Extending Abstract Interpretation to Dependency Analysis of Database Applications

Angshuman Jana¹, Raju Halder¹, K. V. Abhishekh¹, S. D. Ganni², and Agostino Cortesi²
¹Indian Institute of Technology Patna, India
(ajana.pcs13, halder, kalahasti.cs13, ganni.cs13}@iitp.ac.in
²Università Ca’ Foscari Venezia, Italy
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 undecided 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 been longly 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 $\text{Sbonus}$ and by an additional special bonus amount $\text{Sbonus}$ only for aged employees. Observe that the syntactic presence of ‘$\text{sal}$’ 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 ‘$\text{sal}$’ referred in the “WHERE” clauses of $Q_1$ and $Q_3$ do not overlap with each other and this results in an independency between $Q_1$ and $Q_3$. 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 Zanardi [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.

Willmör’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 $\text{sal}$-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 $Q_5$ 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:

```
Start;
Q_0: Connection c = DriverManager.getConnection[......];
Q_1: UPDATE emp SET sal = sal + Sbonus 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 + Sbonus;
Q_5: SELECT AVG(sal) FROM emp;
Stop;
```

Fig. 1: An Introductory Example
improvements in this paper compared to [36] are: (i) only polyherdra abstract domain. To be specific, the preliminary theoretical proposal in [36] considered a set of benchmark codes. We present in section 10 a case study on database code slicing, witnessing an improvement on the evolution of syntax-based dependency computation. In section 8. Section 9 describes the experimental results on the soundness of our approach is proved in section 7. We show, in section 7, how the proposed approach effectively identifies false DD-dependencies in ’Prog’.

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 will 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 dependency computation framework. We discuss in section 11 the current state-of-the-art in the literature. Finally, section 12 concludes our work.

1. This work is a revised and extended version of [36].
2. Available at: https://github.com/angshumanjana/SemDDA.
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 \( Q_1 \rightarrow Q_2 \)) 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 \( C \), \( V_d \) and \( V_a \) be the sets of statements, application-variables and database-attributes in database programs. Let

\[ V = V_d \cup V_a \] where \( V_d \cap V_a = \emptyset \). The construction of syntax-based DOPDG can be formalized based on the two following functions:

\[ \text{USE}: C \rightarrow \phi(V) \]
\[ \text{DEF}: C \rightarrow \phi(V) \]

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.

Fig. 2: Database Code Snippet “Prog”

Fig. 3: Pure Syntax-based DOPDG (★ denotes attribute \( purchase\_amt \)) of “Prog”

Example 1. Consider our running example “Prog” depicted in Figures 2. The control dependencies \( 1 \rightarrow 2, 1 \rightarrow 3, 1 \rightarrow 4 \), 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:
\[ \text{DEF}(2) = [x] \quad \text{DEF}(3) = [y] \]
\[ \text{DEF}(4) = [\text{purchase}\_\text{amt}, \text{delivery}\_\text{charge}, \text{cust}\_\text{name}, \text{wallet}\_\text{bal}, \text{point}] \]
\[ \text{DEF}(5) = [\text{purchase}\_\text{amt}] \quad \text{USE}(5) = [\text{purchase}\_\text{amt}, y] \]
\[ \text{DEF}(6) = [\text{purchase}\_\text{amt}] \quad \text{USE}(6) = [\text{purchase}\_\text{amt}, x] \]
\[ \text{DEF}(7) = [\text{purchase}\_\text{amt}] \]
\[ \text{USE}(7) = [\text{purchase}\_\text{amt}, \text{delivery}\_\text{charge}] \]
\[ \text{USE}(11) = [\text{purchase}\_\text{amt}, \text{cust}\_\text{name}] \]
\[ \text{DEF}(15) = [\text{point}] \]
\[ \text{USE}(15) = [\text{purchase}\_\text{amt}, \text{wallet}\_\text{bal}, \text{point}] \]
\[ \text{USE}(16) = [\text{purchase}\_\text{amt}, \text{wallet}\_\text{bal}, \text{point}] \]

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 \(\rightarrow\) 5, 4 \(\rightarrow\) 6, 4 \(\rightarrow\) 7, 4 \(\rightarrow\) 11, 4 \(\rightarrow\) 15, 4 \(\rightarrow\) 16, 5 \(\rightarrow\) 6, 5 \(\rightarrow\) 7, 5 \(\rightarrow\) 11, 5 \(\rightarrow\) 15, 5 \(\rightarrow\) 16, 6 \(\rightarrow\) 7, 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, 15 \(\rightarrow\) 16

- **PD-dependencies for x and y**: 2 \(\rightarrow\) 6, 3 \(\rightarrow\) 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 \(\rightarrow\) 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:

\[ \text{USE} : C \rightarrow \wp(V \times L) \quad (3) \]
\[ \text{DEF} : C \rightarrow \wp(V \times L) \quad (4) \]

where \( L = \{\bullet, \triangle\} \) 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 \( \triangle \) 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:

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

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

- **DD-dependencies for purchase_amt**: 4 \(\rightarrow\) 5, 4 \(\rightarrow\) 6, 4 \(\rightarrow\) 7, 5 \(\rightarrow\) 6, 5 \(\rightarrow\) 7, 6 \(\rightarrow\) 7, 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, 15 \(\rightarrow\) 16

- **PD-dependencies for x and y**: 2 \(\rightarrow\) 6, 3 \(\rightarrow\) 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 \(\rightarrow\) 11, 5 \(\rightarrow\) 11, 6 \(\rightarrow\) 11, 4 \(\rightarrow\) 15, 5 \(\rightarrow\) 15, 6 \(\rightarrow\) 15, 4 \(\rightarrow\) 16, 5 \(\rightarrow\) 16 and 6 \(\rightarrow\) 16 for purchase_amt. Observe that the DD-dependency 4 \(\rightarrow\) 11 exists for cust_name and dependencies 4 \(\rightarrow\) 15, 4 \(\rightarrow\) 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 \(\rightarrow\) 6 for purchase_amt and 15 \(\rightarrow\) 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
The refined set of data dependencies are:

$\text{Example 3.}$ 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:

- $E^5_{\text{cond}} \rightarrow \Pi_{\text{purchase\_amt}}(\sigma_{\text{purchase\_amt} \geq 1000} \land \sigma_{\text{purchase\_amt} < 3000} \text{ Sales})$
- $E^5_{\text{act}} \rightarrow \epsilon[\text{purchase\_amt}' = \text{purchase\_amt} - 0.05 \times \text{purchase\_amt}] (\sigma_{\text{purchase\_amt} \geq 1000} \land \sigma_{\text{purchase\_amt} < 3000} \text{ Sales})$
- $E^6_{\text{cond}} \rightarrow \Pi_{\text{purchase\_amt}}(\sigma_{\text{purchase\_amt} > 3000} \text{ Sales})$
- $E^6_{\text{act}} \rightarrow \epsilon[\text{purchase\_amt}' = \text{purchase\_amt} - 0.1 \times \text{purchase\_amt}] (\sigma_{\text{purchase\_amt} > 3000} \text{ Sales})$

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 $E_{\text{act}}$ and $(\text{purchase\_amt} > 3000)$ in $E_{\text{cond}}$ are contradictory – meaning that $E_{\text{act}}$ operates on a part of data which is not accessed by $E_{\text{cond}}$. In other words, the action $E_{\text{act}}$ does not affect the condition $E_{\text{cond}}$. Therefore, DD-dependency $5 \to 6$ is false. Similarly we can also identify another false DD-dependency $15 \to 16$. The refined set of data dependencies are:

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

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 $4 \to 11$, $4 \to 15$, $4 \to 16$,
5 \rightarrow 11, \quad 5 \rightarrow 15, \quad 5 \rightarrow 16, \quad 6 \rightarrow 11, \quad 6 \rightarrow 15\text{ and } 6 \rightarrow 16 \text{ in “Prog” due to the presence of multiple definitions of } purchase\_amt \text{ by the statements 5, 6 and 7 in sequence. Moreover, this approach incurs a high computational overhead w.r.t. program size. Observe that the algorithm combining from sections 3.2 and 3.3 will identify a set of false dependencies which is same as the union of the results obtained from both of the algorithms when applied individually.}

The subsequent sections are dedicated to semantic-based DD-dependency (in concrete domain) and DD-independency (in abstract domain) computation of database programs.

4 Formal Syntax and Concrete Semantics of Database Query Languages

In this section, we recall from [39] the formal syntax and concrete semantics of database query languages.

Abstract syntax of database statement is denoted by \( \langle A, \phi \rangle \) where \( A \) represents an action-part and \( \phi \) represents a conditional-part. For instance, the query “UPDATE \( t \) SET \( sal=sal+100 \) WHERE \( age \geq 35 \)” is denoted by \( \langle A, \phi \rangle \) where \( A \) represents “UPDATE \( t \) SET \( sal=sal+100 \)” and \( \phi \) represents “\( age \geq 35 \)”.

Table 1 depicts the syntactic sets and the abstract syntax of database statements. The SQL clauses GROUP BY, ORDER BY, DISTINCT/ALL, and the aggregate functions are denoted by different functions \( g() \), \( f() \), \( r() \), and \( h() \) respectively. A SQL action \( A \) is either “SELECT” or “UPDATE” or “DELETE” or “INSERT”. For example, the abstract syntax of the query above is denoted by

\[
\langle \text{UPDATE}(\overline{v}_d, \overline{e}), \phi \rangle
\]

where \( \phi = (age \geq 35) \) and \( \overline{v}_d = \langle sal \rangle \) and \( \overline{e} = \langle sal + 100 \rangle \).

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

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 \( \text{attr}(t) = \{ a_1, a_2, \ldots, a_i \} \). So, \( t \subseteq D_1 \times D_2 \times \ldots \times D_T \) 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 \( a_i \in \text{attr}(t), \rho_t(a_i) = (\pi_t(l_i)) \mid l_i \in t \) where \( \pi_t \) is the projection operator and \( l_i \) 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 \( \Sigma_{dba} \) be the set of states for the database language under consideration, defined by \( \Sigma_{dba} = \mathcal{E}_{dba} \times \mathcal{E}_a \) where \( \mathcal{E}_{dba} \) and \( \mathcal{E}_a \) denote the set of all database environments and the set of all application environments respectively. Therefore, a state \( \rho \in \Sigma_{dba} \) is denoted by a tuple \( (\rho_d, \rho_a) \) where \( \rho_d \in \mathcal{E}_{dba} \) and \( \rho_a \in \mathcal{E}_a \). The transition relation

\[
\mathcal{T}_{dba} : (C \times \Sigma_{dba}) \mapsto \phi(\Sigma_{dba})
\]

specifies which successor states \( (\rho_d', \rho_a') \in \Sigma_{dba} \) can follow when a statement \( c \in C \) executes on state \( (\rho_d, \rho_a) \in \Sigma_{dba} \). 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:

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

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

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

\[
\mathcal{T}_{dba} Q_{ upd } (\rho_d, \rho_a)
\]

where \( \phi = (age \geq 35) \) and \( \overline{v}_d = \langle sal \rangle \) and \( \overline{e} = \langle sal + 100 \rangle \).

