Semantics-based Dependency Analysis of Database Applications by Abstract Interpretation

Submitted in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

by

Angshuman Jana
(1421CS06)

Under the supervision of

Dr. Raju Halder

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PATNA
PATNA - 801 106, INDIA
June 2019

Dedicated To Our Nation

APPROVAL OF THE VIVA-VOCE BOARD

Certified that the thesis entitled **"Semantics-based Dependency Analysis of Database Applications by Abstract Interpretation"** submitted by **Angshuman Jana** to the Indian Institute of Technology Patna for the award of the degree Doctor of Philosophy has been accepted by the external examiners and that the student has successfully defended the thesis in the Viva-Voce examination held today (21/06/2019).

DECLARATION

I certify that:

- The work contained in this thesis is original and has been done by me under the guidance of my supervisors.
- The work has not been submitted to any other Institute for any degree or diploma.
- I have followed the guidelines provided by the Institute in preparing the thesis.
- I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.
- Whenever I have used materials (data, theory and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the reference section.

Signature of the Student

CERTIFICATE

This is to certify that the thesis entitled **“Semantics-based Dependency Analysis of Database Applications by Abstract Interpretation"**, submitted by **Angshuman Jana** to Indian Institute of Technology, Patna, is a record of bonafide research work under my supervision and I consider it worthy of consideration for the degree of Doctor of Philosophy of the Institute.

Dr. Raju Halder
Dept. of Computer Science & Engineering,
Indian Institute of Technology Patna

Place: I.I.T., Patna
Date: 21st June 2019

Acknowledgment

I would like to express my profound and heartfelt gratitude for my esteemed guide and mentor, Dr. Raju Halder for his ever-present blessings, valuable advice, support, best wishes and encouragement all along the course of this work. I would especially like to thank Prof. Agostino Cortesi for his valuable suggestions during the entire time period. I am also grateful to the members of my Doctoral Committee (Dr. Samrat Mondal, Dr. Jimson Mathew, Dr. Arijit Mondal, Dr. Sudhan Majhi) for their valuable comments and suggestions at every phase of the Ph.D. program.

I would give special thanks to all my family members for supporting me during my Ph.D. Last but not least I thank IIT Patna and MHRD for providing me a research platform, research facilities, and financial support.

Place: I.I.T., Patna
Date: 21st June 2019.

Angshuman Jana

Abstract

Since the pioneer work by Ottenstein and Ottenstein, the notion of Program Dependency Graph (PDG) has attracted a wide variety of compelling applications in software engineering, e.g. program slicing, information flow security analysis, debugging, code-optimization, code-reuse, code-understanding, and many more. Since its inception, a number of variants are also proposed for various programming languages and features, possibly tuning them towards their suitable application domains. In order to exploit the power of dependency graph in solving problems related to relational database applications, Willmor et al. first proposed Database Oriented Program Dependency Graph (DOPDG), an extension of PDG by taking database statements and their dependencies further into consideration. However, we observed that the dependency information generated by the DOPDG construction algorithm is prone to imprecision primarily due to its (partially) syntax-based computation and flow insensitivity, and therefore the approach may increase the susceptibility of false alarms in the above-mentioned application scenarios. Unfortunately, since then no significant contribution is found in this research direction. As the values of database attributes differ from that of imperative language variables, the computation of 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 thesis, we aim to answer the following two main research objectives: (1) How to obtain more precise dependency information (hence more precise DOPDG)? and (2) How to compute them efficiently? To this aim, we define a sound semantics approximation of database applications embedding either Structured Query Language (SQL) or Hibernate Query Language (HQL) by the Abstract Interpretation framework. Considering this as the underlying basis, we instantiate semantics-based dependency computation in various relational and non-relational abstract domains, yielding to a detailed comparative analysis with respect to precision and efficiency. Notably, this framework is powerful enough to provide solution even in the case of undecidable scenario when no initial database state is given. We develop a prototype semDDA, **sem**antics-based **D**atabase **D**ependency **A**nalyzer, 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. Finally, we demonstrate the effect of this semantics-driven dependency refinement on database code slicing and information leakage analysis, as two case studies.

Keywords: Dependency Graph, Static Analysis, Abstract Interpretation, Relational Databases, Structured Query Languages, Hibernate Query Languages, Code Slicing, Information Flow Analysis.

List of Tables

List of Figures

Contents

CHAPTER 1

Introduction

Preface

This chapter starts with a brief introduction on the role of database applications in information systems scenarios and the need of dependency analysis as a way to solve various software engineering problems related to database applications. Then we highlight the motivational factors, possible challenges, and chapter-wise contributions.

Over the decades, database applications are playing a pivotal role in every aspect of our daily lives by providing an easy interface to store, access and process crucial data with the help of Database Management System (DBMS). Their existence are realized everywhere, ranging from simple web applications to the critical systems like banking, health-care, etc. Usually database applications are often written in popular host programming languages such as C, C++, C#, Java, etc., with embedded data access logic expressed declaratively in Structured Query Language (SQL) [38,43,47]. Furthermore, researchers have given enough efforts to minimize paradigm mismatch when database statements are embedded in other host languages. Hibernate Query Language (HQL) is one such instance which remedies the paradigm mismatch between object-oriented languages and relational database models and provides a unified platform for software developers to develop database applications without knowing much details about the underlying databases [11,12,42]. Various methods in "`Session`" are used to propagate object's states from memory to the database (or vice versa) and to synchronize both states when a change is made to persistent objects.

Static program 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 [97]. Over the past several decades, continuous and concerted research efforts in this direction make them powerful enough to solve many non-trivial questions about program's behavior, although they are undecidable in practice [81,97]. Some notable and widely used static analysis techniques include Data-flow analysis [104,114], Control-flow analysis [3], Type-based Theory [98,101,115], Abstract Interpretation [30,31], etc.

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 [59], taint analysis [79], program slicing [116], optimization [17,45], code-reuse [72], code-understanding [102]. A most common representation of these dependencies is *Dependency Graph* [65,100], an intermediate form of programs which consists of both data- and control-dependencies among program components. Since the pioneer work by Ottenstein and Ottenstein [100], a number of variants of dependency graph are proposed for various programming languages with a possible tuning towards their suitable application domains. They are Program Dependency Graph (PDG) for intra-procedural programs [100], System Dependency Graph (SDG) for inter-procedural programs [64], Class Dependency Graph (ClDG) for object-oriented programs [83], Database-Oriented Program Dependency Graph (DOPDG) for database programs [121].

Although dependency analysis has been thoroughly 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. [121] 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 [120] is a well-known static analysis technique to address many software-engineering problems, including code understanding, debugging, maintenance, testing, parallelization, integration, software measurement [78, 82, 102]. 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 [53] 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 [103] 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 [54, 55]. 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 [23] is an 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 [111]. 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 contributions in this thesis on semantics-driven database dependency analyzer meet 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 1.1, which increases salary of all employees by a common bonus amount *Cbonus* and by an additional special bonus amount *Sbonus* only for aged employees. Observe that the syntactic presence of the attribute *sal* as the defined-variable in Q_1 and as the used-variable in Q_3 makes Q_3 syntactically dependent on Q_1. However, a careful observation reveals that syntactic presence of variables as a way of dependency computation may often result in false positives, and thus fails to compute optimal set of dependencies. For instance, it is clear from the code that the values of *sal* referred in the `WHERE` clauses of Q_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

```
Start;
Q0: Connection c =DriverManager.getConnection(......);
Q1: UPDATE emp SET sal := sal + Sbonus WHERE age ⩾ 60;
Q2: SELECT AVG(sal) FROM emp WHERE  age ⩾ 60;
Q3: SELECT AVG(sal) FROM emp WHERE  age < 60
Q4: UPDATE emp SET sal := sal + Cbonus;
Q5: SELECT AVG(sal) FROM emp;
Stop;
```

Figure 1.1: An Introductory Example

statements in concrete domain is in general undecidable [70,81]. This is also true in the case of database applications when the input database instance is *unknown*. Addressing similar problems in imperative languages, Mastroeni and Zanardini [89] introduced the notion of abstract semantics-based data dependency in the Abstract Interpretation framework. Abstract Interpretation [30,31] is a widely used formal method which offers a sound approximation of the program's semantics to answer about the program's runtime behavior including undecidable ones. The intuition of Abstract Interpretation is to lift the concrete semantics to an abstract domain, by replacing concrete values by suitable properties of interests and simulating the operations in the abstract domain w.r.t. its concrete counterparts, in order to ensure sound semantic approximation.

Willmor's definition for DOPDG is not fully semantics-based [121]: 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 [10] 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, according to Willmor's definition of DD-dependency [121], Q_5 in Figure 1.1 is semantically independent on Q_1 as the part of *sal*-values defined by Q_1 is fully redefined by Q_4 and never reaches Q_5. Unfortunately, Condition-Action rules can not capture this independency as the approach checks every pair of database statements independently, and as a result, this finds dependency when the pair Q_1 and Q_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, we draw the following two main research objectives:

- How to obtain more precise dependency information of a database application (hence more precise DOPDG)? and
- How to compute them efficiently?

To achieve our objectives, in this thesis, we formalize the concrete and abstract semantics of database languages SQL and HQL, by extending the Abstract Interpretation framework. This computable abstract semantics serves as a powerful basis to design a static semantics-based dependency analyzer for database applications, resulting into a more precise dependency information by removing false alarms. This is also true for undecidable scenarios when the input database instance is unknown and presence of NULL value in the database.

To summarize, our contributions in this thesis are:

1. We define a sound semantics approximation of both SQL and HQL for static dependency analysis of database applications by the Abstract Interpretation framework.
2. We instantiate dependency computation in various relational and non-relational abstract domains, yielding to a detailed comparative analysis with respect to precision and efficiency.
3. We develop of a prototype semDDA, **sem**antics-based **D**atabase **D**ependency **A**nalyzer, integrated with various abstract domains which enables users to perform precise dependency computation in various abstract domains of interest.
4. Experimental evaluation on a set of open-source database-driven JSP web applications as part of the GotoCode project [1] using our semDDA tool. Experiments

demonstrate the results in different abstract domains with a detailed comparison on precision and efficiency. This clearly shows that our technique improves precision w.r.t. the proposal by Willmor et al. [121].

5. Finally, we demonstrate the effect of this semantics-driven dependency refinement on database code slicing and information leakage analysis, as two case studies.

The structure of the thesis is as follows:

Chapter 2: Preliminaries: Database Languages, Dependency Graphs and Abstract Interpretation
This chapter provides background details of the thesis. The content is divided into three main parts: (i) Database Languages [38,47], (ii) Dependency Graphs [64,83,100], and (iii) Abstract Interpretation Framework [30]. Since our proposals mainly refer to SQL and HQL, we describe their syntax and features as per ANSI standards [69], which are relevant to the subsequent chapters. We then recall various forms of dependency graphs pertaining different programming languages and their roles in software engineering activities. Finally, we provide a technical overview of the Abstract Interpretation framework, covering various relational and non-relational abstract domains.

Chapter 3: Concrete and Abstract Semantics of Structured Query Language (SQL)
In this chapter, we first recall the syntax and the concrete semantics of SQL from the literature [52]. Then we define an abstract semantics of SQL at various levels of abstractions, from non-relational to relational abstract domains such as domains of intervals, octagons, polyhedra and powerset of intervals.

Chapter 4: Concrete and Abstract Semantics of Hibernate Query Language (HQL)
As stated earlier, the thesis refers database applications embedding either SQL and HQL. Therefore, like chapter 3, we also define the formal syntax and concrete semantics of HQL, followed by its abstraction in various domains of interest. In particular, we refer various `session` methods which act as the central interface between an application and its underlying database.

Chapter 5: Semantic-based Dependency Computation of Database Applications

Syntax-based dependency computation often fails to generate an optimal set of dependencies, which increases the susceptibility of false alarms in many software engineering activities. This demands the need of semantics-based dependency computation taking into account variables values rather than their syntactic structures. This chapter is dedicated for this purpose. In particular, considering the previously defined abstract semantics of database applications as the underlying basis, we instantiate semantics-based dependency computation in various relational and non-relational abstract domains tunable with respect to the precision and efficiency. We develop a prototype semDDA, a semantics-based Database Dependency Analyzer, and we present experimental evaluation results in various abstract domains 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.

Chapter 6: Policy-based Database Code Slicing
Program slicing is a static analysis technique which is widely used in various software engineering activities, e.g. debugging, testing, code-understanding, code-optimization, etc. It extracts from programs a subset of statements which is relevant to a given behavior. In this chapter, we introduce a new form of code slicing, known as policy-based slicing, of database applications based on the refined notion of dependency graph. We show how the use of semantics-based dependency, together with semantic relevancy of statements, may improve the precision of the slice w.r.t. a given policy.

Chapter 7: Data Leakage Analysis of Database Applications
Language-based information-flow security analysis has emerged as a promising technique to detect possible information leakage in any software systems. Confidential data stored in an underlying database may be leaked to an unauthorized user due to improper coding of database applications. In this chapter, we extend the full power of the proposed model in [55] to the case of HQL, particularly by focussing on the `session` methods. We define the abstract semantics of HQL over the domain of propositional formulae by considering variables dependencies at each program point. This allows us to

identify illegitimate information flow by checking the satisfiability of propositional formulae with respect to a truth value assignment based on their security levels. Finally we explain how the reduced product of the analysis-results obtained from symbolic propositional formulae domain and numerical abstract domain may further improve the precision.

Chapter 8: Conclusions and Future Directions

In this chapter, we conclude our research works and highlight the possible future research scope.

CHAPTER 2

Preliminaries: Database Languages, Dependency Graphs and Abstract Interpretation

Preface

This chapter provides background details of the thesis. The content is divided into three main parts: (i) Database Languages [38, 47], (ii) Dependency Graphs [64, 83, 100], and (iii) Abstract Interpretation Framework [30]. Since our proposals mainly refer to SQL and HQL, we describe their syntax and features as per ANSI standards [69], which are relevant to the subsequent chapters. We then recall various forms of dependency graphs pertaining different programming languages and their roles in software engineering activities. Finally, we provide a technical overview of the Abstract Interpretation framework, covering various relational and non-relational abstract domains.

2.1 Databases Languages

The database technology is always at the heart of any information systems, facilitating one to store external data into persistent storage and to process them efficiently [47]. Even in the era of big data, a survey by TDWI in 2013 [107] says that, for a quarter of organizations, more than 20% of large volume of data are structured in nature and are stored in the form of relational database. Due to the structured form of stored data, relational database management systems gain immense popularity among the database community. A most common way to develop a database application is to embed relational database languages such as SQL, PL/SQL, HQL, etc., into other host languages like C, C++, Java, etc. [38,48].

Since this thesis primarily focuses database applications involving either SQL or HQL, let us briefly describe them.

2.1.1 Structured Query Language (SQL)

In 1970, IBM researchers developed Structured English QUery Language (SEQUL) for the purpose of manipulating and retrieving data stored in System R (relational database management system of IBM) [6,21]. Later on, the SEQUEL was renamed to Structured

```
SELECT [DISTINCT] < attribute list>
FROM (< table name> {< alias >} | < joined table >) {, (< table name > {< alias >} | < joined table >)}
[WHERE <condition>]
[ GROUP BY < grouping attributes > [ HAVING < group selection condition >]]
[ORDER BY < column name > [< order >] {, < column name > | [< order >] } ]
< attribute list > ::= (∗ | (< column name > | < function > (([DISTINCT] < column name >) | ∗))
                        { , (< column name > | < function > (([DISTINCT] < column name >) | ∗))})
< grouping attributes > ::= < column name > {, < column name >}
< order > ::= (ASC | DESC)
INSERT INTO < table name > [(< column name > {, < column name >})]
(VALUES (< constant value > {,< constant value >}) {, (< constant value > {,< constant value >})}
 | < select statement >)
DELETE FROM < table name >
[WHERE < selection condition >]

UPDATE < table name >
SET < column name >=< value expression > {, < column name >=< value expression >}
[WHERE < selection condition >]
```

Table 2.1: Summary of SQL DML Syntax [69]

Query Language (SQL) and it has been standardized by ANSI and ISO [91]. Today SQL is being used by several RDBMS implementation, like MySql, Oracle, DB2, Sybase, MS Access, etc. [50, 75, 87]. In general, SQL comprises of Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL). In addition, it also supports statements for constraint specification, schema evolution, security enforcement and other features [38]. This is noteworthy to mention that, in this thesis, we restrict our proposal to database applications embedding DML only. Table 2.1 summarizes a part of the DML syntax recalled from [69] for a quick reference.

2.1.2 Hibernate Query Language (HQL)

Although SQL is widely used in many database applications, it suffers from paradigm mismatch between host languages and relational data model. As a remedy of this, Hibernate Query Language (HQL) is introduced as a unified platform for software developers to develop database applications without knowing much details about the database [11,12,42]. More specifically, HQL remedies the paradigm mismatch between object-oriented languages and relational database models. The Object Relational Mapping (ORM) tool of the Hibernate framework automatically translates HQL queries into conventional SQL queries, thus simplifying the data creation, data manipulation and data access.

2.1.2.1 Hibernate Architecture

Different layers of hibernate architecture is depicted in Figure 2.1 [11]. Hibernate framework consists of mapping files and configuration files, along with other elements such as Session Factory, Session, etc.

Mapping file. This is an XML file which contains mapping from a Plain Old Java Object (POJO) class name and its fields to a database table name and its attributes respectively.

Configuration file. This is an XML file, named hibernate.cfg.xml, which consists of several parameters like database name, driver name, user id, password, mapping file name, etc. This is to note that only one configuration file is created for each database.

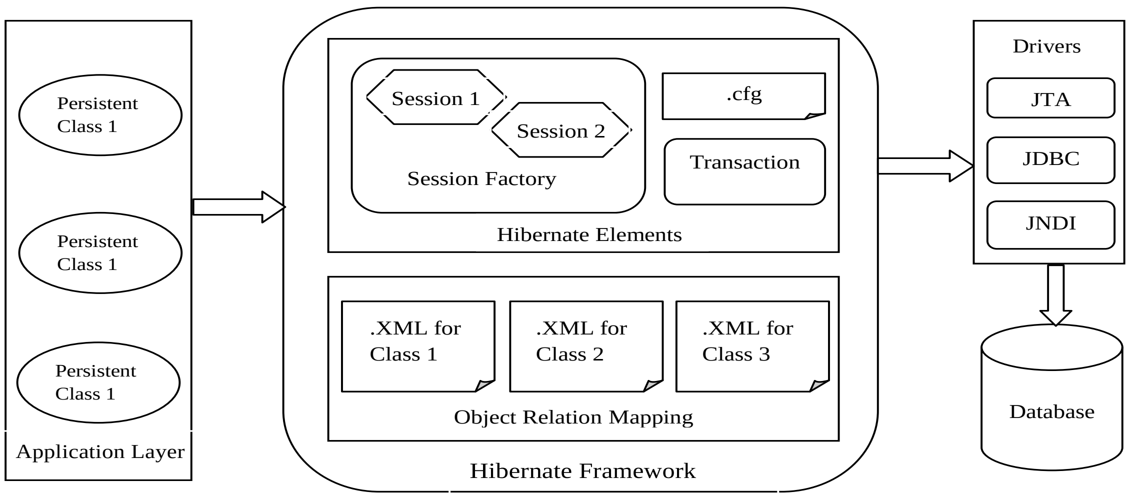


Figure 2.1: Layered form of Hibernate Architecture [99]

Session Factory. This is a heavyweight thread-safe object, which is created by providing a configuration object. Only one session factory is created for each database.

Session. Session objects are used to get a physical connection with a database. It is basically a thread of Session Factory whose responsibilities are to make objects into a persistent form and to store (or retrieve) those persistent objects into (or from) databases. Multiple sessions are used to handle multiple clients' requests. Session consists of several methods which are use to convert HQL into possibly multiple SQL statements.

2.1.2.2 Hibernate Properties

Hibernate creates a POJO class using the following four properties [99]:

- To be persistent, a class must have a default constructor.
- Class should contain an ID in order to allow easy identification of objects within the Hibernate and the database.
- All fields in a class will be declared as private and the class contains public methods **getXXX()** and **setXXX()**, where **getXXX()** has a return type without argument and **setXXX()** accept arguments without any return type.
- Non-final class is Preferable.

In addition, Hibernate system should maintain the followings:

- Different XML files for each persistent class acts as ORM in Hibernate.
- For a particular database only one cfg file is configured.
- There is one service class which consists of **main()** method.

2.1.2.3 Operations in Hibernate Query Language

Let us explain the HQL operations using suitable examples depicted in Figure 2.2. The POJO class `stud` in Figure 2.2(a) contains three private fields *id*, *courseid* and *mark*, and default public methods **getXXX()** and **setXXX()** corresponding to each attribute. Figures 2.2(b) – 2.2(f) depict various database operations Insert, Update, Delete, Select with and without condition respectively. The first statement in each operation creates a session object through which various methods (save(), createQuery(), beginTransaction(), etc.) are invoked. Observe that save() is used to store a `stud` object information into the underlying database table which corresponds to the `stud` class (defined in XML mapping file). The session method createQuery() is used to create database statements for update, delete and select, whereas executeUpdate() and list() methods are used to execute them. The symbol ':' prefixed with a variable indicates that the actual value of this variable will be substituted at runtime using the setParameter() method. Observe that all these database operations will be committed through a Transaction object created using the session method beginTransaction().

2.2 Dependency Graphs and their role in Software Engineering

Dependency Graph [45,100] is an intermediate representation of program which explicits both the data- and control-dependencies among program statements. This provides the basis for powerful programming tool to address a large number of software engineering activities [23, 45, 53, 59, 64, 73, 100, 103]. In this section, we briefly discuss the evolution of dependency graphs and their applications in the field of software engineering.

```
class stud {
 private int id;
 private int courseid;
 private int mark;

 stud() { }

 public int getId() { return id;}
 public void setId(int id) { this.id = id;}
 public String getcourseid() { return courseid;}
 public void setcourseid(int courseid){ this.courseid = courseid;}
 public String getMark() { return mark;}
 public void setMark(int mark) { this.mark = mark;}
}
```

(a) POJO class stud

```
1.   Session session=getSessionFactory().openSession();
2.   Transaction tx=session.beginTransaction();
3.   stud s1=new stud();
4.   s1.setcourseid(1001);
5.   s1.setmark(500);
6.   session.save(s1);
7.   tx.commit();
8.   session.close();
```

(b) Insert

```
1.   Session session = getSessionFactory().openSession();
2.   Transaction tx =session.beginTransaction();
3.   Query query = session.createQuery("UPDATE stud SET mark =: mark WHERE id =: id");
4.   query.setParameter("mark", 700);
5.   query.setParameter("id", 1);
6.   int result = query.executeUpdate();
7.   tx.commit();
8.   session.close();
```

(c) Update

```
1.   Session session = getSessionFactory().openSession();
2.   Transaction tx =session.beginTransaction();
3.   Query query = session.createQuery("DELETE FROM stud WHERE mark < 500 AND courseid = 1001");
4.   int result = query.executeUpdate();
5.   tx.commit();
6.   session.close();
```

(d) Delete

```
1.   Session session = getSessionFactory().openSession();
2.   Transaction tx =session.beginTransaction();
3.   Query query = session.createQuery("FROM stud");
4.   List result = query.list();
5.   tx.commit();
6.   session.close();
```

(e) Select without condition

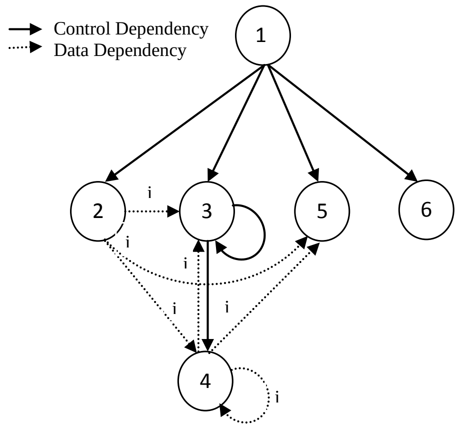
```
1.   Session session = getSessionFactory().openSession();
2.   Transaction tx =session.beginTransaction();
3.   Query query = session.createQuery(" SELECT stud.courseid FROM stud WHERE stud.id > 10 GROUP
     BY stud.mark HAVING MAX(stud.mark) < 700 ORDER BY stud.mark");
4.   List result = query.list();
5.   tx.commit();
6.   session.close();
```

(f) Select with condition

Figure 2.2: Operations of HQL

```
1.  void main() {
2.      int i=1;
3.      while(i<11) {
4.          i=i+1; }
5.      printf("%d", i);
6.  }
```

(a) A code snippet `ExProg`



(b) PDG of `ExProg`

Figure 2.3: An example code snippet and its PDG.

2.2.1 Dependency Graphs

In this section, we recall from the literature a number of variants of dependency graph, such as Program Dependency Graph (PDG) [100], System Dependency Graph (SDG) [64], Class Dependency Graph (ClDG) [83], Database-Oriented Program Dependency Graph (DOPDG) [121], etc., which are proposed for different programming languages and features since the pioneer work by Ottenstein and Ottenstein in 1984 [45].

2.2.1.1 Program Dependency Graph (PDG)

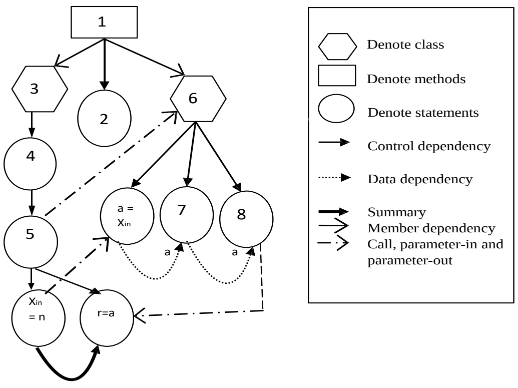
Program Dependency Graph (PDG) [100] is an intermediate representation of programs where nodes represent program statements and edges represent data- and control-dependencies between the statements. However, unlike control-flow graph, the control-dependencies in PDG do not represent any execution-order of the program.

