TM})$

where $\rho_{TM}, \rho_{TM}$ and $\rho_{TM}$ are shown in Table 6 of Example 9.

The abstract semantics of $Q_{ins}$ w.r.t. $\tau$ is

$\mathcal{T}_{dep} \models \langle INSERT((eid, sal, age, dno), (5, 2700, 52, 20)), false \rangle((\rho_T, \rho_\perp)) = (\rho_T, \rho_{\perp}, \rho_{\perp\perp})$

where

$\rho_{\perp\perp} = \rho_{\perp}[eid \leftarrow [5, 5], sal \leftarrow [2700, 2700], age \leftarrow [52, 52], dno \leftarrow [20, 20]]$

The abstract semantics of $Q_{del}$ w.r.t. $\tau$ is

$\mathcal{T}_{dep} \models \langle DELETE((eid, sal, age, dno)), age \geq 61 \rangle((\rho_T, \rho_\perp)) = (\rho_{TM}, \rho_{TM}, \rho_{\perp\perp})$

where

$\rho_{\perp\perp} = \mathcal{T}_d \models age \geq 61 \models (\rho_T) = \rho_T[age \leftarrow [61, 62]]$

$\rho_{\perp\perp} = \mathcal{T}_d \models (\rho_{TM}, \rho_{TM}, \rho_{\perp\perp})$ where

$\rho_{\perp\perp} = \mathcal{T}_d \models age \leq 50 \models (\rho_T) = \rho_T[age \leftarrow [28, 50]]$

$\rho_{\perp\perp} = \mathcal{T}_d \models (\rho_T) = \rho_T[age \leftarrow [51, 62]]$

### 6.2.2 Relational Abstract Domain of Octagons

To yield more precise analysis as compared to the interval abstract domain, Antoine Miné [43] proposed a weekly relational abstract domain – the domain of octagons – which allows an analyzer to discover automatically common errors, such as division by zero, out-of-bound array access or deadlock, and more generally to prove safety properties of programs.

The octagon abstract domain encodes binary constraints between program variables in the form of $kx_i + kx_j \leq k$ where $x_i, x_j$ are program variables, $k, k \in [-1, 0, 1]$ are coefficients and $k$ is a constant in the numerical domain $\mathbb{R}$. Since coefficients can be either -1, 0 or 1, the number of inequalities between any two variables is bounded. The set of points satisfying the conjunction of such constraints forms an octagon.

**Octagonal constraints representation in memory.** The encoding of conjunctions of octagonal constraints makes use of Difference Bound Matrix (DBM) representation. Let us describe DBM first and then an extension to encode the set of octagonal constraints.

**Difference Bound Matrices (DBM) [42].** Given a program $P$ with a finite set of variables $V_P = \{x_1, \ldots, x_n\}$. A Difference Bound Matrix (DBM) $M$ with size $n \times n$ represents a set of invariants each of the form $x_j - x_i \leq k$, where $k \in \mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$ such that:

$M_{ij} = \begin{cases} k & \text{if } (x_j - x_i) \leq k \text{ where } x_i, x_j \in V_P \text{ and } k \in \mathbb{R}_\infty, \\ \infty & \text{otherwise.} \end{cases}$
Example 12. Consider the constraints \( |x_1 - x_2| \leq 3, x_2 - x_3 \leq 4, x_3 - x_1 \leq 5, x_2 - x_4 \leq 4 \). These constraints are represented by the DBM shown below:

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>( x_3 )</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>( x_2 )</td>
<td>( x_1 )</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( x_4 )</td>
<td>( x_3 )</td>
<td>( x_2 )</td>
<td>( x_1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Observe that any constraints of the form \( (x_i < k) \) and \( (x_i > k) \) can be represented as \( (x_i^+ - x_i^- < 2k) \) and \( (x_i^- - x_i^+ < -2k) \) respectively.

Closure. An octagon can be represented by more than one set of inequalities. For instance, the octagonal constraints \( \{(x < 4) \land (y < 6)\} \) and \( \{(x < 4) \land (y < 6) \land (x + y < 10)\} \) represent the same concrete values. Therefore, the use of closure operation ensures a unique representation of any octagonal constraints. The closure operation on DBM follows Floyd–Warshall algorithm [45].

In the rest of the paper, we use the notation \( m \) to represent closed DBM when the context is clear.

Galois Connections. Let \( L_{\epsilon} = \langle \phi(\mathbb{R}^n), \subseteq, \emptyset, \cap, \cup \rangle \) be the concrete lattice. Let \( M \) be the set of all closed DBMs representing the domain of octagons. Let \( M_L = \mathbb{M} \cup \{m_L\} \) where \( m_L \) represents the bottom element that contains an unsatisfiable set of constraints. We define the abstract lattice \( L_{\epsilon} = \langle M_L, \subseteq, m_L, m_T, \cap, \cup \rangle \) where \( m_T \) represents the top element for which the bound for all constraints is \( \infty \). The partial order, meet and join operations in \( L_{\epsilon} \) are defined as follows:

\[
\forall m, n \in M_L: m \sqsubseteq n \iff \forall i, j: m_{ij} \leq n_{ij}.
\]

\[
\forall m, n \in M_L: (m \cap n) = m' \cap n' \ \text{where} \ \forall i, j: m_{ij} \leq \min(m_{ij}, n_{ij}).
\]

\[
\forall m, n \in M_L: (m \cup n) = m' \cup n' \ \text{where} \ \forall i, j: m_{ij} \leq \max(m_{ij}, n_{ij}).
\]

Observe that since the union of two octagons is not always an octagon the result is approximated.

Let \( \Sigma \) be the set of all environments defined as \( \Sigma = \forall x : \mathbb{R} \rightarrow \mathbb{R} \). An environment \( \rho \in \Sigma \) maps each variable to its value. An environment will be understood as a point in \( \mathbb{R}^n \) where \( |V| = n \). The Galois connection between \( L_{\epsilon} \) and \( M \) is formalized as \( \langle L_{\epsilon}, \alpha_M, \gamma_M, L_{\epsilon} \rangle \) where \( \alpha_M \) and \( \gamma_M \) on \( S \in \phi(\mathbb{R}^n) \) and \( m \in M_L \) are defined below:

- \( \alpha_M(S) = m \) if \( S \neq \emptyset \) and \( \alpha_M(S) = m \) where \( m_{ij} \neq \infty \)
- \( \max(\rho(x_i) - \rho(x_0), m_{ij}) \) if \( i = 2k - 1, j = 2l - 1 \)
- \( \max(\rho(x_i) + \rho(x_0), m_{ij}) \) if \( i = 2k - 1, j = 2l \)
- \( \max(-\rho(x_i) + \rho(x_0), m_{ij}) \) if \( i = 2k - 1, j = 2l \)
- \( \emptyset \) if \( m = m_L \)
- \( \mathbb{R}^n \) if \( m = m_T \)
- \( \{k_1, \ldots, k_n \in \mathbb{R}^n \mid (k_1, -k_1, \ldots, -k_n, -k_n) \subseteq \text{dom}(m) \land \forall i: x_i - x_i \leq m_{ij} \} \)

\[
\gamma_M(m) = \{ v \mid e \subseteq \gamma_M(m) \} \}
\]

\[
\gamma_M(m) = \{ v \mid e \subseteq \gamma_M(m) \} \}
\]

Sound operations in octagon domain. Let us recall from [43] some useful sound operations in octagon abstract domain defined in terms of CDBM:

- **Emptiness test:** Let \( m \) be a CDBM and \( S \) be a directed weighted graph of \( m \). We say that the octagon is empty, i.e. \( \gamma(m) = \emptyset \), if and only if \( S \) has a simple cycle with a strictly negative total weight. The well-known Bellman-Ford [46] algorithm is used for such cycle detection.

- **Projection:** Let \( m \) be a CDBM representing a non-empty octagon. We extract the values of the variable \( x_i \) from \( m \) in the form of interval as:

\[
\{ v \mid e \subseteq \gamma_M(m) \} \}
\]

\[
\{ v \mid e \subseteq \gamma_M(m) \} \}
\]

Interested reader may refer to [43] [47] for more abstract operations (closure, widening, etc.) in octagon domain.

Abstract Semantics of Imperative Language in Octagon: A Quick Tour [43].

Given the set of boolean expressions \( B \) and commands \( C \). The concrete denotational semantics functions for state-filtering based on the boolean satisfiability is defined as \( \mathcal{T}_f : (B \rightarrow \phi(\Sigma)) \rightarrow \phi(\Sigma) \). The corresponding sound abstract function \( \mathcal{T}_f \) in the domain of octagons is defined as \( \mathcal{T}_f : (B \rightarrow M_L) \rightarrow M_L \). Similarly, the concrete denotational semantic function for the effects of commands on states is defined as \( \mathcal{T}_e : (C \rightarrow \phi(\Sigma)) \rightarrow \phi(\Sigma) \) and its corresponding sound abstract function \( \mathcal{T}_e \) in octagon domain is defined as \( \mathcal{T}_e : (C \rightarrow M_L) \rightarrow M_L \).

Test: Given a CDBM \( m \) representing abstract state at a program point and a boolean expression \( b \). The state-filtering function \( \mathcal{T}_f \) finds \( m' \) applying \( b \) on \( m \) where \( \gamma(m') = \{ p \in \gamma(m) \mid p \text{ satisfies } b \} \). However, as it is in general impossible to implement such a transition function, an upper approximation result is computed such that

\[
\gamma(m') \supseteq \{ p \in \gamma(m) \mid p \text{ satisfies } b \}
\]

The tests that can be modeled in the octagon domain are:

\[
\begin{align*}
x_0 + x_1 & \leq k, x_0 - x_1 & \leq k, x_0 - x_1 & \leq k, x_0 + x_1 = k, & x_0 \leq k
\end{align*}
\]
and $x_h \geq k$. The state-filtering function $\mathcal{F}_f$ for $x_h + x_l \leq k$ is defined as below:

$$\mathcal{F}_f[x_h + x_l \leq k] = m'$$

where

$$m'_{ij} = \begin{cases} 
\min(m_{ij}, k) & \text{if } (i, j) \in \{(2h, 2l - 1), (2l, 2h - 1)\}, \\
0 & \text{otherwise}
\end{cases}$$

Observe that the entries in the CDBM $m$ corresponding to the cells $(x_h, x_l^r)$ and $(x_l^r, x_h)$ are updated based on the value $k$, resulting into $m'$ which satisfies $x_h + x_l \leq k$. Similarly $\mathcal{F}_f$ for all other tests can also be defined.

Assignment: An assignment is to replace the value of a program variable $x_i$ with the value of an expression $e$. Formally $x_i = e$. Given an abstract state $m$ representing octagonal constraints at a program point and an assignment $x_i = e$, the abstract semantics of the assignment on $m$ results as an upper approximation such that

$$\gamma(m') \supseteq \{p[x_i \leftarrow k] \mid \rho \in \gamma(m) \land k \in \mathcal{F}_f[e \rho] \}$$

where $\mathcal{F}_f$ is the semantic function of arithmetic expression and $\rho[x_i \leftarrow k]$ denote $\rho$ with its $i$th component changed into $k$.