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

\[
\mathcal{T}_{dba}(\langle\text{UPDATE}(sal), (sal + 100)\rangle)(\rho_d, \rho_a)
\]

is Absorbing \( \phi = (age \geq 35) \)
<table>
<thead>
<tr>
<th>Constants:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k \in \mathbb{R}$</td>
</tr>
<tr>
<td>Set of Numerical Constants</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_a \in \mathbb{V}_a$</td>
</tr>
<tr>
<td>Set of Application Variables</td>
</tr>
<tr>
<td>$v_a ::= x</td>
</tr>
<tr>
<td>$v_d \in \mathbb{V}_d$</td>
</tr>
<tr>
<td>Set of Database Attributes</td>
</tr>
<tr>
<td>$v_d ::= a_1</td>
</tr>
<tr>
<td>$\mathbb{V} ::= \mathbb{V}_a \cup \mathbb{V}_d$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expressions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e \in \mathbb{E}$</td>
</tr>
<tr>
<td>Set of Arithmetic Expressions</td>
</tr>
<tr>
<td>$e ::= k</td>
</tr>
<tr>
<td>where $op_a \in {+,-}$ and $op_b \in {+,-,\times,/}$</td>
</tr>
<tr>
<td>$b \in \mathbb{B}$</td>
</tr>
<tr>
<td>Set of Boolean Expressions</td>
</tr>
<tr>
<td>$b ::= \text{true}</td>
</tr>
<tr>
<td>where $op_r \in {\leq,\geq,=,&gt;,\ldots}$ and $\oplus \in {\lor,\land}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL Pre-conditions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau \in \mathbb{T}$</td>
</tr>
<tr>
<td>Set of Terms</td>
</tr>
<tr>
<td>$\tau ::= k</td>
</tr>
<tr>
<td>where $f_n$ is an $n$-ary function.</td>
</tr>
<tr>
<td>$a_f \in \mathbb{A}_f$</td>
</tr>
<tr>
<td>Set of Atomic Formulas</td>
</tr>
<tr>
<td>$a_f ::= R_n(\tau_1, \tau_2, \ldots, \tau_n)</td>
</tr>
<tr>
<td>where $R_n(\tau_1, \tau_2, \ldots, \tau_n) \in {\text{true}, \text{false}}$</td>
</tr>
<tr>
<td>$\phi \in \mathbb{W}$</td>
</tr>
<tr>
<td>Set of Pre-conditions</td>
</tr>
<tr>
<td>$\phi ::= a_f</td>
</tr>
<tr>
<td>where $\land \in {\lor,\land}$ and $\varphi \in {\lor,\land}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SQL Functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(\bar{e}) ::= \text{GROUP \ BY}(\bar{e})</td>
</tr>
<tr>
<td>where $\bar{e} = (e_1, \ldots, e_n</td>
</tr>
<tr>
<td>$r ::= \text{DISTINCT}</td>
</tr>
<tr>
<td>$s ::= \text{AVG}</td>
</tr>
<tr>
<td>$h(\bar{e}) ::= s \circ r(\bar{e})$</td>
</tr>
<tr>
<td>$h(*) ::= \text{COUNT}(\ast)$</td>
</tr>
<tr>
<td>where $\ast$ represents the list of all database attributes denoted by $\mathbb{V}_d$.</td>
</tr>
<tr>
<td>$h(\bar{x}) ::= \langle h_1(x_1), \ldots, h_n(x_n) \rangle$</td>
</tr>
<tr>
<td>where $h = \langle h_1, \ldots, h_n \rangle$ and $x = (x_1, \ldots, x_n</td>
</tr>
<tr>
<td>$f(\bar{e}) ::= \text{ORDER \ BY \ ASC}(\bar{e})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commands:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q \in \mathbb{Q}$</td>
</tr>
<tr>
<td>Set of SQL Statements</td>
</tr>
<tr>
<td>$Q ::= Q_{\text{set}}</td>
</tr>
<tr>
<td>$Q_{\text{set}} ::= \langle A_{\text{set}}, \phi \rangle$</td>
</tr>
<tr>
<td>$\langle \text{SELECT}(f(\bar{e})), r(h(\bar{x})), \phi_2, g(\bar{e}), \phi_1 \rangle$</td>
</tr>
<tr>
<td>$Q_{\text{upd}} ::= \langle A_{\text{upd}}, \phi \rangle$</td>
</tr>
<tr>
<td>$\langle \text{UPDATE}(\bar{v}_d, \bar{e}, \phi \rangle$</td>
</tr>
<tr>
<td>$Q_{\text{ins}} ::= \langle A_{\text{ins}}, \phi \rangle$</td>
</tr>
<tr>
<td>$\langle \text{INSERT}(\bar{v}_d, \bar{e}, false) \rangle$</td>
</tr>
<tr>
<td>$Q_{\text{del}} ::= \langle A_{\text{del}}, \phi \rangle$</td>
</tr>
<tr>
<td>$\langle \text{DELETE}(\bar{v}_d), \phi \rangle$</td>
</tr>
<tr>
<td>$c \in \mathbb{C}$</td>
</tr>
<tr>
<td>Set of Commands</td>
</tr>
<tr>
<td>$c ::= \text{skip}</td>
</tr>
<tr>
<td>$\text{if } b \text{ then } c_1 \text{ else } c_2 \text{ endif}$</td>
</tr>
<tr>
<td>$\text{while } b \text{ do } c \text{ done}$</td>
</tr>
<tr>
<td>$\mathcal{P} ::= c</td>
</tr>
</tbody>
</table>

Program

<table>
<thead>
<tr>
<th>TABLE 1: Abstract Syntax of Programs embedding SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Table 1]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 2: Database before and after the update operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Table 2]</td>
</tr>
</tbody>
</table>

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)). $E \parallel$ is the semantic function for arithmetic expression which maps “$sal + 100”$ to a list of values $(1600, 2600, 3100)$ on the table environment $p_1(age \geq 35)$. The notation $\leftarrow$ denotes a substitution by new values. Observe that the substitution of ‘$sal’ by the list of values in $p_{1(age \geq 35)}$ results in a new table environment $p_r$ (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 \((\rho_1, \rho_2)\) 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 \( e = x^2 + 4w \mod 2 + z'' \). Although in this expression \( e \) 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 \( Q = (A, \phi) \) 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 \( T_{\text{use}}[Q](\rho_1, \rho_2) = (\rho_1, \rho_2) \).

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

\[
A_{\text{def}}(Q, t) = \Delta(\rho_1(\bar{z}), \rho_1(\bar{z})) \tag{6}
\]

\[
A_{\text{use}}(Q, t) = \rho_{1|\phi}(\bar{x}) \cup \rho_{1|\phi}(\bar{y}) \tag{7}
\]

where

- \( t \perp \phi \) : Set of tuples in table \( t \) which satisfies the condition-part \( \phi \).
- \( \rho_{1|\phi}(\bar{x}) \) : Values of \( \bar{x} \) in \( t \perp \phi \).
- \( \rho_{1|\phi}(\bar{y}) \) : Values of \( \bar{y} \) in \( t \perp \phi \).
- \( \Delta \) : 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 1 shown in Table 2(a) and the following update statement:

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

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

\[
A_{\text{use}}(Q_{\text{upd}}, t) = \rho_{1|\phi}(age \geq 35)(\text{sal}) \cup \rho_{1|\phi}(age \geq 35)(age)
\]

\[
A_{\text{def}}(Q_{\text{upd}}, t) = \Delta(\rho_{t}(\text{sal}), \rho_{t}(\text{sal}))
\]

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.

Given two database statements \( Q_1 = \langle A_1, \phi_1 \rangle \) and \( Q_2 = \langle A_2, \phi_2 \rangle \) such that \( \text{target}(Q_1) = t \) and \( T_{\text{use}}[Q_2](\rho_1, \rho_2) = (\rho_1, \rho_2) \) and \( \text{target}(Q_2) = t' \). Following the equations 6 and 7, we can compute the defined part of \( t \) by \( Q_1 \) as \( A_{\text{def}}(Q_1, t) \) and used-part of \( t' \) by \( Q_2 \) as \( A_{\text{use}}(Q_2, t') \). Therefore, we can say \( Q_2 \) is DD-dependent on \( Q_1 \) when \( A_{\text{def}}(Q_1, t) \) and \( A_{\text{use}}(Q_2, t') \) overlap with each other, i.e. \( A_{\text{use}}(Q_2, t') \cap A_{\text{def}}(Q_1, t) \neq \emptyset \). Observe that \( Q_1 \) is either UPDATE, INSERT and DELETE statement which defines the database. This is defined in Definition 5.

**Definition 5** (Semantics-based DD-dependency [27]). A SQL statement \( Q_2 = \langle A_2, \phi_2 \rangle \) with \( \text{target}(Q_2) = t' \) is DD-dependent on another SQL statement \( Q_1 = \langle A_1, \phi_1 \rangle \) with \( \text{target}(Q_1) = t \) (denoted \( Q_1 \rightarrow_{Q_2} Q_2 \) if \( Q_1 \in \{Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{upd}}\} \) and \( T_{\text{use}}[Q_2](\rho_1, \rho_2) = (\rho_{t'}, \rho_2) \) and the overlapping \( \Upsilon = A_{\text{use}}(Q_2, t') \cap A_{\text{def}}(Q_1, t) \).

When an initial database instance is unknown, due to infiniteness of the concrete domains, the computation of concrete semantics of database programs and hence \( A_{\text{use}}, A_{\text{def}} \) and \( \Upsilon \) become undecidable problem. Nevertheless, in case of finite large scale databases, these semantics-based dependency computations also incur in high computational overhead. To ameliorate this performance bottleneck, we apply the Abstract Interpretation theory [9] to compute abstract semantics of database languages, in a decidable way, as a sound approximation of its concrete counterparts.

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
between concrete and abstract semantics domains is usually established by a Galois Connection 4:

**Definition 6 (Galois Connections [9])**. Consider two partial orders \((D, \preceq)\) and \((\overline{D}, \sqsubseteq)\) 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 \(\left\langle (D, \preceq), \alpha, \gamma, (\overline{D}, \sqsubseteq) \right\rangle\) or \((D, \preceq) \Rightarrow (\overline{D}, \sqsubseteq)\) where \(\alpha: D \rightarrow \overline{D}\) and \(\gamma: \overline{D} \rightarrow D\) holds iff:

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

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

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 values and do not preserve any relation among program variables. Non-relational abstract domains are exemplified in Figure 6.

**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.

5. Notice that for some abstract domain the abstraction function may not exist.

---

**Fig. 6: Abstractions of SP by Sign (left) and Interval Properties (right)**

**Fig. 7: Abstractions of SP in Octagon (left) and Polyhedra Domains (right)**

**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 \(t \in \mathfrak{o}(D)\) 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 \(t_i\) 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 \(\left\langle (\mathfrak{o}(D), \subseteq), \alpha, \gamma, (\overline{D}, \sqsubseteq) \right\rangle\). An element \(\overline{t} \in \overline{D}\) is said to be a sound abstraction of the concrete table \(t\) if for all tuples \(l \in t\), \(l \in \gamma(\overline{t})\).

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

**Definition 9 (Abstract Database States).** An abstract database \(\overline{\mathcal{D}}\) is a set of abstract tables \(\{\overline{t}_i | i \in I\}\) for a given set of indexes \(I\). An abstract database environment is defined as a function \(\rho\overline{\mathcal{D}}\) whose domain is \(I\), such that for \(i \in I\), \(\rho\overline{\mathcal{D}}(i) = \overline{t}_i\).

**Definition 10 (Abstract States).** An abstract state \(\overline{\sigma} \in \Sigma_{\mathcal{D}ba}\) for database applications is defined as a tuple \((\rho\overline{\mathcal{D}}, \rho\overline{\sigma})\) where \(\rho\overline{\mathcal{D}} \in \Sigma_{\mathcal{D}ba}\) and \(\rho\overline{\sigma} \in \overline{\Sigma}_a\) are an abstract database environment and an abstract application environment 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 \(\mathcal{T}_{\mathcal{D}ba}\) (defined in equation 5 of section 4):

\[
\mathcal{T}_{\mathcal{D}ba} : C \times \Sigma_{\mathcal{D}ba} \rightarrow \Sigma_{\mathcal{D}ba}
\]

which specifies the successor abstract state \((\rho\overline{\mathcal{D}}, \rho\overline{\sigma}) \in \Sigma_{\mathcal{D}ba}\) when a statement \(c \in C\) executes on an abstract state.
Since our objective is to compute semantics-based DD-independencies, it is important to identify database-parts (identified by the condition $\phi$) before and after performing the action $A$. With this objective, unlike equation 8 which results in a single abstract state $\rho$, we define a variant of the abstract transition relation as follows:

$$\mathcal{F}_{\text{dep}} : \mathbb{C} \times \Sigma_{\text{dba}} \rightarrow (\Sigma_{\text{dba}} \times \Sigma_{\text{dba}})$$

which results in a three-tuple $(\rho_\uparrow, \rho_\downarrow, \rho)$ of abstract database states, where $\rho_\uparrow, \rho_\downarrow, \rho \in \Sigma_{\text{dba}}$. The first component $\rho_\uparrow$ represents an abstract database state which does not satisfy $\phi$, whereas the second component $\rho_\downarrow$ represents an abstract database state which satisfies $\phi$. Observe that an abstract database-part which may or may not satisfy $\phi$ (due to abstraction) will be included in both $\rho_\uparrow$ and $\rho_\downarrow$. The third component $\rho$ is obtained after performing an action $A$ on $\rho$. 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 $L = (\mathbb{R}, \subseteq, 0, R, \cap, \cup)$ be a concrete lattice of the powerset of numerical values $\mathbb{R}$. Let $I = \{[l, h] | l \in \mathbb{R} \cup [-\infty, h) \cup [l, \infty), 1 \leq h \} \cup \perp$ be the abstract domain of intervals forming an abstract lattice $L = (L, \subseteq, 1, \perp)$, such that:

- $[l_1, h_1] \subseteq [l_2, h_2] \iff l_2 \leq l_1 \land h_1 \geq h_2$
- $[l_1, h_1] \cap [l_2, h_2] = [\max(l_1, l_2), \min(h_1, h_2)]$
- $[l_1, h_1] \cup [l_2, h_2] = [\min(l_1, l_2), \max(h_1, h_2)]$

The correspondence between $L$ and $L_a$ is formalized as the Galois connection $(L, \alpha_L, \gamma_L, L_a)$ where $\forall S \in \mathbb{R}$ and $\forall S \subseteq \mathbb{R}$:

$$\alpha_L(S) = \begin{cases} \perp & \text{if } S = \emptyset \\ [l, h] & \text{if } \min(S) = l \land \max(S) = h \\ [-\infty, h) & \text{if } \mathbb{R}(S) \land \max(S) = h \\ [l, +\infty) & \text{if } \min(S) = l \land \mathbb{R}(S) \\ [+\infty, -\infty) & \text{if } \mathbb{R}(S) \land \mathbb{R}(S) \\ \end{cases}$$

The pictorial representation of the Galois connections $(L, \alpha_L, \gamma_L, L_a)$ is shown in Figure 9.