Given two statements S_1 and S_2, the control and data dependencies are defined as follows:

Definition 2.1 (Control Dependency) *The statement S_2 is said to be control dependent on another statement S_1 iff: (i) There exists a path π from S_1 to S_2 such that every statement $S_i \in \pi - \{S_1, S_2\}$ is post-dominated (node n post-dominates node m if every path from node m to the end node e passes through n) by S_2, and (ii) S_2 does not post-dominate S_1.*

```
1.  class X {
2.     int r;
3.     void incr( n ) {
4.        if(r < n) {
5.           r = add(n); }
         }
6.     int add(int a) {
7.        a = a + 1;
8.        return(a); } }
```

(a) class X



(b) SDG of X

Figure 2.4: SDG for inter-procedural dependencies.

Definition 2.2 (Data Dependency) *The statement S_2 is data-dependent on another statement S_1 if there exists some variable x such that: (i) x is defined by S_1, (ii) x is used by S_2, and (iii) there is a x-definition free path from S_2 to S_1.*

Observe that above definition of data dependency is based on the syntactic presence of *"used"* and *"defined"* variables in statements.

Example 1 *Consider the program* `ExProg` *in Figure 2.3(a). The PDG of* `ExProg` *is depicted in Figure 2.3(b). The control-dependencies between program statements (denoted by solid-line) are computed by following the definition 2.1. For instance, the edges* $1 \rightarrow 2, 1 \rightarrow 3, 3 \rightarrow 4$, *etc. represent control dependencies. The data-dependencies (denoted by dotted-line) are computed based on "used" and "defined" variables information in the statements. For instance, the statement* 3 *is data-dependent on statement* 2 *(denoted* $2 \rightarrow 3$*), as the statement* 2 *defines the variable i which is then used by statement* 3. *Similarly, other data-dependencies* $2 \rightarrow 4, 2 \rightarrow 5, 4 \rightarrow 3, 4 \rightarrow 4$ *and* $4 \rightarrow 5$ *are computed.*

2.2.1.2 System Dependency Graph (SDG)

Due to the presence of procedural call in programs, the notion of System Dependency Graph (SDG) [64] is introduced. For each procedure call, nodes corresponding to the calling- and called-procedures are associated with parameter-in and parameter-out nodes. The parameter-in nodes of a calling procedure are then connected by dependency edges with the parameter-in nodes of its corresponding called-procedure. Similar is followed between parameter-out nodes of a calling- and called-procedure nodes, but in a reverse direction.

Example 2 *Consider the program in Figure 2.4(a). Observe that the method* `incr()` *calls another method* `add()` *within the same class* `X`*. The system dependency graph is depicted in Figure 2.4(b). Observe that the long-dash-dotted edges represent dependencies between parameter-in node (labelled by 'x_{in}=n') of calling method (node 5) and parameter-in node (labelled by 'a=x_{in}') of called-method (node 6). Similar is done for parameter-out nodes of 5 and 6 (labelled by '8' and 'r=a' respectively) in reverse direction. The solid-bold edge represents the summary edge which is connected from parameter-in to parameter-out nodes within the same method to model the transitive flow of the dependencies across the method call.*

2.2.1.3 Class Dependency Graph (ClDG)

The Class Dependency Graph (ClDG) [83] is an extension of SDG for Object-Oriented Programming (OOP) languages. This is featured by the following types of dependencies:

(a) Intra-class Intra-method Dependency: This represents the dependencies within the same method of a class, and it follows the PDG-based approach.

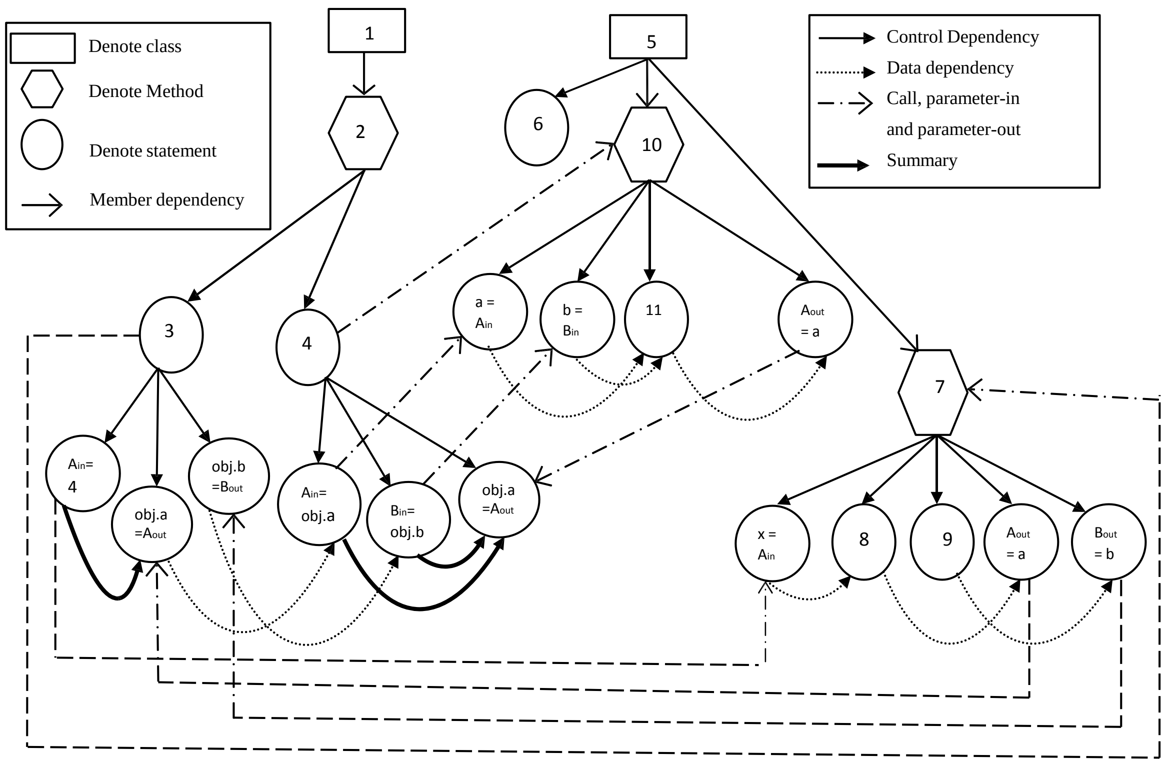
(b) Intra-class Inter-method Dependency: The dependencies between two different methods within the same class is constructed by following the similar approach as in the case of SDG.

(c) Inter-class Inter-method Dependency: Inter-class Inter-method dependencies occur in OOP when a method in one class calls another method in other class. This is done by calling the method through an object of the called-class. Therefore, additional in-parameters corresponding to the object-fields through which the method is called, must be considered. Note that, in this scenario, a constructor-call during object creation is also a part of the graph which follows the same representation as of other inter-class inter-method calls.

Example 3 *Consider the example in Figure 2.5(a). The ClDG of the code is depicted in Figure 2.5(b), which represents the interaction between two methods* `main()` *in class* *`sample` and `mult()` in class `operation`. Observe that when 4 calls 10, the fields of obj (i.e., obj.a and obj.b) are associated with node 4 as in-parameters.*

```
 1.  class sample {
 2.     public static void main(String arg[]){
 3.            operation obj = new operation(4);
 4.            obj.mult(); } }

 5.  class operation {
 6.     int a, b ;
 7.     operation(int x){
 8.            a = x;
 9.            b = 2; }
10.     int mult(){
11.            a = a * b; } }
```

(a) Classes `sample` and operations



(b) ClDG for classes `sample` and `operation`

Figure 2.5: ClDG for inter-class inter-method dependencies.

2.2.1.4 Database-Oriented Program Dependency Graph (DOPDG)

Database-Oriented Program Dependency Graph (DOPDG) [121] is an extension of traditional PDG to the case of database applications. DOPDG considers two additional dependencies: (i) Program-Database dependency (PD-dependency) and (ii) Database-Database dependency (DD-dependency). A PD-dependency represents the dependency between a database statement and an imperative statement, whereas a DD-dependency represents the dependency between two database statements. Let us recall them below:

Definition 2.3 (Program-Database (PD) dependency [121]) *A database statement Q is PD dependent on an imperative statement I for a variable x (denoted $I \xrightarrow{x} Q$) if the following three hold: (i) x is defined by I, (ii) x is used by Q, and (iii) there is no redefinition of x between I and Q.*

The PD-dependency of I on Q is defined similarly.

Definition 2.4 (Database-Database (DD) dependency [121]) *Let $Q.SEL$, $Q.INS$, $Q.UPD$ and $Q.DEL$ denote the parts of database state which are selected, inserted, updated, and deleted respectively by Q. A database statement Q_1 is DD-dependent on another database statement Q_2 iff (i) the database-part defined by Q_2 overlaps the database-part used by Q_1, i.e. $Q_1.SEL \cap (Q_2.INS \cup Q_2.UPD \cup Q_2.DEL) \neq \emptyset$, and (ii) there is no roll-back operation in the execution path p between Q_2 and Q_1 (exclusive) which reverses back the effect of Q_2.*

Example 4 depicts the construction of DOPDG on an example code by following the algorithm proposed in [121].

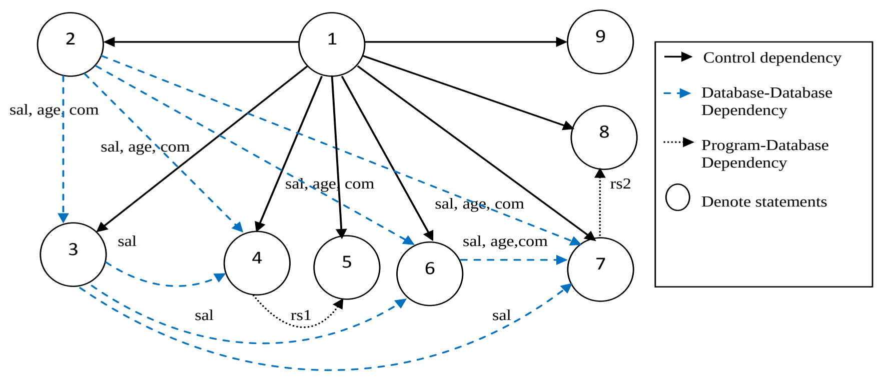
Example 4 *Consider the database application `DbProg` and the associated database `vendor` depicted in Figure 2.6. The DOPDG of `DbProg` is depicted in Figure 2.6(c). The control-dependencies between program statements are computed by following similar approach as in the case of traditional Program Dependency Graphs. For instance, the edges $1 \to 2$, $1 \to 3$, $1 \to 4$, etc., represent control dependencies. We construct DD- and PD-dependencies based on the algorithm proposed in [121]. For instance, edges $2 \to 3$, $2 \to 4$, $2 \to 6$, $2 \to 7$, $3 \to 4$, $3 \to 6$, $3 \to 7$ and $6 \to 7$ represent DD-dependencies (denoted by dashed-lines), whereas edges $4 \to 5$ and $7 \to 8$ represent PD-dependencies (denoted by dotted-line).*

```
1.  start;
2.  Stmt = DriverManager.getConnection(. . . . . . ).createStatement();
3.  Stmt.executeQuery("UPDATE emp SET sal=sal+100 WHERE (age + com) ⩾ 60");
4.  Resultset rs1=Stmt.executeQuery("SELECT avg(sal) from emp WHERE
    (age+com)⩽55");
5.  display(rs1);
6.  Stmt.executeQuery(" DELETE from emp WHERE age ⩾ 61");
7.  Resultset rs2 = Stmt.executeQuery("SELECT * from emp");
8.  display(rs2);
9.  stop;
```

(a) Program `DbProg`

sal	*age*	*com*
1500	35	10
800	28	20
2500	50	10
3000	62	10
2000	30	30
1600	42	20
1000	20	30

(b) Database Table `vendor`



(c) DOPDG of `DbProg`

Figure 2.6: DOPDG for an example database code

Over the past decades, a large number of algorithms using dependency graphs are proposed to address various software-engineering activities, including language-based information flow security analysis, program slicing, code-optimization, code-maintenance, code-understanding, data provenance, etc. [23,45,53,59,64,73,100,103].

2.3 Abstract Interpretation

The Abstract Interpretation theory is proposed by Patrick Cousot and Radhia Cousot in 1977 [30, 32, 33, 34, 35, 96]. It is a method of sound approximation of programs' concrete semantics which enables to provide answers to questions about programs' run-time behaviour (including undecidable ones). The core principle of the Abstract Interpretation theory is that all types of semantics, like operational, denotational, rule-based, axiomatic, etc., can be expressed as fixpoints of monotonic operators in partially ordered structure and can be lifted to an abstract setting by replacing concrete values with suitable properties of interest and simulating concrete operations by sound abstract operations. Moreover, it facilitates the proof of correctness of existing analyses and aids in the design of new analyses [34].

Let $\mathcal{D}$ and $\overline{\mathcal{D}}$ be the concrete and abstract domains respectively. Let us assume that both domains respect the stronger properties like Complete Partial Orderings (CPO) or complete lattices, defined below:

Complete Partial Order (CPO) [32]. A complete partial order $\langle \mathcal{X}, \sqsubseteq \rangle$ is a poset where the set $\mathcal{X}$ is equipped with an ordering relation $\sqsubseteq$ and satisfies the following property: every increasing chain $C \triangleq x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n$ of elements of $\mathcal{X}$ has least upper bound $\bigsqcup C$, which is called the limit of the chain. Note that, since the empty set $\emptyset$ is a chain, a complete partial order has a least element $\bot = \bigsqcup \emptyset$.

Lattice and Complete Lattice [33]. A lattice $\langle \mathcal{X}, \ \sqsubseteq, \sqcup, \sqcap \rangle$ is a poset where each pair of elements $x, y \in \mathcal{X}$ has a least upper bound denoted by $x \sqcup y$ and a greatest lower bound denoted by $x \sqcap y$. A complete lattice $\langle \mathcal{X}, \ \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ is a lattice where any subset $A \subseteq \mathcal{X}$ has a least upper bound $\bigsqcup A \triangleq \bigsqcap \{x \in \mathcal{X} \mid \forall y \in A, \ x \sqsupseteq y\}$ and a greatest lower bound $\bigsqcap A \triangleq \bigsqcup \{x \in \mathcal{X} \mid \forall y \in A, \ x \sqsubseteq y\}$. A complete lattice has both a least element $\bot \triangleq \bigsqcap \mathcal{X}$ and a greatest element $\top \triangleq \bigsqcup \mathcal{X}$. Note that, a complete lattice is always a CPO.

Fixpoint [31]. Given a partially ordered set $\langle \mathcal{X}, \sqsubseteq \rangle$ and a function $F : \mathcal{X} \longrightarrow \mathcal{X}$, a fixpoint of F is an element $y \in \mathcal{X}$ such that $F(y) = y$. A pre-fixpoint is an element $y \in \mathcal{X}$ such that $y \sqsubseteq F(y)$. Dually, $F(y) \sqsubseteq y$ is called post-fixpoint. The least fixpoint of F, denoted by lfp_z F which is greater than or equal to an element $z \in \mathcal{X}$. Dually, the greatest fixpoint of F is denoted by gfp_z F which is less than or equal to an element $z \in \mathcal{X}$. If the order $\sqsubseteq$ is not clear from the context then least and greatest fixpoint of F is written $\mathrm{lfp}_z^{\sqsubseteq}$ F and $\mathrm{gfp}_z^{\sqsubseteq}$ F respectively.

Abstract Interpretation establishes a correspondence between concrete and abstract domains in the form of Galois Connection in order to guarantee the correctness of analysis in the abstract domain.

Definition 2.5 (Galois Connections [30]) *Consider two partial order set $(\mathcal{D}, \leqslant)$ and $(\overline{\mathcal{D}}, \sqsubseteq)$ where the first one represents a concrete domain and the second one represents an abstract domain. Let α: $\mathcal{D} \longrightarrow \overline{\mathcal{D}}$ and γ: $\overline{\mathcal{D}} \longrightarrow \mathcal{D}$ be the abstraction and concretization functions* *respectively. The Galois Connection between $\mathcal{D}$ and $\overline{\mathcal{D}}$ (denoted by $\left\langle(\mathcal{D},\leqslant),\alpha,\gamma,(\overline{\mathcal{D}},\sqsubseteq)\right\rangle$ or $(\mathcal{D},\ \leqslant)\ \underset{\gamma}{\overset{\alpha}{\leftrightarrows}}\ (\overline{\mathcal{D}},\ \sqsubseteq)$) holds iff:*

- $\forall v \in \mathcal{D}.\ v \leqslant \gamma \circ \alpha(v)$.
- $\forall \overline{v} \in \overline{\mathcal{D}}.\ \alpha \circ \gamma(\overline{v}) \sqsubseteq \overline{v}$.
- *α and γ are monotonic.*

In other words, iff $\forall v \in \mathcal{D}, \overline{v} \in \overline{\mathcal{D}}.\ \alpha(v) \sqsubseteq \overline{v} \iff v \leqslant \gamma(\overline{v})$. Notice that for some abstract domains only a concretization function exists, like in the case of Polyhedra.

Formally, the concrete semantic domain $\mathbb{D}$ forms a complete lattice $\langle \mathbb{D}, \subseteq, \bot, \top, \cup, \cap \rangle$. On this domain, a semantics $\mathbf{S}[\![\mathcal{P}]\!]$ of syntactically correct program $\mathcal{P}$ in a given programming language is defined. Observe that the concrete semantics is usually a collecting interpretation. In the same way, an abstract semantics $\overline{\mathbf{S}}[\![\mathcal{P}]\!]$ is defined in an abstract domain $\overline{\mathbb{D}}$, aiming to approximate $\mathbf{S}[\![\mathcal{P}]\!]$ in a computable way. Formally, the abstract semantic domain $\overline{\mathbb{D}}$ has to form a complete lattice $\langle \overline{\mathbb{D}}, \sqsubseteq, \overline{\bot}, \overline{\top}, \sqcup, \sqcap \rangle$. Given a Galois connections $\left\langle(\mathbb{D},\leqslant),\alpha,\gamma,(\overline{\mathbb{D}},\sqsubseteq)\right\rangle$, we say that $\overline{\mathbf{S}}[\![\mathcal{P}]\!]$ is a sound approximation of $\mathbf{S}[\![\mathcal{P}]\!]$ when $\mathbf{S}[\![\mathcal{P}]\!] \subseteq \gamma(\overline{\mathbf{S}}[\![\mathcal{P}]\!])$.

Let $\left\langle(\mathcal{D},\leqslant),\alpha,\gamma,(\overline{\mathcal{D}},\sqsubseteq)\right\rangle$ be a Galois connection, $F : \mathcal{D} \to \mathcal{D}$ be a concrete function and $\overline{F} : \mathcal{D} \to \mathcal{D}$ be an abstract function. $\overline{F}$ is said to be a sound or correct approximation of F iff $F \circ \gamma \sqsubseteq \gamma \circ \overline{F}$. On the other hand, $\overline{F}$ is a complete approximation of F iff $F \circ \gamma = \gamma \circ \overline{F}$, which indicates that no loss of precision is accumulated in the abstract computation through $\overline{F}$. Similarly in case of iterative function, lfp $\overline{F}$ is a sound approximation of lfp F iff lfp $F \sqsubseteq \gamma($lfp $\overline{F})$. Observe that, if the abstract domain respects the ascending chain condition then the computation is guaranteed to terminate. Otherwise the sequence convergence must be assured using a widening operator [86].

A number of abstract domains, non-relational and relational, exist in the literature [30,31,36,93,94]. Let us briefly illustrate them below:

2.3.1 Non-relational Abstract Domains

An abstract domain is said to be non-relational if it does not preserve any relation among program variables. Non-relational abstract domains care only about the actual variables being updated, rather than having potential to change multiple values at once [31]. Some widely used non-relational abstract domains for program analysis include sign domain for sign property analysis, parity domain for parity property analysis, interval domain for division-by-zero or overflows [30]. Analyses in these domains are, although efficient, but imprecise w.r.t. relational abstract domains. Figure 2.7 pictorially depicts a scenario where a set of points SP (indicated by •) on the *xy*-plane are abstracted by sign and interval properties.

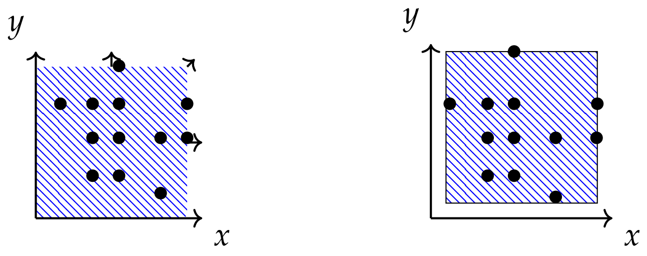


Figure 2.7: Abstractions of SP by Sign (left) and Interval Properties (right)

Let us now describe briefly about the sign and interval domains in the sections below.

2.3.1.1 Sign Domain

Let $\mathsf{L}_c = \langle \wp(\mathbb{R}), \subseteq, \emptyset, \mathbb{R}, \cap, \cup \rangle$ be a concrete lattice of the powerset of numerical values $\mathbb{R}$. Given an abstract domain SIGN$=\{\top, +, -, 0, \pm, 0+, 0-, \bot\}$ representing sign properties of numerical values, let $\mathsf{L}_a = \langle \text{SIGN}, \sqsubseteq, \bot, \top, \sqcap, \sqcup \rangle$ be an abstract lattice.

The correspondence between L_c and L_a is formalized as the Galois connection $\langle \mathsf{L}_c, \alpha_s, \gamma_s, \mathsf{L}_a \rangle$ which is depicted below:

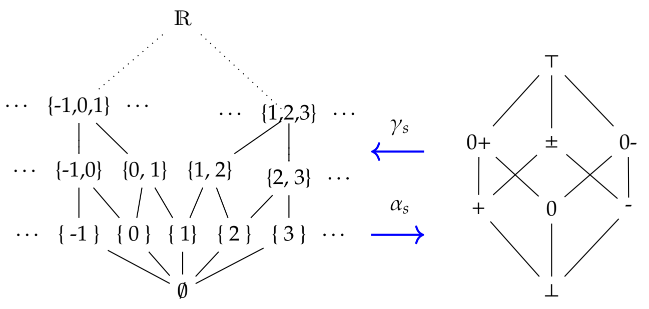


Figure 2.8: Galois Connection between L_c and L_a

The abstraction function α_s and concretization function γ_s between the domains are defined below:

$$\forall S \in \wp(\mathbb{R}) : \ \alpha_s(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ + & \text{if } S = \{\, a \mid a > 0 \,\} \\ 0 & \text{if } S = \{\, 0 \,\} \\ - & \text{if } S = \{\, a \mid a < 0 \,\} \\ \pm & \text{if } S = \{\, a \mid a > 0 \ \vee a < 0 \,\} \\ 0+ & \text{if } S = \{\, a \mid a \geqslant 0 \,\} \\ 0- & \text{if } S = \{\, a \mid a \leqslant 0 \,\} \\ \top & \text{Otherwise} \end{cases}$$

$$\forall \overline{v} \in \text{SIGN}: \gamma_s(\overline{v}) = \begin{cases} \emptyset & \text{if } \overline{v} = \bot \\ \{k \in \mathbb{R} \mid \ k > 0\} & \text{if } \overline{v} = + \\ \{0\} & \text{if } \overline{v} = 0 \\ \{k \in \mathbb{R} \mid \ k < 0\} & \text{if } \overline{v} = - \\ \{k \in \mathbb{R} \mid \ k > 0 \vee k < 0\} & \text{if } \overline{v} = \pm \\ \{k \in \mathbb{R} \mid \ k \geqslant 0\} & \text{if } \overline{v} = 0+ \\ \{k \in \mathbb{R} \mid \ k \leqslant 0\} & \text{if } \overline{v} = 0- \\ \mathbb{R} & \text{Otherwise} \end{cases}$$

The arithmetic operations over the abstract domain are defined accordingly, ensuring the soundness w.r.t. their concrete counter-part [31]. For example, the '$\times$' operation over the concrete domain is mapped to its abstract version '$\overline{\times}$' as follows: $-(\overline{\times})- = +$, $+(\overline{\times})- = -$, $+(\overline{\times})+ = +$, $\top(\overline{\times})+ = \top$, $\bot(\overline{\times})+ = \bot$, and so on.

2.3.1.2 Domain of Intervals

In the interval domain, a set of integers is approximated by a pair $[l, h]$ where l and h represent minimal and maximal element of the set respectively. For example, {2, 1, 100, 4} is represented by [1, 100].