The assignments that can be modeled in octagon domain are: $x_h = x_h + k$ and $x_h = x_h + k$ with $h \neq l$. In the first case $x_h = x_h + k$, we subtract $k$ from inequalities having negative coefficient for $x_h$ and add $k$ to inequalities having positive coefficient for $x_h$. On the other hand, for the second case $x_h = x_h + k$, the inequalities $x_h - x_l \leq k$ and $x_l - x_h \leq -k$ are added into the octagon. Let us illustrate them below:

1) If $x_h = x_h + k$:

$$\mathcal{F}_f[x_h = x_h + k] = m'$$

where

$$m'_{ij} = \begin{cases} 
+1 & \text{if } j = 2h, \\
-1 & \text{if } j = 2h - 1, \\
0 & \text{otherwise}
\end{cases}$$

and

$$\begin{cases} 
\alpha_{ij} = +1 & \text{if } i = 2h, \\
\beta_{ij} = \begin{cases} 
-1 & \text{if } i = 2h - 1, \\
0 & \text{otherwise}
\end{cases}
\end{cases}$$

2) If $x_h = x_h + k$ with $h \neq l$:

$$\mathcal{F}_f[x_h = x_h + k] = m'$$

where

$$m'_{ij} = \begin{cases} 
k & \text{if } (i, j) \in \{(2h, 2l - 1), (2l, 2h - 1)\}, \\
-k & \text{if } (i, j) \in \{(2h, 2l), (2l, 2h - 1)\}, \\
m_{ij} & \text{if } i, j \notin \{(2h, 2l - 1), (2l, 2h - 1)\}, \\
+\infty & \text{otherwise}
\end{cases}$$

Example 14. Consider the statement $c := \text{if } x \geq 5 \text{ then } x = y + 1 \text{ else } x = x - 1$. Let the initial abstract state be $m_T$ (which is the top element in the lattice of octagon domain). The abstract semantics of $c$ w.r.t. $m$ is illustrated below, where $\mathcal{F}_f$ denotes an alternative representation in memory.

$$\mathcal{F}_f[x \geq 5] = \{x \leq 5\} \supseteq m_1$$

$$\mathcal{F}_f[\neg(x \geq 5)] = \{x \leq 4\} \supseteq m_2$$

After recalling the abstract semantics of imperative languages on octagon domains designed by [43], let us move to database languages.

Defining Abstract Semantics of Database Language in Octagon Domain.

In case of database applications, we consider two different environments: database environment $\rho_d \in E_{db}$ and application environment $\rho_a \in E_a$. To determine abstract semantics of database statements in the domain of octagons, we define the abstract state $\overline{\rho} \in E_{db}$ as

$$\overline{\rho} = (m_d, m_a)$$

where $m_d$ and $m_a$ are CDBMs of octagonal constraints as abstraction of database values and application variables values respectively. Therefore, as defined in equation 8, the abstract semantic function for database statements $Q = \langle A, \phi \rangle$ is defined as:

$$\mathcal{F}_d(A, \phi)(m_d, m_a) = \mathcal{F}_d(A, \phi)(m_d, m_a) = (m_d, m_a) \cup (m_T, m_a)$$

where $m_T$ is the octagonal representation of the concrete table $t$ which acts as the target of $Q$ and $m_a$ is the octagonal representation of the resultant table $t'$. Below is the abstract semantics for update statement.

$$\mathcal{F}_d(\langle \text{UPDATE}(\overline{\rho}_d), \phi \rangle)(m_d, m_a)$$

$$\mathcal{F}_d(\langle \text{UPDATE}(\overline{\rho}_d), \phi \rangle)(m_T, m_a) \cup (m_F, m_a)$$

We can define similarly the abstract semantics for other database statements as well.

Example 15. Consider the concrete table $t$ shown in Table 4(a). Consider the concrete application environment $\rho_a = (x \rightarrow 100)$ where $x$ is an application variable. The corresponding abstract representation of $\rho_a$ in octagon domain are represented by CDBM $m_d$ and $m_a$ respectively as

$$m_d \supseteq \{e \leftarrow \neg 1, \text{ eid } \leq 4, \text{ sal } \leq 800, \text{ sal } > 3000, \text{ age } \leq 28, \text{ age } > 62, \text{ dno } \leq -10, \text{ dno } > 20\}$$

$$m_a \supseteq \{x \leq 100, -x \leq -100\}$$

Consider the following UPDATE statement

$$Q_{upd} : \text{UPDATE} t \text{ set sal } = \text{sal } + x \text{ WHERE } \text{age} > 35$$

where $A = \text{UPDATE}(\text{sal}, (\text{sal } + x))$ and $\phi = \text{age} > 35$. The abstract semantics w.r.t. $\overline{\rho} = (m_d, m_a)$ is

$$\mathcal{F}_d(\langle A, \phi \rangle)(m_d, m_a)$$
\[ \text{where} \quad m_{TM} \equiv \{ -e\text{id} \leq -1, e\text{id} \leq 4, -s\text{al} \leq -800, s\text{al} \leq 3000, -a\text{ge} \leq -35, a\text{ge} \leq 62, -d\text{n}\text{o} \leq -10, d\text{n}\text{o} \leq 20 \} \]

\[ m_{FM} \equiv \{ -e\text{id} \leq -1, e\text{id} \leq 4, -s\text{al} \leq -800, s\text{al} \leq 3000, -a\text{ge} \leq -28, a\text{ge} \leq 34, -d\text{n}\text{o} \leq -10, d\text{n}\text{o} \leq 20 \} \]

\[ m_{TM} \equiv \{ -e\text{id} \leq -1, e\text{id} \leq 4, -s\text{al} \leq -900, s\text{al} \leq 3100, -a\text{ge} \leq -35, a\text{ge} \leq 62, -d\text{n}\text{o} \leq -10, d\text{n}\text{o} \leq 20 \} \]

**Observation.** We can follow an alternative equivalent way of abstract state representation by combining both CDBM of \( m_i \) and \( m_s \) for the sake of simplicity. Given \( m_i \) and \( m_s \), CDBM representations of database values and variables values, these can be combined into equivalent CDBM \( m \) defined in \( (p+q) \) – dimension space by merging \( m_i \) and \( m_s \). In the subsequent sections we define abstract semantics w.r.t. abstract state \( \overline{p} = m \).

**Abstract Semantics towards Independency Computation**

Like for the interval domain, the following transition relation is defined, according to equation 9, to compute semantics-based DD-independency in the domain of octagons:

\[ \overline{\text{dep}} : C \times M_\perp \mapsto (M_\perp \times M_\perp \times M_\perp) \]

Below is the definition of \( \overline{\text{dep}} \) for various database statements in octagon domain.

1. **UPDATE:**

\[ \overline{\text{dep}}\langle \text{UPDATE}(\overline{d_i}, \phi) \rangle m = \begin{cases} \langle m_r, m_r, m_r \rangle & \text{if } \phi \equiv \{ k_i x_i + k_j x_j \leq k \} \text{ where } x_i, x_j \in \mathcal{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\ \langle m, m, m' \rangle & \text{otherwise} \end{cases} \]

\[ \overline{\text{dep}}\langle \text{UPDATE}(\overline{d_i}, \phi) \rangle m = m_r \]

\[ \overline{\text{dep}}\langle \text{UPDATE}(\overline{d_i}, \phi) \rangle m = m_r \]

2. **INSERT:**

\[ \overline{\text{dep}}\langle \text{INSERT}(\overline{d_i}, \phi) \rangle m = \langle m, m_s, m_{\text{new}} \rangle \]

\[ \overline{\text{dep}}\langle \text{INSERT}(\overline{d_i}, \phi) \rangle m = m_s \]

\[ \overline{\text{dep}}\langle \text{INSERT}(\overline{d_i}, \phi) \rangle m = m_{\text{new}} \]

where \( m_{\text{new}} \equiv \{ -e\text{id} \leq -5, e\text{id} \leq 5, -s\text{al} \leq -2700, s\text{al} \leq 2700, -a\text{ge} \leq -52, a\text{ge} \leq 52, -d\text{n}\text{o} \leq -20, d\text{n}\text{o} \leq 20 \} \)

The abstract semantics of the \( Q_{\text{upd}} \) with respect to \( m_i \) is \( \overline{\text{dep}}\langle \text{UPDATE}(\overline{d_i}, \phi) \rangle m_i = \langle m_r, m_r, m_r \rangle \)

where \( m_r \) is depicted in Example 15.

The abstract semantics of the \( Q_{\text{ins}} \) w.r.t. \( m_i \) is \( \overline{\text{dep}}\langle \text{INSERT}(\overline{d_i}, \phi) \rangle m_i = \langle m_r, m_r, m_{\text{new}} \rangle \)

where \( m_{\text{new}} \) represents bottom element that contains an unsatisfiable set of constraints.

3. **DELETE:**

\[ \overline{\text{dep}}\langle \text{DELETE}(\overline{d_i}', \phi) \rangle m = \begin{cases} \langle m_r, m_r, m_r \rangle & \text{if } \phi \equiv \{ k_i x_i + k_j x_j \leq k \} \text{ where } x_i, x_j \in \mathcal{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\ \langle m, m, m' \rangle & \text{otherwise} \end{cases} \]

4. **SELECT:**

\[ \overline{\text{dep}}\langle \text{SELECT}(\overline{d_i}, \phi) \rangle m = \begin{cases} \langle m_r, m_r, m_r \rangle & \text{if } \phi \equiv \{ k_i x_i + k_j x_j \leq k \} \text{ where } x_i, x_j \in \mathcal{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\ \langle m, m, m \rangle & \text{otherwise} \end{cases} \]

Observe that \( \phi \equiv \{ k_i x_i + k_j x_j \leq k \} \) checks whether the condition in WHERE clause of database statement respects the form of octagonal constraints.

**Example 16.** Consider the concrete database table \( t \) shown in Table 4(a), and its corresponding abstract representation in the form of CDBM \( m \) in the domain of octagons as

\[ m \equiv \{ -e\text{id} \leq -1, e\text{id} \leq 4, -s\text{al} \leq -800, s\text{al} \leq 3000, -a\text{ge} \leq -28, a\text{ge} \leq 62, -d\text{n}\text{o} \leq -10, d\text{n}\text{o} \leq 20 \} \]

Consider the following statements:

\[ Q_{\text{upd}} = \text{UPDATE} t \text{ SET } s\text{al} = s\text{al} + x \text{ WHERE } a\text{ge} \geq 35 \]

\[ Q_{\text{ins}} = \text{INSERT INTO } t \text{ (e\text{id}, s\text{al}, a\text{ge}, d\text{n}\text{o})VALUES}(5, 2700, 52, 20) \]

\[ Q_{\text{del}} = \text{DELETE FROM } t \text{ WHERE } a\text{ge} \geq 61 \]

\[ Q_{\text{del}} = \text{SELECT s\text{al} FROM } t \text{ WHERE } a\text{ge} \leq 50 \]

The abstract syntax are

\[ Q_{\text{upd}} = \langle \text{UPDATE}(\overline{d_i}, (s\text{al} + 100)) \rangle, a\text{ge} \geq 35 \]

\[ Q_{\text{ins}} = \langle \text{INSERT}(\overline{d_i}, (s\text{al}, a\text{ge}, d\text{n}\text{o}), (5, 2700, 52, 20)) \rangle, \text{false} \]

\[ Q_{\text{del}} = \langle \text{DELETE}(\overline{d_i}, (s\text{al}, a\text{ge}, d\text{n}\text{o})), a\text{ge} \geq 61 \rangle \]

\[ Q_{\text{del}} = \langle \text{SELECT}(\overline{d_i}, a\text{ge} \leq 50) \rangle \]

The abstract semantics of the \( Q_{\text{upd}} \) with respect to \( m_i \) is \( \overline{\text{dep}}\langle \text{UPDATE}(\overline{d_i}, (s\text{al} + 100)), a\text{ge} \geq 35 \rangle m_i = \langle m_r, m_r, m_r \rangle \)

where \( m_r \) and \( m_r \) and \( m_r \) are depicted in Example 15.