### Abstract Semantics of Imperative Language in Interval: A Quick Tour

Consider the set of concrete states $\Sigma : \mathbb{V} \rightarrow \mathbb{R}$ 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 $\mathcal{S} : (E \cup B) \rightarrow (\Sigma \rightarrow \mathbb{R} \cup \{true, false\})$ for expressions evaluation, $\mathcal{S}_f : B \rightarrow (\rho(\Sigma) \rightarrow \rho(\Sigma))$ for state-filtering based on boolean satisfiability and $\mathcal{S}_c : C \rightarrow (\rho(\Sigma) \rightarrow \rho(\Sigma))$ 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 $\Sigma : \mathbb{V} \rightarrow I$ which respects the Galois Connection, i.e. $\forall \rho \in \Sigma, \forall \rho \in \Sigma : a(\rho) \subseteq \rho \iff \rho \in \gamma(\rho)$.

The corresponding sound abstract semantics function $\mathcal{S}_c : (E \cup B) \rightarrow (\Sigma \rightarrow \Sigma)$ for expression evaluation where $\mathcal{T}_B$ denotes “may be true or may be false”, is defined as:

$$\mathcal{S}_c[k] = (\rho, [k]) \iff \rho \in \Sigma$$

$$\mathcal{S}_c[x] = (\rho, \rho(x)) \iff \rho \in \Sigma$$

$$\mathcal{S}_c[e_1 + e_2] = ((\rho, \rho_1, \rho_2) \in \mathcal{S}_c[e_1, \mathcal{S}_c[e_2]]) \Rightarrow ((\rho, \rho_1, \rho_2)) \in \mathcal{S}_c[e_1 + e_2]$$

Examples of sound abstract arithmetic and relational operations $\mathbb{R}$ and $\mathbb{R}$ respectively in the domain of intervals are:

- $[l_1, h_1] + [l_2, h_2] = [l_1 + l_2, h_1 + h_2]$
- $[l_1, h_1] \times [l_2, h_2] = [\min(l_1 \times l_2, l_1 \times h_2, h_1 \times l_2, h_1 \times h_2), \max(l_1 \times l_2, l_1 \times h_2, h_1 \times l_2, h_1 \times h_2)]$

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

Similarly, the abstract semantics functions $\mathcal{S}_f : B \rightarrow (\Sigma \rightarrow \Sigma)$ for abstract state-filtering and $\mathcal{S}_f : C \rightarrow (\Sigma \rightarrow \Sigma)$ for commands are:

$$\mathcal{S}_f[x \leq k] = (\rho, \rho(x) \leftarrow [l, \min(h, k)]) \iff \rho \in \Sigma, \rho(x) = [l, h], l < k$$

$$\mathcal{S}_f[x = k] = (\rho, \rho(x) \leftarrow [l, \min(h, k)]) \iff \rho \in \Sigma, \rho(x) = [l, h], l < k$$

$$\mathcal{S}_f[x \neq k] = (\rho, \rho(x) \leftarrow [l, \min(h, k)]) \iff \rho \in \Sigma, \rho(x) = [l, h], l < k$$

$$\mathcal{S}_f[x < k] = (\rho, \rho(x) \leftarrow [l, \min(h, k)]) \iff \rho \in \Sigma, \rho(x) = [l, h], l < k$$

$$\mathcal{S}_f[x > k] = (\rho, \rho(x) \leftarrow [l, \min(h, k)]) \iff \rho \in \Sigma, \rho(x) = [l, h], l < k$$

$$\mathcal{S}_f[x \leq k] = (\rho, \rho(x) \leftarrow [l, \min(h, k)]) \iff \rho \in \Sigma, \rho(x) = [l, h], l < k$$
Fig. 10: Concrete Denotational Semantics of simple Imperative Language

\[ \mathcal{F}[k] = \{(p, k) \mid p \in \Sigma\} \]
\[ \mathcal{F}[x] = \{(p, v) \mid p \in \Sigma, \rho(x) = v\} \]
\[ \mathcal{F}[c_1 + c_2] = \{(p, v_1 + v_2) \mid (p, v_1) \in \mathcal{F}[c_1], (p, v_2) \in \mathcal{F}[c_2], \}
\[ \quad \rho \in \{+, -, \lor \land \in \{0, 1\} \land v_2 \neq 0\} \]
\[ \mathcal{F}[c_1 \cdot c_2] = \{(p, v_1 \cdot v_2) \mid (p, v_1) \in \mathcal{F}[c_1], (p, v_2) \in \mathcal{F}[c_2], \}
\[ \quad \rho \in \{\lt, \leq, \gg, =\} \]
\[ \mathcal{F}[-b] = \{(p, -v) \mid (p, v) \in \mathcal{F}[b]\} \]
\[ \mathcal{F}[b_1 \cdot b_2] = \{(p, v_1 \cdot v_2) \mid (p, v_1) \in \mathcal{F}[b_1], (p, v_2) \in \mathcal{F}[b_2], \}
\[ \quad \rho \in \{\lor, \land\} \]
\[ \mathcal{F}[\text{if } b \text{ then } c_1 \text{ else } c_2] = \{(p, \rho') \mid (p, \rho) \in \mathcal{F}[b], (p, \rho') \in \mathcal{F}[c_1]\}
\[ \quad \cup \]
\[ \quad (p, \rho'') \mid (p, \rho) \in \mathcal{F}[-b], (p, \rho'') \in \mathcal{F}[c_2]\} \]
\[ \mathcal{F}[\text{while } b \text{ do } c] = \{(p, \rho) \mid (p, \rho') \in \mathcal{F}[\neg b] \circ \mathcal{F}[b] \circ \mathcal{F}[c]\} \]

where \( \cup \) denotes component-wise join operation in the abstract lattice \( \mathcal{L}_a \).

Defining Abstract Semantics of Database Language in Interval Domain.

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

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

\[ \mathcal{F}_\text{dba}(\langle A, \phi \rangle) \bar{p} = \mathcal{F}_\text{dba}(\langle A, \phi \rangle)(\rho_T, \rho_F) \]
\[ = \mathcal{F}_\text{dba}(\langle A, \phi \rangle)(\rho_T, \rho_F) \]
\[ = \mathcal{F}_\text{dba}(\langle A, \phi \rangle)(\rho_T, \rho_F) \]
\[ \quad \text{where } t = \text{target}(A, \phi) \text{ and } \exists \bar{a} \in \bar{a}: t \in \gamma(l) \]
\[ = \mathcal{F}_\text{dba}(\langle A \rangle)(\rho_T \lor \rho_F, \rho_F) \cup (\rho_T \lor \rho_F, \rho_F) \]
\[ = (\rho_T \lor \rho_F, \rho_F) \cup (\rho_T \lor \rho_F, \rho_F) \]

Example 7. Consider the statement \( c := \text{if } x > 5 \text{ then } x = x + y \) else \( x = x - y \). Consider an abstract state in the domain of intervals \( \bar{p} = (x \mapsto [2, 10], y \mapsto [1, 1]) \). The abstract semantics of \( c \) w.r.t. \( \bar{p} \) is illustrated below:

\[ \mathcal{F}_\text{dba}[x > 5] \bar{p} = [5, 10] \]
\[ \mathcal{F}_\text{dba}[x > 5] \bar{p} = [2, 4] \]
\[ \mathcal{F}_\text{dba}[x + y] \bar{p} = [6, 11] \]
\[ \mathcal{F}_\text{dba}[x - y] \bar{p} = [1, 3] \]
\[ \mathcal{F}_\text{dba}[x \times y] \bar{p} = [1, 3] \]
\[ \mathcal{F}_\text{dba}[x \times y] \bar{p} = [1, 1] \]
\[ \mathcal{F}_\text{dba}[x \times y] \bar{p} = [1, 1] \]

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 \( I \) 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).
where

\[(\rho_{TM}, \rho_{T}) \in \mathcal{F} \mathcal{F} \mathcal{F}(\phi(\rho_{T}, \rho_{T})) \text{ and } (\rho_{FM}, \rho_{T}) \in \mathcal{F} \mathcal{F} \mathcal{F}(\neg \phi(\rho_{T}, \rho_{T}))\]

Observe that $\mathcal{T}M$ and $\mathcal{F}M$ are the abstract database states obtained by using the filtering semantics function $\mathcal{F} \mathcal{F} \mathcal{F}$ based on the satisfaction of $\phi$. In particular, $\mathcal{T}M$ denotes the part of the abstract database state for which $\phi$ is true, whereas $\mathcal{F}M$ denotes the abstract database state for which $\phi$ is false. After performing the update action $A$ on $\mathcal{T}M$, the resultant abstract state $\mathcal{T}M'$ is obtained. Finally, component-wise join operation between $\mathcal{T}M'$ and $\mathcal{F}M$ yields the resultant abstract state $\mathcal{P}$. Observe that, in order to ensure the soundness, both $\mathcal{T}M$ and $\mathcal{F}M$ include the information for which $\phi$ results in "may be true or false". We illustrate this in Example 9.

Example 9. Consider the abstract domain of intervals $\mathcal{I}$. Given the concrete database table $t$ shown in Table 4(a), its corresponding abstract version $\mathcal{I}$ replacing concrete values by their properties from $\mathcal{I}$ is depicted in Table 4(b). Similarly, given an application environment $(\mathcal{E}, \mathcal{A})$ defined by the following example.

<table>
<thead>
<tr>
<th>orm</th>
<th>sal</th>
<th>age</th>
<th>dno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1500</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>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 4: Concrete and its corresponding Abstract Database

$\rho_a = (x \iff 1000)$ where $x$ is an application variable, its corresponding abstract application environment in $\mathcal{I}$ is $\rho_T = (x \iff [100, 100])$.

Now consider the following update statement:

$\mathcal{Q}_{upd}: \text{UPDATE } t \text{ SET sal} = \text{sal} + x \text{ WHERE sal} \geq 1500$

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

<table>
<thead>
<tr>
<th>orm</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>2500</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3100</td>
<td>62</td>
<td>10</td>
</tr>
</tbody>
</table>

(a) Abstract table $\mathcal{T}M$

TABLE 5: Execution result $t'$ by $\mathcal{Q}_{upd}$ on $t$

The abstract semantics of $\mathcal{Q}_{upd}$ w.r.t. $\mathcal{T} = (\rho_{T}, \rho_{T})$ is $\mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D}(\mathcal{A}, \phi)(\rho_{T}, \rho_{T})$

$= \mathcal{F} \mathcal{F} \mathcal{F}(\text{UPDATE}((\text{sal}), (\text{sal} + x)), \text{sal} \geq 1500))(\rho_{T}, \rho_{T})$

$= \mathcal{F} \mathcal{F} \mathcal{F}(\text{UPDATE}((\text{sal}), (\text{sal} + x)))(\rho_{TM}, \rho_{T}) \cup (\rho_{FM}, \rho_{T})$

$= (\rho_{TM}, \rho_{T}) \cup (\rho_{FM}, \rho_{T})$

$= (\rho_{TM}, \rho_{T})$

where

$\rho_{TM} = \mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D}(\text{sal} \geq 1500)(\rho_{T}) = \rho_{T}[\text{sal} \leftarrow [1500, 3000]]$

$\rho_{TM} = \mathcal{F} \mathcal{F} \mathcal{F}(\text{sal} \geq 1500)(\rho_{T}) = \rho_{T}[\text{sal} \leftarrow [800, 14999]]$

$\rho_{TM} = \mathcal{F} \mathcal{F} \mathcal{F}(\text{sal} \geq 1500)(\rho_{T}) = \rho_{T}[\text{sal} \leftarrow [1600, 3001]]$

Tables 6(a) and 6(b) depict $\mathcal{T}M$ and $\mathcal{F}M$ respectively. After performing the update action $A$ on $\mathcal{T}M$, the resultant abstract table $\mathcal{T}M'$ is shown in Table 6(c). Finally, component-wise join operation between $\mathcal{T}M'$ and $\mathcal{F}M$ yields the resultant table $\mathcal{P}$ depicted in Table 6(d). Observe that the abstract semantics is sound, i.e. $\mathcal{T} \subseteq \mathcal{P}$.