Let $\mathsf{L}_c = \langle \wp(\mathbb{R}), \subseteq, \emptyset, \mathbb{R}, \cap, \cup \rangle$ be a concrete lattice of the powerset of numerical values $\mathbb{R}$. Let $\mathbb{I} = \{[l,\ h] \mid l \in \mathbb{R} \cup \{-\infty\},\ h \in \mathbb{R} \cup \{+\infty\},\ l \leq h\} \cup \bot$ be the abstract domain of intervals forming an abstract lattice $\mathsf{L}_a = \langle \mathbb{I}, \sqsubseteq, \bot, [-\infty, +\infty], \sqcap, \sqcup \rangle$, such that:

- $[l_1,\ h_1] \sqsubseteq [l_2,\ h_2] \iff l_2 \leqslant l_1 \wedge h_2 \geqslant h_1$
- $[l_1,\ h_1] \sqcap [l_2,\ h_2] = [max(l_1\ l_2),\ min(h_1\ h_2)]$
- $[l_1,\ h_1] \sqcup [l_2,\ h_2] = [min(l_1,\ l_2),\ max(h_1\ h_2)]$

The correspondence between L_c and L_a is formalized as the Galois connection $\langle \mathsf{L}_c, \alpha_{\mathbb{I}},\ \gamma_{\mathbb{I}}, \mathsf{L}_a \rangle$ where $\forall S \in \wp(\mathbb{R})$ and $\forall \overline{v} \in \overline{\mathbb{I}}$:

$$\alpha_{\mathbb{I}}(S) = \begin{cases} \bot & \text{if } S = \emptyset \\ [l, h] & \text{if } inf(S) = l\ \wedge\ sup(S) = h \\ [-\infty, h] & \text{if } \nexists inf(S)\ \wedge\ sup(S) = h \\ [l, +\infty] & \text{if } inf(S) = l\ \wedge\ \nexists sup(S) \\ [+\infty,\ -\infty] & \text{if } \nexists inf(S)\ \wedge\ \nexists sup(S); \end{cases}$$

$$\gamma_{\mathbb{I}}(\overline{v}) = \begin{cases} \emptyset & \text{if } \overline{v} = \bot \\ \{k \in \mathbb{R} \mid\ l \leq k \leq h\} & \text{if } \overline{v} = [l,\ h] \\ \{k \in \mathbb{R} \mid\ k \leq h\} & \text{if } \overline{v} = [-\infty,\ h] \\ \{k \in \mathbb{R} \mid\ l \leq k\} & \text{if } \overline{v} = [l,\ +\infty] \\ \mathbb{R} & \text{if } \overline{v} = [+\infty,\ -\infty]. \end{cases}$$

The pictorial representation of the Galois connections $\langle \mathsf{L}_c, \alpha_{\mathbb{I}},\ \gamma_{\mathbb{I}}, \mathsf{L}_a \rangle$ is shown in Figure 2.9.

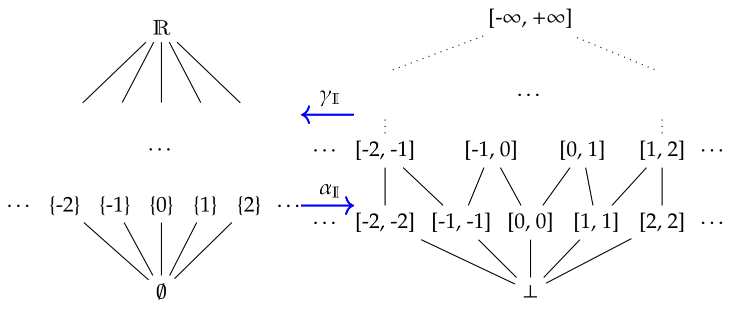


Figure 2.9: Galois Connection between L_c and L_a

2.3.2 Relational Abstract Domains

Unlike non-relational abstract domains, the relational abstract domains preserve relations among program variables [36]. 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 [36,93,94]. Abstractions of the same set of points in the octagon and polyhedra domains are exemplified in Figure 2.10.

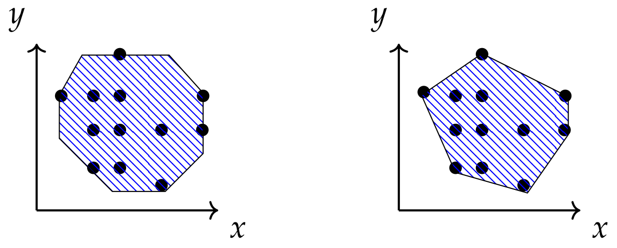


Figure 2.10: Abstractions of SP in Octagon (left) and Polyhedra Domains (right)

Let us now describe briefly about the octagon and polyhedra domains in the sections below.

2.3.2.1 Relational Abstract Domain of Octagons

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

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

Difference Bound Matrices (DBM) [93]. Given a program $\mathcal{P}$ with a finite set of variables $\mathbb{V}_{\mathcal{P}} = \{x_1, \ldots, x_n\}$. A Difference Bound Matrix (DBM) m with size $n \times n$ represents a set of invariants each of the form $x_j - x_i \leqslant k$, where $k \in \mathbb{R}_\infty$ and $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$ such that:

$$\mathsf{m}_{ij} \triangleq \begin{cases} k & \text{if } (x_j - x_i \leqslant k) \text{ where } x_i, x_j \in \mathbb{V}_{\mathcal{P}} \text{ and } k \in \mathbb{R}_\infty, \\ \\ \infty & \text{otherwise.} \end{cases}$$

Example 5 *Consider the constraints* $\{x_1 - x_2 \leqslant 3, x_2 - x_3 \leqslant 4, x_3 - x_1 \leqslant 5, x_2 - x_4 \leqslant 4\}$. *These constraints are represented by the DBM shown below:*

	x_1	x_2	x_3	x_4
x_1	∞	∞	5	∞
x_2	3	∞	∞	∞
x_3	∞	4	∞	∞
x_4	∞	4	∞	∞

Extension to encode octagonal constraints [94]. The above DBM representation over program variables can represent only a subset of octagonal constraints of the form $x_i - x_j \leqslant k$. In order to allow more general form $\pm x_i \pm x_j \leqslant k$ of octagonal constraints, a DBM m of size $n \times n$ defined over $\mathbb{V}_{\mathcal{P}}$ is extended to another DBM m' of size $2n \times 2n$ over the set of enhanced variables $\mathbb{V}'_{\mathcal{P}} = \{x'_1, \ldots, x'_{2n}\}$ where each variable $x_i \in \mathbb{V}_{\mathcal{P}}$ comes in two forms: a positive form x'_{2i-1}, denoted x_i^+ and a negative form x'_{2i}, denoted x_i^-. This extended form of DBM m' is called coherent DBM (CDBM) representing octagon. This is illustrated in the following example.

Example 6 *Consider the octagonal constraints* $\{x_1 + x_2 \leqslant 3, x_1 - x_2 \leqslant 4, -x_1 - x_2 \leqslant 5, x_1 \leqslant 4\}$, *its equivalent CDBM constraints are* $\{x_1^+ - x_2^- \leqslant 3$, $x_2^+ - x_1^- \leqslant 3$, $x_1^+ - x_2^+ \leqslant 4$, $x_2^- - x_1^- \leqslant 4$, $x_1^- - x_2^+ \leqslant 5$, $x_2^- - x_1^+ \leqslant 5$, $x_1^+ - x_1^- \leqslant 8\}$. *These constraints are represented in CDBM shown below:*

	x_1^+	x_1^-	x_2^+	x_2^-
x_1^+	∞	∞	∞	5
x_1^-	8	∞	3	4
x_2^+	4	5	∞	∞
x_2^-	3	∞	∞	∞

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

Closure. An octagon can be represented by more than one set of inequalities. For instance, the octagonal constraints $\{(x \leqslant 4) \wedge (y \leqslant 6)\}$ and $\{(x \leqslant 4) \wedge (y \leqslant 6) \wedge (x + y \leqslant 10)\}$

represent the same concrete values. Therefore, the use of closure operation ensures a unique representation of any octagonal constraints. The closure operation on CDBM follows Floyd – Warshall algorithm [27].

In the rest of the paper, we use the notation m to represent closed CDBM when the context is clear.

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

- $\forall \mathsf{m}, \mathsf{n} \in \mathbb{M}_\perp$: $\mathsf{m} \sqsubseteq \mathsf{n} \iff \forall i, j$: $\mathsf{m}_{ij} \leqslant \mathsf{n}_{ij}$.

- $\forall \mathsf{m}, \mathsf{n} \in \mathbb{M}_\perp$: $(\mathsf{m} \sqcap \mathsf{n}) = \mathsf{m}'$ where $\forall i, j$: $\mathsf{m}'_{ij} \triangleq \min(\mathsf{m}_{ij}, \mathsf{n}_{ij})$.

- $\forall \mathsf{m}, \mathsf{n} \in \mathbb{M}_\perp$: $(\mathsf{m} \sqcup \mathsf{n}) = \mathsf{m}'$ where $\forall i, j$: $\mathsf{m}'_{ij} \triangleq \max(\mathsf{m}_{ij}, \mathsf{n}_{ij})$.

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

Let Σ be the set of all environments defined as $\Sigma : \mathbb{V} \mapsto \mathbb{R}$. An environment $\rho \in \Sigma$ maps each variable to its value. An environment will be understood as a point in $\mathbb{R}^n$ where $|\mathbb{V}| = n$. The Galois connection between L_c and L_a is formalized as $\langle \mathsf{L}_c, \alpha_{\mathbb{M}}, \gamma_{\mathbb{M}}, \mathsf{L}_a \rangle$ where $\alpha_{\mathbb{M}}$ and $\gamma_{\mathbb{M}}$ on $S \in \wp(\mathbb{R}^n)$ and $\mathsf{m} \in \mathbb{M}_\perp$ are defined below:

- if $S = \emptyset$: $\alpha_{\mathbb{M}}(S) \triangleq \mathsf{m}_\perp$

- if $S \neq \emptyset$: $\alpha_{\mathbb{M}}(S) = \mathsf{m}$ where $\mathsf{m}_{ij} \triangleq$
$$\begin{cases} \max\{\rho(x_l) - \rho(x_k) \mid \rho \in S\} & \text{when } i = 2k-1,\ j = 2l-1 \text{ or } i = 2l,\ j = 2k \\ \max\{\rho(x_l) + \rho(x_k) \mid \rho \in S\} & \text{when } i = 2k,\ j = 2l-1 \\ \max\{-\rho(x_l) - \rho(x_k) \mid \rho \in S\} & \text{when } i = 2k-1,\ j = 2l \end{cases}$$

$$\gamma_{\mathbb{M}}(\mathsf{m}) = \begin{cases} \emptyset & \text{if } \mathsf{m} = \mathsf{m}_{\bot} \\ \mathbb{R}^n & \text{if } \mathsf{m} = \mathsf{m}_{\top} \\ \Big\{(k_1, \dots, k_n) \in \mathbb{R}^n \mid (k_1, -k_1 \dots k_n, -k_n) & \\ \quad \in \mathtt{dom}(\mathsf{m}) \text{ and } \forall i, j : x_j - x_i \leqslant \mathsf{m}_{ij}\Big\} & \text{otherwise} \end{cases}$$

Sound operations in octagon domain. Let us recall from [94] some useful sound operations in octagon abstract domain defined in terms of CDBM:

- *Emptiness test*: Let m be a CDBM and $\mathtt{G}$ be a directed weighted graph of m. We say that the octagon is empty, i.e. $\gamma(\mathsf{m}) = \emptyset$, if and only if $\mathtt{G}$ has a simple cycle with a strictly negative total weight. The well-known Bellman-Ford [13] 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:

$$\begin{aligned} &\{v \mid \exists(k_1 \dots k_n) \in \gamma(\mathsf{m}) \text{ such that } k_i = v\} \\ &\quad = [-\mathsf{m}_{2i\ 2i+1}/2,\ \ \mathsf{m}_{2i+1\ 2i}/2] \end{aligned}$$

Interested reader may refer to [94] [95] for more abstract operations (closure, widening, etc.) in octagon domain.

2.3.2.2 Relational Abstract Domain of Polyhedra

The regions in n-dimensional space $\mathbb{R}^n$ bounded by finite sets of hyperplanes are called polyhedra. Let $\mathbb{V}_{\mathcal{P}} = \{x_1, x_2, \dots\ x_n\}$ be the set of variables in program $\mathcal{P}$. We represent by $\vec{v} = \langle v_1, v_2, \dots\ v_n\rangle \in \mathbb{R}^n$, an n-tuple (vector) of real numbers. By $\beta = \vec{v}.\vec{x} \geqslant k$ where $\vec{v} \neq \vec{0}$, $\vec{x} = \langle x_1, x_2, \dots, x_n\rangle$, $k \in \mathbb{R}$, we represent a linear inequality over $\mathbb{R}^n$. A linear inequality defines an affine half-space of $\mathbb{R}^n$. If $\mathtt{P}$ is expressed as the intersection of a finite number of affine half-spaces of $\mathbb{R}^n$, then $\mathtt{P} \in \mathbb{R}^n$ is a convex polyhedron. Formally, a convex polyhedron $\mathtt{P} = (\Theta, n)$ is a set of linear inequalities $\Theta = \{\beta_1, \beta_2 \dots \beta_m\}$ on $\mathbb{R}^n$. Equivalently, $\mathtt{P}$ can be represented by frame representation which is a collection of generators i.e. *vertices* and *rays* [25]. On the other hand, given a set of linear inequalities Θ on $\mathbb{R}^n$, a set of solutions or points defines a polyhedron $\mathtt{P} = (\Theta, n)$.

Concretization Function. Let $\mathsf{L}_c = \langle \wp(\mathbb{R}^n), \subseteq, \emptyset, \mathbb{R}^n, \cap, \cup \rangle$ be the concrete lattice defined over the concrete domain. The set of polyhedra $\mathbb{P}$ with partial order $\sqsubseteq$ forms an abstract lattice L_a=$\langle \mathbb{P}, \sqsubseteq, \mathsf{P}_\perp, \mathsf{P}_\top, \sqcap, \sqcup \rangle$. Given P_1, $\mathsf{P}_2 \in \mathbb{P}$, the partial order, meet and join operations are defined below:

- $\mathsf{P}_1 \sqsubseteq \mathsf{P}_2$ if and only if $\gamma(\mathsf{P}_1) \subseteq \gamma(\mathsf{P}_2)$, where $\gamma(\mathsf{P})$ represents the set of solutions or points in P as concrete values.

- $\mathsf{P}_1 \sqcap \mathsf{P}_2$ is the convex polyhedron containing exactly the set of points $\gamma(\mathsf{P}_1) \cap \gamma(\mathsf{P}_2)$.

- $\mathsf{P}_1 \sqcup \mathsf{P}_2$ is not necessarily a convex-polyhedron. Therefore, the least polyhedron enclosing this union is computed in terms of convex hull.

An environment $\rho \in \Sigma \triangleq \mathbb{V} \mapsto \mathbb{R}$ map each variable to its value in $\mathbb{R}$. Given $\mathsf{P} \in \mathbb{P}$, $\gamma_{\mathbb{P}}$ is defined below:

$$\gamma_{\mathbb{P}}(\mathsf{P}) = \begin{cases} \emptyset & \text{if } \mathsf{P} = \mathsf{P}_\perp \\ \mathbb{R}^n & \text{if } \mathsf{P} = \mathsf{P}_\top. \\ \left\{\rho \in \Sigma \mid \forall(\vec{v}.\vec{x} \geqslant k) : \vec{v}.\rho(\vec{x}) \geqslant k\right\} & \text{otherwise} \end{cases}$$

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 [36]. Therefore, in this case we denote by $\alpha_{\mathbb{P}}(\mathsf{S})$ a (possibly minimal) polyhedron in $\mathbb{P}$ such that $\gamma_{\mathbb{P}}(\alpha_{\mathbb{P}}(\mathsf{S})) \supseteq \mathsf{S}$.

Sound operations in polyhedra domain. Let us recall from [7, 22, 36] 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 [76] is used for checking feasibility of such constraint system. For example, adding a new constraint $\vec{v}.\vec{x} \geqslant k$ to a non empty polyhedra P, we can solve the LP problem μ = min $\vec{v}.\vec{x}$ 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:

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

This is computed by eliminating all occurrences of x_i in the constraints of P by using the Fourier-Motzkin algorithm [67] as below:

$$\begin{aligned} F(\mathsf{P},x_i) \triangleq \{(\Sigma_i v_i x_i \geqslant k) \in \Theta^i \mid v_i = 0\} \cup \{(-v_i^-)\beta^+ + v_i^+\beta^- \\ \mid \beta^+ = (\Sigma_i v_i^+ x_i \geqslant k^+) \in \Theta^+,\ v_i^+ > 0, \beta^- = (\Sigma_i v_i^- x_i \geqslant k^-) \in \Theta^-,\ v_i^- < 0\} \end{aligned}$$

where v_i^+ and v_i^- represent positive and negative coefficients for x_i respectively. The algorithm partitions the set of liner inequalities $\Theta = \{\beta_1, \beta_2 \ldots \beta_m\}$ into Θ^+, Θ^i and Θ^-, corresponding to inequalities that have positive, zero and negative coefficients for x_i. For each pair (β^+, β^-) of inequalities drown from $\Theta^+ \times \Theta^-$, the algorithm multiplies β^+ by the absolute value of x_i-*th* coefficient ($|v_i^-|$) in β^- and similarly multiplies β^- by x_i-*th* coefficient v_i^+ in β^+. 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 $\mathsf{P}_2 \sqsubseteq \mathsf{P}_1$) 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 $\mu = \min \vec{v}.\vec{x}$ subject to P_1 for each $\vec{v}.\vec{x} \geqslant k$. If $\mu < k$ then inclusion does not hold.

- *Redundancy removal*: In order to improve the efficiency in memory, it is desirable to remove the redundant constraints. Given $\mathsf{P} = (\Theta, n)$, an inequality $\beta \in \Theta$ is said to be redundant when β can be entailed by the other constraints in P, i.e. $\mathsf{P} / \{\beta\} \models \beta$. The LP solver is used for such verification. For example, in order to check whether $\beta = (\vec{v}.\vec{x} \geqslant k)$ is redundant, we can compute $\mu = \max \vec{v}.\vec{x}$ subject to P. If $k \leqslant \mu$, then β is redundant and can be eliminate from P.

2.3.2.3 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 [8]. 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.

Let $\mathsf{L}_c = \langle \mathbb{D}, \leqslant, \bot_c, \top_c, \cap_c, \cup_c \rangle$ be a concrete lattice and $\mathsf{L}_a = \langle \overline{\mathbb{D}}, \sqsubseteq, \bot_a, \top_a, \sqcap_a, \sqcup_a \rangle$ an abstract lattice over an abstract domain $\mathbb{A}$. The L_c and L_a are related by the Galois connection $(\mathsf{L}_c, \alpha, \gamma, \mathsf{L}_a)$. Considering the powerset abstract domain, the powerset of $\overline{\mathbb{D}}$ denoted by $\wp(\overline{\mathbb{D}})$ with the order relations $\preceq$ forms an abstract lattice $\mathsf{L}_p = \langle \wp(\overline{\mathbb{D}}), \preceq, \emptyset, \overline{\mathbb{D}}, \curlywedge, \curlyvee \rangle$. The partial order, meet and join operations in this abstract domain are defined as follows:

- $\forall S_1, S_2 \in \wp(\overline{\mathbb{D}}) : S_1 \preceq S_2 \Leftrightarrow \forall \overline{v}_i \in S_1 : \exists \overline{v}_j \in S_2 . \ \overline{v}_i \sqsubseteq \overline{v}_j$.
- $\forall S_1, S_2 \in \wp(\overline{\mathbb{D}}) : S_1 \curlywedge S_2 = \{\overline{v}_i \sqcap \overline{v}_j \mid \forall \overline{v}_i \in S_1, \forall \overline{v}_j \in S_2\}$.
- $\forall S_1, S_2 \in \wp(\overline{\mathbb{D}}) : S_1 \curlyvee S_2 = S_1 \cup S_2$.

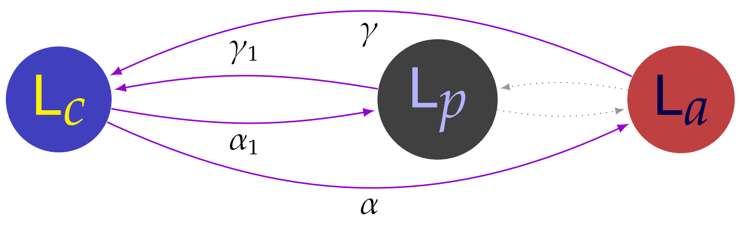
Observe that in powerset abstract domain the meet operation $S_1 \curlywedge S_2$ is defined by the pairwise meet of the elements from S_1 and S_2, whereas the join operation $S_1 \curlyvee S_2$ reduces to a set union.

The L_c and L_p are related by a Galois connections $\langle \mathsf{L}_c, \alpha_1, \ \gamma_1, \mathsf{L}_p \rangle$ where α_1 and γ_1 on $\forall X \in \mathbb{D}$ and $\forall Y \in \wp(\overline{\mathbb{D}})$ are defined below:

$$\alpha_1(X) = \begin{cases} \emptyset & \text{if } X = \bot_c \\ \overline{\mathbb{D}} & \text{if } X = \top_c \\ \big\{\alpha(X)\big\} & \text{otherwise} \end{cases}$$

$$\gamma_1(Y) = \begin{cases} \bot_c & \text{if } Y = \emptyset \\ \top_c & \text{if } Y = \overline{\mathbb{D}}. \\ \bigcup \big\{\gamma(\overline{v}) \mid \overline{v} \in Y\big\} & \text{otherwise} \end{cases}$$

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



To summarize, as we move from non-relational to relational abstract domain, the precision of the analysis-result improves by tolerating an increased computational cost. Therefore, a wise-choice of abstract domain as a trade-off between precision and efficiency is the key factor here.

CHAPTER 3

Concrete and Abstract Semantics of Structured Query Language (SQL)

Preface

In this chapter, we first recall the syntax and the concrete semantics of SQL from the literature [52]. Then we define an abstract semantics of SQL at various levels of abstractions, from non-relational to relational abstract domains such as domains of intervals, octagons, polyhedra and powerset of intervals.

3.1 Formal Syntax of SQL

In this section, we recall from [52] the formal syntax of database applications embedding SQL. Table 3.1 depicts the syntactic sets and the formal syntax. In particular, the syntax supports imperative programming paradigm combined with SQL statements. Formally, a SQL statement Q is denoted by $\langle A, \phi \rangle$ where A represents an action-part and ϕ represents a conditional-part. The action-part A includes `SELECT`, `UPDATE`, `DELETE` and `INSERT` operations which are denoted by A_{sel}, A_{upd}, A_{del} and A_{ins} respectively. The conditional-part ϕ represents the condition under the `WHERE` clause of the statement, which follows first-order logic formula. The imperative counterpart of the language includes `skip`, `assignment`, `conditional` and `iteration`. Intuitively, any program in this language involves two types of variables: application variables (denoted $\mathbb{V}_a$) and database variables (denoted $\mathbb{V}_d$). For the sake of simplicity, without compromising the generality, this is assumed that attributes names are unique. The SQL clauses `GROUP BY` and `ORDER BY` are denoted by the functions $g(\vec{e})$ and $f(\vec{e})$ respectively where $\vec{e}$ represents an order sequence of arithmetic expressions. Based on the values of $\vec{e}$ over database tuples, $g(\vec{e})$ results maximal partitions of the tuples and $f(\vec{e})$ sorts the tuples in either ascending or descending order. The aggregate functions (`AVG`, `SUM`, `MAX`, `MIN` and `COUNT`) in `SELECT` query are denoted by s. The ordered sequence of aggregate functions operating on an ordered sequence of arguments $\vec{x}$ is denoted by $\vec{h}(\vec{x})$ where each function $h_i \in \vec{h}$ operates on the corresponding argument $x_i \in \vec{x}$. Observe that the argument '$*$' in case of `count` denotes the sequence of all attributes of the target relation. This is to observe that, in the thesis, we consider only numerical domain for variables values including NULL. However, there is a scope to enhance our work by considering other data-type as well, such as string [28].