The abstract semantics of the \( Q_{\text{ins}} \) w.r.t. \( m_i \) is \( \overline{\text{dep}}\langle \text{INSERT}(\overline{d_i}, (s\text{al}, a\text{ge}, d\text{n}\text{o}), (5, 2700, 52, 20)), \text{false} \rangle m_i = \langle m_r, m_r, m_{\text{new}} \rangle \)

where \( m_{\text{new}} \) represents bottom element that contains an unsatisfiable set of constraints.
The abstract semantics of the \( Q_{\text{ot}} \) w.r.t. \( m_r \) is
\[
\mathcal{F}_{\text{dep}}[\text{SELECT}(\text{age})], \text{ age } \leq 50 \] \( m_r = (m_r, m_r, m_r) \)
where
\[
m_r \equiv [ - \text{eid} \leq -1, \text{eid} \leq 4, -\text{sal} \leq -800, \text{sal} \leq 3000, \text{ age } \leq 60, \]
\[
- \text{age} \leq -28, -\text{dno} \leq -10, \text{dno} \leq 20 \]

The abstract semantics of the \( Q_{\text{ot}} \) w.r.t. \( m_r \) is
\[
\mathcal{F}_{\text{dep}}[\text{SELECT}(\text{age})], \text{ age } \leq 50 \] \( m_r = (m_r, m_r, m_r) \)
where
\[
m_r \equiv [ - \text{eid} \leq -1, \text{eid} \leq 4, -\text{sal} \leq -800, \text{sal} \leq 3000, \text{ age } \leq 50, \]
\[
- \text{age} \leq -28, -\text{dno} \leq -10, \text{dno} \leq 20 \]

The abstract semantics of the \( Q_{\text{ot}} \) w.r.t. \( m_r \) is
\[
\mathcal{F}_{\text{dep}}[\text{SELECT}(\text{age})], \text{ age } \leq 50 \] \( m_r = (m_r, m_r, m_r) \)
where
\[
m_r \equiv [ - \text{eid} \leq -1, \text{eid} \leq 4, -\text{sal} \leq -800, \text{sal} \leq 3000, \text{ age } \leq 62, \]
\[
- \text{age} \leq -51, -\text{dno} \leq -10, \text{dno} \leq 20 \]

**Limitations.** The octagon abstract domain is a weakly relational domain that allows a limited number of relations between program variables. Due to this bottleneck, analyses in this abstract domain may fail to produce precise results. For example, consider the following statement:
\[
\text{UPDATE} \ 1 \ \text{SET} \ a = a + 1 \ \text{WHERE} \ a + b + c > 35 \]

Let us briefly recall some basics. The regions in \( n \)-dimensional space \( \mathbb{R}^n \) bounded by a finite set of hyperplanes are called polyhedra. Let \( V_P = \{x_1, x_2, \ldots, x_n\} \) be the set of variables in program \( P \). We represent by \( \mathcal{S} = \{v_1, v_2, \ldots, v_n\} \subset \mathbb{R}^n \), an \( n \)-tuple (vector) of real numbers. By \( \beta = \bar{v} \bar{x} \geq k \) where \( \bar{v} \neq 0 \), \( \bar{x} = (x_1, x_2, \ldots, x_n) \), \( k \in \mathbb{R} \), we represent a linear inequality over \( \mathbb{R}^n \). A linear inequality defines an affine half-space of \( \mathbb{R}^n \). If \( P \) is expressed as the intersection of a finite number of affine half-spaces of \( \mathbb{R}^n \), then \( P \in \mathbb{R}^n \) is a convex polyhedron. Formally, a convex polyhedron \( P = (\Theta, n) \) is a set of linear inequalities \( \Theta = \{\beta_1, \beta_2, \ldots, \beta_m\} \subset \mathbb{R}^n \).

**Sound operations in polyhedra domain.** Let us recall from [41], [52], [53] some useful operations in the abstract domain of polyhedra:

- **Emptiness test:** Program analyzers during their analysis may encounter constraints present in program statements. Addition of a constraint to a non-empty polyhedron may lead to an empty polyhedron. A polyhedron is empty if and only if its constraint set is infeasible. The Linear Programming (LP) solver [54] is used for checking feasibility of such constraint systems. For example, adding a new constraint \( \bar{v} \bar{x} \geq k \) to a non-empty polyhedron \( P \), we can solve the LP problem \( \mu = \min \bar{v} \bar{x} \) subject to \( P \). If \( k > \mu \), then new polyhedron is empty. Alternatively, in generator representation a polyhedron is empty if and only if its set of vertices and rays are empty.

- **Projection:** Let \( P \) be a non-empty polyhedron. The projection operation removes all constraints information from \( P \) corresponding to a variable \( x_i \) without affecting the relational information between other variables, defined as:

\[
\Pi_{x_i}(P) = \{\{v[\forall v_i] | \nu \in \gamma(P), v \in \mathbb{R}\} \}
\]

This is computed by eliminating all occurrences of \( x_i \) in the constraints of \( P \) by using the Fourier-Motzkin algorithm [55] as below:

\[
F(\nu_0, P, x_i) = \{\nu \in \mathbb{R}^n | \nu \leq k\} \cap \Theta^+ \cup \{(-v_i^+)^+ + v_i^- v_i^- | v_i = (\Sigma v_i^+ x_i > k^+) \in \Theta^+, v_i^+ > 0, \}
\]
\[
- v_i^- v_i^- | v_i = (\Sigma v_i^+ x_i > k^+) \in \Theta^+, v_i^- < 0\}
\]

where \( v_i^+ \) and \( v_i^- \) represent positive and negative coefficients for \( x_i \) respectively. The
algorithm partitions the set of linear inequalities \( \Theta = \{ \beta_1, \beta_2, \ldots, \beta_n \} \) into \( \Theta^+, \Theta^- \) and \( \Theta^* \), corresponding to inequalities that have positive, zero and negative coefficients for \( x_i \). For each pair \( (\beta^+, \beta^-) \) of inequalities drawn from \( \Theta^+ \times \Theta^- \), the algorithm multiplies \( \beta^+ \) by the absolute value of \( x_i \)-th coefficient \( |v_i^+| \) in \( \beta^+ \) and similarly multiplies \( \beta^- \) by \( x_i \)-th coefficient \( v_i^- \) in \( \beta^- \). The combination of these two results finally removes \( x_i \) as the resultant coefficient becomes zero.

- **Inclusion test:** Let \( P_1 \) and \( P_2 \) be non empty polyhedra. The inclusion test (denoted \( P_2 \subseteq P_1 \)) reduces to the problem of checking whether each inequality in \( P_2 \) is entailed by \( P_1 \), which can be implemented using LP. For example, we can compute \( \mu = \min \{ x \} \) subject to \( P_1 \) for each \( \forall x \geq k \). If \( \mu < k \) then inclusion does not hold.

**Abstract Semantics of Imperative Language in Polyhedra – A Quick Tour [41].**

Given the concrete denotational semantics of \( \forall \) : \( \mathbb{B} \mapsto \mathbb{P}(\mathbb{E}) \mapsto \mathbb{P}(\mathbb{E}) \), the corresponding sound abstract function \( \mathcal{F}_f \) in the domain of polyhedra is defined as \( \mathcal{F}_f : \mathbb{B} \mapsto \mathbb{P} \). Similarly, given the concrete denotational semantic function \( \forall \) : \( \mathbb{C} \mapsto \mathbb{P}(\mathbb{E}) \mapsto \mathbb{P}(\mathbb{E}) \), its corresponding sound abstract function \( \mathcal{F}_c \) in polyhedra domain is defined as \( \mathcal{F}_c : \mathbb{C} \mapsto \mathbb{P} \).

**Test:** Let \( b \) be a boolean expression in the form of linear inequalities \( \forall x \geq k \) and the abstract state in the form of polyhedra \( \mathbb{P} \). The state-filtering function \( \mathcal{F}_f \) finds \( \mathbb{P}' \) applying \( b \) on \( \mathbb{P} \) defined as \( \mathcal{F}_f(\forall x \geq k)\mathbb{P} = \mathbb{P}' \) where \( \mathbb{P}' = \mathbb{P} \cap b \).

**Example 17.** Given \( \mathbb{P} = \{x \geq 8, y \geq 6, 2\} \). The equivalent generators representation (vertices and rays) of \( \mathbb{P} \) is \( V = \{8, 6\} \) and \( R = \{1, 0\} \). The abstract semantics of boolean expression \( x \geq 20 \) is defined as: \( \mathcal{F}_f(x \geq 20)\mathbb{P} = \mathbb{P}' \) where \( \mathbb{P}' = \{x \geq 20, y \geq 6, 2\} \) and its equivalent generators representation is \( V' = \{20, 6\} \) and \( R' = \{(1, 0), (0, 1)\} \).

**Assignment statement:** \( \mathcal{F}_c[x = e] \mathbb{P} = \mathbb{P}' \) where \( \mathbb{P}' \) is obtained as follows: (i) **Case-1:** If \( e \) is non-expression or the assignment is non-invertible, then we simply project-out the corresponding variable from the linear inequalities in \( \mathbb{P} \), resulting into a new polyhedron \( \mathbb{P}' \); (ii) **Case-2:** otherwise, we introduce a fresh variable \( x' \) to hold the value of \( e \), then we project out \( x \) and finally we reuse \( x' \) in place of \( x \) which results into \( \mathbb{P}' \).

**Example 18.** Given \( \mathbb{P} = \{x \geq 3, y \geq 2, 2\} \). The equivalent generators representation (vertices and rays) of \( \mathbb{P} \) is \( V = \{3, 2\} \) and \( R = \{1, 0\} \). The \( \mathcal{F}_c \) of assignment \( x = x + y \) is defined as \( \mathcal{F}_c(x = x + y)\mathbb{P} = \mathbb{P}' \) where \( \mathbb{P}' = \{(x - y \geq 3, y \geq 2, 2\) and its equivalent generators representation is \( V' = \{(5, 2)\} \) and \( R' = \{(1, 0), (-1, -1)\} \).

**Defining Abstract Semantics of Database Language in Polyhedra Domain.**

Let us define the abstract semantics for four database operations in the domain of polyhedra. Like octagon domain, given \( \mathbb{P} = \{p_d, p_a\} \in \mathcal{F}_d \) where \( p_d \) and \( p_a \) are polyhedra representation of database values and application variables values respectively. According to equation 8, the abstract semantic function for database statements \( Q = \{A, \phi\} \) is defined as: \( \mathcal{F}_d(\langle A, \phi\rangle)(\{p_d, p_a\}) = \mathcal{F}_d(\langle A, \phi\rangle)(\{p_d, p_a\}) = \{P, \mathbb{P} \} \) where \( P \) is the polyhedron representation of the concrete table \( t \) which acts as the target of \( Q \), and \( P \) is the polyhedron representation of the resultant table \( t' \).

Alternatively, like octagon abstract domain, for the sake of simplicity, we may combine both \( P_d \) and \( P_a \) into a single polyhedra \( P \) as an abstract program state. In the subsequent sections we define abstract semantics suitable for independency computations w.r.t. an abstract state \( \mathbb{P} \in \mathbb{P} \) in the domain of polyhedra.