TABLE 6: Execution results of $\mathcal{Q}_{upd}$ on $t$

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

TABLE 7: Part of execution results of $\mathcal{Q}_{upd}$ on $\mathcal{P}$

Abstract Semantics towards Independency Computation.

In order to compute semantics-based DD-independencies, we define $\mathcal{T} \mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D} \mathcal{D}$ according to equations 9, in $\mathcal{I}$ for database statements as follows:

$\mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D} \mathcal{D}(\mathcal{A}, \phi)(\mathcal{T})$

$= \mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D} \mathcal{D}(\mathcal{A}, \phi)(\mathcal{T})$

$= \mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D}(\mathcal{A}, \phi)(\mathcal{T})$

$= \mathcal{F} \mathcal{D} \mathcal{D} \mathcal{D}(\mathcal{A}, \phi)(\mathcal{T})$

where $t = \text{target}(\mathcal{A}, \phi)$ and $\mathcal{E} \in \mathcal{A}: t \in \gamma(\mathcal{I})$

$= (\rho_{TM}, \rho_{TM}, \rho_{TM})$

(11)

where

- $\mathcal{F} \mathcal{F} \mathcal{F}(\rho_{T}, \rho_{T}) = (\rho_{TM}, \rho_{T})$
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 $kx_i + kx_j ≤ k$ where $x_i, x_j$ are program variables, $k_i, k_j ∈ [-1, 0, 1]$ 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.

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 concrete abstract domain encodes binary constraints between program variables in the form of $kx_i + kx_j ≤ k$ where $x_i, x_j$ are program variables, $k_i, k_j ∈ [-1, 0, 1]$ 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 $\mathcal{P}$ with a finite set of variables $\mathbb{V}_p = \{x_1, \ldots, x_n\}$. A Difference Bound Matrix (DBM) $\mathcal{M}$ with size $n \times n$ represents a set of invariants each of the form $x_j - x_i ≤ k$, where $k ∈ \mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$ 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.

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

\[
\overline{F}_{dep}(\langle \text{UPDATE}(s) \rangle, \overline{p}) = \langle \rho_{TM}, \rho_{TM}, \rho_{TM} \rangle
\]

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. $\overline{p}$ is

\[
\overline{F}_{dep}(\langle \text{INSERT}(e) \rangle, \overline{p}) = \langle \rho_{TM}, \rho_{TM}, \rho_{TM} \rangle
\]

Observe that the select operation does not change any information.

Example 11. Consider the abstract state $\overline{p} = (\overline{p}_1, \overline{p}_2)$ and $\rho_\overline{p} = \{x \mapsto [100, 100]\}$ where $\overline{p}$ is depicted in Table 4(b), as defined in Example 9. Consider the following statements:

- $Q_{ins} =$UPDATE t SET sal = sal + x WHERE sal ≥ 1500
- $Q_{ins} =$INSERT INTO t (eid, sal, age, dno)VALUES(5, 2700, 52, 20)
- $Q_{del} =$DELETE FROM t WHERE age ≥ 61
- $Q_{sel} =$SELECT age FROM t WHERE age ≤ 50

The abstract syntax of the statements are:

- $Q_{ins} =$UPDATE(sal, sal + x), sal ≥ 1500
- $Q_{ins} =$INSERT(eid, sal, age, dno), (5, 2700, 52, 20), false
- $Q_{del} =$DELETE(eid, age, dno), age ≥ 61
- $Q_{sel} =$SELECT(age), age ≤ 50

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

\[
\overline{F}_{dep}(\langle \text{UPDATE}(s) \rangle, \rho_{TM}, \rho_{TM}, \rho_{TM})
\]
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:


g |
\hline
x_1 & x_2 & x_3 & x_4 \\
\hline
x_1 & 5 & 0 & 0 \\
x_2 & 0 & 0 & 4 \\
x_3 & 0 & 4 & 0 \\
x_4 & 0 & 4 & 0 \\
\hline

Extension to encode octagonal constraints [43]. The above DBM representation over program variables can represent only a subset of octagonal constraints of the form \(x_i - x_j \leq k\). In order to allow more general form \(\pm x_i \pm x_j \leq k\) of octagonal constraints, a DBM of size \(n \times n\) defined over \(\mathbb{V}_p\) is extended to another DBM \(m\) of size \(2n \times 2n\) over the set of enhanced variables \(\mathbb{V}_p = \{x'_1, \ldots, x'_{2n}\}\) where each variable \(x_i \in \mathbb{V}_p\) comes in two forms: a positive form \(x'_{2i-1}\), denoted \(x_i^p\) and a negative form \(x'_{2i}\), denoted \(x_i^n\). This extended form of DBM \(m\) is called coherent DBM (CDBM) representing octagon. This is illustrated in the following example.

Example 13. Consider the octagonal constraints \(x_1 + x_2 \leq 3, x_1 - x_2 \leq 4, -x_1 - x_2 \leq 5, x_1 \leq 4\), its equivalent CDBM constraints are \(x_i^p - x_i^n \leq 3, x_i^p - x_i^n \leq 3, x_i^p - x_i^n \leq 4, x_i^p - x_i^n \leq 4, x_i^p - x_i^n \leq 5, x_i^p - x_i^n \leq 5, x_i^p - x_i^n \leq 8\). These constraints are represented in CDBM shown below:


g |
\hline
x_1 & x_2 & x_3 & x_4 \\
\hline
x_1 & 0 & 0 & 0 \\
x_2 & 4 & 5 & 0 \\
x_3 & 5 & 0 & 0 \\
x_4 & 0 & 0 & 0 \\
\hline

Observe that any constraints of the form \((x_i \leq k)\) and \((x_i \geq k)\) can be represented as \((x_i^p - x_i^n \leq 2k)\) and \((x_i^p - x_i^n \leq -2k)\) respectively.

Closure. An octagon can be represented by more than one set of inequalities. For instance, the octagonal constraints \(\{(x \leq 4) \land (y \leq 6)\}\) and \(\{(x \leq 4) \land (y \leq 6) \land (x + y \leq 10)\}\) 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 CDBM when the context is clear.

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

\[
\forall m, n \in \mathbb{M}_L: m \subseteq n \iff \forall i, j: m_{ij} \leq n_{ij}.
\]

\[
\forall m, n \in \mathbb{M}_L: (m \cap n) = m \lor n \forall i, j: (m \cap n)_{ij} \leq \min(m_{ij}, n_{ij}).
\]

\[
\forall m, n \in \mathbb{M}_L: (m \lor n) = m \land n \forall i, j: (m \lor n)_{ij} \geq \max(m_{ij}, n_{ij}).
\]

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 \(\Sigma = \mathbb{V} \rightarrow \mathbb{R}\). An environment \(\rho\) of \(\Sigma\) maps each variable to its value. An environment will be understood as a point in \(\mathbb{R}^n\) where \(|\mathbb{V}| = n\). The Galois connection between \(\mathbb{L}_c\) and \(\mathbb{L}_a\) is formalized as \(\langle \mathbb{L}_c, \alpha_{BM}, \gamma_{BM}, \mathbb{L}_a \rangle\) where \(\alpha_{BM}\) and \(\gamma_{BM}\) on \(S \in \phi(\mathbb{R}^n)\) and \(m \in \mathbb{M}_L\) are defined below:

\[
\begin{align*}
\text{if } S = \emptyset: & \quad \alpha_{BM}(S) \equiv m_{\bot} \\
\text{if } S \neq \emptyset: & \quad \alpha_{BM}(S) = m \text{ where } m_{ij} \equiv \begin{cases} \\
\max(\rho(x_i) - \rho(x_j) | \rho \in S) & \text{if } i = 2k - 1, j = 2l - 1 \\
\max(\rho(x_i) + \rho(x_j) | \rho \in S) & \text{if } i = 2k, j = 2l - 1 \\
\max(-\rho(x_i) - \rho(x_j) | \rho \in S) & \text{if } i = 2k - 1, j = 2l \\
\end{cases}, \quad \text{or } i = 2l, j = 2k \\
\gamma_{BM}(m) = & \begin{cases} \\
\emptyset & \text{if } m = m_{\bot} \\
\mathbb{R}^n & \text{if } m = m_{\top} \\
\left\{(k_1, \ldots, k_n) \in \mathbb{R}^n | (k_1, -k_1 \ldots, -k_n) \in \text{dom}(m) \land \forall i, j: x_i - x_j \leq m_{ij}\right\} & \text{otherwise}
\end{cases}
\end{align*}
\]

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

- **Empitness test:** Let \(m\) be a CDBM and \(\mathbb{G}\) be a directed weighted graph of \(m\). We say that the octagon is empty, i.e. \(\gamma(m) = \emptyset\), if and only if \(\mathbb{G}\) 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:

\[
[v | \exists (k_1 \ldots k_n) \in \gamma(m) \; \text{such that} \; k_i = v] = [-m_{2i+1}/2, m_{2i+1}/2]
\]

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

**Test:** Given a CDBM \(m\) representing abstract state at a program point and a boolean expression \(\rho\). The state-filtering function \(\mathcal{T}_f\) finds \(m'\) applying \(\rho\) on \(m\) where \(\gamma(m') = \{p \in \gamma(m) | \rho\) 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) | \rho\) satisfies \(b\}\n\]

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

\[
x_i + x_j \leq k, x_i - x_j \leq k, -x_i - x_j \leq k, x_i + x_j = k, x_i \leq k
\]
and $x_h \geq k$. The state-filtering function $\overline{\mathcal{F}_j}$ for $x_h + x_l \leq k$ is defined as below:

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

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

Observe that the entries in the CDBM $m$ corresponding to the cells $(x_h', x_l')$ and $(x_h^c, x_l^c)$ are updated based on the value $k$, resulting into $m'$ which satisfies $x_h + x_l \leq k$. Similarly $\overline{\mathcal{F}_j}$ 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 in $m'$ as an upper approximation such that

$$\gamma(m') \trianglerighteq \nu \{ x_i \leftarrow k \} \cap \gamma(m) \land k = \overline{\mathcal{F}_j}[c]\rho$$

where $\mathcal{F}_j$ 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: $x_h = x_h + 1$ 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 we 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 $x_h - x_k \leq -k$ are added into the octagon. Let us illustrate them below:

1) If $x_h = x_h + k$:
   $$\overline{\mathcal{F}_j}[x_h = x_h + k] = m'$$ where $m'_{ij} = m_{ij} + (\alpha_{ij} + \beta_{ij})k$
   with
   $$\alpha_{ij} = \begin{cases} 
+1 & \text{if } j = 2h, \\
-1 & \text{if } j = 2h - 1, \\
0 & \text{otherwise}
\end{cases}$$
   and
   $$\beta_{ij} = \begin{cases} 
-1 & \text{if } i = 2h, \\
+1 & \text{if } i = 2h - 1, \\
0 & \text{otherwise}
\end{cases}$$

2) If $x_h = x_h + 1$ with $h \neq l$:
   $$\overline{\mathcal{F}_j}[x_h = x_h + 1] = m'$$ where
   $$m'_{ij} = \begin{cases} 
2h & \text{if } (i, j) \in \{(2h, 2l - 1), (2l, 2h - 1)\}, \\
2l - 1 & \text{if } (i, j) \in \{(2h, 2l - 1), (2l, 2h - 1)\}, \\
m_{ij} & \text{otherwise}
\end{cases}$$