Let us illustrate the formal syntax using a suitable examples shown below:

Example 7 *Consider the following database statements:*

$$Q_{upd} = \texttt{UPDATE}\ t\ \texttt{SET}\ sal := sal + 100\ \texttt{WHERE}\ age \geqslant 35$$

$$Q_{sel} = \texttt{SELECT}\ dno,\ \texttt{MAX}(sal),\ \texttt{AVG}(\texttt{DISTINCT}\ age),\ \texttt{COUNT}(*)\ \texttt{FROM}\ t\ \ \texttt{WHERE}\ sal\ \geq 1000$$
$$\texttt{GROUP BY}\ dno\ \texttt{HAVING}\ \texttt{MAX}(sal) < 4000,\ \texttt{ORDER BY}\ dno$$

Constants:			
k	$\in$	$\mathbb{R}$	Set of Numerical Constants
Variables:			
v_a	$\in$	$\mathbb{V}_a$	Set of Application Variables
v_a	$::=$	$x \mid y \mid z \mid \ldots$	
v_d	$\in$	$\mathbb{V}_d$	Set of Database Attributes
v_d	$::=$	$a_1 \mid a_2 \mid a_3 \mid \ldots$	
$\mathbb{V}$	$::=$	$\mathbb{V}_a \cup \mathbb{V}_d$	
Expressions:			
e	$\in$	$\mathbb{E}$	Set of Arithmetic Expressions
e	$::=$	$k \mid v_d \mid v_a \mid op_u\ e \mid e_1\ op_b\ e_2$	where $op_u \in \{+,-\}$ and $op_b \in \{+,-,*,/\}$
b	$\in$	$\mathbb{B}$	Set of Boolean Expressions
b	$::=$	true | false | $e_1\ op_r\ e_2 \mid \neg b \mid b_1 \oplus b_2$	
		where $op_r \in \{\leq, \geq, ==, >, \neq, \ldots\}$ and $\oplus \in \{\vee, \wedge\}$	
SQL Pre-conditions:			
τ	$\in$	$\mathbb{T}$	Set of Terms
τ	$::=$	$k \mid v_a \mid v_d \mid f_n(\tau_1, \tau_2, ..., \tau_n)$	where f_n is an n-ary function.
a_f	$\in$	$\mathbb{A}_f$	Set of Atomic Formulas
a_f	$::=$	$R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 == \tau_2$	where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$
ϕ	$\in$	$\mathbb{W}$	Set of Pre-conditions
ϕ	$::=$	$a_f \mid \neg\phi \mid \phi_1 \oplus \phi_2 \mid \otimes\, v\, \phi$	where $\oplus \in \{\vee, \wedge\}$ and $\otimes \in \{\forall, \exists\}$
SQL Functions:			
$g(\vec{e})$	$::=$	`GROUP BY`$(\vec{e}) \mid id$	
		where $\vec{e} = \langle e_1, ..., e_n \mid e_i \in \mathbb{E}\rangle$ and id denotes identity function	
r	$::=$	`DISTINCT` | `ALL`	
s	$::=$	`AVG` | `SUM` | `MAX` | `MIN` | `COUNT` | id	
$h(e)$	$::=$	$s \circ r(e)$	
$h(*)$	$::=$	`COUNT(*)`	
		where * represents a list of database attributes denoted by $\vec{v_d}$	
$\vec{h}(\vec{x})$	$::=$	$\langle h_1(x_1), ..., h_n(x_n)\rangle$	
		where $\vec{h} = \langle h_1, ..., h_n\rangle$ and $\vec{x} = \langle x_1, ..., x_n \mid x_i = e \vee x_i = *\rangle$	
$f(\vec{e})$	$::=$	`ORDER BY ASC`$(\vec{e})$ | `ORDER BY DESC`$(\vec{e}) \mid id$	
Commands:			
Q	$\in$	$\mathbb{Q}$	Set of SQL Statements
Q	$::=$	$Q_{sel} \mid Q_{upd} \mid Q_{ins} \mid Q_{del}$	
Q_{sel}	$::=$	$\langle A_{sel},\ \phi\rangle$	
	$::=$	$\Big\langle$`SELECT`$\big(v_a, f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ \phi_1\Big\rangle$	
Q_{upd}	$::=$	$\langle A_{upd},\ \phi\rangle$	
	$::=$	$\langle$`UPDATE`$(\vec{v_d}, \vec{e}),\ \phi\rangle$	
Q_{ins}	$::=$	$\langle A_{ins},\ \phi\rangle$	
	$::=$	$\langle$`INSERT`$(\vec{v_d}, \vec{e}),\ false\rangle$	
Q_{del}	$::=$	$\langle A_{del},\ \phi\rangle$	
	$::=$	$\langle$`DELETE`$(\vec{v_d}),\ \phi\rangle$	
c	$\in$	$\mathbb{C}$	Set of Commands
c	$::=$	$skip \mid v_a := e \mid Q \mid$ if b then c endif	
	|	if b then c_1 else c_2 endif | while b do c done	
$\mathcal{P}$	$::=$	$c \mid c\ ;\ \mathcal{P}$	Program

Table 3.1: Formal Syntax of SQL embedded Programs [52]

The abstract syntax of the above statements are denoted by

$Q_{upd} = \langle A,\ \phi\rangle = \langle \texttt{UPDATE}(\vec{v_d},\ \vec{e}),\ \phi\rangle$ *where*

- $\phi = (age \geqslant 35)$
- $\vec{v_d} = \langle sal\rangle$
- $\vec{e} = \langle sal + 100\rangle$

and $Q_{sel}{=}\langle A,\ \phi\rangle{=}\ \langle \texttt{SELECT}\big(v_a, f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\big),\ \phi_1\rangle$ *where*

- $\phi_1 = sal \geq 1000,$
- $\vec{e} = \langle\ dno, sal, age\ \rangle,$
- $g(\overrightarrow{e}) = \texttt{GROUP BY}(dno),$
- $\phi_2 = (\texttt{MAX} \circ \texttt{ALL}(sal)) < 4000,$
- $\overrightarrow{h}(\overrightarrow{x}) = \langle\ \texttt{DISTINCT}(dno), \texttt{MAX} \circ \texttt{ALL}(sal), \texttt{AVG} \circ \texttt{DISTINCT}(age), \texttt{COUNT}(*)\ \rangle,$
- $f(\overrightarrow{e'}) = \texttt{ORDER BY}\ ASC(dno),$
- v_a = *ResultSet type application variable with fields* $\vec{w} = < w_1, w_2, \ldots w_n >.$

Observe that the syntax is consistent with the SQL definition given by ANSI [69]. Therefore, the formalism supports different RDBMS implementations, like Oracle, MySQL or IBM DB2. Its equivalence with relational algebra is reported in [52].

3.2 Concrete Semantics

In this section, we describe the concrete semantics of database applications embedding SQL, by recalling the formal concrete semantics of the imperative [30] as well as database statements [52].

3.2.1 Concrete Semantics of Imperative Statements

Since the imperative counterpart of the language does not involve any database variable, let us define the set of concrete states $\Sigma : \mathbb{V}_a \to \mathbb{R}$, which represents a mapping of imperative program variables to their semantic domain values. Given the set of arithmetic expressions $\mathbb{E}$ and the set of boolean expressions $\mathbb{B}$, the concrete denotational semantics functions $\mathscr{T}_e : (\mathbb{E} \cup \mathbb{B}) \to (\Sigma \to \mathbb{R} \cup \{true, false\})$ for expressions evaluation and $\mathscr{T}_f : \mathbb{B} \to (\wp(\Sigma) \to \wp(\Sigma))$ for state-filtering based on boolean satisfiability are defined below:

$$
\begin{aligned}
\mathscr{T}_e[\![k]\!] =& \{(\rho, k) \mid \rho \in \Sigma\} \\
\mathscr{T}_e[\![x]\!] =& \{(\rho, v) \mid \rho \in \Sigma,\ \rho(x) = v\} \\
\mathscr{T}_e[\![e_1 \oplus e_2]\!] =& \{(\rho, v_1 \oplus v_2) \mid\ (\rho, v_1) \in \mathscr{T}_e[\![e_1]\!], (\rho, v_2) \in \mathscr{T}_e[\![e_2]\!], \\
& \oplus \in \{+, -, \times\} \vee (\oplus \in \{/, \%\} \wedge v_2 \neq 0)\} \\
\mathscr{T}_e[\![e_1 \circledcirc e_2]\!] =& \{(\rho, u_1 \circledcirc u_2) \mid (\rho, u_1) \in \mathscr{T}_e[\![e_1]\!], (\rho, u_2) \in \mathscr{T}_e[\![e_2]\!], \\
& \circledcirc \in \{\geqslant, \leqslant, <, >, =\}\} \\
\mathscr{T}_e[\![\neg b]\!] =& \{(\rho, \neg w) \mid (\rho, w) \in \mathscr{T}_e[\![b]\!]\} \\
\mathscr{T}_e[\![b_1 \circledast b_2]\!] =& \{(\rho, w_1 \circledast w_2) \mid (\rho, w_1) \in \mathscr{T}_e[\![b_1]\!], (\rho, w_2) \in \mathscr{T}_e[\![b_2]\!], \\
& \circledast \in \{\vee, \wedge\}\} \\
\mathscr{T}_f[\![b]\!] =& \{(\rho, \rho)\ \mid\ \rho \in \Sigma,\ (\rho, true) \in \mathscr{T}_e[\![b]\!]\}
\end{aligned}
$$

Observe that the operators $\oplus$, $\circledcirc$ and $\circledast$ on the left hand side represent syntactic part of the language, whereas the same on the right hand side represent the operations on the corresponding semantic values.

Given the set of commands $\mathbb{C}$, the concrete denotational semantics function $\mathscr{T}_c : \mathbb{C} \to (\Sigma \to \Sigma)$ specifying effects of commands on states is defined below:

$$
\begin{aligned}
\mathscr{T}_c[\![skip]\!] =& \{(\rho, \rho)\ \mid\ \rho \in \Sigma\} \\
\mathscr{T}_c[\![x := e]\!] =& \{(\rho, \rho[x \leftarrow v])\ \mid\ \rho \in \Sigma,\ (\rho, v) \in \mathscr{T}_e[\![e]\!]\} \\
\mathscr{T}_c[\![c_1; c_2]\!] =& \mathscr{T}_c[\![c_2]\!] \circ \mathscr{T}_c[\![c_1]\!] \\
\mathscr{T}_c[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!] =& \{(\rho\ ,\ \rho')\ \mid\ (\rho\ ,\ \rho) \in \mathscr{T}_f[\![b]\!],\ (\rho\ ,\ \rho') \in \mathscr{T}_c[\![c_1]\!]\} \\
& \cup
\end{aligned}
$$

$$\{(\rho\ ,\ \rho'')\ \mid\ (\rho,\ \rho) \in \mathscr{T}_f[\![\neg b]\!],\ (\rho,\ \rho'') \in \mathscr{T}_c[\![c_2]\!]\}$$

$$\mathscr{T}_c[\![\text{while } b \text{ do } c]\!] = \{(\rho\ ,\ \rho')\ \mid\ (\rho\ ,\ \rho') \in$$

$$\mathscr{T}_f[\![\neg b]\!] \circ \text{lfp}\ \lambda Y.(\rho \cup \mathscr{T}_c[\![c]\!] \circ \mathscr{T}_f[\![b]\!]Y)\}$$

Observe that, in case of assignment statement $x := e$, the semantic function $\mathscr{T}_c$ returns a new state by substituting the semantic value of e into x w.r.t. the given state. The semantics of conditional and iterative statements are expressed in terms of the function $\mathscr{T}_f$ which filters the memory domains and considers only those satisfying the boolean expression b. The join operation $\cup$ merges the semantics of both branches of the conditional statement. On the other hand, the semantics of iterative statement is defined in terms of a loop invariant which is obtained via a least fixpoint computation.

3.2.2 Concrete Semantics of SQL Statements

Before describing the semantics of database languages, let us recall the definition of environments and state from [52]. This is to note that database language may involves variables from both $\mathbb{V}_a$ and $\mathbb{V}_d$. Therefore, both the notions of database environment and application environment are relevant in defining state.

Application Environment. Given the set of application variables $\mathbb{V}_a$ and the domain of values `Val`, let $\mathfrak{E}_a : \mathbb{V}_a \to$ `Val` be the set of all functions with domain $\mathbb{V}_a$ and range included in `Val`. An application environment $\rho_a \in \mathfrak{E}_a$ maps application variables to their values in `Val`.

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

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

State. Let Σ_{dba} be the set of states for the database language under consideration, defined by $\Sigma_{dba} \triangleq \mathfrak{E}_{dbs} \times \mathfrak{E}_a$ where $\mathfrak{E}_{dbs}$ and $\mathfrak{E}_a$ 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 (ρ_d, ρ_a) where $\rho_d \in \mathfrak{E}_{dbs}$ and $\rho_a \in \mathfrak{E}_a$.

Transition Function. The transition function

$$\mathscr{T}_{dba} : (\mathbb{C} \times \Sigma_{dba}) \to \wp(\Sigma_{dba}) \tag{3.1}$$

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

Concrete Semantics. Given a database statement $Q = \langle A, \phi \rangle$ and a concrete database state $\rho = (\rho_d\ ,\ \rho_a)$, the concrete semantics of Q w.r.t. ρ is defined below:

$$\begin{aligned}
&\mathscr{T}_{dba}[\![\langle A, \phi\rangle]\!]\rho\\
&\quad= \mathscr{T}_{dba}[\![\langle A, \phi\rangle]\!](\rho_d\ ,\ \rho_a)\\
&\quad= \mathscr{T}_{dba}[\![\langle A, \phi\rangle]\!](\rho_t\ ,\ \rho_a)\\
&\qquad\quad \text{where } t = \mathit{target}(\langle A, \phi\rangle)\\
&\quad= \mathscr{T}_{dba}[\![\langle A\rangle]\!](\rho_{(t\downarrow\phi)}\ ,\ \rho_a)\ \sqcup\ (\rho_{\neg(t\downarrow\phi)}\ ,\ \rho_a)\\
&\quad= (\rho_{t'}\ ,\ \rho_a)\ \sqcup\ (\rho_{\neg(t\downarrow\phi)}\ ,\ \rho_a)\\
&\quad= (\rho_{t'}\ \sqcup\ \rho_{\neg(t\downarrow\phi)}\ ,\ \rho_a\ \sqcup\ \rho_a)\\
&\quad= (\rho_{t''}\ ,\ \rho_a)
\end{aligned} \tag{3.2}$$

where the function target(Q) returns the table on which the operations in Q are restricted and the notation $(t \downarrow \phi)$ denotes the set of tuples in t for which ϕ is true. Observe that the semantics function $\mathscr{T}_{dba}$ on the action-part A w.r.t. the table environment $\rho_{t\downarrow\phi}$ yields a new state $\rho_{t'}$. Observe that, as concrete table environment is defined in such a way to capture multiset semantics of attributes, the concrete semantics of SQL respect this as well. As this is clear from the context, in this semantics definition the state of the tables other than ρ_t remained unchanged. Therefore, we have not specified this explicitly. Let us illustrate the concrete semantics of an update statement in Example 8.

(a) table t

eid	*sal*	*age*	*dno*
1	1500	35	10
2	800	28	20
3	2500	50	10
4	3000	62	10

(b) table t''

eid	*sal*	*age*	*dno*
1	1600	35	10
2	800	28	20
3	2600	50	10
4	3100	62	10

Table 3.2: Database before and after the update operation

Example 8 *Consider the database table t in Table 3.2(a) and the following update statement:*

$$Q_{upd} : \texttt{UPDATE } t \texttt{ SET } sal := sal + 100 \texttt{ WHERE } age \geqslant 35$$

The abstract syntax is denoted by $\langle \texttt{UPDATE}(\vec{v_d}, \vec{e}),\ \phi\rangle$ *where* $\phi = (age \geqslant 35)$ *and* $\vec{v_d} = \langle sal\rangle$ *and* $\vec{e} = \langle sal + 100\rangle$.

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

$$\begin{aligned}
&\mathscr{T}_{dba}[\![Q_{upd}]\!](\rho_d, \rho_a)\\
&=\mathscr{T}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle),\ (age \geqslant 35)\Big\rangle]\!](\rho_d, \rho_a)\\
&=\mathscr{T}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle),\ (age \geqslant 35)\Big\rangle]\!](\rho_t, \rho_a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\textbf{\textit{Since}},\ target(Q_{upd})=\{t\}\\
&=\mathscr{T}_{dba}[\![\texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle)]\!](\rho_{t\downarrow(age\geqslant 35)}, \rho_a)\\
&\qquad\quad\sqcup\\
&\quad(\rho_{t\downarrow\neg(age\geqslant 35)}, \rho_a) \qquad\qquad \textbf{\textit{Absorbing}}\ \phi = (age \geqslant 35)\\
&=(\rho_{t'}, \rho_a)\ \sqcup\ (\rho_{t\downarrow\neg(age\geqslant 35)}, \rho_a)\\
&=(\rho_{t'} \sqcup \rho_{t\downarrow\neg(age\geqslant 35)},\ \rho_a)\\
&=(\rho_{t''}, \rho_a)
\end{aligned}$$

where

$$\rho_{t'} \equiv \rho_{t\downarrow(age\geqslant 35)}\Big[sal \ \leftarrow\ \mathscr{T}_e[\![sal\ +\ 100]\!](\rho_{t\downarrow(age\geqslant 35)},\ \rho_a)\Big]$$

$$=\rho_{t\downarrow(age \geqslant 35)}\,[sal \;\; \leftarrow \langle 1600,\;\; 2600,\;\; 3100\rangle]$$

The notation $(t \downarrow (age \geqslant 35))$ *denotes the set of tuples in t for which* $(age \geqslant 35)$ *is true (denoted by red part in t of Table 3.2(a)).* $\mathscr{T}_e$ *is the semantic function for arithmetic expression which maps "sal* $+ 100$*" to a list of values* $\langle 1600, 2600, 3100\rangle$ *on the table environment* $\rho_{t\downarrow(age \geqslant 35)}$*. The notation* $\leftarrow$ *denotes a substitution by new values. Observe that the substitution of 'sal' by the list of values in* $\rho_{t\downarrow(age \geqslant 35)}$ *results in a new table environment* $\rho_{t'}$ *(denoted by red part in Table 3.2(b)). Finally, the least upper bound (denoted* $\sqcup$*) which is defined over the lattice of table environments partially ordered by* $\subseteq$*, results in a new state* $(\rho_{t''}, \rho_a)$ *where* t'' *is depicted in Table 3.2(b).*

3.3 Abstract Semantics

Let us first recall from [29,36,94] the abstract semantics of imperative statements. Then we define the abstract semantics of database statements at various levels of abstractions, from non-relational to relational domains.

3.3.1 Abstract Semantics of Imperative Statements

Let $\mathsf{L}_c = \langle \mathbb{D}, \subseteq, \bot_c, \top_c, \cap_c, \cup_c\rangle$ be a concrete lattice and $\mathsf{L}_a = \langle \overline{\mathbb{D}}, \sqsubseteq, \bot_a, \top_a, \sqcap_a, \sqcup_a\rangle$ an abstract lattice. The L_c and L_a are related by the Galois connection $(\mathsf{L}_c, \alpha, \gamma, \mathsf{L}_a)$ such that $\alpha(X) \sqsubseteq Y \iff X \subseteq \gamma(Y)$ where $X \in \mathbb{D}$ and $Y \in \overline{\mathbb{D}}$. The set of abstract states is defined as $\overline{\Sigma} : \mathbb{V}_a \rightarrow \overline{\mathbb{D}}$ which respects the Galois Connection, i.e. $\forall\, \rho \in \Sigma, \forall\, \overline{\rho} \in \overline{\Sigma}$: $\alpha(\rho) \sqsubseteq \overline{\rho} \iff \rho \subseteq \gamma(\overline{\rho})$.

3.3.1.1 Domain of Interval

Given an abstract domain $\mathbb{I}$ of intervals, the set of abstract states is defined as $\overline{\Sigma} : \mathbb{V}_a \rightarrow \mathbb{I}$ which respects the Galois Connection, i.e. $\forall\, \rho \in \Sigma, \forall\, \overline{\rho} \in \overline{\Sigma}$: $\alpha(\rho) \sqsubseteq \overline{\rho} \iff \rho \subseteq \gamma(\overline{\rho})$.

The corresponding sound abstract semantics function $\overline{\mathscr{T}_e} : (\mathbb{E} \cup \mathbb{B}) \rightarrow (\overline{\Sigma} \rightarrow \mathbb{I} \cup \{true, false, \top_B\})$ for expression evaluation where $\top_B$ denotes " *may be true or may be false*", is defined as:

$$\overline{\mathscr{T}_e}[\![k]\!] = \{(\overline{\rho}, [k, k]) \mid \overline{\rho} \in \overline{\Sigma}\}$$

$$\begin{aligned}
\overline{\mathscr{T}_e}[\![x]\!] &= \{(\overline{\rho}, \overline{\rho}(x)) \mid \overline{\rho} \in \overline{\Sigma}\}\\
\overline{\mathscr{T}_e}[\![e_1 \;\oplus\; e_2]\!] &= \{(\overline{\rho}, \overline{v}_1 \mathbin{\overline{\oplus}} \overline{v}_2) \mid (\overline{\rho}, \overline{v}_1) \in \overline{\mathscr{T}_e}[\![e_1]\!], (\overline{\rho}, \overline{v}_2) \in \overline{\mathscr{T}_e}[\![e_2]\!]\}\\
\overline{\mathscr{T}_e}[\![e_1 \;\circledcirc\; e_2]\!] &= \{(\overline{\rho}, \overline{v}_1 \mathbin{\overline{\circledcirc}} \overline{v}_2) \mid (\overline{\rho}, \overline{v}_1) \in \overline{\mathscr{T}_e}[\![e_1]\!], (\overline{\rho}, \overline{v}_2) \in \overline{\mathscr{T}_e}[\![e_2]\!]\}
\end{aligned}$$

Examples of sound abstract arithmetic and relational operations $\overline{\oplus}$ and $\overline{\circledcirc}$ respectively in the domain of intervals are:

$$\begin{aligned}
[l_1,\ h_1] \mathbin{\overline{+}} [l_2,\ h_2] &= [l_1 + l_2,\ h_1 + h_2]\\
[l_1,\ h_1] \mathbin{\overline{-}} [l_2,\ h_2] &= [l_1 - l_2,\ h_1 - h_2]\\
[l_1,\ h_1] \mathbin{\overline{\times}} [l_2,\ h_2] &= [\min(l_1 \times l_2, l_1 \times h_2, h_1 \times l_2, h_1 \times h_2),\\
&\qquad \max(l_1 \times l_2, l_1 \times h_2, h_1 \times l_2, h_1 \times h_2)]\\
[l_1,\ h_1] \mathbin{\overline{/}} [l_2,\ h_2] &= [l_1,\ h_1] \overline{\times} [\frac{1}{l_2},\ \frac{1}{h_2}],\ 0 \notin [l_2,\ h_2]\\
[l_1,\ h_1] \mathbin{\overline{\geqslant}} [l_2,\ h_2] &= \begin{cases} true & \text{if } l_1 \geqslant h_2\\ false & \text{if } h_1 < l_2\\ \top_B & \text{otherwise}\end{cases}
\end{aligned}$$

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

Similarly, the abstract semantics functions $\overline{\mathscr{T}_f} :\ \mathbb{B} \to (\overline{\Sigma} \to \overline{\Sigma})$ for abstract state-filtering and $\overline{\mathscr{T}_c} :\ \mathbb{C} \to (\overline{\Sigma} \to \overline{\Sigma})$ for commands are:

$$\begin{aligned}
\overline{\mathscr{T}_f}[\![x \leqslant k]\!] &= \Big\{(\overline{\rho}, \overline{\rho}[x \leftarrow [l, \min(h,k)]]) \mid \overline{\rho} \in \overline{\Sigma}, \overline{\rho}(x) = [l,\ h],\ l \leqslant k\Big\}\\
\overline{\mathscr{T}_f}[\![x < k]\!] &= \Big\{(\overline{\rho}, \overline{\rho}[x \leftarrow [l, \min(h,k-1)]]) \mid \overline{\rho} \in \overline{\Sigma}, \overline{\rho}(x) = [l,\ h],\ l \leqslant k-1\Big\}\\
\overline{\mathscr{T}_f}[\![x \geqslant k]\!] &= \Big\{(\overline{\rho}, \overline{\rho}[x \leftarrow [\max(l,k), h]]) \mid \overline{\rho} \in \overline{\Sigma}, \overline{\rho}(x) = [l,\ h],\ h \geqslant k\Big\}\\
\overline{\mathscr{T}_f}[\![x > k]\!] &= \Big\{(\overline{\rho}, \overline{\rho}[x \leftarrow [\max(l,k+1), h]]) \mid \overline{\rho} \in \overline{\Sigma}, \overline{\rho}(x) = [l,\ h],\ h \geqslant k+1\Big\}\\
\overline{\mathscr{T}_f}[\![x = k]\!] &= \Big\{(\overline{\rho}, \overline{\rho}[x \leftarrow [k,k]]) \mid \overline{\rho} \in \overline{\Sigma}, \overline{\rho}(x) = [l,\ h],\ l \ \leqslant\ k \leqslant\ h\Big\}\\
\overline{\mathscr{T}_c}[\![skip]\!] &= \Big\{(\overline{\rho}, \overline{\rho}) \ \mid\ \overline{\rho} \in \overline{\Sigma}\Big\}\\
\overline{\mathscr{T}_c}[\![x := e]\!] &= \Big\{(\overline{\rho}, \overline{\rho}[x \leftarrow \overline{v}]) \mid (\overline{\rho}, \overline{v}) \in \overline{\mathscr{T}_e}[\![e]\!]\Big\}
\end{aligned}$$

$$\overline{\mathscr{T}_c}[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!] = \{(\overline{\rho},\ \overline{\rho}_3)\ \mid\ (\overline{\rho},\ \overline{\rho}_1) \in \overline{\mathscr{T}_f}[\![b]\!],\ (\overline{\rho}_1,\ \overline{\rho}_3) \in \overline{\mathscr{T}_c}[\![c_1]\!]\}$$

$$\begin{aligned}&\sqcup\\&\{(\overline{\rho},\ \overline{\rho}_4)\ \mid\ (\overline{\rho},\ \overline{\rho}_2)\in\overline{\mathscr{T}_f}[\![\neg b]\!],\ (\overline{\rho}_2,\ \overline{\rho}_4)\in\overline{\mathscr{T}_c}[\![c_2]\!]\}\\=\ &\{(\overline{\rho},\ \overline{\rho}_3\sqcup\overline{\rho}_4)\ \mid\overline{\rho}\in\overline{\Sigma}\}\end{aligned}$$

where $\sqcup$ denotes component-wise join operation in the abstract lattice L_a.

$$\overline{\mathscr{T}_c}[\![\text{while } b \text{ do } c]\!] = \{(\overline{\rho}\ ,\ \overline{\rho}_1)\mid(\overline{\rho}\ ,\ \overline{\rho}_1)\in \overline{\mathscr{T}}_f[\![\neg b]\!]\circ \text{lfp}\ \lambda Y.(Y\nabla(\overline{\rho}\ \sqcup\ \overline{\mathscr{T}}_c[\![c]\!]\circ\overline{\mathscr{T}}_f[\![b]\!]Y))\}$$

where $\nabla : (\mathbb{I}\times\mathbb{I})\rightarrow\mathbb{I}$ is a widening operator, if:

- for each $x, y \in \mathbb{I}$: $x \sqsubseteq x\nabla y$ and $y \sqsubseteq x\nabla y$.
- for each increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq \dots$, the increasing chain defined by $y_0 = x_0$, $y_{n+1} = y_n\nabla x_{n+1}$ for $n \in \mathbb{N}$, is not strictly increasing.