**Abstract Semantics towards Independency Computation.**

Let us define the transition relation \( \mathcal{F}_d : \mathbb{C} \times \mathbb{P} \mapsto (\mathbb{P} \times \mathbb{P} \times \mathbb{P}) \) to compute semantics-based DD-independency in the domain of polyhedra for database statements:

1. **UPDATE:** \( \mathcal{F}_d(\langle \text{UPDATE}(d_i, e), \phi\rangle)\mathbb{P} = \{P_r, P_T, P_T\} \) where \( \mathcal{F}_d(\langle \text{UPDATE}(d_i, e), \phi\rangle)\mathbb{P} = \{P_r, P_T, P_T\} \).

2. **INSERT:** \( \mathcal{F}_d(\langle \text{INSERT}(d_i, e), \phi\rangle)\mathbb{P} = \{P, P_{\perp}, P_{\text{new}}\} \) where \( \mathcal{F}_d(\langle \text{INSERT}(d_i, e), \phi\rangle)\mathbb{P} = \{P, P_{\perp}, P_{\text{new}}\} \).

3. **DELETE:** \( \mathcal{F}_d(\langle \text{DELETE}(d_i), \phi\rangle)\mathbb{P} = \{P_r, P_T, P_{\perp}\} \).

4. **SELECT:** \( \mathcal{F}_d(\langle \text{SELECT}(d_i, c(i)), \phi_1, \phi_2, \phi_3\rangle)\mathbb{P} = \{P_r, P_T, P_T\} \).

**Example 19.** Consider the database table \( t \) in Table 4(a) and its corresponding abstract representation \( P_t = (\Theta, n) \) in the form of polyhedra, where

\[ P_t = \{cid \geq 1, -cid \geq -4, sal \geq 800, -sal \geq -3000, \\
\text{age} \geq 28, -\text{age} \geq -62, dno \geq 10, -dno \geq -20\} \]
Consider the following statements:

\[ Q_{\text{upd}} = \text{UPDATE} \ I \ \text{SET} \ sal = sal + sal \times 0.2 \ \text{WHERE} \ dno + age \geq 60 \]
\[ Q_{\text{ins}} = \text{INSERT INTO} \ I \ \text{(eid, sal, age, dno)}\text{VALUES}(5, 2700, 52, 20) \]
\[ Q_{\text{del}} = \text{DELETE FROM} \ I \ \text{WHERE} \ age \geq 61 \]
\[ Q_{\text{sel}} = \text{SELECT age FROM} \ I \ \text{WHERE} \ age + dno \leq 60 \]

The equivalent abstract syntax are:

\[ Q_{\text{upd}} = (\text{UPDATE}((\text{sal}, \langle \text{sal} + \text{sal} \times 0.2 \rangle)), \ dno + age \geq 60) \]
\[ Q_{\text{ins}} = (\text{INSERT}((\text{eid, sal, age, dno}), (5, 2700, 52, 20)), \ false) \]
\[ Q_{\text{del}} = (\text{DELETE}((\text{eid, sal, age, dno})), \ age \geq 61) \]
\[ Q_{\text{sel}} = (\text{SELECT}((\text{age})), \ age + dno \leq 60) \]

The abstract semantics of \( Q_{\text{upd}} \) w.r.t. \( P_i \) is

\[ \{ \langle \text{sal}, \langle \text{sal} + \text{sal} \times 0.2 \rangle \rangle, \ dno + age \geq 60 \} P_i \]

where

\[ P_T = \{ \text{eid} \geq 1, \dno \leq 62, \text{dno} \geq 10, \text{dno} \geq 20, \text{dno} + age \geq 60 \} \]

The abstract semantics of \( Q_{\text{ins}} \) w.r.t. \( P_i \) is

\[ \{ \langle \text{sid}, \text{sal}, \text{age}, \text{dno} \rangle, (5, 2700, 52, 20) \} P_i \]

where

\[ P_T = \{ \text{eid} \geq 1, \dno \leq 62, \text{dno} \geq 10, \text{dno} \geq 20, \text{dno} + age \geq 60 \} \]

The abstract semantics of \( Q_{\text{del}} \) w.r.t. \( P_i \) is

\[ \{ \langle \text{age} \rangle, \ age + dno \leq 60 \} P_i \]

where

\[ P_T = \{ \text{eid} \geq 1, \dno \leq 62, \text{dno} \geq 10, \text{dno} \geq 20, \text{dno} + age \geq 60 \} \]

6.2.4 Powerset Abstract Domain

The finite powerset construction of an abstract domain yields a new abstract domain which improves the precision of the analysis as compared to the original one [56]. For example, application of condition-part in many cases may result in multiple abstract values for an attribute. In such cases the powerset representation of abstract state is more suitable in terms of precision. Due to the scattered nature of data in the database, the semantics-based dependency analysis of database applications in the above-mentioned abstract domains may often be highly over-approximated. Thus powerset abstract domains, on top of the existing relational- and non-relational abstract domains, may capture the database values as a way of refined approximation, improving the analysis results significantly.

Let \( L_c = (D_c, \leq_c, \top_c, \bot_c, \cup_c, \cap_c) \) be a concrete lattice and \( L_p = (\bar{D}_c, \subseteq_p, \top_p, \bot_p, \cup_p, \cap_p) \) an abstract lattice over an abstract domain \( A \). The \( L_c \) and \( L_p \) are related by the Galois connection \((L_c, \alpha, \gamma, L_p)\). Considering the powerset abstract domain, the powerset of \( \bar{D} \) denoted by \( \varphi(\bar{D}) \) with the order relations \( \leq \) forms an abstract lattice \( L_p = (\varphi(\bar{D}), \subseteq, \emptyset, \emptyset, \varphi(D)) \). The partial order, meet and join operations in this abstract domain are defined as follows:

- \( \forall S_1, S_2 \in \varphi(D) : S_1 \leq S_2 \leftrightarrow \forall \bar{v}_j \in S_1 : \exists \bar{v}_j \in S_2, \bar{v}_j \subseteq \bar{v}_j \).
- \( \forall S_1, S_2 \in \varphi(D) : S_1 \cap S_2 = [\bar{v}_j \cap \bar{v}_j] \cup [\bar{v}_j \cup \bar{v}_j \in S_2] \).
- \( \forall S_1, S_2 \in \varphi(D) : S_1 \cup S_2 = S_1 \cup S_2 \).

Observe that in powerset abstract domain the meet operation \( S_1 \cap S_2 \) is defined by the pairwise meet of the elements from \( S_1 \) and \( S_2 \), whereas the join operation \( S_1 \cup S_2 \) reduces to a set union.

The \( L_c \) and \( L_p \) are related by a Galois connections \((L_c, \alpha, \gamma, L_p)\) where \( \alpha \) and \( \gamma \) on \( \forall X \in D \) and \( \forall Y \in \varphi(D) \) are defined below:

\[ \alpha_1(X) = \begin{cases} \emptyset & \text{if } X = \bot_c \\ \bar{D}_c & \text{if } X = \top_c \\ \{ \alpha(X) \} & \text{otherwise} \end{cases} \]

\[ \gamma_1(Y) = \begin{cases} \bot_c & \text{if } Y = \emptyset \\ \top_c & \text{if } Y = \bar{D}_c \\ \cup \{ \gamma(Y) \mid \bar{v} \in Y \} & \text{otherwise} \end{cases} \]

The pictorial representation of the Galois Connection among the concrete domain \( L_c \), the abstract domain \( L_p \), and the powerset of the abstract domain \( L_p \) is shown below:

![Diagram](image)

Let us explain the powerset construction over the interval abstract domain. Consider the interval abstract domain \( \bar{I} = [l, h] \cup [l, h] \cup [-\infty, h] \cup [h, +\infty), l \leq h \cup \bot \) forming an
abstract lattice \( L_\rho = (\mathcal{I}, \subseteq, \cup, \rho) \). The powerset of the intervals denoted by \( \wp(\mathcal{I}) \) forms the abstract lattice \( L_\wp = (\wp(\mathcal{I}), \subseteq, \emptyset, \wp(\mathcal{I}), \gamma) \).

The correspondence between \( L_\rho \) and \( L_\wp \) is formalized as the Galois connections \((\rho_\gamma, \alpha_1, \gamma_1, L_\rho)\). The partial order, meet and join operations in the powerset domain of intervals can be defined accordingly.

Given a powerset abstract domain of intervals \( \wp(\mathcal{I}) \), the set of abstract states \( \Sigma : \forall \rho \in \Sigma, \wp(\mathcal{I}) \models \wp(\mathcal{I}) \). The corresponding sound abstract semantics function \( \mathcal{F} \) for abstract state-filtering and join operations in the powerset domain of intervals can be defined accordingly.

The corresponding sound abstract semantics function \( \mathcal{F} \) for abstract state-filtering and join operations in the powerset domain of intervals can be defined accordingly.

Example 20. Consider the statement \( c := x > 5 \) then \( x = x + y \) else \( x = x - y \). Consider an abstract state in the powerset domain of intervals \( \wp(\Sigma) \models \wp(\Sigma) \) is illustrated below:

\[
\mathcal{F} \models c \models \wp(x > 5) \models \wp(x = x + y) \models \wp(x = x - y) \models \wp(x = x - y)
\]

The abstract semantics of \( c \) w.r.t. \( \wp(\Sigma) \) is illustrated below:

\[
\mathcal{F} \models c \models \wp(x = x + y) \models \wp(x = x - y) \models \wp(x = x - y)
\]

The corresponding sound abstract semantics function \( \mathcal{F} \) for abstract state-filtering and join operations in the powerset domain of intervals can be defined accordingly.

### 6.3 A Summary on Abstract Domains

Let us summarize the strengths and limitations of various relational and non-relational abstract domains which we have used above to define abstract semantics of database applications. As abstraction in the interval domain does not capture any relation among variables or attributes, this yields a highly approximated analysis-results. On the other hand, although abstract semantics in both octagon and polyhedra domains capture relationships among variables or attributes, the octagon domain allows a weak form of constraints compared to that in polyhedra domain. Due to this reason, analysis in octagon domain is less precise than that in the polyhedra domain. Intuitively, preciseness of the analysis in relational abstract domain improves significantly when more number of relations among variables or attributes is present in the program itself, e.g. in the WHERE clause or in the conditional or iterative statements. In terms of algorithmic efficiency, octagon domain always lies between interval and polyhedra. Analyses (involving all common operations e.g. emptiness test, inclusion, etc.) in polyhedra abstract domain experience an exponential \((O(2^n))\) worst-case time complexity [54], whereas in octagonal domain the graph-based analysis algorithms for all common operations experience \(O(n^2)\) worst-case time complexity, where \( n \) is the number of variables in the program [43]. Powerset operator, on the other hand, can generate very expressive interpretations. In fact, the powerset abstract domain gains the capability of expressing the logical disjunction of the properties represented by the original domain. A summary on the strength and weakness of domains is reported in Table 8.