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

$$\overline{\mathcal{F}_j}[x \geq 5] \overline{\mathcal{F}_j}[x = y + 1] \overline{\mathcal{F}_j}[x = x - 1]$$

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 $\rho_d \in \mathcal{E}_{dbu}$ and application environment $\rho_a \in \mathcal{E}_a$. To determine abstract semantics of database statements in the domain of octagons, we define the abstract state $\overline{\rho}$ in $\mathcal{E}_{dbu}$ as

$$\overline{\rho} = \langle m_d, m_a \rangle$$

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: $\overline{\mathcal{F}_d\Phi}[A, \phi] = \overline{\mathcal{F}_d\Phi}(\langle A, \phi \rangle)(m_d, m_a) = (m_d, m_a)$ where $m$ is the octagonal representation of the concrete table $t$ which acts as the target of $Q$ and $m'$ is the octagonal representation of the resultant table $t'$. Below is the abstract semantics for update statement.

$$\overline{\mathcal{F}_d\Phi}[\langle UPDATE(\bar{x}_d, \bar{c}_d), \phi \rangle](m_d, m_a) = \overline{\mathcal{F}_d\Phi}[\langle UPDATE(\bar{x}_d, \bar{c}_d) \rangle](m_{TM}, m_a) \sqcup (m_{FM}, m_a)$$

where

$$\overline{\mathcal{F}_d\Phi}[(m_d, m_a)] = (m_{TM}, m_a) \sqcup (m_{FM}, m_a)$$

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 $\rho_a = \langle x \mapsto 100 \rangle$ where $x$ is an application variable. The corresponding abstract representation of $\rho_a$ and $\rho_d$ in octagon domain are represented by CDBM $m_d$ and $m_a$, respectively as

$$m_d \overset{\text{def}}{=} \{ - e id \leq -1, - e id \leq -1, - e sal \leq -800, - e sal \leq 3000, - e ag e \leq -28, e ag e \leq 62, - e dno \leq -10, e dno \leq 20 \},$$

and

$$m_a \overset{\text{def}}{=} \{ x \leq 100, x \leq -100 \}.$$
of Observation the numbers of database variables and application variables
Like for the interval domain, the following transition re-
and
\[ \frac{t = \text{UPDATE}(\langle \phi \rangle)}{m_{TM}, m_a} \cup (m_{FM}, m_a) \]
= \langle m_{TM} \cup m_{TM}, m_a \cup m_a \rangle
= \langle m', m_a \rangle

where
\[ m_{TM} \equiv \{ - eid < -1, eid < 4, sal < 800, sal < 3000, age \leq 35, age < 62, -dno < -10, dno < 20 \} \]
\[ m_{FM} \equiv \{ - eid < -1, eid < 4, sal < 800, sal < 3000, age \leq 28, age < 34, -dno < -10, dno < 20 \} \]
\[ m_{TM'} \equiv \{ - eid < -1, eid < 4, sal < 2900, sal < 3100, age \leq 35, age < 62, -dno < -10, dno < 20 \} \]

Observation. We can follow an alternative equivalent way of abstract state representation by combining both CDBM of \( m_I \) and \( m_a \) for the sake of simplicity. Let \( p \) and \( q \) denote the numbers of database variables and application variables respectively. Given \( m_I \) and \( m_a \) as 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 \( m_a \). In the subsequent sections we define abstract semantics w.r.t. abstract state \( \tilde{p} = m \).

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:
\[ \tilde{\mathcal{F}}_{dep} : C \times M_\perp \rightarrow (M_\perp \times M_\perp \times M_\perp) \]
Below is the definition of \( \tilde{\mathcal{F}}_{dep} \) for various database statements in octagon domain.

1. UPDATE:
\[ \tilde{\mathcal{F}}_{dep} \langle \text{UPDATE}(\vec{\phi}, \phi) \rangle m = \left\{ \begin{array}{ll}
\langle m_I, m_T, m_T \rangle & \text{if } \phi \in [k_i, x_j + k_j, x_j] \leq k \text{ where } x_i, x_j \in \mathbb{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\
\langle m, m, m' \rangle & \text{otherwise}
\end{array} \right. \]

where \( \tilde{\mathcal{F}}_{dep} \langle \phi \rangle m = m_I \) and \( \tilde{\mathcal{F}}_{dep} \langle \phi \rangle m = m_T \)

\[ \tilde{\mathcal{F}}_{dep} \langle \text{UPDATE}(\vec{\phi}, \phi) \rangle m_T = m_T [\vec{\phi} \leftarrow \tilde{\mathcal{F}}_{dep} \langle \phi \rangle m_T] = m_T. \]
\[ \tilde{\mathcal{F}}_{dep} \langle \text{UPDATE}(\vec{\phi}, \phi) \rangle m = m'. \]

2. INSERT:
\[ \tilde{\mathcal{F}}_{dep} \langle \text{INSERT}(\vec{\phi}, false) \rangle m = \langle m, m, m_\perp \rangle \]
where \( \tilde{\mathcal{F}}_{dep} \langle false \rangle m = m \) and \( \tilde{\mathcal{F}}_{dep} \langle \phi \rangle m = m_\perp \)

where \( m_\perp \) represents bottom element that contains an unsatisfiable set of constraints.

3. DELETE:
\[ \tilde{\mathcal{F}}_{dep} \langle \text{DELETE}(\vec{\phi}, \phi) \rangle m = \left\{ \begin{array}{ll}
\langle m_I, m_T, m_T \rangle & \text{if } \phi \in [k_i, x_j + k_j, x_j] \leq k \text{ where } x_i, x_j \in \mathbb{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\
\langle m, m, m_\perp \rangle & \text{otherwise}
\end{array} \right. \]

4. SELECT:
\[ \tilde{\mathcal{F}}_{dep} \langle \text{SELECT}(\vec{\phi}, r, \phi) \rangle m = \left\{ \begin{array}{ll}
\langle m_I, m_T, m_T \rangle & \text{if } \phi \in [k_i, x_j + k_j, x_j] \leq k \text{ where } x_i, x_j \in \mathbb{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\
\langle m, m, m \rangle & \text{otherwise}
\end{array} \right. \]

Observe that \( \phi \in [k_i, x_j + k_j, x_j] \leq k \) 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 \( m_I \) in the domain of octagons as
\[ m_I \equiv \{ - eid < -1, eid < 4, sal < 800, sal < 3000, age < 35, age < 62, -dno < -10, dno < 20 \}. \]

Consider the following statements:
\[ Q_{\text{upd}} = \text{UPDATE} t \text{ SET } sal = sal + x \text{ WHERE } age \geq 35 \]
\[ Q_{\text{ins}} = \text{INSERT INTO } t(eid, sal, age, dno) \text{VALUES}(5, 2700, 52, 20) \]
\[ Q_{\text{del}} = \text{DELETE FROM } t \text{ WHERE } age \geq 61 \]
\[ Q_{\text{sel}} = \text{SELECT } sal \text{ FROM } t \text{ WHERE } age \leq 50 \]

The abstract syntax are
\[ Q_{\text{ins}} *= \text{UPDATE} t * \text{ WHERE } age = 35 \]
\[ Q_{\text{ins}} *= \text{INSERT} t * \text{ WHERE } age = (5, 2700, 52, 20) \]
\[ Q_{\text{del}} *= \text{DELETE} t * \text{ WHERE } age = 61 \]
\[ Q_{\text{sel}} *= \text{SELECT} t * \text{ WHERE } age = 50 \]

The abstract semantics of the \( Q_{\text{upd}} \) with respect to \( m_I \) is
\[ \tilde{\mathcal{F}}_{dep} \langle \text{UPDATE}(sal, x) \rangle m_I = \{ m_I, m_T, m_T \} \]
where \( m_{new} \equiv \{ - eid < -5, eid < 5, sal < 2700, sal < 2700, age < 52, age < 52, -dno < -20, dno < 20 \} \)

The abstract semantics of the \( Q_{\text{ins}} \) w.r.t. \( m_I \) is
\[ \tilde{\mathcal{F}}_{dep} \langle \text{INSERT}(eid, sal, age, dno) \rangle m_I = \{ m_I, m_T, m_T \} \]

The abstract semantics of the \( Q_{\text{del}} \) w.r.t. \( m_I \) is
\[ \tilde{\mathcal{F}}_{dep} \langle \text{DELETE}(eid, sal, age, dno) \rangle m_I = \{ m_I, m_T, m_T \} \]

The abstract semantics of the \( Q_{\text{sel}} \) w.r.t. \( m_I \) is
\[ \tilde{\mathcal{F}}_{dep} \langle \text{SELECT}(eid, sal, age, dno) \rangle m_I = \{ m_I, m_T, m_T \} \]
m_r ≡ \{ \begin{align*}
& \text{age} \leq -61, -\text{dno} \leq -10, \text{dno} \leq 20 \\
& \text{age} \leq -51, -\text{dno} \leq -10, \text{dno} \leq 20
\end{align*} \}
\]

The abstract semantics of the $Q_{\text{eq}}$ w.r.t. $m_r$ is
\[
\mathcal{T}_\text{dep}(\text{SELECT}(\text{age})), \text{age} \leq 50 \rangle m_r = \langle m_r, m_t, m_r \rangle
\]

where
\[
\begin{align*}
& m_t ≡ \{ -\text{eid} \leq -1, \text{eid} \leq 4, -\text{sal} \leq -800, \text{sal} \leq 3000, \text{age} \leq 60, \\
& \text{age} \leq -28, -\text{dno} \leq -10, \text{dno} \leq 20 \}
\end{align*}
\]

**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: `update t set a = a + 1 where a + b + c > 35`. Given an abstract state $m$ in the domain of octagons, the abstract semantics function $\mathcal{T}_\text{dep}$ fails to capture $m_{TM}$ and $m_M$ precisely as the constraint $a + b + c > 35$ involves more than two variables and hence cannot be represented in octagonal constraint form.

### 6.2.3 Relational Abstract Domain of Polyhedra

The preciseness of the analysis in relational abstract domain improves significantly if more number of relations among variables or attributes are in consideration when analyzing the programs. Thus, the analysis in the polyhedra abstract domain, although computationally costly, improves the precision significantly compared to the octagon abstract domain, P. Cousot and N. Halbwachs in their seminal work [41] first introduced the polyhedra abstract domain for static determination of linear equality and inequality relations among program variables, and over the past several decades this has been widely used in several engineering problems such as static analysis of gated Data Dependence Graphs (gated DDGs) [48], Information flow analysis to detect possible information leakages combining symbolic propositional formulas domain and numerical polyhedra domain [49], Hybrid systems verification tool SpaceEx [50], etc.

Let us briefly recall some basics. The regions in $n$-dimensional space $R^n$ 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 $\vec{v} = (v_1, v_2, \ldots v_n) \in R^n$, an $n$-tuple (vector) of real numbers. By $\beta = \vec{v} \cdot \vec{x} \geq k$ where $\vec{v} \neq \vec{0}$, $\vec{x} = (x_1, x_2, \ldots, x_n)$, $k \in R$, we represent a linear inequality over $R^n$. A linear inequality defines an affine half-space of $R^n$. If $P$ is expressed as the intersection of a finite number of affine half-spaces of $R^n$, then $P \in R^n$ is a convex polyhedron. Formally, a convex polyhedron $P = (\Theta, n)$ is a set of linear inequalities $\Theta = \{ \beta_1, \beta_2, \ldots \beta_m \}$ on $R^n$. Equivalently, $P$ can be represented by frame representation which is a collection of generators i.e. vertices and rays [51]. On the other hand, given a set of linear inequalities $\Theta$ on $R^n$, a set of solutions or points defines a polyhedron $P = (\Theta, n)$. **Concrete Function.** Let $L_r = (\mathcal{P}(R^n), \subseteq, \mathcal{R}(R^n), \cap, \cup)$ be the concrete lattice defined over the concrete domain. The set of polyhedra $P$ with partial order $\subseteq$ forms an abstract lattice $L_a = (P_a, \subseteq, P_r, \cap, \cup)$. Given $P_1, P_2 \in P$, the partial order, meet and join operations are defined below:

- $P_1 \subseteq P_2$ if and only if $\gamma(P_1) \subseteq \gamma(P_2)$, where $\gamma(P)$ represents the set of solutions or points in $P$ as concrete values.
- $P_1 \cap P_2$ is the convex polyhedron containing exactly the set of points $\gamma(P_1) \cap \gamma(P_2)$.
- $P_1 \cup P_2$ is not necessarily a convex-polyhedron. Therefore, the least polyhedron enclosing this union is computed in terms of convex hull.