Example 9 *Consider the statement c ::= if $x \geqslant 5$ then $x := x + y$ else $x := x - y$. Consider an abstract state in the domain of intervals $\overline{\rho}$=$\langle x \rightarrow [2, 10], y \rightarrow [1, 1]\rangle$. The abstract semantics of c w.r.t. $\overline{\rho}$ is illustrated below:*

$$\begin{aligned}\overline{\mathscr{T}_f}[\![x\geqslant 5]\!](\overline{\rho}) =&\overline{\rho}[x\leftarrow[5,10]] = \overline{\rho}_1\\\overline{\mathscr{T}_f}[\![\neg(x\geqslant 5)]\!](\overline{\rho}) =&\overline{\rho}[x\leftarrow[2,4]] = \overline{\rho}_2\\\overline{\mathscr{T}_e}[\![x+y]\!](\overline{\rho}_1) =&\overline{\rho}_1(x)\ \overline{+}\ \overline{\rho}_1(y) = [6,11]\\\overline{\mathscr{T}_e}[\![x-y]\!](\overline{\rho}_2) =&\overline{\rho}_2(x)\ \overline{-}\ \overline{\rho}_2(y) = [1,3]\\\overline{\mathscr{T}_c}[\![x:=x+y]\!](\overline{\rho}_1) =&\overline{\rho}_1[x\leftarrow[6,11]] = \overline{\rho}_3\\\overline{\mathscr{T}_c}[\![x:=x-y]\!](\overline{\rho}_2) =&\overline{\rho}_2[x\leftarrow[1,3]] = \overline{\rho}_4\\\overline{\mathscr{T}_c}[\![\textit{if } x\geqslant 5 \textit{ then } x:=x+y \textit{ else } x:=x-y]\!](\overline{\rho})&\\=&\ (\overline{\rho}_3\sqcup\overline{\rho}_4) = \langle[6,11]\sqcup[1,3],\ [1,1]\sqcup[1,1]\rangle\\=&\ \langle[1,11],\ [1,1]\rangle = \overline{\rho}_5\end{aligned}$$

Example 10 *Consider the simple code fragment $x = 1$; while$(x < 100)\{x = x + 1;\}$. Figure 3.1 illustrates a data flow-based analysis of the code in $\mathbb{I}$ for the absence of runtime errors. The data-flow equation for each node is mentioned on the controlling edge of the corresponding node.*

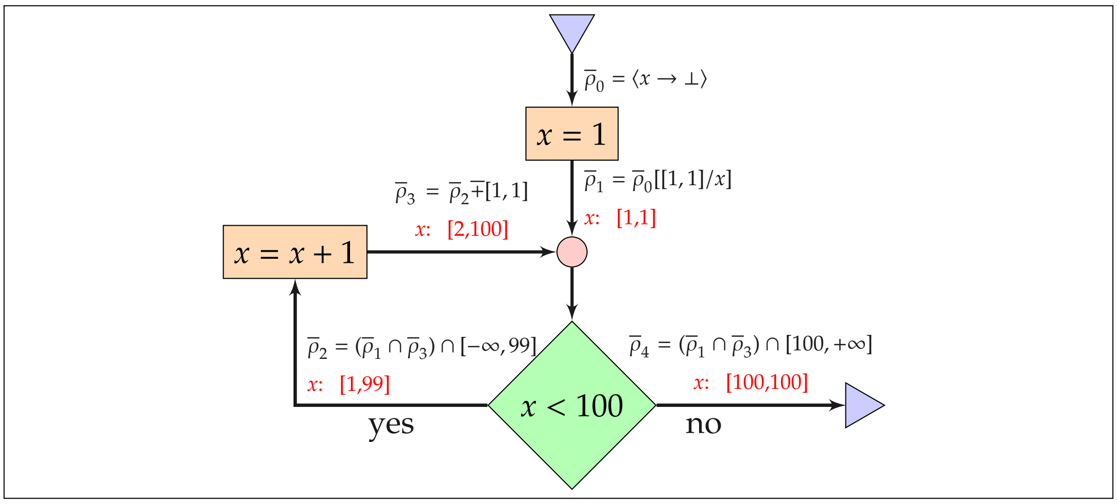

$\overline{\rho}_0 = \langle x \to \bot \rangle$
$x = 1$
$\overline{\rho}_3 = \overline{\rho}_2 \overline{+} [1,1]$
x: [2,100]
$\overline{\rho}_1 = \overline{\rho}_0[[1,1]/x]$
x: [1,1]
$x = x + 1$
$\overline{\rho}_2 = (\overline{\rho}_1 \cap \overline{\rho}_3) \cap [-\infty, 99]$
x: [1,99]
$\overline{\rho}_4 = (\overline{\rho}_1 \cap \overline{\rho}_3) \cap [100, +\infty]$
x: [100,100]
$x < 100$
yes
no

Figure 3.1: An example of interval analysis

The fix-point solution of these equations represent the abstract collecting semantics (denoted by red color).

3.3.1.2 Domain of Octagons

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 $\mathscr{T}_f : \big(\mathbb{B} \to \wp(\Sigma)\big) \to \wp(\Sigma)$. The corresponding sound abstract function $\overline{\mathscr{T}_f}$ in the domain of octagons is defined as $\overline{\mathscr{T}_f} : (\mathbb{B} \to \mathbb{M}_\bot) \to \mathbb{M}_\bot$. Similarly, the concrete denotational semantic function for the effects of commands on states is defined as $\mathscr{T}_c : \big(\mathbb{C} \to \wp(\Sigma)\big) \to \wp(\Sigma)$ and its corresponding sound abstract function $\overline{\mathscr{T}_c}$ in octagon domain is defined as $\overline{\mathscr{T}_c} : (\mathbb{C} \to \mathbb{M}_\bot) \to \mathbb{M}_\bot$.

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

$$\gamma(\mathsf{m}') \supseteq \{\rho \in \gamma(\mathsf{m}) \mid \rho \text{ satisfies } b\}$$

The tests that can be modeled in the octagon domain are: $x_h + x_l \leqslant k$, $x_h - x_l \leqslant k$, $-x_h - x_l \leqslant k$, $x_h + x_l = k$, $x_h \leqslant k$ and $x_h \geqslant k$. The state-filtering function $\overline{\mathscr{T}_f}$ for $x_h + x_l \leqslant k$ is defined as below:

$$\overline{\mathscr{T}_f}[\![x_h + x_l \leqslant k]\!]\mathsf{m} = \mathsf{m}' \text{ where}$$

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

Observe that the entries in the CDBM m corresponding to the cells (x_h^-, x_l^+) and (x_l^-, x_h^+) are updated based on the value k, resulting into m' which satisfies $x_h + x_l \leqslant k$. Similarly $\overline{\mathscr{T}_f}$ for all others 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 m' as an upper approximation such that

$$\gamma(\mathsf{m}') \supseteq \{\rho[x_i \leftarrow k] \mid \rho \in \gamma(\mathsf{m}) \wedge k = \mathscr{T}_e[\![e]\!]\rho\} \text{ where}$$

$\mathscr{T}_e$ is the semantic function of arithmetic expression and $\rho[x_i \leftarrow k]$ denote ρ with its i^{th} component changed into k.

The assignments that can be modeled in octagon domain are: $x_h = x_h + k$ and $x_h = x_l + k$ 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 $x_h = x_l + k$, the inequalities $x_h - x_l \leqslant k$ and $x_l - x_h \leqslant -k$ are added into the octagon. Let us illustrate them below:

1. If $x_h = x_h + k$:
   $\overline{\mathscr{T}_c}[\![x_h = x_h + k]\!]\mathsf{m} = \mathsf{m}'$ where $\mathsf{m}'_{ij} \triangleq \mathsf{m}_{ij} + (\alpha_{ij} + \beta_{ij})k$ with

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

and

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

2. If $x_h = x_l + k$ with $h \neq l$:
   $\overline{\mathcal{T}_c}[\![x_h = x_l + k]\!]\mathsf{m} = \mathsf{m}'$ where

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

Example 11 *Consider the statement $c ::= if\ x \geqslant 5\ then\ x := y+1\ else\ x := y-1$. Let the initial abstract state be $m_\top$ (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 $\overset{rep}{=}$ denotes an alternative representation in memory.*

$$\overline{\mathcal{T}_f}[\![x \geqslant 5]\!]m_\top = \{-x \leqslant -5\} \overset{rep}{=} m_1$$

$$\overline{\mathcal{T}_f}[\![\neg(x \geqslant 5)]\!]m_\top = \{x \leqslant 4\} \overset{rep}{=} m_2$$

$$\overline{\mathcal{T}_c}[\![x := y+1]\!]m_1 = \{x - y \leqslant 1, y - x \leqslant -1, -x \leqslant -5\} \overset{rep}{=} m_3$$

$$\overline{\mathcal{T}_c}[\![x := y-1]\!]m_1 = \{x - y \leqslant -1, y - x \leqslant 1, x \leqslant 4\} \overset{rep}{=} m_4$$

$$\overline{\mathcal{T}_c}[\![if\ x \geqslant 5\ then\ x := y+1\ else\ x := y-1]\!]m_\top = (m_3 \sqcup m_4)$$

$$\overset{rep}{=} \langle\{x - y \leqslant 1, y - x \leqslant -1, -x \leqslant -5\} \sqcup$$

$$\{x - y \leqslant -1, y - x \leqslant 1, x \leqslant 4\}\rangle$$

$$= \langle\{x - y \leqslant 1, y - x \leqslant 1\}\rangle = \mathsf{m}_5$$

After recalling the abstract semantics of imperative languages on octagon domains designed by [94], let us move to database languages.

3.3.1.3 Domain of Polyhedra

Given the concrete denotational semantics functions $\mathscr{T}_f : \big(\mathbb{B} \to \wp(\Sigma)\big) \to \wp(\Sigma)$, the corresponding sound abstract function $\overline{\mathscr{T}_f}$ in the domain of polyhedra is defined as $\overline{\mathscr{T}_f} : (\mathbb{B} \to \mathbb{P}) \to \mathbb{P}$. Similarly, given the concrete denotational semantic function $\mathscr{T}_c : \big(\mathbb{C} \to \wp(\Sigma)\big) \to \wp(\Sigma)$, its corresponding sound abstract function $\overline{\mathscr{T}_c}$ in polyhedra domain is defined as $\overline{\mathscr{T}_c} : (\mathbb{C} \to \mathbb{P}) \to \mathbb{P}$.

Test: Let b be a boolean expression in the form of linear inequalities $\vec{v}.\vec{x} \geqslant k$ and the abstract state in the form of polyhedron P. The state-filtering function $\overline{\mathscr{T}_f}$ finds P' applying b on P define as

$$\overline{\mathscr{T}_f}[\![\vec{v}.\vec{x} \geqslant k]\!]\mathsf{P} = \mathsf{P}'$$

where $\mathsf{P}' = \mathsf{P} \sqcap b$.

Example 12 *Given P=($\{x \geqslant 8,\ y \geqslant 6\}$, 2). The equivalent generators representation (*vertices *and* rays*) of P is V={(8, 6)} and R={(1, 0), (0, 1)}. The abstract semantics of boolean expression $x \geqslant 20$ is defined as: $\overline{\mathscr{T}_f}[\![x \geqslant 20]\!]\mathsf{P} = \mathsf{P}'$ where $\mathsf{P}' = (\{x \geqslant 20, y \geqslant 6\}, 2)$ and its equivalent generators representation is V′={(20, 6)} and R′={(1, 0), (0, 1)}.*

Assignment statement: $\overline{\mathscr{T}_c}[\![x_j = e]\!](\mathsf{P}) = \mathsf{P}'$ where P' is obtained as follows: (*i*) **Case**-1: If e is non-linear 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 $x_j{}'$ to hold the value of e, then we project out x_j and finally we reuse $x_j{}'$ in place of x_j which results into P'.

Example 13 *Given P=($\{x \geqslant 3,\ y \geqslant 2\}$, 2). The equivalent generators representation (*vertices *and* rays*) of P is V={(3, 2)} and R={(1, 0), (0, 1)}. The $\overline{\mathscr{T}_c}$ of assignment $x := x + y$ is define as*

$$\overline{\mathscr{T}_c}[\![x := x + y]\!](\{x \geqslant 3, y \geqslant 2\}, 2) = \mathsf{P}' \qquad \textit{where}$$

$P' = (\{x - y \geqslant 3, y \geqslant 2\}, 2)$ *and its equivalent generators representation is V′={(5, 2)} and R′={(1, 0), (-1, -1)}.*

3.3.1.4 Powerset Abstract Domain

Let us explain the powerset construction over the interval abstract domain. Consider the interval abstract domain $\overline{\mathbb{I}} = \{[l,\ h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\} \cup \bot$ forming an abstract lattice $\mathsf{L}_a = \langle \overline{\mathbb{I}}, \sqsubseteq, \bot, [-\infty, +\infty], \sqcap, \sqcup \rangle$. The powerset of the intervals denoted by $\wp(\overline{\mathbb{I}})$ forms the abstract lattice $\mathsf{L}_p = \langle \wp(\overline{\mathbb{I}}), \preceq, \emptyset, \overline{\mathbb{I}}, \curlywedge, \curlyvee \rangle$.

The correspondence between L_a and L_p is formalized as the Galois connections $\langle \mathsf{L}_a, \alpha_1, \gamma_1, \mathsf{L}_p \rangle$. The partial order, meet and join operations in the powerset domain of intervals can be defined accordingly.

Given a powerset abstract domain of intervals $\wp(\overline{\mathbb{I}})$, the set of abstract states $\overline{\Sigma} : \mathbb{V} \to \wp(\overline{\mathbb{I}})$ respects the Galois Connection, i.e. $\forall\ \rho \in \Sigma, \forall\ \overline{\rho} \in \overline{\Sigma}$: $\alpha_1(\rho) \sqsubseteq \overline{\rho} \iff \rho \subseteq \gamma_1(\overline{\rho})$.

Given $S_1, S_2 \in \wp(\overline{\mathbb{I}})$, the sound abstract arithmetic operations $\overline{\oplus}$ in the powerset abstract domain of intervals are defined as:

$$\forall S_1, S_2 \in \wp(\overline{\mathbb{I}}) : S_1 \overline{\oplus} S_2 = \{\overline{v}_i \,\overline{\oplus}\, \overline{v}_j \mid \forall \overline{v}_i \in S_1, \forall \overline{v}_j \in S_2\}$$

The corresponding sound abstract semantics function $\overline{\mathscr{T}_e} : (\mathbb{E} \cup \mathbb{B}) \to (\overline{\Sigma} \to \wp(\overline{\mathbb{I}}) \cup \{true, false, \top_B\})$ for expression evaluation where $\top_B$ denotes *“ may be true or may be false”*, the abstract semantics functions $\overline{\mathscr{T}_f} : \mathbb{B} \to (\overline{\Sigma} \to \overline{\Sigma})$ for abstract state-filtering and $\overline{\mathscr{T}_c} : \mathbb{C} \to (\overline{\Sigma} \to \overline{\Sigma})$ for commands are defined accordingly in the powerset domain of intervals. Example 14 illustrates this.

Example 14 *Consider the statement c ::= if* $x \geqslant 5$ *then* $x := x + y$ *else* $x := x - y$*. Consider an abstract state in the powerset domain of interval* $\overline{\rho}$=$\langle x \to \{[2,6],[8,10]\}, y \to \{[1,1]\}\rangle$*. The abstract semantics of c w.r.t.* $\overline{\rho}$ *is illustrated below:*

$$\begin{aligned}
\overline{\mathscr{T}_f}[\![x \geqslant 5]\!](\overline{\rho}) &= \overline{\rho}[x \leftarrow \{[5,6],[8,10]\}] = \overline{\rho}_1 \\
\overline{\mathscr{T}_f}[\![\neg(x \geqslant 5)]\!](\overline{\rho}) &= \overline{\rho}[x \leftarrow \{[2,4]\}] = \overline{\rho}_2 \\
\overline{\mathscr{T}_e}[\![x + y]\!](\overline{\rho}_1) &= \overline{\rho}_1(x) \mathbin{\overline{+}} \overline{\rho}_1(y) = \{[6,7],[9,11]\}
\end{aligned}$$

$$\overline{\mathscr{T}_e}[\![x - y]\!](\overline{\rho}_2) = \overline{\rho}_2(x) \,\overline{-}\, \overline{\rho}_2(y) = \{[1,3]\}$$

$$\overline{\mathscr{T}_c}[\![x := x + y]\!](\overline{\rho}_1) = \overline{\rho}_1[x \leftarrow \{[6,7],[9,11]\}] = \overline{\rho}_3$$

$$\overline{\mathscr{T}_c}[\![x := x - y]\!](\overline{\rho}_2) = \overline{\rho}_2[x \leftarrow \{[1,3]\}] = \overline{\rho}_4$$

$$\overline{\mathscr{T}_c}[\![\mathit{if}\ x \geqslant 5\ \mathit{then}\ x := x + y\ \mathit{else}\ x := x - y]\!](\overline{\rho})$$

$$= (\overline{\rho}_3 \vee \overline{\rho}_4) = \langle \{[6,7],[9,11]\} \vee \{[1,3]\},\ \{[1,1]\} \vee \{[1,1]\} \rangle$$

$$= \langle \{[1,3],[6,7],[9,11]\},\ \{[1,1]\} \rangle = \overline{\rho}_5$$

3.3.2 Abstract Semantics of SQL Statements

Motivated from the abstract semantics of database statements defined only in the domain of intervals [52], we are now in a position to enrich it to more precise abstract domains, namely the domain of octagons, polyhedra and powerset of intervals. To this aim, let us first define abstract database states and the abstract semantic transition function in an abstract domain of interest w.r.t. its concrete counterpart.

Definition 3.1 (Abstract Table) *Given a concrete table $t \in \wp(D)$ where $D = D_1 \times D_2 \times .... \times D_k$ such that* `attr`*(t)=$\{a_1, a_2, \dots, a_k\}$ and a_i is the attribute corresponding to the typed domain D_i. Let $\overline{D}$ be an abstract domain which represents properties of the attributes of t establishing the Galois Connection* [1] $\left\langle (\wp(D), \subseteq), \alpha, \gamma, (\overline{D}, \sqsubseteq) \right\rangle$. *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 3.2 (Abstract Table Environment) *Given an abstract table $\overline{t}$, an abstract table environment $\rho_{\overline{t}}$ is defined as $\rho_{\overline{t}}(a_i) = \overline{\pi}_i(\overline{t})$ for any attribute $a_i \in$* `attr`*$(\overline{t})$, where $\overline{\pi}$ is the projection operator in the abstract domain and $\overline{\pi}_i(\overline{t})$ represents the projected abstract values corresponding to the i^{th} attribute in $\overline{t}$.*

Definition 3.3 (Abstract Database States) *An abstract database $\overline{d}$ is a set of abstract tables $\{\overline{t_i} \mid i \in I_x\}$ for a given set of indexes I_x. An abstract database environment is defined as a function $\rho_{\overline{d}}$ whose domain is I_x, such that for $i \in I_x$, $\rho_{\overline{d}}(i) = \overline{t_i}$.*

Definition 3.4 (Abstract States) *An abstract state $\overline{\rho} \in \overline{\Sigma}_{dba}$ for database applications is defined as a tuple $(\rho_{\overline{d}}, \rho_{\overline{a}})$ where $\rho_{\overline{d}} \in \overline{\mathfrak{E}}_{dbs}$ and $\rho_{\overline{a}} \in \overline{\mathfrak{E}}_{aps}$ are an abstract database environment and an abstract application environment respectively.*

[1]Notice that for some abstract domain the abstraction function may not exist.

Example 15 illustrates the abstraction of a simple table in various relational and non-relational abstract domains.

Example 15 *Consider a concrete database that contains a table t shown in Figure 3.2(a). The table provides information about the employees of a company. Let us consider the abstract domain of intervals. Considering an abstraction where employees id, salaries ages and department number of the employees are abstracted by the elements from the domain of intervals. The abstract table $\bar{t}$ corresponding to t is depicted in Figure 3.2(b). Similarly one can abstract the*

eid	*sal*	*age*	*dno*
1	1500	35	10
2	800	28	20
3	2500	50	10
4	3000	62	10

(a) Concrete table t

eid	*sal*	*age*	*dno*
[1,4]	[800,3000]	[28,62]	[10, 20]

(b) Abstract table $\bar{t}$ in interval domain

Figure 3.2: Concrete and its corresponding Abstract Database

table using relational abstract domains also. The corresponding abstract representation of t in octagons and polyhedra domain are represented by CDBM m_t and P_t respectively as

$$\mathsf{m}_t \overset{rep}{=} \Big\{ -eid \leqslant -1,\ eid \leqslant 4,\ -sal \leqslant -800,\ sal \leqslant 3000,$$
$$-age \leqslant -28, age \leqslant 62, -dno \leqslant -10, dno \leqslant 20 \Big\},\ and$$
$$\mathsf{P}_t \overset{rep}{=} \Big\{ eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000,$$
$$age \geqslant 28, -age \geqslant -62, dno \geqslant 10, -dno \geqslant -20 \Big\}$$

Beside these classical abstract domains, one may also consider other kinds of treatment to database instances as abstraction. For example, sets of abstract rows when each row corresponds to the union of abstractions of some partitions of the concrete table. Similarly, the form of conditional tables (c-tables) proposed in [68] can also be viewed as an abstraction.

In order to formalize the abstract semantics of database applications, we define the following sound abstract transition function corresponding to its concrete counterpart $\mathscr{T}_{dba}$ (defined in equation 3.1):

$$\overline{\mathscr{T}}_{dba} : \mathbb{C} \times \overline{\Sigma}_{dba} \rightarrow \overline{\Sigma}_{dba} \tag{3.3}$$

which specifies the successor abstract state $(\rho_{\overline{d'}}, \rho_{\overline{a'}}) \in \overline{\Sigma}_{dba}$ when a statement $c \in \mathbb{C}$

executes on an abstract state $(\rho_{\overline{d}}\,,\ \rho_{\overline{a}}) \in \overline{\Sigma}_{dba}$. The soundness of the abstract semantics relies on the soundness of $\overline{\mathscr{T}}_{dba}$ w.r.t. $\mathscr{T}_{dba}$.

Observe that since our abstraction over-approximates the attributes values, the multiplicity of attribute values may not be taken into account in abstract database representation and hence in abstract semantics, as per the definition of abstract table environment.

The abstract semantics of SQL statements in various abstract domains following equation 3.3 are defined below.

3.3.2.1 Domain of Intervals

Let us recall the semantic function $\overline{\mathscr{T}}_{dba}$ defined in equation 3.3 which specifies the successor abstract state $(\rho_{\overline{d'}}\,,\ \rho_{\overline{a'}}) \in \overline{\Sigma}_{dba}$ when a statement $c \in \mathbb{C}$ executes on an abstract state $(\rho_{\overline{d}}\,,\ \rho_{\overline{a}}) \in \overline{\Sigma}_{dba}$.

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

$$\begin{aligned}
&\overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!]\overline{\rho} \\
&\quad = \overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!](\rho_{\overline{d}}\,,\ \rho_{\overline{a}}) \\
&\quad = \overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!](\rho_{\overline{t}}\,,\ \rho_{\overline{a}}) \\
&\qquad\quad \text{where } t = \mathit{target}(\langle A, \phi \rangle) \text{ and } \exists \overline{t} \in \rho_{\overline{d}} :\ t \in \gamma(\overline{t}) \\
&\quad = \overline{\mathscr{T}}_{dba}[\![\langle A \rangle]\!](\rho_{\overline{TM}}\,,\ \rho_{\overline{a}}) \sqcup (\rho_{\overline{FM}}\,,\ \rho_{\overline{a}}) \\
&\quad = (\rho_{\overline{TM'}}\,,\ \rho_{\overline{a}}) \sqcup (\rho_{\overline{FM}}\,,\ \rho_{\overline{a}}) \\
&\quad = (\rho_{\overline{TM'}} \sqcup \rho_{\overline{FM}}\,,\ \rho_{\overline{a}} \sqcup \rho_{\overline{a}}) \\
&\quad = (\rho_{\overline{t'}}\,,\ \rho_{\overline{a}})
\end{aligned} \tag{3.4}$$

where

$$(\rho_{\overline{TM}}\,,\ \rho_{\overline{a}}) \in \overline{\mathscr{T}_f}[\![\phi]\!](\rho_{\overline{t}}\,,\ \rho_{\overline{a}}) \text{ and } (\rho_{\overline{FM}}\,,\ \rho_{\overline{a}}) \in \overline{\mathscr{T}_f}[\![\neg\phi]\!](\rho_{\overline{t}}\,,\ \rho_{\overline{a}})$$

Observe that $\overline{TM}$ and $\overline{FM}$ are the abstract database states obtained by using the filtering semantics function $\overline{\mathscr{T}_f}$ based on the satisfaction of ϕ. In particular, $\overline{TM}$ denotes the part of the abstract database state for which ϕ is true, whereas $\overline{FM}$ denotes the abstract database state for which ϕ is false. After performing the update action A on $\overline{TM}$, the resultant abstract state $\overline{TM'}$ is obtained. Finally, component-wise join operation between $\overline{TM'}$ and $\overline{FM}$ yields the resultant abstract state $\overline{t'}$. Observe that, in order to ensure the soundness, both $\overline{TM}$ and $\overline{FM}$ include the information for which ϕ results in *"may be true or false"*. We illustrate this in Example 16.

Example 16 *Consider the abstract domain of intervals $\mathbb{I}$. Given the concrete database table t shown in Table 3.3(a), its corresponding abstract version $\bar{t}$ replacing concrete values by their properties from $\mathbb{I}$ is depicted in Table 3.3(b). Similarly, given an application environment*

(a) Concrete table t

eid	*sal*	*age*	*dno*
1	1500	35	10
2	800	28	20
3	2500	50	10
4	3000	62	10

(b) Abstract table $\bar{t}$ in interval domain

eid	*sal*	*age*	*dno*
[1,4]	[800,3000]	[28,62]	[10, 20]

Table 3.3: Concrete and its corresponding Abstract Database

$\rho_a = \langle x \rightarrow 100\rangle$ where x is an application variable, its corresponding abstract application environment in $\mathbb{I}$ is $\rho_{\bar{a}} = \langle x \rightarrow [100, 100]\rangle$.