### 6.4 Algorithm to Compute Abstract Semantics of Database Applications

We now design the algorithm \texttt{CompAbsSem}, depicted in Algorithm 1, which makes use of the semantics function \( \mathcal{F}_\sigma \) and computes abstract states w.r.t. abstract domain \( D \) at each program point of the database program. The algorithm is based on the data flow analysis considering various control-flow nodes: \texttt{start}, \texttt{DB-connect}, \texttt{assignment}, \texttt{test}, \texttt{update}, \texttt{delete}, \texttt{insert}, \texttt{select}, \texttt{join}, \texttt{end}. We denote by \texttt{pred(c)} and \texttt{AS(c)} the set of predecessor of \( c \) and the abstract state at \( c \) respectively. The algorithm starts in step 2 with undefined abstract state at each program point and then applies in step 3 all the data-flow equations (defined in steps 4-25)
until least fixed point solution is reached. After obtaining the abstract state at each program point in the form of collecting semantics, step 26 applies \( \overline{\mathcal{T}}_{dep} \) in order to get state-representation in the form of three-tuples \((\rho, p, x)\) (as defined in equation 9). This abstract semantics is used to compute used- and defined-parts and hence the semantics-based independencies (described next). Observe that if the initial database is unknown then the domain range of each attribute and other integrity constraints are considered to represent the initial abstraction of database as an overapproximation of all possible initial database states, as defined in steps 1 and 9.

6.5 Approximating used- and defined Database Parts in Various Abstract Domains

Given a database statement \( Q \), let \( \overline{\rho} = (\rho_\mathcal{P}, \rho_\mathcal{D}, \rho_\mathcal{X}) \) be an abstract state at \( Q \) obtained by following Algorithm 1. In order to determine abstract DD-dependency between two database statements, we need to identify abstract database-parts to be defined or used by \( Q \). To this aim, let us define sound abstract functions \( \overline{\mathcal{A}}_{\text{def}} \) and \( \overline{\mathcal{A}}_{\text{use}} \) w.r.t. their concrete counterparts already defined in equations 6 and 7 respectively. Suppose \( D^Q \) and \( U^Q \) denote the defined and the used abstract database-parts by \( Q \) respectively. Therefore,

\[
D^Q = \overline{\mathcal{A}}_{\text{def}}(Q, \overline{\rho}) = (\rho_\mathcal{P}, \rho_\mathcal{D}, \rho_\mathcal{X}) \tag{12}
\]

\[
U^Q = \overline{\mathcal{A}}_{\text{use}}(Q, \overline{\rho}) = (\rho_\mathcal{P}) \tag{13}
\]

Observe that \( \overline{\mathcal{A}}_{\text{use}} \) maps a query \( Q \) to the abstract database-part used by it, whereas \( \overline{\mathcal{A}}_{\text{def}} \) defines the changes occurred in the abstract database states after performing the action in \( Q \). We represent \( D^Q \) in the form of two-tuple where \( \rho_\mathcal{P} \) and \( \rho_\mathcal{X} \) respectively represent the true-part before and the updated-part after executing \( Q \) on the abstract database. Note that although the defined-part can be computed by following the abstract difference operation \( \overline{\Delta} \) (corresponding to \( \Delta \) defined in equation 6), however, to avoid computational complexity in dependency computation, we keep both of these separated. Table 9 depicts defined and used parts by different database statements in various abstract domains.

6.6 Dependency Computations

We are now in a position to compute DD-independencies among database statements based on the information on used- and defined-parts as obtained in the previous section.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Invariants</th>
<th>Time complexity</th>
<th>Memory complexity</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval</td>
<td>( x \in [l, h] \mid l, h \in \mathbb{R}, l \leq h, x \in \mathbb{V} )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>low</td>
</tr>
<tr>
<td>Octagon</td>
<td>( \pm x_i \pm x_j \leq k ), ( x_i, x_j \in \mathbb{V} )</td>
<td>( O(n^3) )</td>
<td>( O(n^n) )</td>
<td>medium</td>
</tr>
<tr>
<td>Polyhedra</td>
<td>( \sum_{i=1}^{n} a_i x_i \leq q, x_i \in \mathbb{V} )</td>
<td>( O(2^n) )</td>
<td>( O(2^n) )</td>
<td>high</td>
</tr>
<tr>
<td>PowerSet</td>
<td>( \varphi(\mathcal{D}) )</td>
<td>Depends on ( \mathcal{D} )</td>
<td>Depends on ( \mathcal{D} )</td>
<td>Improves w.r.t. ( \mathcal{D} )</td>
</tr>
</tbody>
</table>

### Algorithm 1: CompAbsSem

**Input:** Database program \( \mathcal{P} \) containing \( n \) statements, Initial database \( DB \), and Abstract domain \( \overline{\mathcal{D}} \).

**Output:** Abstract state at each program point of \( \mathcal{P} \).

1. Compute \( \overline{\rho} = (\rho_\mathcal{P}, \rho_\mathcal{D}, \rho_\mathcal{X}) \) using the abstraction function \( a \) following the Galois connection \( \langle (\varphi(\mathcal{D}), \mathcal{L}), \alpha, \gamma, (\overline{\mathcal{D}}, \mathcal{X}) \rangle \) as formalized in Definition 7.
2. \( \forall i \in [1, \ldots, n] \), \( AS(c_i) = \bot \). \hspace{1cm} \text{// Initializing } AS(c_i) \text{ as initial abstract collecting semantics.}
3. Apply data flow equations defined in steps 4 - 25 until least fix-point is reached.
4. For \( i = 1 \) to \( n \) do

\[
\text{// Defining data flow equation for CFG-node corresponding to a statement } c_i \text{ in } \mathcal{P}.
\]

\[
\text{switch } (c_i) \\
\quad \text{case start:} \\
\quad \text{case DB-connect:} \\
\quad \text{case assignment:} \\
\quad \text{case test:} \\
\quad \text{case update:} \\
\quad \text{case delete:} \\
\quad \text{case insert:} \\
\quad \text{case select:} \\
\quad \text{case join:} \\
\quad \text{case end:}
\]

26. Apply the abstract transition relation \( \overline{\mathcal{T}}_{dep} \) on the abstract state \( \overline{AS}(c_i) \) obtained at each program point.

Let \( \overline{\rho}_1 = (\rho_\mathcal{P}^1, \rho_\mathcal{D}^1, \rho_\mathcal{X}^1) \) and \( \overline{\rho}_2 = (\rho_\mathcal{P}^2, \rho_\mathcal{D}^2, \rho_\mathcal{X}^2) \) be the abstract states at \( Q_1 \) and \( Q_2 \) respectively. The defined-part by \( Q_1 \) and the used-part by \( Q_2 \) are:

\[
D^{Q_1} = \overline{\mathcal{A}}_{\text{def}}(Q_1, \overline{\rho}_1) = (\rho_\mathcal{P}^1, \rho_\mathcal{D}^1, \rho_\mathcal{X}^1)
\]

TABLE 9: Abstract defined- and used-part of database by SQL statements in various abstract domains

\[
\mathcal{U}_Q = \bar{\mathsf{A}}_{\text{used}}(Q_2, \rho_1^Q) = \langle \rho_1^Q \rangle
\]

The semantic dependency and independency of \(Q_2\) on \(Q_1\) are determined based on the following four cases:

- **Case 1.** If \(\rho_1^Q \cap \rho_1^Q \neq \emptyset \) and \(\rho_1^Q \cap \rho_1^Q = \emptyset\).
- **Case 2.** If \(\rho_1^Q \cap \rho_1^Q = \emptyset \) and \(\rho_1^Q \cap \rho_1^Q \neq \emptyset\).
- **Case 3.** If \(\rho_1^Q \cap \rho_1^Q = \emptyset \) and \(\rho_1^Q \cap \rho_1^Q = \emptyset\).
- **Case 4.** If \(\rho_1^Q \cap \rho_1^Q \neq \emptyset \) and \(\rho_1^Q \cap \rho_1^Q = \emptyset\).

The pictorial representation of the above cases are depicted in Figure 12. Observe that only case 4 indicates a semantic independency between \(Q_1\) and \(Q_2\) whereas all other cases indicate a semantic dependency between them. Therefore, \(Q_2\) is DD-independent on \(Q_1\) iff \(\mathcal{D}_Q \cap \mathcal{U}_Q = \emptyset\); that is

\[
\rho_1^Q \cap \rho_1^Q = \emptyset \quad \land \quad \rho_1^Q \cap \rho_1^Q = \emptyset
\]

Theorem 1 states that, given an abstract domain, equation 14 is necessary and sufficient condition for abstract DD-independency. Observe that this theorem does not establish anything about its soundness w.r.t. its concrete counterpart.

**Theorem 1.** Given an abstract domain, the necessary and sufficient condition for a SQL statement \(Q_2\) to be abstract DD-independent on another statement \(Q_1\) is \(\mathcal{D}_Q \cap \mathcal{U}_Q = \emptyset \); that is

\[
\rho_1^Q \cap \rho_1^Q = \emptyset \quad \land \quad \rho_1^Q \cap \rho_1^Q = \emptyset
\]

**Proof.** Consider two database statements \(Q_1 = \langle A_1, \phi_1 \rangle\) and \(Q_2 = \langle A_2, \phi_2 \rangle\). Given the abstract states \(\overline{\mathcal{P}}\) and \(\overline{\mathcal{P}}\) at \(Q_1\) and \(Q_2\) respectively which are obtained in step 25 of Algorithm 1. Let the abstract semantics applying \(\overline{\mathsf{T}}_{\text{dep}}\) in step 26 be \(\overline{\mathsf{T}}_{\text{dep}}(Q_1, \overline{\mathcal{P}}) = \langle \rho_1^Q, \rho_1^Q, \rho_1^Q \rangle\) and \(\overline{\mathsf{T}}_{\text{dep}}(Q_2, \overline{\mathcal{P}}) = \langle \rho_2^Q, \rho_2^Q, \rho_2^Q \rangle\). Intuitively, we can say that \(Q_2\) is abstract DD-dependent on \(Q_1\) when any modification on the abstract database by \(Q_1\) affects the abstract database-part to be accessed by \(Q_2\). The following three kinds of affects may happen on \(Q_2\) due to \(Q_1\):

1. **Inclusion of new information:** Because of the modification by \(Q_1\) some new data may be accessed by \(Q_2\) satisfying \(\phi_2\). This is captured in Case-2.

2. **Removal of existing information:** As a result of the modification done by \(Q_1\), some information (which was previously accessed by \(Q_2\)) now can not be accessed by \(Q_2\) due to the unsatisfiability of \(\phi_2\). This is captured in Case-1.

3. **Access of modified information:** \(Q_2\) can access now modified values, instead of their original values, of some attributes due to the application of \(Q_1\). This is captured in Case-3.

Therefore, we can say \(Q_2\) is semantically abstract DD-independent on \(Q_1\) when the above three effects do not take place. In other words, the abstract database-part \(\rho_1^Q\) accessed by \(Q_2\) overlaps with the parts \(\rho_1^Q\) and \(\rho_1^Q\) referred by \(Q_1\) operations. This is captured in Case-4.

**Algorithm to Compute Semantics-based DD-dependencies.** The algorithm \(\text{semDOPDG}\) in Algorithm 2 takes a list of used- and defined-parts at each program point of the database program \(P\) and computes semantics-based DD-dependency among database statements. The algorithm, in step 2, first identifies all database statements present in the program. Step 5 inside the loops checks whether the defined-part by \(Q_1\) overlaps with the used-part by \(Q_1\), and accordingly DD-dependency edge is created between them in step 6 and the flag is set to true in step 7. If dependency exists between \(Q_1\) and \(Q_2\) and flag is true, then in the next step 11 the algorithm checks the condition \(\mathcal{D}_Q \subseteq \mathcal{D}_Q\) in order to verify whether defined-part at program point \(Q_1\) is fully covered by the defined-part at program point \(Q_1\). If yes, the execution immediately breaks the inner loop and does not check for dependency of the subsequent database statements (after \(Q_1\)) on \(Q_1\), and hence disregards the false dependencies which may occur due to redefinition of attributes values by \(Q_1\).