An environment $\rho \in \Sigma = V \mapsto R$ map each variable to its value in $R$. Given $P \in P$, $\gamma_P$ is defined below:

$$\gamma_P(P) = \begin{cases}
\emptyset & \text{if } P = P_\bot \\
\rho \in \Sigma & \text{if } P = P_\top \\
\rho \in \Sigma & \{ \rho \in \Sigma | \forall (\vec{v}, \vec{x}) \geq k : \vec{v} \cdot \rho(\vec{x}) \geq k \} & \text{otherwise}
\end{cases}$$

Note that there is no abstraction function in polyhedra abstract domain because some vector sets do not have a best over-approximation as a convex closed polyhedron [41]. Therefore, in this case we denote by $\alpha_P(S)$ a (possibly minimal) polyhedron in $P$ such that $\gamma_P(\alpha_P(S)) \geq S$.

**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 system. For example, adding a new constraint $\vec{v} \cdot \vec{x} \geq k$ to a non empty polyhedron $P$, we can solve the LP problem $\mu = \min \vec{v} \cdot \vec{x}$ 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:

$$\Pi_{x_i}(P) = \{ \rho[v_i/x_i] \mid \rho \in \gamma(P), v \in R \}$$

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

$$F(P, x_i) = \{ (\Sigma, x_i > k) \in \Theta^+ | v_i = 0 \} \cup \{ (\Sigma, x_i > k) \in \Theta^- | v_i > 0 \} \cup \{ (\Sigma, x_i > k) \in \Theta^+ | v_i < 0 \} \cup \{ (\Sigma, x_i > k) \in \Theta^- | v_i < 0 \}$$

where $v_i^+$ and $v_i^-$ represent positive and negative coefficients for $x_i$ respectively. The
algorithm partitions the set of liner inequalities \( \mathcal{A} = \{a_1, a_2, \ldots, a_n\} \) into \( \mathcal{A}_1, \mathcal{A}_2 \) and \( \mathcal{A}_3 \), corresponding to inequalities that have positive, zero and negative coefficients for \( x_i \). For each pair \((\mathcal{A}_i, \mathcal{A}_j)\) of inequalities drawn from \( \mathcal{A}_1 \times \mathcal{A}_2 \), the algorithm multiplies \( \beta_i \) by the absolute value of \( x_i \)-th coefficient \( |a_{ij}| \) in \( \mathcal{A}_i \) and similarly multiplies \( \beta_j \) by \( x_j \)-th coefficient \( a_{ji} \) in \( \mathcal{A}_j \). 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 \( \mu = \min \bar{\delta}x \) subject to \( P_1 \) for each \( \bar{\delta}x \geq k \). If \( \mu < k \) then inclusion does not hold.

### Abstract Semantics of Imperative Language in Polyhedra - A Quick Tour [41].

Given the concrete denotational semantic functions \( \mathcal{F}_c : (\mathcal{B} \mapsto \wp(\Sigma)) \mapsto \wp(\Sigma) \), the corresponding sound abstract function \( \mathcal{F}_a \) in polyhedra is defined as \( \mathcal{F}_a : (\mathcal{B} \mapsto \wp(\Sigma)) \mapsto \wp(\Sigma) \). Similarly, given the concrete denotational semantic function \( \mathcal{F}_c : (\mathcal{C} \mapsto \wp(\Sigma)) \mapsto \wp(\Sigma) \), its corresponding sound abstract function \( \mathcal{F}_a \) in polyhedra domain is defined as \( \mathcal{F}_a : (\mathcal{C} \mapsto \wp(\Sigma)) \mapsto \wp(\Sigma) \).

**Test**: Let \( \bar{a} \) be a boolean expression in the form of linear inequalities \( \bar{\delta}x \geq k \) and the abstract state in the form of polyhedron \( P \). The state-filtering function \( \mathcal{F}_f \) finds \( P' \) applying \( b \) on \( P \) define as

\[
\mathcal{F}_f[\bar{\delta}x \geq k]P = P'
\]

where \( P' = P \cap b \).

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

**Assignment statement**: \( \mathcal{F}_a[\{x : e\}]P = P' \) where \( P' \) 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 \( P \), resulting into a new polyhedron \( P' \); (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 \( P' \).

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

\[
\mathcal{F}_a[x = x + y]P = P'
\]

where \( P' = \{(x - y \geq 3, y \geq 2), 2\} \) 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 \( \bar{p} = \langle p_d, p_s \rangle \in \mathcal{F}_{\text{db}} \) where \( p_d \) and \( p_s \) are polyhedra representation of database values and application variables values respectively. According to equation 8, the abstract semantic function for database statements \( Q = \langle A, \phi \rangle \) is defined as: \( \mathcal{F}_{\text{db}}[\langle A, \phi \rangle](\bar{p}_d, \bar{p}_s) = \mathcal{F}_{\text{db}}[\langle A, \phi \rangle](\bar{p}_d, \bar{p}_s) = \langle p_t, \bar{p}_s \rangle \) where \( p_t \) is the polyhedron representation of the concrete table \( \ell \) which acts as the target of \( Q \), and \( p_t \) is the polyhedron representation of the resultant table \( \ell' \).

Alternatively, like octagon abstract domain, for the sake of simplicity, we may combine both \( p_d \) and \( p_s \) into a single polyhedra \( P \) as an abstract program state. In the subsequent sections we define abstract semantics suitable for indepen-
dency computations w.r.t. an abstract state \( \bar{p}=P \) in the domain of polyhedra.

### Abstract Semantics towards Independency Computation.

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

1. **UPDATE**: \( \mathcal{F}_{\text{dep}}[\langle \text{UPDATE}(\bar{a}_d, \bar{e}), \phi \rangle]P = \langle P_F, P_T, P_T \rangle \) where

\[
\mathcal{F}_{\text{dep}}[\neg \phi]P = P_F,
\]

\[
\mathcal{F}_{\text{dep}}[\phi]P = P_T,
\]

\[
\mathcal{F}_{\text{dep}}[\text{UPDATE}(\bar{a}_d, \bar{e})]P_T = \mathcal{F}_a[\bar{a}_d \mapsto \bar{e}]P_T = P_T.
\]

We denote by the notation \( \bar{e}_d \mapsto \bar{e} \) a series of assignments \( \langle \bar{e}_1= e_1, \bar{e}_2= e_2, \ldots, \bar{e}_n= e_n \rangle \) where \( \bar{e}_d = \langle \bar{e}_1, \bar{e}_2, \ldots, \bar{e}_n \rangle \) and \( \bar{e} = \langle e_1, e_2, \ldots, e_n \rangle \), which follow the transition semantic definition for the assignment statement.

2. **INSERT**: \( \mathcal{F}_{\text{dep}}[\langle \text{INSERT}(\bar{a}_d, \bar{e}), \text{false} \rangle]P = \langle P, P_L, P_{\text{new}} \rangle \) where

\[
\mathcal{F}_{\text{dep}}[\neg \text{false}]P = P,
\]

\[
\mathcal{F}_{\text{dep}}[\text{false}]P = P_L,
\]

\[
\mathcal{F}_{\text{dep}}[\text{INSERT}(\bar{a}_d, \bar{e})]P_L = P_L[\bar{a}_d \mapsto \mathcal{F}_a[\bar{a}_d]P_L] = P_{\text{new}}
\]

\( P_{\text{new}} \) represents a polyhedron corresponding to the new inserted tuple values.

3. **DELETE**: \( \mathcal{F}_{\text{dep}}[\langle \text{DELETE}(\bar{a}_d), \phi \rangle]P = \langle P_F, P_T, P_L \rangle \)

4. **SELECT**: \( \mathcal{F}_{\text{dep}}[\langle \text{SELECT}(\bar{a}_d, \bar{u}(\bar{x})), \phi_2, \bar{g}(\bar{c}), \phi_1 \rangle]P = \langle P_T, P_T, P_T \rangle \)

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

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

\[ Q_{\text{upd}} = \text{UPDATE } t \text{ SET } \text{sal} = \text{sal} + \text{sal} \times 0.2 \text{ WHERE } \text{dno} + \text{age} \geq 60 \]

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

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

\[ Q_{\text{sel}} = \text{SELECT age FROM } t \text{ WHERE } \text{age} + \text{dno} \leq 60 \]

The equivalent abstract syntax are:

\[ Q_{\text{upd}} = (\text{UPDATE}(\langle \text{sal} \rangle, \langle \text{sal} + \text{sal} \times 0.2 \rangle), \text{dno} + \text{age} \geq 60) \]

\[ Q_{\text{ins}} = (\text{INSERT}(\langle \text{eid}, \text{sal}, \text{age}, \text{dno} \rangle), (5, 2700, 52, 20), \text{false}) \]

\[ Q_{\text{del}} = (\text{DELETE}(\langle \text{eid}, \text{sal}, \text{age}, \text{dno} \rangle), \text{age} \geq 61) \]

\[ Q_{\text{sel}} = (\text{SELECT}(\langle \text{age} \rangle), \text{age} + \text{dno} \leq 60) \]

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

\[
\overrightarrow{\text{dep}}(\langle \text{sal} \rangle, \langle \text{sal} + \text{sal} \times 0.2 \rangle, \text{dno} + \text{age} \geq 60) \| P_t = \langle P_T, P_T, P_T \rangle
\]

where

\[
P_T = \langle \text{eid} \geq 1, \text{age} \geq 49, \text{dno} \geq 10, \text{dno} - \text{dno} \geq -59 \rangle
\]

\[
P_{\bot} = \langle \text{age} \geq 49, \text{dno} \geq 10, \text{dno} - \text{dno} \geq -59 \rangle
\]

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

\[
\overrightarrow{\text{dep}}(\langle \text{eid}, \text{sal}, \text{age}, \text{dno} \rangle, (5, 2700, 52, 20), \text{false}) \| P_t = \langle P_T, P_{\bot}, P_{\text{new}} \rangle
\]

where

\[
P_{\text{new}} = \langle \text{age} \geq 52, \text{dno} \geq 20, \text{dno} - \text{dno} \geq -20 \rangle
\]

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

\[
\overrightarrow{\text{dep}}(\langle \text{age} \rangle, \text{age} + \text{dno} \leq 60) \| P_t = \langle P_T, P_T, P_T \rangle
\]

where

\[
P_T = \langle \text{age} \geq 49, \text{dno} \geq 10, \text{dno} - \text{dno} \geq -59 \rangle
\]

\[
P_{\bot} = \langle \text{age} \geq 49, \text{dno} \geq 10, \text{dno} - \text{dno} \geq -59 \rangle
\]

\[
P_{\text{new}} = \langle \text{age} \geq 52, \text{dno} \geq 20, \text{dno} - \text{dno} \geq -20 \rangle
\]

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 \( L_c = (D_c, \emptyset, \bot_c, T_c, \cap_c, \cup_c) \) be a concrete lattice and \( L_p = (\overline{D_p}, \emptyset, \perp_p, T_p, \cap_p, \cup_p, \emptyset) \) an abstract lattice over an abstract domain \( A \). The \( L_c \) and \( L_p \) are related by the Galois connection \( (L_c, \alpha_1, \gamma_1, L_p) \). Considering the powerset abstract domain, the powerset of \( \overline{D} \) denoted by \( \wp(\overline{D}) \) with the order relations \( \preceq \) forms an abstract lattice \( L_p = (\wp(\overline{D}), \subseteq, \emptyset, \overline{D}, \land, \lor) \). The partial order, meet and join operations in this abstract domain are defined as follows:

\[
\forall S_1, S_2 \in \wp(\overline{D}) : S_1 \preceq S_2 \iff \forall \overline{v}_i \in S_1 : \exists \overline{v}_i \in S_2, \overline{v}_i \subseteq \overline{v}_j.
\]

\[
\forall S_1, S_2 \in \wp(\overline{D}) : S_1 \land S_2 = \{ \overline{v}_1 \cap \overline{v}_j \mid \forall \overline{v}_j \in S_1, \overline{v}_j \in S_2 \}.
\]

\[
\forall S_1, S_2 \in \wp(\overline{D}) : S_1 \lor S_2 = S_1 \cup S_2.
\]

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

The \( L_c \) and \( L_p \) are related by a Galois connections \( (L_c, \alpha_1, \gamma_1, L_p) \) where \( \alpha_1 \) and \( \gamma_1 \) on \( \forall X \in D \) and \( \forall Y \in \wp(D) \) are defined below:

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

\[
\gamma_1(Y) = \begin{cases} \bot_c & \text{if } Y = \emptyset \\
T_c & \text{if } Y = \overline{D} \\
\bigcup \{ \gamma(\overline{y}) \mid \overline{y} \in Y \} & \text{otherwise} \end{cases}
\]

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:
abstract lattice \( L_\varphi = (\mathcal{I}, \sqsubseteq, \bot, [\infty, -\infty], \cap, \cup) \). The powerset of the intervals denoted by \( \varphi(\mathcal{I}) \) forms the abstract lattice \( L_\varphi = (\varphi(\mathcal{I}), \subseteq, \emptyset, \varphi(\mathcal{I}), \lor, \land) \).