Now consider the following `UPDATE` *statement:*

$$Q_{upd} : \texttt{UPDATE}\ t\ \texttt{SET}\ sal = sal + x\ \texttt{WHERE}\ sal \geqslant 1500$$

Here $A = \texttt{UPDATE}(\langle sal\rangle, \langle sal + x\rangle)$ and $\phi = sal \geqslant 1500$. The concrete semantics yields the resultant table t' shown in Table 3.4.

eid	*sal*	*age*	*dno*
1	1600	35	10
2	800	28	20
3	2600	50	10
4	3100	62	10

Table 3.4: Execution result t' by Q_{upd} on t

The abstract semantics of Q_{upd} w.r.t. $\overline{\rho} = (\rho_{\bar{t}}\ ,\ \rho_{\bar{a}})$ is

$$\overline{\mathscr{T}}_{dba}[\![\langle A, \phi\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}})$$
$$=\overline{\mathscr{T}}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + x\rangle),\ sal \geqslant 1500\Big\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}})$$

$$=\overline{\mathscr{T}}_{dba}[\![\Big\langle \mathit{UPDATE}(\langle sal\rangle, \langle sal+x\rangle)\Big\rangle]\!](\rho_{\overline{TM}}, \rho_{\overline{a}}) \sqcup (\rho_{\overline{FM}}, \rho_{\overline{a}})$$

$$=(\rho_{\overline{TM'}}\,,\ \rho_{\overline{a}}) \ \sqcup\ (\rho_{\overline{FM}}\,,\ \rho_{\overline{a}})$$

$$=(\rho_{\overline{TM'}} \sqcup \rho_{\overline{FM}}\,,\ \rho_{\overline{a}} \sqcup \rho_{\overline{a}})$$

$$=(\rho_{\overline{t'}}\,,\ \rho_{\overline{a}})$$

where

$$\rho_{\overline{TM}} = \overline{\mathscr{T}_f}[\![sal \ \geqslant\ 1500]\!](\rho_{\overline{t}}) \ =\ \rho_{\overline{t}}\Big[sal \ \leftarrow\ [1500,\ 3000]\Big]$$
$$\rho_{\overline{FM}} = \overline{\mathscr{T}_f}[\![\neg(sal \ \geqslant\ 1500)\ ]\!](\rho_{\overline{t}}) \ =\ \rho_{\overline{t}}\Big[sal \ \leftarrow\ [800,\ 1499]\Big]$$

$$\begin{aligned}\rho_{\overline{TM'}} &= \overline{\mathscr{T}}_{dba}[\![\mathit{UPDATE}(\langle sal\rangle, \langle sal+x\rangle)]\!](\rho_{\overline{TM}}\,,\ \rho_{\overline{a}})\\ &= \overline{\mathscr{T}_c}[\![sal = sal + x]\!](\rho_{\overline{TM}}\,,\ \rho_{\overline{a}})\\ &= \overline{\mathscr{T}_c}[\![sal = sal + [100,100]]\!](\rho_{\overline{TM}}\,,\ \rho_{\overline{a}})\\ &= \rho_{\overline{TM}}\Big[sal \leftarrow [1600,3100]\Big]\end{aligned}$$

Tables 3.5(a) and 3.5(b) depict $\overline{TM}$ and $\overline{FM}$ respectively. After performing the update action A on $\overline{TM}$, the resultant abstract table $\overline{TM'}$ is shown in Table 3.5(c). Finally, component-wise join operation between $\overline{TM'}$ and $\overline{FM}$ yields the resultant table $\overline{t'}$ depicted in Table 3.5(d). Observe that the abstract semantics is sound, i.e. $t' \in \gamma(\overline{t'})$.

(a) Abstract table $\overline{TM}$

eid	*sal*	*age*	*dno*
[1,4]	[1500,3000]	[28,62]	[10, 20]

(b) Abstract table $\overline{FM}$

eid	*sal*	*age*	*dno*
[1,4]	[800,1499]	[28,62]	[10, 20]

(c) Abstract table $\overline{TM'}$

eid	*sal*	*age*	*dno*
[1,4]	[1600,3100]	[28,62]	[10, 20]

(d) Abstract table $\overline{t'} = \overline{TM'} \sqcup \overline{FM}$

eid	*sal*	*age*	*dno*
[1,4]	[800,3100]	[28,62]	[10, 20]

Table 3.5: Execution results of Q_{upd} on $\overline{t}$

3.3.2.2 Domain of Octagons

In case of database applications, we consider two different environments: database environment $\rho_d \in \mathfrak{E}_{dbs}$ and application environment $\rho_a \in \mathfrak{E}_a$. To determine abstract semantics of database statements in the domain of octagons, we define the abstract state $\overline{\rho} \in \overline{\Sigma}_{dba}$ as

$$\overline{\rho} = \langle \mathsf{m}_d, \mathsf{m}_a \rangle$$

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 3.3, the abstract semantic function for database statements $Q = \langle A, \phi \rangle$ is defined as: $\overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!](\mathsf{m}_d, \mathsf{m}_a) = \overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!](\mathsf{m}_t\ ,\ \mathsf{m}_a) = (\mathsf{m}_{t'}\ ,\ \mathsf{m}_a)$ where m_t is the octagonal representation of the concrete table t which acts as the target of Q and $\mathsf{m}_{t'}$ is the octagonal representation of the resultant table t'. Below is the abstract semantics for update statement.

$$\begin{aligned}
&\overline{\mathscr{T}}_{dba}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}), \phi \rangle]\!](\mathsf{m}_t\ ,\ \mathsf{m}_a) \\
&= \overline{\mathscr{T}}_{dba}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}) \rangle]\!](\mathsf{m}_{TM}\ ,\ \mathsf{m}_a) \sqcup (\mathsf{m}_{FM}\ ,\ \mathsf{m}_a) \\
&= (\mathsf{m}_{TM'}\ ,\ \mathsf{m}_a) \sqcup (\mathsf{m}_{FM}\ ,\ \mathsf{m}_a) \\
&= (\mathsf{m}_{TM'} \sqcup \mathsf{m}_{FM}\ ,\ \mathsf{m}_a \sqcup \mathsf{m}_a) \\
&= (\mathsf{m}_{t'}\ ,\ \mathsf{m}_a) \qquad\qquad \text{where}
\end{aligned}$$

$$\overline{\mathscr{T}_f}[\![\neg\phi]\!](\mathsf{m}_t, \mathsf{m}_a) = (\mathsf{m}_{FM}, \mathsf{m}_a) \text{ and } \overline{\mathscr{T}_f}[\![\phi]\!](\mathsf{m}_t, \mathsf{m}_a) = (\mathsf{m}_{TM}, \mathsf{m}_a)$$

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

Example 17 *Consider the concrete table t shown in Table 3.3(a). Consider the concrete application environment $\rho_a = \langle x \longrightarrow 100 \rangle$ where x is an application variable. The corresponding abstract representation of ρ_t and ρ_a in octagon domain are represented by CDBM m_t and m_a respectively as*

$$\begin{aligned}
\mathsf{m}_t \overset{rep}{=} &\Big\{ -eid \leqslant -1,\ eid \leqslant 4,\ \ -sal \leqslant -800,\ sal \leqslant 3000, \\
&-age \leqslant -28, age \leqslant 62, -dno \leqslant -10, dno \leqslant 20 \Big\},\ \textit{and} \\
\mathsf{m}_a \overset{rep}{=} &\Big\{ x \leqslant 100, -x \leqslant -100 \Big\}.
\end{aligned}$$

Consider the following `UPDATE` *statement*

$$Q_{upd}: \texttt{UPDATE}\ t\ \texttt{SET}\ sal = sal + x\ \texttt{WHERE}\ age \geqslant 35$$

where $A = \texttt{UPDATE}(\langle sal\rangle, \langle sal{+}x\rangle)$ *and* $\phi = age \geqslant 35$. *The abstract semantics w.r.t.* $\overline{\rho} = (\mathsf{m}_t, \mathsf{m}_a)$ *is*

$$\begin{aligned}
&\overline{\mathscr{T}}_{dba}[\![\langle A, \phi\rangle]\!](\mathsf{m}_t, \mathsf{m}_a)\\
&\quad=\overline{\mathscr{T}}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + x\rangle),\ age \geqslant 35\Big\rangle]\!](\mathsf{m}_t, \mathsf{m}_a)\\
&\quad=\overline{\mathscr{T}}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + x\rangle)\Big\rangle]\!](\mathsf{m}_{TM}, \mathsf{m}_a) \sqcup (\mathsf{m}_{FM}, \mathsf{m}_a)\\
&\quad=(\mathsf{m}_{TM'}, \mathsf{m}_a)\ \sqcup\ (\mathsf{m}_{FM}, \mathsf{m}_a)\\
&\quad=(\mathsf{m}_{TM'} \sqcup \mathsf{m}_{FM}\ ,\ \mathsf{m}_a \sqcup \mathsf{m}_a)\\
&\quad=(\mathsf{m}_{t'}\ ,\ \mathsf{m}_a)
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{m}_{TM} \overset{rep}{=} \Big\{&-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000, -age\\
&\leqslant -35,\ age \leqslant 62,\ -dno \leqslant -10,\ dno \leqslant 20\Big\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{m}_{FM} \overset{rep}{=} \Big\{&-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000, -age\\
&\leqslant -28,\ age \leqslant 34,\ -dno \leqslant -10, dno \leqslant 20\Big\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{m}_{TM'} \overset{rep}{=} \Big\{&-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -900, sal \leqslant 3100, -age\\
&\leqslant -35,\ age \leqslant 62,\ -dno \leqslant -10,\ dno \leqslant 20\Big\}
\end{aligned}$$

Note that we can follow an alternative equivalent way of abstract state representation by combining both CDBM of m_d and m_a for the sake of simplicity. Let p and q denote the numbers of database variables and application variables respectively. Given m_d 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_d

and m_a. In the subsequent chapters we define abstract semantics w.r.t. abstract state $\overline{\rho} = \mathsf{m}$.

3.3.2.3 Domain of Polyhedra

Let us define the abstract semantics for four database operations in the domain of polyhedra. Like octagon domain, given $\overline{\rho} = \langle \mathsf{P}_d, \mathsf{P}_a \rangle \in \overline{\Sigma}_{dba}$ where P_d and P_a are polyhedra representation of database values and application variables values respectively. According to equation 3.3, the abstract semantic function for database statements $Q = \langle A, \phi \rangle$ is defined as: $\overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!](\mathsf{P}_d, \mathsf{P}_a) = \overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!](\mathsf{P}_t\ ,\ \mathsf{P}_a) = (\mathsf{P}_{t'}\ ,\ \mathsf{P}_a)$ where P_t is the polyhedron representation of the concrete table t which acts as the target of Q, and $\mathsf{P}_{t'}$ is the polyhedron representation of the resultant table t'. Below is the abstract semantics for update statement.

$$
\begin{aligned}
&\overline{\mathscr{T}}_{dba}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}), \phi \rangle]\!](\mathsf{P}_t\ ,\ \mathsf{P}_a) \\
&\quad = \overline{\mathscr{T}}_{dba}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}) \rangle]\!](\mathsf{P}_{TM}\ ,\ \mathsf{P}_a) \sqcup (\mathsf{P}_{FM}\ ,\ \mathsf{P}_a) \\
&\quad = (\mathsf{P}_{TM'}\ ,\ \mathsf{P}_a) \sqcup (\mathsf{P}_{FM}\ ,\ \mathsf{P}_a) \\
&\quad = (\mathsf{P}_{TM'} \sqcup \mathsf{P}_{FM}\ ,\ \mathsf{P}_a \sqcup \mathsf{P}_a) \quad = \quad (\mathsf{P}_{t'}\ ,\ \mathsf{P}_a)
\end{aligned}
$$

where

$$
\overline{\mathscr{T}_f}[\![\neg\phi]\!](\mathsf{P}_t, \mathsf{P}_a) = (\mathsf{P}_{FM}, \mathsf{P}_a) \text{ and } \overline{\mathscr{T}_f}[\![\phi]\!](\mathsf{P}_t, \mathsf{P}_a) = (\mathsf{P}_{TM},\ \mathsf{P}_a)
$$

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

Example 18 *Consider the concrete table t shown in Table 3.3(a). Consider the concrete application environment $\rho_a = \langle x \rightarrow 0.2 \rangle$ where x is an application variable. The corresponding abstract representation of ρ_t and ρ_a in polyhedra domain are represented by P_t and P_a respectively as*

$$
\begin{aligned}
\mathsf{P}_t &\overset{rep}{=} \Big\{ eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, \\
&\quad age \geqslant 28, -age \geqslant -62, dno \geqslant 10, -dno \geqslant -20 \Big\}, \text{ and} \\
\mathsf{P}_a &\overset{rep}{=} \Big\{ -x \geqslant -0.2, x \geqslant 0.2 \Big\}.
\end{aligned}
$$

Consider the following `UPDATE` *statement*

$$Q_{upd} = \texttt{UPDATE}\ t\ \texttt{SET}\ sal = sal + sal \times x\ \texttt{WHERE}\ dno + age \geqslant 60$$

where $A = \texttt{UPDATE}(\langle sal\rangle, \langle sal \times x\rangle)$ *and* $\phi = dno + age \geqslant 60$. *The abstract semantics w.r.t.* $\overline{\rho} = (\mathsf{P}_t, \mathsf{P}_a)$ *is*

$$\begin{aligned}
&\overline{\mathscr{T}}_{dba}[\![\langle A, \phi\rangle]\!](\mathsf{P}_t, \mathsf{P}_a)\\
&=\overline{\mathscr{T}}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal \times x\rangle),\ dno + age \geqslant 60\Big\rangle]\!](\mathsf{P}_t, \mathsf{P}_a)\\
&=\overline{\mathscr{T}}_{dba}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal \times x\rangle)\Big\rangle]\!](\mathsf{P}_{TM}, \mathsf{P}_a) \sqcup (\mathsf{P}_{FM}, \mathsf{P}_a)\\
&=(\mathsf{P}_{TM'}, \mathsf{P}_a)\ \sqcup\ (\mathsf{P}_{FM}, \mathsf{P}_a)\\
&=(\mathsf{P}_{TM'} \sqcup \mathsf{P}_{FM}\ ,\ \mathsf{P}_a \sqcup \mathsf{P}_a)\\
&=(\mathsf{P}_{t'}\ ,\ \mathsf{P}_a)
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{P}_{TM} \overset{rep}{=} \Big\{&eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 40,\\
&-age \geqslant -62, dno \geqslant 10, -dno \geqslant -20, dno + age \geqslant 60\Big\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{P}_{FM} \overset{rep}{=} \Big\{&eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 28,\\
&-age \geqslant -49, dno \geqslant 10, -dno \geqslant -20, -dno - age \geqslant -59\Big\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{P}_{TM'} \overset{rep}{=} \Big\{&eid \geqslant 1, -eid \geqslant -4, sal \geqslant 960, -sal \geqslant -3600, age \geqslant 40,\\
&-age \geqslant -62, dno \geqslant 10, -dno \geqslant -20, dno + age \geqslant 60\Big\}
\end{aligned}$$

Observe that alternatively, like octagon abstract domain, for the sake of simplicity, we may combine both P_d and P_a into a single polyhedra P as an abstract program state. In the subsequent chapters we define abstract semantics suitable for independency computations w.r.t. an abstract state $\overline{\rho}= \mathsf{P}$ in the domain of polyhedra.

3.4 Correctness

Lemma 3.1 *Given an abstract state $\overline{\rho} = (\rho_{\overline{t}}, \rho_{\overline{a}})$, the abstract semantics function $\overline{\mathscr{T}}_{dba}$ is correct w.r.t. the concretization function γ if $\forall Q \in \mathbb{Q}, \forall \rho_t \in \gamma(\rho_{\overline{t}}), \ \forall \rho_a \in \gamma(\rho_{\overline{a}}) \ : \mathscr{T}_{dba}[\![Q]\!](\rho_t, \rho_a) \subseteq \gamma(\overline{\mathscr{T}}_{dba}[\![Q]\!](\rho_{\overline{t}}, \rho_{\overline{a}}))$.*

Proof 1 *Given an abstract state $\overline{\rho}$, the abstract semantics of $Q = \langle A, \phi \rangle \in \mathbb{Q}$ w.r.t. $\overline{\rho}$ is defined as $\overline{\mathscr{T}}_{dba}[\![Q]\!]\overline{\rho} = \overline{\mathscr{T}}_{dba}[\![\langle A, \phi \rangle]\!]\overline{\rho} = \overline{\mathscr{T}}_{dba}[\![\langle A \rangle]\!]\rho_{\overline{TM}} \sqcup \rho_{\overline{FM}} = \rho_{\overline{TM'}} \sqcup \rho_{\overline{FM}} = \overline{\rho}'$, where $\rho_{\overline{TM}}$ represents abstract database state which satisfies ϕ and $\rho_{\overline{FM}}$ represents abstract database state which does not satisfy ϕ. Abstract state which, due to abstraction, may satisfy ϕ is included in both $\rho_{\overline{TM}}$ and $\rho_{\overline{FM}}$. The state $\rho_{\overline{TM'}}$ is obtained by performing A on $\rho_{\overline{TM}}$. Now let us consider a concrete state $\rho \in \gamma(\overline{\rho})$. The concrete semantics of $Q = \langle A, \phi \rangle$ w.r.t. ρ is $\mathscr{T}_{dba}[\![Q]\!]\rho = \mathscr{T}_{dba}[\![\langle A, \phi \rangle]\!]\rho = \mathscr{T}_{dba}[\![\langle A \rangle]\!]\rho_T \cup \rho_F = \rho_{T'} \cup \rho_F = \rho'$, where ρ_T and ρ_F represent concrete database states based on the satisfaction and dissatisfaction of ϕ respectively. As ϕ in the abstract domain considers three valued logic due to the imprecision introduced in the abstraction, and since both $\rho_{\overline{TM}}$ and $\rho_{\overline{FM}}$ include the database state for which ϕ evaluates to* "may be true or false"*, assuming local correctness of the functions and relations involved in ϕ we get $\rho_T \in \gamma(\rho_{\overline{TM}})$ and $\rho_F \in \gamma(\rho_{\overline{FM}})$. Similarly, the local correctness of the operations involved in A guarantees $\rho_{T'} \in \gamma(\rho_{\overline{TM'}})$ [52]. Considering the Galois connection between concrete and abstract database and application domains, we therefore get $(\rho_{T'} \cup \rho_F) \in \gamma(\rho_{\overline{TM'}} \sqcup \rho_{\overline{FM}})$ and so $\rho' \in \gamma(\overline{\rho}')$. This is depicted below:*

$$\begin{tikzcd}
\rho \arrow[r, "\mathscr{T}_{dba}[\![Q]\!]"] & \rho' \subseteq \gamma(\overline{\rho}')) \\
\overline{\rho} \arrow[u, "\gamma"'] \arrow[r, "\overline{\mathscr{T}}_{dba}[\![Q]\!]"'] & \overline{\rho}' \arrow[u, "\gamma"']
\end{tikzcd}$$

3.5 Discussions

Let us summarize the strengths and limitations of various relational and non-relational abstract domains which we have used above to define abstract semantics of database applications. As abstraction in the interval domain does not capture any relation among variables or attributes, this yields a highly approximated analysis-results. On the other hand, although abstract semantics in both octagon and polyhedra domains capture relationships among variables or attributes, the octagon domain allows a weak form of constraints compared to that in polyhedra domain. Due to this reason, analysis in octagon domain is less precise than that in the polyhedra domain. Intuitively, preciseness of the analysis in relational abstract domain improves significantly when more number of relations among variables or attributes is present in the program itself, e.g. in the `WHERE` clause or in the conditional or iterative statements. In terms of algorithmic efficiency, octagon domain always lies between interval and polyhedra. Analyses (involving all common operations e.g. emptiness test, inclusion, etc.) in polyhedra abstract domain experience an exponential ($O(2^n)$) worst-case time complexity [76], 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 [94]. 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 3.6. This is to observe that, in relational database, the semantics of NULL value is as follows [44]:

1. Value unknown (exists but is not known).

2. Value not available (exists but is purposely withheld).

3. Value not applicable (the attribute is undefined for this tuple).

When initial database in unknown, the database is over-approximated by considering the top values of attributes from their corresponding abstract domains. Therefore, this captures the above-mentioned semantics of NULL value as well. Moreover, we have defined the abstract semantic functions $\overline{\mathscr{T}}_{dba}$ and $\overline{\mathscr{T}}_{dep}$ in terms of $\rho_{\overline{TM}}$, $\rho_{\overline{FM}}$ and $\rho_{\overline{TM}'}$,

Domain	**Invariants**	**Time cost**	**Memory cost**	**Precision**
Interval	$x \in \Big\{[l,\ h] \mid l,\ h \in \mathbb{R}, l \leq h, x \in \mathbb{V}\Big\}$	$O(n)$	$O(n)$	*low*
Octagon	$\pm x_i \pm x_j \leqslant k,\ x_i, x_j \in \mathbb{V} \wedge k \in \mathbb{R} \cup \{\infty\}$	$O(n^3)$	$O(n^2)$	*medium*
Polyhedra	$\Sigma_{i=1}^{n} a_i x_i \geqslant k, x_i \in \mathbb{V} \wedge a_i, k \in \mathbb{R}^n$	$O(2^n)$	$O(2^n)$	*high*
Powerset	$\wp(\overline{\mathbb{D}})$	*Depends on* $\overline{\mathbb{D}}$	*Depends on* $\overline{\mathbb{D}}$	*Improves w.r.t* $\overline{\mathbb{D}}$

Table 3.6: A summary on various abstract domains

which also include the results when the condition-part ϕ leads to "*may be true or false*" in three-valued logic and the results on applying the action-part A on NULL values. This guarantees a sound approximation of real database systems where we often experience NULL values in the database.

3.6 Conclusions

In this chapter, we define an abstract semantics of database languages embedding SQL in the Abstract Interpretation framework. We consider various non-relational and relational abstract domains of interests. We prove that the abstraction of database language, following the Abstract Interpretation framework, always guarantees the soundness property. In the subsequent part of the thesis, we show how this semantic formalism may serve as a powerful dependency analysis framework, even in the case of undecidable scenarios, enabling one to tune between precision and efficiency.

CHAPTER 4

Concrete and Abstract Semantics of Hibernate Query Language (HQL)

Preface

As stated earlier, the thesis refers database applications embedding either SQL and HQL. Therefore, like chapter 3, we also define the formal syntax and concrete semantics of HQL, followed by its abstraction in various domains of interest. In particular, we refer various `session` methods which act as the central interface between an application and its underlying database.

4.1 Formal Syntax of HQL

Hibernate query language (HQL) is an Object-Relational Mapping (ORM) tool which remedies the paradigm mismatch between object-oriented languages and relational database models and hence simplifies the data creation, data manipulation, data access [11, 12, 42]. It provides a unified platform for the programmers to develop object-oriented applications to interact with databases, without knowing much details about the underlying databases. The attractive feature of Hibernate is the presence of `Hibernate Session` which provides a central interface between the application and database and acts as a persistence manager. In HQL, an object is transient if it has just been instantiated using the new operator. Transient instances will be destroyed by the garbage collector if the application does not hold a reference anymore. A persistent instance, on the other hand, has a representation in the database and an identifier value assigned to it. Given an object, the `Hibernate Session` is used to make the object persistent. Various methods in `Hibernate Session` are used to propagate object's states from memory to the database (or vice versa) and to synchronize both the states when a change is made to the persistent objects [99].

Syntax of HQL is similar to the object oriented constructs along with SQL variants through `Session` objects. The syntactic sets and the abstract syntax of HQL is depicted in Table 4.1. Like Object-Oriented Program (OOP) [86], HQL programs are composed of a set of classes including `main` class. That is, a HQL program $\mathcal{P}$ is defined as $\mathcal{P} = \langle c_{main}, \texttt{L}\rangle$ where $c_{main} \in \texttt{Class}$ is the main class and $\texttt{L} \subset \texttt{Class}$ are the other classes. Similarly, a class $c \in \texttt{Class}$ contains a set of fields and methods, and therefore, is defined as a triplet $c = \langle \texttt{init}, \texttt{F}, \texttt{M}\rangle$, where `init` is the constructor, `F` is the set of fields, and `M` is the set of member methods.

In abstract syntax, we denote a `Session` method by a triplet $\langle \texttt{C}, \phi, \texttt{OP}\rangle$ where `OP` is the operation to be performed on the database tuples corresponding to a set of objects of classes $c \in \texttt{C}$ satisfying the condition ϕ. For instance, consider the following update statement which is invoked through a session object '`ses`':

$$Query\ Q = ses.createQuery(\text{“}\texttt{UPDATE std SET}\ rank = rank + 1\ \texttt{WHERE}\ mark > 500\text{”})$$

The abstract syntax of Q is denoted by $\langle \texttt{C}, \phi, \texttt{OP}\rangle = \langle \{\texttt{std}\}, mark > 500, rank = rank + 1\rangle$

Constants and Variables			
n	$\in$	$\mathbb{N}$	Set of Integers
v	$\in$	$\mathbb{V}$	Set of Variables
Arithmetic and Boolean Expressions			
exp	$\in$	$\mathbb{E}$	Set of Arithmetic Expressions
exp	::=	$n \mid v \mid exp_1 \oplus exp_2$	
		where $\oplus \in \{+, -, *, /\}$	
b	$\in$	$\mathbb{B}$	Set of Boolean Expressions
b	::=	true | false | $exp_1 \otimes exp_2 \mid \neg b \mid b_1 \oslash b_2$	
		where $\otimes \in \{\leq, \geq, ==, >, \neq, \dots\}$ and $\oslash \in \{\vee, \wedge\}$	
Well-formed Formulas			
τ	$\in$	$\mathbb{T}$	Set of Terms
τ	::=	$n \mid v \mid f_n(\tau_1, \tau_2, ..., \tau_n)$	
		where f_n is an n-ary function.	
a_f	$\in$	$\mathbb{A}_f$	Set of Atomic Formulas
a_f	::=	$R_n(\tau_1, \tau_2, ..., \tau_n) \mid \tau_1 == \tau_2$	
		where $R_n(\tau_1, \tau_2, ..., \tau_n) \in \{true, false\}$	
ϕ	$\in$	$\mathbb{W}$	Set of Well-formed Formulas
ϕ	::=	$a_f \mid \neg\phi \mid \phi_1 \oslash \phi_2$	
		where $\oslash \in \{\vee, \wedge\}$	
HQL Functions			
$g(\vec{e})$	::=	`GROUP BY`($\vec{exp}$) | id	
		where $\vec{exp} = \langle exp_1, ..., exp_n \mid exp_i \in \mathbb{E}\rangle$	
r	::=	`DISTINCT` | `ALL`	
s	::=	`AVG` | `SUM` | `MAX` | `MIN` | `COUNT`	
$h(exp)$	::=	$s \circ r(exp)$ | `DISTINCT`(exp) | id	
$h(*)$	::=	`COUNT(*)`	
		where * represents a list of database attributes denoted by $\vec{v_d}$	
$\vec{h}(\vec{x})$	::=	$\langle h_1(x_1), ..., h_n(x_n)\rangle$	
		where $\vec{h} = \langle h_1, ..., h_n\rangle$ and $\vec{x} = \langle x_1, ..., x_n \mid x_i = exp \vee x_i = *\rangle$	
$f(\vec{exp})$	::=	`ORDER BY ASC`($\vec{exp}$) | `ORDER BY DESC`($\vec{exp}$) | id	
`Session` Methods			
c	$\in$	`Class`	Set of Classes
c	::=	⟨`init`,`F`,`M`⟩	
		where `init` is the constructor, `F` ⊆ `Var` is the set of fields, and `M` is the set of methods.	
m_{ses}	$\in$	$\mathtt{M}_{ses}$	Set of `Session` methods
m_{ses}	::=	⟨`C`,ϕ,`OP`⟩	
		where `C` ⊆ `Class`	
`OP`	::=	`SEL`($v_a, f(\vec{exp'}), r(\vec{h}(\vec{x})), \phi, g(\vec{exp})$)	
	|	`UPD`($\vec{v}, \vec{exp}$)	
	|	`SAVE(obj)`	
	|	`DEL()`	
		where ϕ represents '`HAVING`' clause and `obj` denotes an instance of a class.	