**7 Illustration on the Running Example**

Now we illustrate our approach on the running example “Proc” in section 2. The semantic-based data independencies are computed applying the following steps in different abstract domains:

- Compute abstract semantics using Algorithm 1 at each program point of “Proc”.
- Compute defined- and used-parts based on the abstract semantics.
- Refinement of syntactic dependencies in “Proc” based on the semantics-based independencies using Algorithm 2.

A comparative result of the analysis in various abstract domains is depicted in Table 10. Let us explain briefly few scenarios by illustrating our approach.

For the sake of simplicity, since statements 5 and 6 involve only the attribute ‘purchase_amt’ and the applications variables ‘x’ and ‘y’, let us consider the abstract initial state \(\overline{\mathcal{P}}\) taking those variables into account with an assumption that ‘purchase_amt’ is typed with unsigned smallint. Therefore, \(\overline{\mathcal{P}} = \overline{\mathcal{P}}\).
The **defined**-part by statement 5 and the used-part by statement 6 are defined as follows:

\[
D^5 = A_{df}(\rho^5, 5) = (\rho^5_{TM}, \rho^5_{TM}) \quad \text{and} \quad U^6 = A_{used}(\rho^6, 6) = (\rho^6_{TM})
\]

Therefore, the dependency \( 5 \to 6 \) does not exist semantically as

\[
D^5 \cap U^6 = \emptyset \implies \rho^5_{TM} \cap \rho^6_{TM} = \emptyset \land \rho^5_{TM} \cap \rho^6_{TM} = \emptyset
\]

This way we can easily capture semantics independencies. Note that interval analysis is not yet an optimal setting to capture all such independencies in “Prog”, for instance \( 15 \to 16 \).

On the other hand, consider the domain of polyhedra. Consider the statements 15 and 16 which involve attributes ‘purchase_amt’ , ‘wallet_bal’ and ‘point’. Let us consider the abstract initial database state in the form of polyhedron \( P_{db} \) based on the assumption that \( purchase\_amt \) is typed with unsigned smallint and the integrity constraints \( 0 \leq point < 100 \) and \( 100 \leq wallet\_bal \leq 90000 \) are defined on ‘point’ and ‘wallet_bal’. Therefore, the abstract state at program point 4 is:

\[
P_{db}^{11} = \{ purchase\_amt \geq 0, \neg purchase\_amt \geq -65000, point \geq 0, \neg point \geq -100, wallet\_bal \geq 100, \neg wallet\_bal \geq -90000 \}
\]

The abstract semantics of statement 15 is defined as:

\[
\overrightarrow{db}_{\text{dep}}[15](p_{db}^{11}) = p_{db}^{15} \quad \text{and} \quad \overrightarrow{db}_{\text{dep}}[15](p_{db}^{11}) = (p_{F}^{15}, p_{T}^{15}, p_{F}^{15})\]

where \( p_{F}^{15} = \langle purchase\_amt \geq 0, purchase\_amt \geq -65000, point \geq 0, \neg point \geq -100, wallet\_bal \geq 100, \neg wallet\_bal \geq -90000 \rangle \) and

\[
p_{T}^{15} = \langle purchase\_amt \geq 0, purchase\_amt \geq -9899, point \geq 0, \neg point \geq -100, wallet\_bal \geq 100, \neg wallet\_bal \geq -9999, 5000 \leq purchase\_amt + wallet\_bal \leq 9999 \rangle
\]

Similarly, the abstract semantics of statement 16 is:

\[
\overrightarrow{db}_{\text{dep}}[16](p_{db}^{15}) = p_{db}^{16} \quad \text{and} \quad \overrightarrow{db}_{\text{dep}}[16](p_{db}^{15}) = (p_{F}^{16}, p_{T}^{16}, p_{F}^{16})\]

where \( p_{F}^{16} = \langle purchase\_amt \geq 0, purchase\_amt \geq -9899, point \geq 0, \neg point \geq -100, wallet\_bal \geq 100, \neg wallet\_bal \geq -9999, purchase\_amt + wallet\_bal \leq 9999 \rangle \)
TABLE 10: Representation of dependency results on “Prog” in various approaches

<table>
<thead>
<tr>
<th>Data dependency</th>
<th>pure syntax-based</th>
<th>Improved syntax-based</th>
<th>Condition-Action rule-based</th>
<th>Interval domain</th>
<th>Octagon domain</th>
<th>Polyhedra domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>DD-dependency</td>
<td>4 → [5, 6, 7, 11, 15, 16]</td>
<td>4 → [5, 6, 7, 11, 15, 16]</td>
<td>4 → [5, 6, 7, 11, 15, 16]</td>
<td>4 → [5, 6, 7, 11, 15, 16]</td>
<td>4 → [5, 6, 7, 11, 15, 16]</td>
<td></td>
</tr>
<tr>
<td>PD-dependency</td>
<td>2 → 6, 3 → 5</td>
<td>2 → 6, 3 → 5</td>
<td>2 → 6, 3 → 5</td>
<td>2 → 6, 3 → 5</td>
<td>2 → 6, 3 → 5</td>
<td></td>
</tr>
</tbody>
</table>

**Lemma 2.** Let \( \overline{p} \) be an abstract state. The abstract semantic function \( F_{dep} \) is sound w.r.t. \( \gamma \) if \( \forall Q \in Q, \forall \rho \in \gamma(\overline{p}) \): \( F_{dep}[Q]\rho \subseteq \gamma(\overline{F_{dep}[Q]p}) \). **Proof.** Given an abstract state \( \overline{p} \) and a database statement \( Q = (A, \phi) \in Q \), the abstract semantic function \( F_{dep} \) on \( \overline{p} \) computes abstract database state in the form of three-tuple as follows:

\[
F_{dep}[Q]\rho = (\rho_{\phi}, \rho_{\phi}), \quad \rho_{\phi} = (\rho_{\phi}, \rho_{\phi})
\]

where \( \rho_{\phi} \) represents abstract database state which must (or may) not satisfy \( \phi \), whereas \( \rho_{\phi} \) represents abstract database state which must (or may) satisfy \( \phi \). Now let \( \rho \) be a concrete state such that \( \rho \in \gamma(\overline{p}) \), the abstract semantics similarly is defined as

\[
\overline{F_{dep}[Q]p} = (\rho_{\phi}, \rho_{\phi})
\]

where \( \rho_{\phi} \) and \( \rho_{\phi} \) represent concrete database state based on the satisfaction and dissatisfaction of \( \phi \) respectively, and \( \rho_{\phi} \) is obtained after performing an action \( A \) on \( \rho_{\phi} \). Like in lemma 1, because of three-valued logic of \( \phi \) due to the imprecision introduced in the abstract domain and the local correctness of the operations in \( A \), we get \( \rho_{\phi} \in \gamma(\overline{p}) \) and \( \rho_{\phi} \in \gamma(\overline{p}) \) implies that \( \overline{F_{dep}[Q]p} \subseteq \gamma(\overline{F_{dep}[Q]p}) \).

**Lemma 3.** Let \( \overline{p} \) be an abstract state. The abstract function \( \overline{A}_{def} \) is sound w.r.t. \( \gamma \) if \( \forall \rho \in \gamma(\overline{p}), \forall Q \in Q: \gamma(\overline{A}_{def}(Q, \rho)) \supseteq \overline{A}_{def}(Q, \rho) \).

**Proof.** Given a database statement \( Q = (A, \phi) \in Q \) and an abstract state \( \overline{p} \), the abstract semantics (based on equation 9) is defined as

\[
\overline{A}_{def}[Q]p = \overline{A}_{def}(A, \phi)(\overline{p})
\]

As per the equation 12, the abstract defined-part is

\[
\overline{A}_{def}(Q, \overline{p}) = (\rho_{\phi}, \rho_{\phi})
\]

Now given a concrete state \( \rho = (p_{\phi}, p_{\phi}) \in \gamma(\overline{p}) \), we get the concrete semantics of \( Q \) according to equation 5, as

\[
\overline{A}_{def}[Q]p = \overline{A}_{def}(A, \phi)(p_{\phi}, p_{\phi}) = (p_{\phi}, p_{\phi})
\]

...
Alternatively, $\mathcal{J}_{dep}$ on $\rho$ computes concrete semantics of $Q$ as $\mathcal{J}_{dep}[\mathcal{Q}, \rho] = \mathcal{J}_{dep}[[A, \phi]] = (\rho_\phi, \rho_Q, \rho_\bullet)$. As per the equation 6, we can define the defined-part in the concrete domain by defining $\Delta$, which computes the difference between database states before and after applying $Q$, in the form below:

$$A_{dep}(Q, \rho_1) = (\rho_\phi, \rho_1) = (\rho_\phi, \rho_Q, \rho_\bullet)$$

Assuming the local correctness of $\phi$ and $A$, we get $\rho_\phi \subseteq \gamma(\rho_\phi)$ and $\bullet \subseteq \gamma(\rho_\bullet)$ respectively. Therefore, $\gamma(\mathcal{J}_{dep}(Q, \rho)) \supseteq A_{use}(Q, \rho)$.

**Lemma 4.** Let $\overline{\rho}$ be an abstract state. The abstract function $\overline{A_{use}}$ is sound w.r.t. $\gamma$ if $\forall \rho \in \gamma(\overline{\rho})$, $\forall Q : \gamma(\overline{A_{use}}(Q, \overline{\rho})) \supseteq A_{use}(Q, \rho)$

**Proof 5.** Proof is same as lemma 3.

**Soundness.** The semantics-based independency computation is sound if and only if an absence of dependency in the abstract domain guarantees that no dependency is present in the concrete domain.

**Theorem 2 (Soundness of semantic independencies).** Given two database statements $Q_1$ and $Q_2$, let $\overline{\rho}$ and $\overline{\rho'}$ be the abstract states at $Q_1$ and $Q_2$ respectively. The computation of semantic independency is sound if

$$\forall X \in \gamma(\overline{A_{df}}(Q_1, \overline{\rho})), \forall Y \in \gamma(\overline{A_{use}}(Q_2, \overline{\rho'})) : X \cap Y \subseteq \gamma(\overline{A_{use}}(Q_2, \overline{\rho'}))$$

which implies $\overline{A_{df}}(Q_1, \overline{\rho}) \cap \overline{A_{use}}(Q_2, \overline{\rho'}) = \emptyset \Rightarrow X \cap Y = \emptyset$.

**Proof 6.** Consider two database statements $Q_1$ and $Q_2$. Let $\overline{\rho} = (\rho_\phi, \rho_Q, \rho_\bullet)$ and $\overline{\rho'} = (\rho'_\phi, \rho'_Q, \rho'_\bullet)$ be the abstract states at $Q_1$ and $Q_2$ respectively, which are obtained by applying Algorithm 1. According to equations 12 and 13, we get the defined-part by $Q_1$ and the used-part by $Q_2$ as $\overline{A_{df}}(Q_1, \overline{\rho}) = (\rho_\phi, \rho_Q, \rho_\bullet)$ and $\overline{A_{use}}(Q_2, \overline{\rho'}) = (\rho'_\phi, \rho'_Q, \rho'_\bullet)$ respectively.