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

Given a powerset abstract domain of intervals \( \varphi(\mathcal{I}) \), the set of abstract states \( \Sigma = \mathcal{V} \rightarrow \varphi(\mathcal{I}) \) respects the Galois Connection, i.e. \( \forall \rho \in \Sigma, \forall \rho' \in \Sigma : \alpha_1(\rho) \subseteq \rho \Leftrightarrow \rho \subseteq \gamma_1(\rho) \).

Given \( S_1, S_2 \in \varphi(\mathcal{I}) \), the sound abstract arithmetic operations \( \varphi(\mathcal{I}) \) in the powerset abstract domain of intervals are defined as:

\[
\forall S_1, S_2 \in \varphi(\mathcal{I}) : S_1 \varphi S_2 = \{ \emptyset, \varphi(\mathcal{I}) \mid \forall \varphi \in S_1, \forall \varphi' \in S_2 \}
\]

The corresponding sound abstract semantics function \( \mathcal{F} : (E \cup B) \rightarrow (\Sigma \rightarrow \varphi(\mathcal{I}) \cup \{ \text{true, false} \}, \tau_R) \) for expression evaluation where \( \tau_R \) denotes “may be true or may be false”, the abstract semantics functions \( \mathcal{F} : B \rightarrow (\Sigma \rightarrow \Sigma) \) for abstract state-filtering and \( \mathcal{F} : C \rightarrow (\Sigma \rightarrow \Sigma) \) for commands are defined accordingly in the powerset domain of intervals. Example 20 illustrates this.

**Example 20.** Consider the statement \( c := \text{if } x \geq 5 \text{ then } x = x + y \) else \( x = x - y \). Consider an abstract state in the powerset domain of interval \( \tilde{\varphi}(x = [2, 6], [8, 10]), y = [1, 1]) \). The abstract semantics of \( c \) w.r.t. \( \tilde{\varphi} \) is illustrated below:

\[
\begin{align*}
\mathcal{F}[x = 5](\tilde{\varphi}) &= \tilde{\varphi}(x = [5, 6], [8, 10]) = \tilde{\varphi}_1 \\
\mathcal{F}[x = 5](\tilde{\varphi}) &= \tilde{\varphi}(x = [2, 4]) = \tilde{\varphi}_2 \\
\mathcal{F}[x + y](\tilde{\varphi}_1) &= \tilde{\varphi}_1(x = 5, y = [6, 7], [9, 11]) = \tilde{\varphi}_3 \\
\mathcal{F}[x - y](\tilde{\varphi}_2) &= \tilde{\varphi}_2(x = 2, y = [1, 3]) = \tilde{\varphi}_4 \\
\mathcal{F}[x = x + y](\tilde{\varphi}_1) &= \tilde{\varphi}_1(x = [6, 7], [9, 11]) = \tilde{\varphi}_3 \\
\mathcal{F}[x = x - y](\tilde{\varphi}_2) &= \tilde{\varphi}_2(x = [1, 3]) = \tilde{\varphi}_4 \\
\mathcal{F}[x \geq 5 \text{ then } x = x + y \text{ else } x = x - y](\tilde{\varphi}) &= (\tilde{\varphi}_3 \cup \tilde{\varphi}_4) \\
&= \{(\tilde{\varphi}_3 \cup \tilde{\varphi}_4) \mid ([6, 7], [9, 11]) \cup ([1, 3]) \cup ([1, 1]) \}
\end{align*}
\]

**Defining Abstract Semantics of Database Language in the Powerset of Interval**

Given the semantic domain \( \varphi(\mathcal{I}) \), the abstract state is defined as \( \tilde{\varphi} = (\rho_T, \rho_S) \) where \( \rho_T : \text{attr}(t) \rightarrow \varphi(\mathcal{I}) \) and \( \rho_S : \mathcal{V} \rightarrow \varphi(\mathcal{I}) \).

Like other domains, according to equation 9, the abstract semantics in the powerset abstract domain for database statements are similarly defined below:

**UPDATE:**

\[
\mathcal{F}_{\text{db}}(\text{UPDATE}(\vec{v}_d, \vec{c}), \phi) = (\rho_T, \rho_S) = (\rho_{TM}, \rho_{TM})
\]

**INSERT:**

\[
\mathcal{F}_{\text{db}}(\text{INSERT}(\vec{v}_d, \vec{c}), \phi) = (\rho_T, \rho_S) = (\rho_{TM}, \rho_{TM})
\]

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 (including all common operations e.g. emptiness test, inclusion, etc.) in polyhedra abstract domain experience an exponential \( O(2^{n^2}) \) worst-case time complexity [54], whereas in octagonal domain the graph-based analysis algorithms for all common operations experience \( O(n^3) \) 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 CompAbsSem, depicted in Algorithm 1, which makes use of the semantics function \( \mathcal{F}_{\text{db}} \) and computes abstract states w.r.t. abstract domain \( \mathcal{D} \) at each program point of the database program. The algorithm is based on the data flow analysis considering various control-flow nodes: start, DB-connect, assignment, test, update, delete, insert, select, join, end. We denote by pred(c_i) and AS(c_i) the set of predecessor of \( c_i \) and the abstract state at \( c_i \) 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)
TABLE 8: A summary on various abstract domains

<table>
<thead>
<tr>
<th>Domain</th>
<th>Invariants</th>
<th>Time cost</th>
<th>Memory cost</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} \land k \in \mathbb{R} \cup {\infty})</td>
<td>(O(n^3))</td>
<td>(O(n^2))</td>
<td>medium</td>
</tr>
<tr>
<td>Polyhedra</td>
<td>(\sum_{i \in \mathbb{I}} a_i x_i \geq k, x_i \in \mathbb{V} \land a_i, k \in \mathbb{R}^n)</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>

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

Given a database statement \(Q\), let \(\bar{p} = (\rho_\text{df}, \rho_\text{def}, \rho_\text{assoc})\) 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 \(\bar{A}_\text{def}\) and \(\bar{A}_\text{used}\) 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,

\[
D^Q = \bar{A}_\text{def}(Q, \bar{p}) = (\rho_\text{df}, \rho_\text{assoc})
\]

\[
U^Q = \bar{A}_\text{used}(Q, \bar{p}) = (\rho_\text{def})
\]

Observe that \(\bar{A}_\text{used}\) maps a query \(Q\) to the abstract database-part used by it, whereas \(\bar{A}_\text{def}\) 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 \(\rho_\text{df}\) and \(\rho_\text{assoc}\) 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 \(\Delta\) (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 9: Abstract defined- and used-part of database by SQL statements in various abstract domains

<table>
<thead>
<tr>
<th>SQL</th>
<th>Abstract Domain of Intervals $\mathcal{P}$</th>
<th>Abstract Domain of Polyhedra $\mathcal{P}$</th>
<th>Abstract Domain of Octagons $\mathcal{P}$</th>
<th>Abstract Domain of Polyhedra $\mathcal{P}$</th>
<th>Powerset of Interval Domain $\mathcal{P}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update $Q_{\text{upd}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
</tr>
<tr>
<td>Delete $Q_{\text{del}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
</tr>
<tr>
<td>Insert $Q_{\text{ins}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
</tr>
<tr>
<td>Select $Q_{\text{sel}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
<td>$\Delta_{\text{def}}$, $\Delta_{\text{use}}$</td>
</tr>
</tbody>
</table>

Theorem 1 states that, given an abstract domain, equation 14 is necessary and sufficient condition for abstract DD-independence. 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 $\Delta_{\text{def}} \cap \Delta_{\text{use}} = \emptyset$ for $Q_1 = Q_2$.

Proof 1. 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 $\overline{p}$ and $\overline{p}'$ at $Q_1$ and $Q_2$ respectively which are obtained in step 25 of Algorithm 1. Let the abstract semantics applying $\overline{\mathcal{F}}_{\text{dep}}$ in step 26 be $\overline{\mathcal{F}}_{\text{dep}}(Q_1, \overline{p}) = \langle \overline{\rho}_{Q_1}, \overline{\rho}_{Q_1}, \overline{\rho}_{Q_2} \rangle$. 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 cannot 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 affects do not take place. In other words, the abstract database-part $\overline{\rho}_{Q_2}$ accessed by $Q_2$ overlaps with the parts $\overline{\rho}_{Q_1}$ and $\overline{\rho}_{Q_2}$ referred by $Q_1$ operations. This is captured in Case-4.

Algorithm to Compute Semantics-based DD-dependencies. The algorithm semDOPDG in Algorithm 2 takes a list of used- and defined-parts at each program point of the database program $\mathcal{P}$ 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 $Q_2$ 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 $\text{flag} is true$, then in the next step 11 the algorithm checks the condition $\Delta_{\text{use}} \subseteq \Delta_{\text{def}}$ in order to verify whether defined-part at program point $Q_2$ 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 $Q_2$, 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 $\overline{p}$ taking those variables into account with an assumption that “purchase_amt” is typed with unsigned smallint. Therefore, $\overline{p} =$
A \rightarrow 6 \) does not exist semantically as
\[ D^5 \cap U^6 = \emptyset \Rightarrow \rho^5_{TM} \cap \rho^6_{TM} = \emptyset \wedge \rho^5_{TM} \cap \rho^6_{TM} = \emptyset \]

This way one 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 \rightarrow 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 \( \leq 100 \) and 100 \( \leq \) wallet_bal \( \leq 9000 \) are defined on `point` and `wallet_bal`. Therefore, the abstract state at program point 4 is:
\[
P^{11}_{DB} = \{ \text{purchase_amt} \geq 0, \text{wallet_bal} \geq 65000, \text{point} \geq 0, \text{wallet_bal} \geq -9000 \}
\]

The abstract semantics of statement 15 is defined as:
\[
\mathcal{T}_{dbu}15(p_{11}^{15}) = p_{15}^{15} \text{ and } \mathcal{T}_{dep}15(p_{15}^{11}) = (p_{15}^F, p_{15}^I, p_{15}^U)
\]

where \( p^{15}_{11} = \{ \text{purchase_amt} \geq 0, \text{wallet_bal} \geq 65000, \text{point} \geq 0, \text{wallet_bal} \geq -9000 \} \) and \( p^{15}_{15} = \{ \text{purchase_amt} \geq 0, \text{purchase_bal} \geq 5000, \text{point} \geq 0, \text{purchase_bal} \geq -9999 \} \) and 5000 \( \leq \) purchase_bal + wallet_bal \( \leq 9999 \)

Similarly, abstract semantics of statement 16 is:
\[
\mathcal{T}_{dbu}16(p_{15}^{16}) = p_{16}^{15} \text{ and } \mathcal{T}_{dep}16(p_{16}^{15}) = (p_{16}^F, p_{16}^I, p_{16}^U)
\]

where \( p^{16}_{15} = \{ \text{purchase_amt} \geq 0, \text{wallet_bal} \geq 5000, \text{point} \geq 0, \text{wallet_bal} \geq -9999 \} \) and 5000 \( \leq \) purchase_bal + wallet_bal \( \leq 9999 \)

---

**Algorithm 2: semDOPDG**

**Input:** used- and defined-parts at each program point in the database program \( \mathcal{P} \).

**Output:** Semantic-based DD-dependency

1. Set \( \text{flag} = \text{true} \)
2. Identify database statements present in \( \mathcal{P} \). Let \( m \) be the number of database statements.
3. for \( i=1 \) to \( m-1 \) do
4.     for \( j=i+1 \) to \( m \) do
5.         if \( D^i \cap U^j \neq \emptyset \) then
6.             Add edges from \( i \)-th statement to \( j \)-th statement \((i \rightarrow j)\)
7.             Set \( \text{flag} = \text{true} \);
8.         else
9.             Set \( \text{flag} = \text{false} \);
10.        if \( \text{flag} = \text{true} \) then
11.            if \( D^i \subseteq D^j \) then
12.                break;
13.        End
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>4 → [5, 6, 7, 11, 15, 16]</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>2 → 6, 3 → 5</td>
</tr>
</tbody>
</table>

\[
\mathcal{P}_T^{16} = \left\{ \text{purchase\_amt} \geq 0, -\text{purchase\_amt} \geq -65000, \text{point} \geq 0, \\
- \text{point} \geq -100, \text{wallet\_bal} \geq 100, -\text{wallet\_bal} \geq -90000 \\
\text{purchase\_amt} + \text{wallet\_bal} \geq 10000 \right\}
\]

\[
\mathcal{P}_T^{16} = \left\{ \text{purchase\_amt} \geq 0, -\text{purchase\_amt} \geq -65000, \text{point} \geq 4, \\
- \text{point} \geq -104, \text{wallet\_bal} \geq 100, -\text{wallet\_bal} \geq -90000 \\
\text{purchase\_amt} + \text{wallet\_bal} \geq 10000 \right\}
\]

The defined-part by statement 15 and the used-part by statement 16 are computed as follows:

\[
\mathcal{D}^{15} = \mathcal{X}_{\text{def}}(\mathcal{P}_T^{15}, 15) = (\mathcal{P}_T^{15}, \mathcal{P}_T^{15}) \text{ and } \mathcal{U}^{16} = \mathcal{X}_{\text{use}}(\mathcal{P}_T^{16}, 16) = (\mathcal{P}_T^{16})
\]

Therefore the dependency \(15 \rightarrow 16\) does not exist semantically, as

\[
\mathcal{D}^{15} \cap \mathcal{U}^{16} = \emptyset \implies \mathcal{P}_T^{15} \cap \mathcal{P}_T^{16} = \emptyset \land \mathcal{P}_T^{15} \cap \mathcal{P}_T^{16} = \emptyset
\]

This way other data independencies can also be captured under polyhedral analysis.