Table 4.1: Formal Syntax of HQL `Session` Methods

The descriptions of `OP` in various `Session` methods are as follows:

- $\left\langle \texttt{C}, \phi, \texttt{SAVE(obj)} \right\rangle = \left\langle \{c\}, \mathit{false}, \texttt{SAVE(obj)} \right\rangle$: Stores the state of the object `obj` in the database table t, where t corresponds to the POJO class c and `obj` is the instance of c. The pre-condition ϕ is false as the method does not identify any existing tuples in the database.
- $\left\langle \texttt{C}, \phi, \texttt{UPD}(\vec{x}, \vec{exp}) \right\rangle = \left\langle \{c\}, \phi, \texttt{UPD}(\vec{v}, \vec{exp}) \right\rangle$: Updates the attributes corresponding to the class fields $\vec{x}$ by $\vec{exp}$ in the database table t for the tuples satisfying ϕ, where t corresponds to the POJO class c.
- $\left\langle \texttt{C}, \phi, \texttt{DEL}() \right\rangle = \left\langle \{c\}, \phi, \texttt{DEL}() \right\rangle$: Deletes the tuples satisfying ϕ in t, where t is the database table corresponding to the POJO class c.
- $\left\langle \texttt{C}, \phi, \texttt{SEL}\left(v_a, f(\vec{exp'}),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})\right) \right\rangle$: Selects information from the database tables corresponding to the set of POJO classes `C`, and returns the equivalent representations in the form of objects.

It is immediate that in case of `SAVE`() the condition ϕ is false and `C` is singleton set $\{c\}$. As `UPD`() and `DEL`() always target single class, the set `C` is also singleton $\{c\}$ in those cases. However, `C` may not be singleton in case of `SEL`().

4.2 Concrete Semantics of HQL

In this section, we define the semantics of HQL by (*i*) extending the of semantics Object-Oriented Programming (OOP) [86] and (*ii*) defining the semantics of `Session` methods in terms of the semantics of database statements [52].

4.2.1 Concrete Semantics of OOP

Let us first recall from [86] the concrete semantics of object-oriented programming languages. Object-oriented programming languages consist of a set of classes including a main class from where execution starts. Each class contains a set of attributes and a set of methods - called members of the class. Therefore, a program $\mathcal{P}$ in OOP is defined as $\mathcal{P} = \langle c_{main}, \texttt{L} \rangle$ where `Class` denotes the set of classes, $c_{main} \in \texttt{Class}$ is the main class, $\texttt{L} \subset \texttt{Class}$ are the other classes present in $\mathcal{P}$.

A class $c \in \texttt{Class}$ is defined as a triplet $c = \langle \texttt{init}, \texttt{F}, \texttt{M} \rangle$ where `init` is the constructor, `F` is the set of fields, and `M` is the set of member methods in c.

Let `Var`, `Val` and `Loc` be the set of variables, the domain of values and the set of memory locations respectively. The set of environments, stores and states are defined below:

- The set of environments is defined as $\texttt{Env} : [\texttt{Var} \longrightarrow \texttt{Loc}]$
- The set of stores is defined as $\texttt{Store} : [\texttt{Loc} \longrightarrow \texttt{Val}]$
- The set of states is defined as $\Sigma : \texttt{Env} \times \texttt{Store}$. So, a state $\rho \in \Sigma$ is denoted by a tuple $\langle e, s \rangle$ where $e \in \texttt{Env}$ and $s \in \texttt{Store}$.

4.2.1.1 Trace Semantics

Let $c = \langle \texttt{init}, \texttt{F}, \texttt{M} \rangle$ and m be `init` or $m \in \texttt{M}$. Let pc_{exit} and pc_{in} be the entry and exit point of the m. Furthermore, let $\rightarrow \subseteq (\texttt{Env} \times \texttt{Store}) \times (\texttt{Env} \times \texttt{Store})$ be a transition relation and $S_0 \in \wp(\texttt{Env} \times \texttt{Store})$ be a set of methods' initial states. The trace semantics of m, $\mathbb{W}[\![m]\!] \in (\wp(\texttt{Env} \times \texttt{Store}) \rightarrow \wp(\texttt{Env} \times \texttt{Store}))$, is

$$\begin{aligned}\mathbb{W}[\![m]\!](S_0) = &\mathrm{lfp}_{\emptyset}^{\subseteq}\, \lambda\, X.\, S_0 \cup \Big\{\rho_0 \rightarrow \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \rho_{n+1} \;\mid\; \rho_{n+1} \in (\texttt{Env} \times \texttt{Store}) \wedge \\ &\rho_0 \rightarrow \rho_1 \rightarrow \cdots \rightarrow \rho_n \in X \wedge \rho_n \rightarrow \rho_{n+1}\Big\} \cup \Big\{\rho_0 \rightarrow \ldots \rho_n \mid \rho_0 \rightarrow \ldots \rho_n \in X\Big\}\end{aligned}$$

Let $\mathcal{P} = \langle c_{main}, \texttt{L} \rangle$ be an object-oriented program. Let $\rightarrow \subseteq (\texttt{Env} \times \texttt{Store}) \times (\texttt{Env} \times \texttt{Store})$ be a transition relation and $S_0 \in \wp(\texttt{Env} \times \texttt{Store})$ be a set of initial states such that $\forall \rho_0 \in S_0.\ \rho_0(currentMethod) = c_{main}$ and $\rho_0(pc) = pc_{main}$ where pc_{main} is the entry point of main method in c_{main}. The semantic of $\mathcal{P}$ is defined as

$$\begin{aligned}S[\![P]\!](S_0) = &\mathrm{lfp}_{\emptyset}^{\subseteq}\, \lambda\, X.\, S_0 \cup \Big\{\rho_0 \rightarrow \rho_1 \rightarrow \cdots \rightarrow \rho_n \rightarrow \rho_{n+1} \;\mid\; \rho_{n+1} \in (\texttt{Env} \times \texttt{Store}) \wedge \\ &\rho_0 \rightarrow \rho_1 \rightarrow \cdots \rightarrow \rho_n \in X \wedge \rho_n \rightarrow \rho_{n+1}\Big\}\end{aligned}$$

4.2.1.2 Constructor and Method Semantics

The semantics of constructor and methods are defined in terms of final state abstraction of their own trace semantics. The class constructor is invoked when an object of that class is created and initialized, given a store s, the constructor maps its fields to fresh locations and then assigns values into those locations. Constructor never returns any output.

Definition 4.1 (Constructor Semantics) *Given a store s. Let $\{a_{in}, a_{pc}\} \subseteq \texttt{Loc}$ be the free locations, $\texttt{Val}_{in} \subseteq \texttt{Val}$ be the semantic domain for input values. Let $v_{in} \in \texttt{Val}_{in}$ and pc_{exit} be the input value and the exit point of the constructor. The semantic of the class constructor* `init`*, $S[\![\texttt{init}]\!] \in (\texttt{Store} \times \texttt{Val} \rightarrow \wp(\texttt{Env} \times \texttt{Store}))$, is defined by*

$$\begin{aligned} S[\![\texttt{init}]\!](v_{in}, s) =& \Big\{(e_0, s_0) \ \mid \ (e_0 \triangleq V_{in} \rightarrow a_{in}, pc \rightarrow a_{pc}) \wedge (s_0 \triangleq s[a_{in} \rightarrow v_{in}, a_{pc} \rightarrow pc_{exit}])\Big\} \\ & \quad in\ \alpha_{\dashv}(\mathbb{W}[\![\texttt{init}]\!](\Big\{(e_0, s_0)\Big\})) \end{aligned}$$

where the final state abstraction $\alpha_{\dashv}$ is defined as

$$\alpha_{\dashv}(T) = \{\sigma \in \Sigma \mid \exists \tau \in T.\ \tau \textit{ is maximal, } \tau(len(\tau) - 1) = \sigma \ \textit{ and } T \textit{ is the sets of finite traces}\}$$

Definition 4.2 (Method Semantics) *Let $\texttt{Val}_{in} \subseteq \texttt{Val}$ and $\texttt{Val}_{out} \subseteq \texttt{Val}$ be the semantic domains for the input values and the output values respectively. Let $v_{in} \in \texttt{Val}_{in}$ be the input values, a_{in} and a_{pc} be the fresh memory locations, and pc_{exit} be the exit point of the method m. The semantic of a method m, $S[\![m]\!] \in (\texttt{Env} \times \texttt{Store} \times \texttt{Val}_{in} \rightarrow \wp(\texttt{Store} \times \texttt{Env} \times \texttt{Val}_{out})$, is defined as*

$$\begin{aligned} S[\![m]\!](e, s, v_{in}) =& \Big\{(e', s', v_{out}) \ \mid \ (e' \triangleq e[V_{in} \rightarrow a_{in}, pc \rightarrow a_{pc}]) \wedge \\ & \qquad (s' \triangleq s[a_{in} \rightarrow v_{in}, a_{pc} \rightarrow pc_{exit}]) \wedge v_{out} \in \texttt{Val}_{out}\Big\} \\ & in\ let\ S_f = \alpha_{\dashv}(\mathbb{W}[\![\texttt{init}]\!](\Big\{(e', s')\Big\})) \\ & in\ \Big\{\langle \rho_f(v_{out}, e_f, s_f)\rangle \mid \rho_f = \langle e_f, s_f)\rangle \in S_f\Big\} \end{aligned}$$

Example 19 *Consider the example of Figure 4.1. The class constructor* `Sample()` *creates a new environment consists of field a. The semantics of constructor* `Sample()` *and the semantics of the methods* `parity()` *and* `incr()` *are defined below:*

$$S[\![\texttt{Sample()}]\!](s, i) = \Big\{(e_0, s_0) \ \mid \ (e_0 \triangleq a \rightarrow a_{in}, pc \rightarrow a_{pc}) \wedge (s_0 \triangleq s[a_{in} \rightarrow i, a_{pc} \rightarrow 5])\Big\}$$

$$S[\![\texttt{parity()}]\!](e, s, \varnothing) = \Big\{(e, s', v_{out}) \ \mid \ (s' \triangleq s[e(pc) \rightarrow 10]) \wedge (v_{out} = if(s(e(a))\%2)\ ?1 : 0)\Big\}$$

```
1.    class Sample {
2.       int a;

3.       Sample(int i) {
4.          a = i;
5.       }
6.       int parity() {
7.          if(a % 2 == 0)
8.               return 1;
9.          else return 0;
10.      }
11.      int * incr( int j ) {
12.         a = a + j;
13.         return &a;
14.      }
15.  }
```

Figure 4.1: An example class

$$S[\![\texttt{incr()}]\!](e,s,j) = \Big\{(e,s',v_{out}) \mid (s' \triangleq s[e(a) \to s(e(a)) + j, e(pc) \to 14]) \wedge v_{out} = e(a)\Big\}$$

Observe that `parity()` *takes no input and returns an integer value as output, whereas* `incr()` *takes an integer value as input and returns an address as output.*

4.2.1.3 Object and Class Semantics

Object semantics is defined in terms of interaction history between the program-context and the object. A direct interaction takes place when the program-context calls any member-method of the object, whereas an indirect interaction occurs when the program-context updates any address escaped from the object's scope. However, both direct or indirect interaction can cause a change in an interaction state (see definition 4.3).

Definition 4.3 (Interaction States) *The set of interaction states is defined by*

$$\Sigma = \texttt{Env} \times \texttt{Store} \times \texttt{Val}_{out} \times \wp(\texttt{Loc})$$

where `Env`, `Store`, $\texttt{Val}_{out}$, *and* `Loc` *are the set of application environments, the set of stores, the set of output values, and the set of addresses respectively.*

Definition 4.4 (Initial Interaction States) *Let* $v_{in} \in \texttt{Val}_{in}$ *be an input to the class constructor* `init` *when creating an object. Let* $s \in \texttt{Store}$ *be a store. Then the set of initial interaction* *states is defined by*

$$\mathcal{I}_0 = \Big\{ \langle e_0, s_0, \phi, \emptyset \rangle \mid v_{in} \in \mathtt{Val}_{in},\ s \in \mathtt{Store},\ S[\![\mathtt{init}]\!](v_{in}, s) \ni \langle e_0, s_0 \rangle \Big\}$$

Observe that ϕ denotes no output produced by the constructor and $\emptyset$ represents the empty context with no escaped address.

Example 20 (Initial Interaction States) *Consider the example of Figure 4.1. The input to the constructor is i. Given a store s, the initial interaction states are*

$$\begin{aligned}\mathcal{I}_0 &= \Big\{ \langle e_0, s_0, \phi, \emptyset \rangle \mid S[\![\mathtt{Sample()}]\!](i, s) \ni (e_0, s_0) \Big\} \\ &= \Big\{ \langle e_0, s_0, \phi, \emptyset \rangle \mid (e_0 \triangleq a \to a_{in}, pc \to a_{pc}) \wedge (s_0 \triangleq s[a_{in} \to i, a_{pc} \to 5]) \Big\}\end{aligned}$$

Observe that the third element in an initial state is ϕ because constructor does not return any value as output. Similarly the fourth element is $\emptyset$ because no address is escaped from the object's scope after execution of `sample()`.

4.2.1.4 Transition Function.

Let $\mathtt{Lab} = (\mathtt{M} \times \mathtt{Val}_{in}) \cup \{\mathtt{upd}\}$ be a set of labels, where $\mathtt{M}$ is the set of class-methods, $\mathtt{Val}_{in}$ is the set of input values and `upd` denotes an indirect update operation by the context.

The transition function $\mathscr{T} : \Sigma \to \wp(\Sigma \times \mathtt{Lab})$ specifies which successor interaction states $\sigma' = \langle e', s', v', \mathtt{Esc}' \rangle \in \Sigma$ can follow

1. when an object's methods $m \in \mathtt{M}$ with input $v_{in} \in \mathtt{Val}_{in}$ is directly invoked on an interaction state $\sigma = \langle e, s, v, \mathtt{Esc} \rangle$ (**direct interaction**), or

2. the context indirectly updates an address escaped from an object's scope (**indirect interaction**).

Definition 4.5 (Direct Interaction $\mathscr{T}_{dir}$) *Transition on Direct Interaction is defined below:*

$$\begin{aligned}\mathscr{T}_{dir}(\langle e, s, v, \mathit{Esc} \rangle) = \Big\{ \big(\langle e', s', v', \mathit{Esc}' \rangle, (m, v_{in}) \big) \mid S[\![m]\!](\langle e, s, v_{in} \rangle) \ni \langle e', s', v' \rangle \\ \wedge\ \mathit{Esc}' = \mathit{Esc} \cup \mathit{reach}(v', s') \Big\}\end{aligned}$$

where

$$\mathtt{reach}(v,s) = \begin{cases} \textit{if } v \in \mathtt{Loc} \\ \qquad \{v\} \cup \{\mathtt{reach}(e'(f),s) \mid \exists B.\ B = \{\mathtt{init}, \mathtt{F}, \mathtt{M}\},\ f \in \mathtt{F}, \\ \qquad s(v) \textit{ is an instance of } B,\ s(s(v)) = e' \\ \\ \textit{else } \emptyset \end{cases}$$

Example 21 (Direct interaction $\mathscr{T}_{dir}$) *Consider the example of Figure 4.1. The context can invoke any one of the two methods of* `Sample` *class. Therefore given an interaction state $\sigma = \langle e,s,v,\mathtt{Esc}\rangle$, the set of successor interaction states are*

$$\mathscr{T}_{dir}(\langle e,s,v,\mathtt{Esc}\rangle) = \Big\{\Big(\langle e,s',v',\mathtt{Esc}\rangle, (\mathtt{parity()},\phi)\Big) \mid S[\![\mathtt{parity()}]\!](\langle e,s,\phi\rangle) \ni \langle e,s',v'\rangle\Big\}$$
$$\bigcup \Big\{\Big(\langle e,s',v',\mathtt{Esc'}\rangle, (\mathtt{incr()},j)\Big) \mid S[\![\mathtt{incr()}]\!](\langle e,s,j\rangle) \ni \langle e,s',v'\rangle$$
$$\wedge\ \mathtt{Esc'} = \mathtt{Esc} \cup \{v'\}\Big\}$$

Definition 4.6 (Indirect Interaction $\mathscr{T}_{ind}$) *Transition on Indirect Interaction is defined below:*

$$\mathscr{T}_{ind}(\langle e,s,v,\mathtt{Esc}\rangle) = \Big\{\Big(\langle e,s',v,\mathtt{Esc}\rangle, \mathtt{upd}\Big) \mid \exists a \in \mathtt{Esc}.\ \mathtt{Update}(a,s) \ni s'\Big\}$$

where $\mathtt{Update}(a,s) = \{s' \mid \exists v \in \mathtt{Val}.\ s' = s[a \leftarrow v]\}$

Definition 4.7 (Transition function $\mathscr{T}$) *Let $\sigma \in \Sigma$ be an interaction state. The transition function $\mathscr{T} : \Sigma \to \wp(\Sigma \times \mathtt{Lab})$ is defined as $\mathscr{T} = \mathscr{T}_{dir} \cup \mathscr{T}_{ind}$, where $\mathscr{T}_{dir}$ and $\mathscr{T}_{ind}$ represent direct and indirect transitions respectively.*

Let us denote a transition between interaction states σ_1 and σ_2 by $\sigma_1 \xrightarrow{\ell} \sigma_2$ where $\ell \in \mathtt{Lab}$.

4.2.1.5 Objects Fix-point Semantics

Given a store $s \in \mathtt{Store}$, the set of initial interaction states is defined as

$$\mathcal{I}_0 = \Big\{\langle e_0, s_0, \phi, \emptyset\rangle \mid S[\![\mathtt{init}]\!](v_{in}, s) \ni \langle e_0, s_0\rangle, v_{in} \in \mathtt{Val}_{in}\Big\}$$

The fix-point trace semantics of `obj`, according to [31], is defined as

$$\mathscr{T}[\![\texttt{obj}]\!](\mathcal{I}_0) = \text{lfp}^{\subseteq}_{\emptyset}\mathcal{F}(\mathcal{I}_0) = \bigcup_{i\leq\omega}\mathcal{F}^i(\mathcal{I}_0)$$

$$\text{where } \mathcal{F}(\mathcal{I}) = \quad \lambda\mathcal{T}.\,\mathcal{I}\cup\Big\{\sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \xrightarrow{\ell_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{\ell_0} \dots \xrightarrow{\ell_{n-1}} \sigma_n \in \mathcal{T}\wedge (\sigma_{n+1},\ell_n)\in\mathscr{T}(\sigma_n)\Big\}$$

4.2.1.6 Class Semantics

Class is nothing but a description of the set of objects. The semantics of a class `c` is defined as

$$S[\![\texttt{c}]\!] = \cup\Big\{\mathscr{T}[\![\texttt{obj}]\!](\mathcal{I}_0) \mid \text{"obj" is an instance of a class } \texttt{c} \text{ and } \mathcal{I}_0 \text{ is the set of initial interaction states}\Big\}$$

Observe that the semantic definitions of objects and classes aim at verifying invariance properties of classes.

4.2.2 Concrete Semantics of `Session` Methods

The semantics of conventional constructors, methods, objects, classes in HQL are defined in the same way as in the case of OOP. The `Session` methods require an 'ad-hoc' treatment. We define its concrete semantics by specifying how the methods are executed on (e, s, ρ_d) where $e \in \texttt{Env}$ is an environment, $s \in \texttt{Store}$ is a store and $\rho_d \in \mathfrak{E}_d$ is a database environment (defined in chapter 3), resulting into new state $(e', s', \rho_{d'})$. The semantic definitions are expressed in terms of the semantics of database statements `SELECT, INSERT, UPDATE, DELETE` defined in chapter 3.

We use the following functions in the subsequent part: *map(v)* maps v to the underlying database object; *var(exp)* returns the variables appearing in *exp*; *attr(t)* returns the attributes associated with table *t*; *dom(f)* returns the domain of f.

The semantics function is defined as:

$S[\![(\texttt{C}, \phi, \texttt{op})]\!](e, s, \rho_d)$

$$= \begin{cases} S[\![(\mathtt{C},\phi,\mathtt{op})]\!](e,s,\rho_{t'}) \text{ if } \exists t_1,\ldots,t_n \in dom(\rho_d):\ \mathtt{C}=\{c_1,\ldots,c_n\} \\ \qquad\qquad\qquad \wedge(\forall i\in[1\ldots n].\ t_i = map(c_i)) \wedge t' = t_1\times t_2\times\cdots\times t_n. \\ \\ \emptyset \quad \textit{otherwise.} \end{cases}$$

4.2.2.1 Semantics of `Session` Method `SAVE()`.

Let `obj` be an instance of a class c. Consider the `Session` method $\langle\{c\},\phi,\mathtt{SAVE(obj)}\rangle$. The semantics is defined as follows:

$$S[\![\langle\{c\},\phi,\mathtt{SAVE(obj)}\rangle]\!]$$

$$\begin{aligned} &= S[\![\langle\{c\}, false, \mathtt{SAVE(obj)}\rangle]\!] \\ &= \lambda(e,s,\rho_t).\ \text{let } c = \langle\mathtt{init},\mathtt{F},\mathtt{M}\rangle \text{ such that } map(\mathtt{F}) = attr(t) = \vec{a}, \text{ and let} \\ &\quad s(e(\mathtt{obj})) = e' \text{ such that } s(e'(\mathtt{F})) = \vec{val}, \text{ in} \\ &\quad \Big\{\langle e,s,\rho_{t'}\rangle \ \mid\ \rho_{t'} \in S[\![\Big\langle \mathtt{INSERT}(\vec{a},\vec{val}), false\Big\rangle]\!](\rho_t)\Big\}. \end{aligned}$$

Note that our semantic definition presents a state transition from ρ_t to ρ'_t on insertion of new object into the database table t. Observe that this definition does not capture the existence of any underlying database constraints.

4.2.2.2 Semantics of `Session` Method `UPD()`.

Consider the `Session` method $\langle\{c\},\phi,\mathtt{UPD}(\vec{v},\vec{exp})\rangle$.

Let $\mathtt{PE}[\![X]\!]$ (which stands for partial evaluation) be an auxiliary function which converts variables in X into the corresponding database objects. This is defined by

$$\mathtt{PE}[\![X]\!](e,s,\mathtt{F}) = X'$$

where $X' = X[x_i/v_i]$ for all $v_i \in var(X)$ and $x_i = \begin{cases} map(v_i) & \text{if } v_i \in \mathtt{F} \\ \\ E[\![v_i]\!](e,s) & \text{otherwise} \end{cases}$

The semantics is defined below[1]:

$S[\![\langle\{c\}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp})\rangle]\!]$

$=\lambda(e, s, \rho_t)$. let $c = \langle \mathtt{init}, \mathtt{F}, \mathtt{M}\rangle$ such that $map(\mathtt{F}) = attr(t)$ and $map(\vec{v}) = \vec{a} \subseteq attr(t)$ where $\vec{v} \subseteq \mathtt{F}$, and let $\phi_d = \mathtt{PE}[\![\phi]\!](e, s, \mathtt{F})$ and $\vec{exp}_d = \mathtt{PE}[\![\vec{exp}]\!](e, s, \mathtt{F})$ in $\left\{\langle e, s, \rho_{t'}\rangle \mid \rho_{t'} \in S[\![\left\langle \mathtt{UPDATE}(\vec{a}, \vec{exp}_d), \phi_d\right\rangle]\!](\rho_t)\right\}$.