Now, the semantics independency in abstract domain can be defined as $\overline{\rho} \cap \overline{\rho'} = \emptyset \wedge \rho_Q \cap \rho'_Q \neq \emptyset$. Given the concrete states $\rho = (\rho_\phi, \rho_Q, \rho_\bullet) \cap (\rho'_\phi, \rho'_Q, \rho'_\bullet)$ where $\rho \in \gamma(\overline{\rho})$ and $\rho' \in \gamma(\overline{\rho'})$, the semantics independency between $Q_1$ and $Q_2$ in the concrete domain is defined as $\rho_\phi \cap \rho'_\phi = \emptyset \wedge \rho_\phi \cap \rho'_\phi \neq \emptyset$. From lemma 3 and 4, we get $\gamma(\overline{A_{df}}(Q_1, \overline{\rho})) \supseteq A_{df}(Q_1, \rho)$ and $\gamma(\overline{A_{use}}(Q_2, \overline{\rho'})) \supseteq A_{use}(Q_2, \rho')$ respectively. This implies that $\gamma(\overline{A_{df}}(Q_1, \overline{\rho}) \cap \overline{A_{use}}(Q_2, \overline{\rho'})) \supseteq A_{df}(Q_1, \rho) \cap A_{use}(Q_2, \rho')$. Therefore, $\overline{A_{df}}(Q_1, \overline{\rho}) \cap \overline{A_{use}}(Q_2, \overline{\rho'}) = \emptyset \Rightarrow X \cap Y = \emptyset$ where $X = \overline{A_{df}}(Q_1, \rho) \in \gamma(\overline{A_{df}}(Q_1, \overline{\rho}))$ and $Y = \overline{A_{use}}(Q_2, \rho') \in \gamma(\overline{A_{use}}(Q_2, \overline{\rho'}))$.

9 **Implementation and Experimental Evaluation.**

We have implemented a prototype tool SemDDA – Semantics-based Database Dependency Analyzer – following the Algorithms 1 and 2 to perform experimental evaluation on a set of open-source database-driven JSP web applications as part of the GotoCode project [37].

6. The source code is available on github: https://github.com/angshumanjana/SemDDA.
7. The original website “http://www.gotocode.com” does no longer exist at this moment. We have archived the benchmark codes at “https://github.com/angshumanjana/GotoCode”.
8. These benchmark codes are used by many authors in their experiments, such as [57], [58], [59], [60].

**9.1 The SemDDA Tool**

The aim of designing SemDDA is to provide a user-friendly interface for the users to compute both syntax and semantics-based DD-dependency in various abstract domains of interest. The current implementation is in its preliminary stage which accepts only database-driven JSP codes. We provide a modular-based design and implementation of our tool, facilitating an easy expansion in future. The tool consists of two major components: (i) Syntax-based module, and (ii) Semantic-based module. Figure 13 depicts the overall architecture of the tool, where database program and underlying database are provided as input and a set of syntax-based dependencies and its refinement based on the abstract semantics are generated as output. The code is implemented in Java version 1.7. We used Eclipse version 4.2 as the development platform and Java applet technology for designing User Interfaces of semSSA.

(i) **Proformat:** The module “Proformat” accepts database code written in JSP embedding SQL, and preprocesses it to add line numbers (starting from zero) to all statements, ignoring comments. Assuming input programs syntactically correct, the module separates program’s statements based on the predefined delimiters and right braces. During this process, it also computes Non-Comment Lines of Code (NCLOC) and the number of SQL statements present in the program. In particular, the presence of Data Manipulation Language (DML) statements is identified based on the presence of keywords such as SELECT, UPDATE, DELETE and INSERT in the statements.

(ii) **ExtractInfo:** This module extracts detail information about input programs, i.e. control statements, defined variables, used variables, etc. for all statements in the program.

Modules “Proformat” and “ExtractInfo” currently support only JSP embedded database code. The extension of these modules to support other programming languages does not require major design efforts, and it is currently in the to-do list for the next version of our analyser.

(iii) **Dependency:** The “Dependency” module computes syntax-based dependencies among program statements using the information computed by “ExtractInfo” module.

(iv) **Tuning:** At this preliminary stage of implementation, this module supports three abstract domains (Interval, Octagon, and Polyhedra). The module automatically picks the
best domain based on the attribute relationships present in SQL statements. If none of the statements contains any relationship among attributes, then "Tuning" module automatically picks interval domain. On the other hand, either octagon or polyhedra abstract domain is chosen if at least one SQL statement contains respectively octagonal or polyhedral form of constraint. Moreover, users can also select one of the abstract domains of her choice based on the importance of computational cost and analysis-precision.

(v) Abstraction: The module "Abstraction" computes abstract semantics in the chosen abstract domain based on the data-flow analysis. Currently the module supports intervals, octagons, polyhedra, and powerset of intervals abstract domains.

(vi) Overlap: Finally this module identifies false dependency (if any) based on the semantics-based approximation of used and defined parts and their overlapping.

9.2 Experimental Results

We have used semDDA to perform experiments on a set of benchmark programs which are open-source database-driven web applications in JSP as part of the GotoCode project [37]. A brief description of these benchmark codes are mentioned in Table 11. The experiment is performed on a system configured with Intel i3 processor, 1.80GHz clock speed, Windows 7 Professional 64-bit Operating System with 8GB RAM.

In the following sections, we provide experimental results in various approaches on a set of benchmark codes under consideration.

9.2.1 DD-dependency results in pure syntax-based approach

The DD-dependency results on the benchmark codes in pure syntax-based approach is depicted in the 5th column of Table 12. It is worthwhile to mention that, for the given benchmark codes, the improved syntax-based approach generates same results as that by pure syntax-based approach.

9.2.2 DD-dependency results in Condition-Action rules-based approach

We implemented Condition-Action rules using Satisfiability Modulo Theories (SMT). In particular, we used Z3 [61], a high-performance SMT Solver implemented in C++ and developed by Microsoft Research. For this purpose, we performed the following steps: (i) Selection of database statements in pairs according to their order of occurrences in the program, (ii) Conversion of these database statements into Static Single Assignment (SSA) form, (iii) Generation of Verification Condition (VC) from each pair by extracting predicates from the action- and condition-parts of the first statement and the condition-part of the second statement in the pair, and finally (iv) Dependency verification based on the satisfiability of VCs using Z3 tool. We used the online version of the Z3 tool available at “https://rise4fun.com/z3". We encoded VCs by following Z3 language syntax (which is an extension of the one defined by the SMT-LIB 2.0 standard). After compilation and execution by Z3, the output "UNSAT" for a pair indicates that the second database statement is not dependent on the first one in the pair. Let us explain this with the following simple example.

Example 21. Consider the following pair of database statements:

- \( Q_1 : \text{UPDATE } emp \text{ SET } hra = hra + 100 \text{ WHERE } da + hra \geq 1000 \)
- \( Q_2 : \text{SELECT } hra \text{ FROM } emp \text{ WHERE } da + hra \leq 5000 \)

The equivalent SSA form of these statements are:

- \( Q_1 : \text{UPDATE } emp \text{ SET } hra2 = hra1 + 100 \) \text{ WHERE } da1 + hra1 \geq 1000
- \( Q_2 : \text{SELECT } hra2 \text{ FROM } emp \text{ WHERE } da1 + hra2 \leq 5000 \)

The VC of this pair of statements is:

\[ V_c = (hra2 == hra1 + 100) \land (da1 + hra1 \geq 1000) \land (da1 + hra2 \leq 5000) \]

As the Z3 reports this formula as satisfiable (Z3 output is "SAT"), this indicates that \( Q_2 \) depend on \( Q_1 \).

The DD-dependency results on the benchmark codes using this approach is depicted in the 6th column of Table 12. This shows an improvement in the precision over the syntax-based results. In fact, on the given benchmark codes, an average of 12% improvement is observed as compared to the syntax-based approach.

9.2.3 Results based on the Abstract Semantics

Columns 7th, 8th, 9th and 10th of Table 12 depict DD-dependency results in the domains of intervals, octagons, polyhedra and powerset of intervals respectively. It is worthwhile to note that the analysis-results for five benchmark codes (‘EditOfficer’, ‘EditMember’, ‘BookMaint’, ‘EditorialsRecord’, ‘BugRecord’) in the interval domain improves w.r.t. their syntax-based results. On the other hand, analysis in the domain of octagons for ‘EmployeeMaint’, ‘ProjectMaint’, ‘EventNew’ and ‘EmployeeMaint’ results in more precise dependency information compared to that in the interval domain, due to the presence of restricted attributes relationship (which involves at most two attributes) in SQL statements. Similarly, polyhedra domain analysis captures more precise DD-dependency results, shown in the case of ‘LedgerRecord’ and ‘EmpsRecord’, compared to their interval and octagon counterparts, as they allow unrestricted relationship among attributes. We obtain an
Applications | Number of Files Tested | Descriptions
---|---|---
Events | 1 | It is a basic online event management system. It includes many features like event information (event name, year, presenter, etc.), users administration, etc.
Ledger | 1 | It is an example implementation of a web-based ledger which allows a user to track bank deposits, withdrawals, commission and view current balance.
Portal | 2 | It is a fully functional online web-based Portal which is useful for small organizations, clubs, user groups, and schools. It provides several functionalities like user registration, news section, list of club officers and etc. The considering files mainly work on the administration of club officers and members.
EmplDir | 2 | It is a basic employee directory that may use as an online system for small companies. It serves deferent searching facilities (e.g. by name, email) to the user. The selected files are dealing to store the employee and departmental information.
Bookstore | 2 | It is an online store system that keeps various books information, articles and other items. It has many features like user registrations, shopping cart, administration of credit card types and etc. It utilizes VeriSign’s payflow link system to verify and charge credit cards.
BugTrack | 3 | It is a basic fully functional web-based bug tracking system which may useful for small teams working on software projects. It keeps projects information and its associated employee’s detail (consider files work for this purpose), also provides many searching options.

<table>
<thead>
<tr>
<th>Applications (File Names)</th>
<th>NCLOC</th>
<th>Number of SQL Stmts</th>
<th>Number of Attributes</th>
<th>Pure Syntax-based</th>
<th>Condition-Action Rule-based</th>
<th>Interval Domain</th>
<th>Octagon Domain</th>
<th>Polyhedra Domain</th>
<th>Powerset of Intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events(EventNew.jsp)</td>
<td>334</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Ledger(LedgerRecord.jsp)</td>
<td>436</td>
<td>9</td>
<td>8</td>
<td>22</td>
<td>18</td>
<td>22</td>
<td>22</td>
<td>16</td>
<td>22</td>
</tr>
<tr>
<td>Portal(EditOfficer.jsp)</td>
<td>300</td>
<td>7</td>
<td>4</td>
<td>21</td>
<td>21</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>Portal(EditMembers.jsp)</td>
<td>362</td>
<td>10</td>
<td>5</td>
<td>16</td>
<td>15</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>EmplDir(DepsRecord.jsp)</td>
<td>285</td>
<td>4</td>
<td>3</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>EmplDir(EmpsRecord.jsp)</td>
<td>435</td>
<td>9</td>
<td>7</td>
<td>23</td>
<td>21</td>
<td>23</td>
<td>22</td>
<td>14</td>
<td>23</td>
</tr>
<tr>
<td>Bookstore(EditorialsRecord.jsp)</td>
<td>294</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Bookstore(BookMaint.jsp)</td>
<td>357</td>
<td>6</td>
<td>5</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>BugTrack(ProjectMaint.jsp)</td>
<td>307</td>
<td>7</td>
<td>4</td>
<td>15</td>
<td>13</td>
<td>15</td>
<td>13</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>BugTrack(EmployeeMaint.jsp)</td>
<td>316</td>
<td>6</td>
<td>5</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>BugTrack(BugRecord.jsp)</td>
<td>336</td>
<td>6</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

TABLE 11: Description of the benchmark programs [37]

improvement in the precision for two benchmark codes ‘BugRecord’ and ‘BookMaint’ w.r.t. the analysis-results in other domains when we consider an abstract representation of initial databases in the powerset of intervals domain. Overall, we achieved an improvement in the precision on an average of 6% in the interval domain, 11% in the octagon, 21% in the polyhedra domain and 7% in the powerset of intervals domain, as compared to the syntax-based approach for the chosen set of benchmark codes. Figure 14 compares all DD-dependency results.