8 Soundness of the Analysis

**Lemma 1.** Given an abstract state \(\overline{p} = (p_1, p_2)\), the abstract semantics function \(\mathcal{T}_{\text{abs}}\) is sound w.r.t. the concretization function \(\gamma\) if \(\forall Q \in Q, \forall \rho_1 \in \gamma(p_1), \forall \rho_2 \in \gamma(p_2) : \mathcal{T}_{\text{abs}}[Q](\rho_1, \rho_2) \subseteq \gamma(\mathcal{T}_{\text{abs}}[Q](p_1, p_2))\).

**Proof 2.** Given an abstract state \(\overline{p}\), the abstract semantics of \(Q = (A, \phi)\) w.r.t. \(\overline{p}\) is defined as \(\mathcal{T}_{\text{abs}}[Q](\overline{p}) = \mathcal{T}_{\text{abs}}[Q](A, \phi)(\overline{p}) = \mathcal{T}_{\text{abs}}[Q](A, \phi)(\rho_T, \rho_T) \subseteq \gamma(\mathcal{T}_{\text{abs}}[Q](p_1, p_2))\), where \(\rho_T\) and \(\rho_T\) represent abstract database states which satisfies \(\phi\) and \(\rho_T\) represents abstract database state which does not satisfy \(\phi\). Abstract state which, due to abstraction, may satisfy \(\phi\) is included in both \(\rho_T\) and \(\rho_T\). The state \(\rho_T\) is obtained by performing \(A\) on \(\rho_T\). Now let us consider a concrete state \(\rho \in \gamma(\overline{p})\). The concrete semantics of \(Q = (A, \phi)\) w.r.t. \(\rho\) is \(\mathcal{T}_{\text{abs}}[Q][\rho] = \mathcal{T}_{\text{abs}}[Q](A, \phi)[\rho] = \mathcal{T}_{\text{abs}}[Q](A, \phi)(\rho_T, \rho_T) \subseteq \gamma(\mathcal{T}_{\text{abs}}[Q](p_1, p_2))\), where \(\rho_T\) and \(\rho_T\) represent concrete database states based on the satisfaction and dissatisfaction of \(\phi\) respectively. As \(\phi\) in the abstract domain considers three valued logic due to the imprecision introduced in the abstraction, and since both \(\rho_T\) and \(\rho_T\) include the database state for which \(\phi\) evaluates to “may be true or false”, assuming local correctness of the functions and relations involved in \(\phi\) we get \(\rho_T \in \gamma(\rho_T)\) and \(\rho_T \in \gamma(\rho_T)\). Considering the Galois connection between concrete and abstract database and application domains, we therefore get \((\rho_T \cup \rho_T) \in \gamma(\rho_T \cup \rho_T)\) and so \(\rho' \in \gamma(\rho')\). This is depicted below:

**Lemma 2.** Let \(\overline{p}\) be an abstract state. The abstract semantic function \(\mathcal{T}_{\text{abs}}[Q]\) is sound w.r.t. \(\gamma\) if \(\forall Q \in Q, \forall \rho \in \gamma(\overline{p}) : \mathcal{T}_{\text{abs}}[Q][\rho] \subseteq \gamma(\mathcal{T}_{\text{abs}}[Q][\rho])\).

**Proof 3.** Given an abstract state \(\overline{p}\) and a database statement \(Q = (A, \phi)\) in \(Q\), the abstract semantic function \(\mathcal{T}_{\text{abs}}[Q]\) on \(\overline{p}\) computes abstract database state in the form of three-tuple as follows:

\[
\mathcal{T}_{\text{abs}}[Q][\rho] = \mathcal{T}_{\text{abs}}[Q][A, \phi][\rho] = (\rho_T, \rho_T, \rho_T)
\]

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

\[
\mathcal{T}_{\text{abs}}[Q][\rho] = \mathcal{T}_{\text{abs}}[Q][A, \phi][\rho] = (\rho_T, \rho_T, \rho_T)
\]

where \(\rho_T\) and \(\rho_T\) represent concrete database state based on the satisfaction and dissatisfaction of \(\phi\) respectively, and \(\rho_T\) is obtained after performing \(A\) on \(\rho_T\). 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 \(\rho_T \in \gamma(\rho_T)\), \(\rho_T \in \gamma(\rho_T)\) and \(\rho_T \in \gamma(\rho_T)\), which implies that \(\mathcal{T}_{\text{abs}}[Q][\rho] \subseteq \gamma(\mathcal{T}_{\text{abs}}[Q][\rho])\).
Alternatively, \( \mathcal{I}_{dep} \) on \( \rho \) computes concrete semantics of \( Q \) as \( \mathcal{I}_{dep}(Q, \rho) = \mathcal{I}_{dep}(A, \phi) \| \rho = (\rho_{\sigma}, \rho_{\phi}, \rho_{\mathcal{A}}) \). 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:

\[
A_{def}(Q, \rho_1) = \Delta(\rho_x, \rho_1) = (\rho_{\sigma_x}, \rho_{\phi}, \rho_{\mathcal{A}})
\]

Assuming the local correctness of \( \phi \) and \( A \), we get \( \rho_{\sigma} \subseteq \gamma(\rho_{\sigma}) \) and \( \rho_{\phi} \subseteq \gamma(\rho_{\phi}) \) respectively. Therefore, \( \gamma(A_{def}(Q, \rho)) \supseteq A_{use}(Q, \rho) \).

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

**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 \( \overline{p} \) and \( \overline{p}' \) be the abstract states at \( Q_1 \) and \( Q_2 \) respectively. The computation of semantic independency is sound if

\[
\forall X \in \gamma(A_{def}(Q_1, \overline{p})), \forall Y \in \gamma(A_{use}(Q_2, \overline{p}')):
X \cap Y \subseteq \gamma(A_{def}(Q_1, \overline{p})) \cap \gamma(A_{use}(Q_2, \overline{p}'))
\]

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

**Proof 6.** Consider two database statements \( Q_1 \) and \( Q_2 \). Let \( \overline{p} = (\rho_{\sigma}, \rho_{\phi}, \rho_{\mathcal{A}}) \) and \( \overline{p}' = (\rho'_{\sigma}, \rho'_{\phi}, \rho'_{\mathcal{A}}) \) 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 \( A_{def}(Q_1, \overline{p}) = (\rho_{\sigma}, \rho_{\phi}, \rho_{\mathcal{A}}) \) and \( A_{use}(Q_2, \overline{p}') = (\rho'_{\sigma}, \rho'_{\phi}, \rho'_{\mathcal{A}}) \) respectively.

Now, the semantics independency in abstract domain can be defined as \( \rho_{\sigma} \cap \rho'_{\sigma} = \emptyset \) \( \land \rho_{\phi} \cap \rho'_{\phi} = \emptyset \). Given the concrete states \( \rho = (\rho_{\sigma_{\mathcal{A}}}, \rho_{\phi}, \rho_{\mathcal{A}}) \) and \( \rho' = (\rho'_{\sigma_{\mathcal{A}}}, \rho'_{\phi}, \rho'_{\mathcal{A}}) \) where \( \rho \in \gamma(\overline{p}) \) and \( \rho' \in \gamma(\overline{p}') \), the semantics independency between \( Q_1 \) and \( Q_2 \) in the concrete domain is defined as \( \rho_{\sigma} \cap \rho'_{\sigma} = \emptyset \) \( \land \rho_{\phi} \cap \rho'_{\phi} = \emptyset \). From lemma 3 and 4, we get \( \gamma(A_{def}(Q_1, \overline{p})) \supseteq A_{def}(Q_1, \rho) \) and \( \gamma(A_{use}(Q_2, \overline{p}')) \supseteq A_{use}(Q_2, \rho') \) respectively. This implies that \( \gamma(A_{def}(Q_1, \overline{p}) \cap A_{use}(Q_2, \overline{p}')) \supseteq A_{def}(Q_1, \rho) \cap A_{use}(Q_2, \rho') \). Therefore, \( A_{def}(Q_1, \overline{p}) \cap A_{use}(Q_2, \overline{p}') = \emptyset \) \( \Rightarrow X \cap Y = \emptyset \) where \( X = A_{def}(Q_1, \rho) \) \( \in \gamma(A_{def}(Q_1, \overline{p})) \) and \( Y = A_{use}(Q_2, \rho') \) \( \in \gamma(A_{use}(Q_2, \overline{p}')). \)

9 Implementation and Experimental Evaluation

We have implemented a prototype tool \( \text{SemDDA} \) – 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].

The aim of designing \( \text{SemDDA} \) is to provide a user-friendly interface for the users to compute both syntax and semantic-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 \( \text{SemSSA} \).

(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:

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

The equivalent SSA form of these statements are:

\[ Q_1 : \text{UPDATE hra2 = hra1 + 100 WHERE da1 + hra1} \geq 1000 \]
\[ Q_2 : \text{SELECT hra2 FROM emp WHERE da1 + hra2} \leq 5000 \]

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', 'EditorialRecord', '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 different 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 11: Description of the benchmark programs [37]

<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>9</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 12: DD-dependency results in various approaches (NCLOC denotes Non-Comment Lines of Code)

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.

Table 13 reports the execution time (in milliseconds) of the analysis in the interval, octagon, polyhedra and powerset of intervals abstract domains. This is to mention that we do not observe any notable variation in the execution time across multiple trials. The variation of execution time for various benchmarks is depicted in Figure 15. Observe that we represent along y-axis the execution time (in milliseconds) in $\log_{10}$ scale, as the data range over several orders of magnitude. The reason behind the massive growth of execution time in polyhedra domain for 'EmpsRecord' and 'LedgerRecord' is exponential time complexity of the analysis w.r.t. the number of attributes (as reported in Table 12).

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 $\psi = \langle 16, \text{point} \rangle$ 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: $5 \rightarrow 6$, $4 \rightarrow 11$, $5 \rightarrow 11$, $6 \rightarrow 11$, $4 \rightarrow 15$, $5 \rightarrow 15$, $6 \rightarrow 15$, $4 \rightarrow 16$, $5 \rightarrow 16$, $6 \rightarrow 16$, $15 \rightarrow 16$. 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.
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 (CDG) 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 dependencies: Program-Database and Database-Database dependencies. 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], [39] 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 $Q_1$ is contained in a query $Q_2$ if and only if the result of applying $Q_1$ to any database $D$ is contained in the result of applying $Q_2$ to the same database $D$. Formally, a query $Q_1$ is said to be contained in a query $Q_2$, denoted $Q_1 \subseteq Q_2 \iff \forall D \left( Q_1(D) \subseteq Q_2(D) \right)$. 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 FileràSicura, Italy.

References