To finally conclude, our observation on the experimental evaluation results indicates that proper tuning of abstract domains from coarse to fine level in precision, perhaps compromising the computational costs, plays a crucial role to meet the analysis-objectives.

10 Case Study

Program slicing [13] is an effective static analysis technique which extracts from programs a subset of statements relevant to a given behavior. This allows engineers to address several software-related problems, including program understanding, debugging, maintenance, testing, parallelization, integration, software measurement, etc. Since the pioneer work of Mark Weiser [23] who introduces the notion
of static program slicing using program dependency graph, different algorithms to compute slice and different slicing variants (e.g. dynamic, conditioned, amorphous) are proposed by tuning them towards specific program analysis aim [26], [27], [62], [63], [64].

Let us apply our proposed dependency refinement in computing a slice of our running example program “Prog”.

Consider the slicing criterion \( \psi = (16, \text{point}) \) where point is an attribute of interest at program point 16. The objective of the backward slicing is to extract only semantically relevant statements affecting the values of point at 16. Let us perform the below steps:

- Construction of syntax-based DOPDG of "Prog".
- Refinement based on our proposed approach.
- Finally, computation of backward slicing w.r.t. \( \psi \) on the refined DOPDG.

According to the first step, the syntax-based DOPDG of “Prog” is depicted in Figure 3 (Section 3). Following our semantics-based refinement analysis in polyhedra abstract domain, we identify the following false dependencies: \( 5 \rightarrow 6, 4 \rightarrow 11, 5 \rightarrow 11, 6 \rightarrow 11, 4 \rightarrow 15, 5 \rightarrow 15, 6 \rightarrow 15, 4 \rightarrow 16, 5 \rightarrow 16, 6 \rightarrow 16 \) and \( 15 \rightarrow 16 \). The refined DOPDG discarding all these false dependencies is depicted in Figure 16(a). Finally, according to the third step, we traverse the DOPDG in a backward direction starting from node 16 considering point as the variable of interest which produces a sub-DOPDG shown in Figure 16(b). The corresponding slice code is shown in Figure 17. Observe that the resultant slice is more precise as it is able to capture statement 15 as semantically irrelevant compared to other syntactic approaches.
0. public class saleOffer
1. public static void main(String[] args) throws SQLException
2. float x = 0.1;
3. float y = 0.05;
4. try { Statement con = DriverManager.getConnection("jdbc mysql: . . . ", "scott", "tiger").createStatement();
5. con.executeUpdate("UPDATE Sales SET purchase_amt = purchase_amt - y * purchase_amt WHERE purchase_amt BETWEEN 1000 AND 3000 ");
6. con.executeUpdate("UPDATE Sales SET purchase_amt = purchase_amt - x * purchase_amt WHERE purchase_amt > 3000 ");
7. con.executeUpdate("UPDATE Sales SET purchase_amt = purchase_amt - delivery_charge ");
8. con.executeUpdate("UPDATE Sales SET point = point + 4 WHERE (purchase_amt + wallet_bal) > 10000 "); catch (Exception e) { ... } }

Fig. 17: Program “Prog”

11 Related Works

Ferrante et al. [14] first introduced the notion of Program Dependency Graph (PDG) aiming program optimization. Since then, PDG is playing crucial roles in a wide range of software-engineering activities, e.g. program slicing [13], code-reuse [16], language-based information flow security analysis [11], [12], [29], code-understanding [17]. Over the time, various forms of dependency graphs for various programming languages are proposed in order to address several language-specific problems. In [65], Zhao proposed a static dependency analysis algorithm for concurrent Java programs based on Multi-thread Dependency Graph (MDG). An MDG consists of a collection of TDGs (Thread Dependency Graphs) each of which represents a single thread. Cheng [66] proposed a PDG for parallel and distributed programs. In [67], the authors introduced the notion of Concurrency Program Dependency Graph (CPDG) to represent concurrent programs written using Unix primitives. It represents various aspects of concurrent programs in a hierarchical fashion. Horwitz et al. [20] introduced System Dependency Graph (SDG) in case of inter-procedural programs. Class Dependency Graph (CIDG) is introduced for Object Oriented Programming (OOP) languages in [21]. Willmor et.al. [22] introduced a variant of program dependence graph, known as Database-Oriented Program Dependence Graph (DOPDG), by considering two additional types of data dependences: Program-Database and Database-Database dependences. The authors observed that, although the generation of used and defined sets of variables is straightforward, but the identification of overlap of database parts by different statements is more challenging. To this aim, they refer to the Condition-Action rules introduced by Baralis and Widom in [35]. The propagation algorithm based on Condition-Action rules predicts how the action of one rule can affect the condition of another. In other words, the analysis checks whether the condition sees any data inserted or deleted or modified due to the action.

Mastroeni and Zanardini [34] first introduced the notion of semantic data independencies following the Abstract Interpretation framework at expression-level. This leads to generate more precise semantics-based PDGs by removing false data dependencies w.r.t. the traditional syntactic PDGs. Our previous attempts to refine dependencies and hence more precise code analysis for database programs are reported in [27], [36], [38]. [38] applied predicate transformer (weakest precondition) to apply on dependency tree among a series of attribute-defining statements, whereas [27], [36] formalized the semantics for dependency refinement in a simple setting following the Abstract Interpretation as an initial attempt.

The authors in [68] and [69] addressed a closely related problem, known as query containment problem, which checks whether, for every database, the result of one query is a subset of the result of another query. For instance, a query Q1 is said to be contained in a query Q2 if and only if the result of applying Q1 to any database D is contained in the result of applying Q2 to the same database D. Formally, a query Q1 is said to be contained in a query Q2, denoted Q1 ⊊ Q2 ⇐⇒ ∀D Q1(D) ⊆ Q2(D) and Q1 ≡ Q2 ⇐⇒ Q1 ⊊ Q2 ∧ Q2 ⊊ Q1, where Q(D) represents the result of query Q on database D. The computational complexity of conjunctive query containment is NP-complete [68]. Query containment is useful for the various purposes of query optimization, detecting independency of queries from database updates, rewriting queries using views, etc. As dependency computations of database applications consider DML commands (INSERT, UPDATE, DELETE), the solutions proposed in [68], [69] for only conjunctive queries is, therefore, unable to provide a complete solution in our case which involves both write-write and write-read operations.

The authors in [70] addressed an undecidable problem which aims to identify all possible values that may occur as results of string expressions. Few interesting applications of the solution, among many others, include static analysis of validity of dynamically generated XML documents in the JWIG extension of Java, static syntax checking of dynamically generated queries in database programs. The authors proposed a static analysis technique for extracting context-free grammar from a given program and applied a variant of the Mohri-Nederhof approximation algorithm to approximate the possible values of string expressions in Java programs. A static analysis framework is proposed in [71] to automatically identify possible SQL injection attacks, SQL query performance optimization and data integrity violations in database programs. For this purpose, the framework adapts data and control flow analysis of traditional optimizing compilers techniques by leveraging understanding of data access APIs. [72] proposed a sound static analysis technique for verifying the correctness of
dynamically generated SQL query strings in database applications. The technique is based on a combination of automata-theoretic techniques and a variant of the context-free language reachability algorithm. A new framework [73] is proposed for context-sensitive program analysis. The concept of deductive database technology is used here to create a higher abstraction for this cloning-based approach to context sensitivity. The framework allows users to express whole-program analysis succinctly with a small number of Datalog rules that operate on a cloned call graph. In [74], the authors proposed the constraint coverage criteria and the column coverage criteria for testing the specification of integrity constraints in a relational database schema. They expressed integrity constraints as predicates with constraint coverage, whereas they generated test requirements with the column coverage for checking integrity constraints.

12 Conclusion and Future Works

Dependency analysis of database programs plays a crucial role in different fields of software engineering. Some applications among many others include Program Slicing, Language-based Information Flow Security Analysis, Data Provenance, Concurrent System Modeling, Materialization View Creation. Although syntax-based dependency computation is straightforward, its semantics-based refinement is quite challenging when considering attributes’ values in possible database instances. This paper proposes a novel approach to compute semantics-based independencies among database statements, based on the Abstract Interpretation framework. This steers construction of semantic-based DOPDGs with more precise set of dependencies. Most importantly, this serves as a powerful basis to give solution even in case of undecidable scenario when no initial database state is provided. The comparative study among various approaches and various abstract domains in terms of precision and efficiency clearly indicates that a trade-off in choosing appropriate abstract domains or their combination is very crucial to meet the objectives. There are many application areas where false dependence information could lead to huge financial loss while proving crucial properties of software products. Information flow security analysis of critical softwares is one such example. In such case, precision dominates over the analysis cost and a choice of stronger abstract domain, e.g. polyhedra domain, may be a good choice. On the other hand, when development speed is an important factor, choice of weakly relational or even non-relational abstract domain may be a wise decision. Experimental evaluation on the benchmarks set reports a precision improvement in the range of 6% - 21% under various levels of abstractions. This proves that the approach may impact significantly when to deal with large-scale complex software systems involving huge variables set and millions of lines of codes.

Some interesting future research scopes identified in this direction are designing suitable reduced products on multiple abstract domains, designing new ad-hoc abstract domains in the context of database states, extending the analysis to a distributed scenario with multiple transactions and heterogeneous database systems. As our future work, we shall extend our tool SemDDA to support some popular languages, such as C, Python, Java, in its next release. Besides, we shall also consider string data-type [75] along with numerical attributes.

Acknowledgement

This work is partially supported by the research grant (SB/FTP/ETA-315/2013) from the Science & Engineering Research Board (SERB), Department of science and Technology, Government of India, and by CINI Cybersecurity National Laboratory within the project FileriSicura, Italy.

References

Angshuman Jana is currently working toward the Ph.D. degree in the Department of Computer Science and Engineering, IIT Patna, India. He received the B.Tech. degree from MAKAUT, India, in 2011 and the M.Tech. degree from NIT Durgapur, India, in 2013, both in the discipline of computer science and engineering. His primary research focus is on Program Analysis and Verification using Formal Methods.

Raju Halder is an assistant professor in the Department of Computer Science and Engineering, IIT Patna, India. He received the doctoral degree from Università Ca’ Foscari Venezia, Italy, in 2012. Before joining IIT Patna, he served as a post doctoral researcher at Macquarie University, Australia. He worked with the Robotics team at HASLab (University of Minho), Portugal, in 2016. Prior to his Ph.D., he had also worked as an associate system engineer at IBM India Pvt. Ltd. during 2007-2008. His areas of research interests include Program Analysis and Verification, Formal Methods, Robotics, Database Languages, Data Privacy and Security, etc.

Kalahasti Venkata Abhishekh is currently working as Software Development Engineer at Amazon Development Center, Hyderabad, India. He is a part of Value Added Services team whose primary business focus is in United States. He had recently graduated from IIT Patna in Computer Science and Engineering in 2017. He loves to code and develop new software applications. His areas of research interests include Artificial Intelligence and Machine Learning.