4.2.2.3 Semantics of `Session` Method `DEL()`.

Consider the `Session` method $\langle\{c\}, \phi, \mathtt{DEL}()\rangle$. The semantics is defined below:

$S[\![\langle\{c\}, \phi, \mathtt{DEL}()\rangle]\!]$

$=\lambda(e, s, \rho_t)$. let $c = \langle \mathtt{init}, \mathtt{F}, \mathtt{M}\rangle$ such that $map(\mathtt{F}) = attr(t) = \vec{a}$ and let $\phi_d = \mathtt{PE}[\![\phi]\!](e, s, \mathtt{F})$ in $\left\{\langle e, s, \rho_{t'}\rangle \mid \rho_{t'} \in S[\![\left\langle \mathtt{DELETE}(\vec{a}), \phi_d\right\rangle]\!](\rho_t)\right\}$

4.2.2.4 Semantics of `Session` Method `SEL()`.

The semantics of `Session` method $\langle \mathtt{C}, \phi, \mathtt{SEL}\big(v_a, f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})\big)\rangle$ is defined as:

$S[\![\langle \mathtt{C}, \phi, \mathtt{SEL}\big(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})\big)\rangle]\!]$

$=\lambda(e, s, \rho_t)$. let $\mathtt{C} = \{\langle \mathtt{init}_i, \mathtt{F}_i, \mathtt{M}_i\rangle \mid i = 1, \ldots, n\}$, and $\mathtt{F} = \bigcup_{i=1,\ldots,n} \mathtt{F}_i$, and $\langle \vec{exp}'_d,\ \vec{x}_d,\ \phi'_d,\ \vec{exp}_d,\ \phi_d\rangle = \mathtt{PE}[\![\langle \vec{exp}',\ \vec{x},\ \phi',\ \vec{exp},\ \phi\rangle]\!](e, s, \mathtt{F})$, and let $\rho_{t'} = S[\![\langle \mathtt{SELECT}(v_a, f(\vec{exp}'_d),\ r(\vec{h}(\vec{x}_d)),\ \phi'_d,\ g(\vec{exp}_d)), \phi_d\rangle]\!](\rho_t)$ and $(e', s') = \bigsqcup_{\forall l_i \in t'} S[\![\mathtt{Object()}]\!](s, \mathtt{val}(l_i))$ in $\left\{\langle e', s', \rho_t\rangle\right\}$.

Observe that $\mathtt{val}(l_i)$ converts each tuple $l_i \in t'$ into input values, and $S[\![\mathtt{Object()}]\!](s, \mathtt{val}(l_i))$ invokes the object constructor `Object()` which creates an object by initializing the fields

[1]Observe that, for the sake of simplicity, we do not consider here the method REFRESH() which synchronize the in-memory objects state with that of the underlying database.

with $\texttt{val}(l_i)$. This is done for all tuples $l_i \in t'$, resulting into new (e', s'). As we already mentioned in the case of concrete semantics of SQL, in this semantics definition also the state of the tables other than ρ_t remained unchanged and this is clear from the context.

4.2.2.5 Fix-point Semantics of `Session` Objects

Let `Env` and `Store` be the set of HQL environments and stores respectively. Let $\mathfrak{E}_d$ be the set of database environments. The set of interaction states of `Session` objects is defined below:

Definition 4.8 (Interaction States of `Session` Objects) *The set of interaction states of `Session` objects is defined by*

$$\Sigma = \mathit{Env} \times \mathit{Store} \times \mathfrak{E}_d \times \wp(\mathit{Loc})$$

Therefore, an interaction state of a `Session` object is a triplet $\langle e, s, \rho_d, \mathit{Esc}\rangle$, where $e \in \mathit{Env}$, $s \in \mathit{Store}$, $\rho_d \in \mathfrak{E}_d$ and $\mathit{Esc} \in \wp(\mathit{Loc})$.

Because of nondeterministic executions, the transition function is defined as $\mathscr{T} : \mathtt{M}_{ses} \times \Sigma \to \wp(\Sigma)$ specifying which successor interaction states $\sigma' = \langle e', s', \rho_{d'}, \mathtt{Esc}'\rangle \in \Sigma$ can follow when a `Session` method $m_{ses} = \langle \mathtt{C}, \phi, \mathtt{op}\rangle \in \mathtt{M}_{ses}$ is invoked on an interaction state $\sigma = \langle e, s, \rho_d, \mathtt{Esc}\rangle$. That is,

$$\mathscr{T}[\![m_{ses}]\!](\langle e, s, \rho_d, \mathtt{Esc}\rangle) = \Big\{\langle e', s', \rho_{d'}, \mathtt{Esc}'\rangle \mid S[\![m_{ses}]\!](\langle e, s, \rho_d\rangle) \ni \langle e', s', \rho_{d'}\rangle \wedge m_{ses} \in \mathtt{M}_{ses}\Big\}$$

We denote a transition by $\sigma \xrightarrow{m_{ses}} \sigma'$ when application of a `Session` method m_{ses} on interaction state σ results into a new state σ'.

Let $\mathcal{I}_0$ be the set of initial interaction states. The semantics of `Session` object $\mathtt{obj}_{ses}$ is defined as

$$\mathscr{T}[\![\mathtt{obj}_{ses}]\!](\mathcal{I}_0) = \mathrm{lfp}^{\subseteq}_{\emptyset}\mathcal{F}(\mathcal{I}_0) = \bigcup_{i \leq \omega} \mathcal{F}^i(\mathcal{I}_0)$$

$$\text{where } \mathcal{F}(\mathcal{I}) = \lambda\mathcal{T}.\, \mathcal{I} \cup \Big\{\sigma_0 \xrightarrow{m_0} \dots \xrightarrow{m_{n-1}} \sigma_n \xrightarrow{m_n} \sigma_{n+1} \mid \sigma_0 \xrightarrow{m_0} \dots \xrightarrow{m_{n-1}} \sigma_n \in \mathcal{T} \\ \wedge \sigma_n \xrightarrow{m_n} \sigma_{n+1} \in \mathscr{T}\Big\}$$

4.2.2.6 `Session` Class Semantics

The semantics of the `Session` class c_{ses} is defined as

$$S[\![c_{ses}]\!] = \cup \Big\{ \mathscr{T}[\![\texttt{obj}_{ses}]\!](\mathcal{I}_0) \ \mid \ \text{"obj}_{ses}\text{" is an instance of a } \texttt{Session} \text{ class c}_{ses} \text{ and } \mathcal{I}_0 \text{ is the set of initial interaction states} \Big\}$$

4.2.3 Illustration using an Example

Consider the HQL program `HProg` depicted in Figure 4.2 and the initial database table t_1 depicted in Table 4.3(a). The fields *id, age, dno, sal* of POJO class `emp` correspond to the attributes *tid, tage, tdno, tsal* of table t_1 respectively. The `Session` methods of the class `Service` allows manipulating the employee's information such as inserting a new record in the database table, updating the values of the attributes *tage* and *tsal*, removing existing record from the database table and to make simple queries to that database [2].

Now we will describe our semantic formalism on the Session methods at various program points of `HProg`.

The formal syntax of `SAVE`() at program point 12 is defined as $\Big\langle \{\texttt{emp}\}, false, \texttt{SAVE(obj)} \Big\rangle$. Given the table environment ρ_{t_1} in Figure 4.3(a), the semantics is defined as below:

$$[\![\Big\langle \{\texttt{emp}\}, false, \texttt{SAVE(obj)} \Big\rangle]\!]$$

$$\begin{aligned} =&\lambda(e, s, \rho_{t_1}).\ \text{let } \texttt{emp} = \langle \texttt{emp()}, \texttt{F}, \texttt{M} \rangle \text{ such that } \texttt{F} = \langle id, age, dno, sal \rangle \text{ and} \\ &\quad map(\texttt{F}) = attr(t) = \langle tid, tage, tdno, tsal \rangle \text{ and let} \\ &\quad s(e(obj)) = e' \text{ such that } s(e'(\langle id, age, dno, sal \rangle)) = \langle 4, 32, 1, 1000 \rangle,\ \text{in} \\ &\Big\{ \langle e, s, \rho_{t_2} \rangle \ \mid \ \rho_{t_2} \in S[\![\Big\langle \texttt{INSERT}(\langle tid, tage, tdno, tsal \rangle, \langle 4, 32, 1, 1000 \rangle), false \Big\rangle]\!](\rho_{t_1}) \Big\}. \end{aligned}$$

where ρ_{t_2} is depicted in Table 4.3(b).

The formal syntax of the `UPD`() to the statements 13-15 is $\Big\langle \{c\}, \phi, \texttt{UPD}(\vec{v}, \vec{exp}) \Big\rangle$, where

[2]Observe at program points 13, 14-16, 17-18 that the basic differences between HQL and SQL.

```
class emp {
        private int id;
        private int age;
        private int dno;
        private int sal;
        emp(){ }
        emp(int x){ this.id=x; }
        public int getId() { return id;}
        public void setId(int id) { this.id = id;}
        public int getage() {return age;}
        public void setage(int age) { this.age = age;}
        public int getdno() { return dno;}
        public void setdno(int dno) { this.dno = dno;}
        public int getsalary() { return sal;}
        public void setsalary(int salary) { this.sal = salary;}    }
```

(a) Class emp

```
public class Service{
    public static void main(String[] args) {
        Configuration cfg=new Configuration();
        cfg.configure("hibernate.cfg.xml");
        SessionFactory sf=cfg.buildSessionFactory();
        Session ses=sf.openSession();
        Transaction tr=ses.beginTransaction();
% Creating emp object and stores into database %
        emp obj=new emp( 4 );
        obj.setage(32);
        obj.setdno(1);
        obj.setsalary(1000);
        ses.save( obj );
% Updating persistent emp objects %
        Query q1 = ses.createQuery("UPDATE emp SET emp.age= emp.age+1,
                        emp.sal= emp.sal + :inc×2 WHERE emp.sal > 1600");
        q1.setParameter("inc",100);
        int r1 = q1.executeUpdate();
% Selecting from persistent emp objects %
        Query q2 = session.createQuery("SELECT emp.dno, MAX(emp.sal),
                        AVG(DISTINCT emp.age) FROM emp
                        WHERE emp.sal ≥ 1000 GROUP BY emp.dno HAVING
                        MAX(emp.sal) < 4000 ORDER BY emp.dno");
        List r2 = q2.list();
% Deleting persistent emp objects %
        Query q3 = ses.createQuery("DELETE FROM emp WHERE
                        emp.age > 50");
        int r3 = q3.executeUpdate();
        tr.commit();
        ses.close();
    } }
```

(b) Class Service

Figure 4.2: A HQL Program HProg

tid	*tage*	*tdno*	*tsal*
1	35	3	1600
2	19	2	900
3	50	3	2550

(a) Initial Table t_1

tid	*tage*	*tdno*	*tsal*
1	35	3	1600
2	19	2	900
3	50	3	2550
4	32	1	1000

(b) Table t_2: After executing statement 12

tid	*tage*	*tdno*	*tsal*
1	35	3	1600
2	19	2	900
3	51	3	2750
4	32	1	1000

(c) Table t_3: After executing statements 13-15

tid	*tage*	*tdno*	*tsal*
1	35	3	1600
2	19	2	900
3	51	3	2750
4	32	1	1000

(d) Table t_4: After executing statement 16-17 (no change in database)

tid	*tage*	*tdno*	*tsal*
1	35	3	1600
2	19	2	900
4	32	1	1000

(e) Table t_5: After executing statements 18-19

tdno	MAX(*tsal*)	AVG(*tage*)
1	1000	32
3	2750	43

(f) Table t_{sel}: Result of Selection at 16-17

Figure 4.3: Snapshot of database states after executing various Session methods

- {c}= {emp},
- ϕ= "emp.$sal > 1600$",
- $\texttt{UPD}(\vec{v}, \vec{exp})$=$\texttt{UPD}\Big(\langle age, sal\rangle,\ \langle age+1, sal+:inc\times 2\rangle\Big)$

Given the table environment ρ_{t_2} in Figure 4.3(b), the semantics is defined as:

$$S[\![\Big\langle\{\texttt{emp}\},\ (\texttt{emp}.sal > 1600),\ \texttt{UPD}\Big(\langle age, sal\rangle,\ \langle age+1, sal+:inc\times 2\rangle\Big)\Big\rangle]\!]$$

$=\lambda(e, s, \rho_{t_2})$. let $\texttt{emp} = \langle \texttt{emp()}, \texttt{F}, \texttt{M}\rangle$ such that $\texttt{F} = \langle id, age, dno, sal\rangle$ and

$map(\texttt{F}) = attr(t) = \langle tid, tage, tdno, tsal\rangle$ and

$map(\vec{v}) = map(\langle age, sal\rangle) = \langle tage, tsal\rangle \subseteq attr(t)$, and let

$(tsal > 1600) = \texttt{PE}[\![(\texttt{emp}.sal > 1600)]\!](e, s, \texttt{F})$ and

$\langle tage+1, tsal+100\times 2\rangle = \texttt{PE}[\![\langle age+1, sal+:inc\times 2\rangle]\!](e, s, \texttt{F})$ in

$$\Big\{\langle e, s, \rho_{t_3}\rangle \mid \rho_{t_3} \in S[\![\Big\langle \texttt{UPDATE}(\langle tage, tsal\rangle, \langle tage+1, tsal+100\times 2\rangle),\ (tsal > 1600)\Big\rangle]\!](\rho_{t_2})\Big\}.$$

where ρ_{t_3} is depicted in Table 4.3(c).

The formal syntax of the DEL() at program point 18-19 is defined as $\Big\langle \{\texttt{emp}\}, \phi, \texttt{DEL}() \Big\rangle$, where ϕ = "emp.$age > 50$".

Given the table environment ρ_{t_4} in Figure 4.3(d), the semantics is defined as below:

$$S[\![\Big\langle \{\texttt{emp}\}, (\texttt{emp}.age > 50), \texttt{DEL}() \Big\rangle]\!]$$

$$\begin{aligned}
=&\lambda(e, s, \rho_{t_4}).\ \text{let } \texttt{emp} = \langle \texttt{emp()}, \texttt{F}, \texttt{M} \rangle \text{ such that } \texttt{F} = \langle id, age, dno, sal \rangle \text{ and} \\
&\quad map(\texttt{F}) = attr(t) = \langle tid, tage, tdno, tsal \rangle \text{ and let} \\
&\quad (tage > 50) = \texttt{PE}[\![(\texttt{emp}.age > 50)]\!](e, s, \texttt{F}) \text{ in} \\
&\Big\{\langle e, s, \rho_{t_5} \rangle \mid \rho_{t_5} \in S[\![\Big\langle \texttt{DELETE}(\langle tid, tage, tdno, tsal \rangle, (tage > 50)) \Big\rangle]\!](\rho_{t_4})\Big\}.
\end{aligned}$$

where ρ_{t_5} is depicted in Table 4.3(e).

The formal syntax of the SEL() to the statements 16-17 is:

$$\Big\langle \texttt{C}, \phi, \texttt{SEL}(v_a, f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})) \Big\rangle \quad \text{where}$$

- C={c}= {emp},
- ϕ= "emp.$sal \geq 1000$",
- $\vec{exp} = \langle$emp.dno$\rangle$,
- $g(\vec{exp})$ = GROUP BY($\langle$emp.dno$\rangle$),
- ϕ'="MAX(emp.sal)<4000",
- $\vec{x}$= $\langle$emp.dno, emp.sal, emp.$age\rangle$,
- $\vec{h}$= $\langle$DISTINCT, MAX$\circ$ALL, AVG$\circ$DISTINCT$\rangle$,
- $r(\vec{h}(\vec{x}))$= DISTINCT$\Big($DISTINCT(emp.dno), MAX$\circ$ALL(emp.sal), AVG$\circ$DISTINCT(emp.age)$\Big)$,
- $\vec{exp}'$=$\langle$emp.dno$\rangle$,

- $f(\vec{exp'})$=ORDER BY ASC($\langle$emp.dno$\rangle$)
- v_a = ResultSet type with fields $\vec{w} =< w_1, w_2, \ldots w_n >$

Given the table environment ρ_{t_3} in Figure 4.3(c), the semantics is defined as:

$$S[\![\Big\langle \{\texttt{emp}\},\ \phi,\ \texttt{SEL}\Big(v_a, f(\vec{exp'}),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp})\Big)\Big\rangle]\!]$$

$=\lambda(e, s, \rho_{t_2})$. let $\texttt{emp} = \langle \texttt{emp()}, \texttt{F}, \texttt{M}\rangle$ such that

$\texttt{F} = \langle id, age, dno, sal\rangle$ and let

$\langle \vec{exp'_d},\ \vec{x_d},\ \phi'_d,\ \vec{exp_d},\ \phi_d\rangle = \texttt{PE}[\![\langle \vec{exp'},\ \vec{x},\ \phi',\ \vec{exp},\ \phi\rangle]\!](e, s, \texttt{F})$ *where*

ϕ_d = "$tsal \geq 1000$"

$\vec{exp_d} = \langle tdno\rangle$

ϕ'_d = "$\texttt{MAX}(tsal) < 4000$"

$\vec{x_d} = \langle tdno, tsal, tage\rangle$

$\vec{exp'_d} = \langle tdno\rangle$

and $S[\![\Big\langle \texttt{SELECT}(v_a, f(\vec{exp'_d}),\ r(\vec{h}(\vec{x_d})),\ \phi'_d,\ g(\vec{exp_d})), \phi_d\Big\rangle]\!](\rho_{t_3}) = \rho_{t_{sel}}$ (See Figure 4.3(f))

in $\Big\{\langle e', s', \rho_t\rangle \mid (e', s') = \bigsqcup_{l_i \in t_{sel}} S[\![\texttt{Object()}]\!](s, val(l_i))\Big\}$.

4.3 Abstract Semantics of HQL

As it is usual, in the Abstract Interpretation framework, once the concrete semantics is formulated, it can be lifted to an abstract semantics by simply making correspondence of concrete objects (variables values, object instances, stores, states, traces, etc.) into abstract ones representing partial information on them. Here we extend the semantics abstraction of OOP to the case of HQL in the same line as proposed in [86]. This will allow us to capture reachable database state semantics, and hence database invariant in an abstract domain of interest.

The semantics of a method can be abstracted to set of interaction states by considering their reachability on all possible execution of the method.

Definition 4.9 (Collecting Session Method Semantics) *Let m be a method and $\mathcal{X} \in \wp(\Sigma)$ be a set of interaction states. Then the collecting* `Session` *method semantics of m_{ses}, $\mathbb{M}[\![m_{ses}]\!] \in$*

$(\wp(\Sigma) \rightarrow \wp(\Sigma))$ *is defined as*

$$\mathbb{M}[\![m_{ses}]\!](\mathcal{X}) = \Big\{\langle e', s', \rho_{d'}, \mathtt{Esc}\rangle \mid \langle e, s, \rho_d, \mathtt{Esc}\rangle \in \mathcal{X}, v_{in} \in \mathtt{Val}_{in}, S[\![m_{ses}]\!](\langle e, s, \rho_d\rangle) \ni \langle e', s', \rho_{d'}\rangle\Big\}$$

Reachable States Session Object Semantics. Let $v_{in} \in \mathtt{Val}_{in}$ and $s \in \mathtt{Store}$. Let $\mathtt{obj}_{ses}$ is an object instance of `Session` class $c_{ses} = \langle \mathtt{init}, \mathtt{F}, \mathtt{M}_{ses}\rangle$. The reachable states fixpoint semantics of $\mathtt{obj}_{ses}$, defined as a function $\mathscr{T}_r[\![\mathtt{obj}_{ses}]\!] \in (\mathtt{Val}_{in} \times \mathtt{Store} \rightarrow \wp(\Sigma))$, is as follows:

$$\mathscr{T}_r[\![\mathtt{obj}_{ses}]\!](v_{in}\ ,\ s) = \mathrm{lfp}_{\emptyset}^{\subseteq} \lambda\mathcal{K}. \Big(\mathcal{I}_0 \ \cup \bigcup_{m_{ses} \in \mathtt{M}_{ses}} \mathbb{M}[\![m_{ses}]\!](\mathcal{K})\Big)$$

Now, $\mathscr{T}_r[\![\mathtt{obj}_{ses}]\!]$ is a sound approximation of its concrete semantics. That is,

$$\alpha_\Sigma(\mathscr{T}[\![\mathtt{obj}_{ses}]\!](v_{in}\ ,\ s)) \subseteq \mathscr{T}_r[\![\mathtt{obj}_{ses}]\!](v_{in}\ ,\ s) \quad \text{where}$$

$$\alpha_\Sigma(T) = \{\sigma \in \Sigma \mid \exists \tau \in T.\ \exists i.\ \tau(i) = \sigma \text{ and } T \text{ is the sets of finite traces}\}$$

Reachable States Session Class Semantics. The reachable states semantics $S_r[\![c_{ses}]\!] \in \wp(\Sigma)$ of `Session` class c_{ses} is defined as:

$$S_r[\![c_{ses}]\!] = \bigcup\{\mathscr{T}_r[\![\mathtt{obj}_{ses}]\!](v_{in}\ ,\ s) \mid \mathtt{obj}_{ses} \text{ is an instance of } c_{ses},\ v_{in} \in \mathtt{Val}_{in}\ ,\ s \in \mathtt{Store}\}$$

Observe that $S_r[\![c_{ses}]\!]$ is a sound approximation of the `Session` class concrete semantics, i.e. $\alpha_\Sigma(S[\![c_{ses}]\!]) \subseteq S_r[\![c_{ses}]\!]$. It can be expressed in fixpoint form as

$$S_r[\![c_{ses}]\!] = \mathrm{lfp}_{\emptyset}^{\subseteq} \lambda\mathcal{K}. \Big(\mathcal{I}[\![\mathtt{init}]\!] \ \cup \bigcup_{m_{ses} \in \mathtt{M}_{ses}} \mathbb{M}[\![m_{ses}]\!](\mathcal{K})\Big)$$

where $\mathcal{I}[\![\mathtt{init}]\!] = \mathcal{I}_0 \in \wp(\Sigma)$ are the states reached after an invocation of the class constructor.

State-based Session Class Invariant. The equations that characterize a `Session` class invariant can be derived directly from the above defined fixpoint formulation of the `Session` class reachable states. That is,

$$\mathcal{K} = \mathcal{I}[\![\mathtt{init}]\!] \ \cup \bigcup_{m_{ses} \in \mathtt{M}_{ses}} \mathbb{M}[\![m_{ses}]\!](\mathcal{K})$$

In its turn, such an equation can be rewritten as the following system of recursive equations, where n is the number of method calls of `Session` class c_{ses}.

$$\begin{aligned}
&\mathcal{K} = \mathcal{K}_0 \cup \bigcup_{1 \leqslant i \leqslant n} \mathcal{K}_i\ ,\\
&\mathcal{K}_0 = \mathcal{I}[\![\texttt{init}]\!],\\
&\mathcal{K}_i = \mathbb{M}[\![m_{ses_i}]\!](\mathcal{K}_{i-1}), \qquad\qquad 1 \leqslant i \leqslant n
\end{aligned}$$

A solution of the above system of equations is a tuple of sets of states $\langle \mathcal{K},\ \mathcal{K}_0,\ \mathcal{K}_1 \ldots \mathcal{K}_n \rangle$ where

- $\mathcal{K}$ are the states reached after the execution of a class constructor and at the entry and exit point of any method in the `Session` class;
- $\mathcal{K}_0$ are the states reached after the execution of the `Session` class constructor;
- $\mathcal{K}_i$ are the states reached after the execution of the `Session` method m_{ses_i}.

Let $\langle \overline{\mathbb{D}},\ \sqsubseteq,\ \perp,\ \top,\ \sqcup,\ \sqcap \rangle$ be an abstract domain approximating sets of interaction states, establishing the following Galois connection:

$$\langle \wp(\Sigma),\ \subseteq,\ \emptyset,\ \Sigma,\ \cap,\ \cup \rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle \overline{\mathbb{D}},\ \sqsubseteq,\ \perp,\ \top,\ \sqcup,\ \sqcap \rangle \tag{4.1}$$

Let us consider the abstract counterpart of the constructor semantics so that the initial states are approximated by a function $\overline{\mathcal{I}}[\![\texttt{init}]\!] \in \overline{\mathbb{D}}$ such that

$$\mathcal{I}[\![\texttt{init}]\!] \subseteq \gamma(I[\![\texttt{init}]\!])$$

The collecting `Session` method semantics is approximated by an abstract semantic function $\overline{\mathbb{M}}[\![m_{ses_i}]\!] \in (\overline{\mathbb{D}} \rightarrow \overline{\mathbb{D}})$. In fact, $\overline{\mathbb{M}}[\![m_{ses_i}]\!]$ is a sound approximation of the collecting semantics of m_{ses_i}. That is,

$$\forall \mathcal{X} \in \wp(\Sigma).\ \ \mathbb{M}[\![m_{ses_i}]\!](\mathcal{X}) \subseteq \gamma(\overline{\mathbb{M}}[\![m_{ses_i}]\!](\alpha(\mathcal{X}))) \tag{4.2}$$

Given $\overline{\mathcal{I}}[\![\texttt{init}]\!] \in \overline{\mathbb{D}}$ and $\overline{\mathbb{M}}[\![m_{ses_i}]\!] \in (\overline{\mathbb{D}} \rightarrow \overline{\mathbb{D}})$, the tuple $\langle \overline{\mathcal{K}},\ \overline{\mathcal{K}_0},\ \overline{\mathcal{K}_1} \ldots \overline{\mathcal{K}_n} \rangle \in \overline{\mathbb{D}}^{n+2}$ is a solution of the following recursive equations system:

$$\begin{aligned}
\overline{\mathcal{K}} &= \overline{\mathcal{K}}_0 \sqcup \bigsqcup_{1 \leqslant i \leqslant n} \overline{\mathcal{K}}_i\,, \\
\overline{\mathcal{K}}_0 &= \overline{\mathcal{I}}[\![\texttt{init}]\!], \\
\overline{\mathcal{K}}_i &= \overline{\mathbb{M}}[\![m_{ses_i}]\!](\overline{\mathcal{K}}_{i-1}), \qquad\qquad 1 \leqslant i \leqslant n
\end{aligned}$$

Observe that $S_r[\![c_{ses}]\!] \subseteq \gamma(\overline{\mathcal{K}})$, $\mathcal{I}[\![\texttt{init}]\!] \subseteq \gamma(\overline{\mathcal{K}}_0)$ and $\mathbb{M}[\![m_{ses_i}]\!](S_r[\![c_{ses}]\!]) \subseteq \gamma(\overline{\mathcal{K}}_i)$.

Therefore, for the verification of database invariant property, one has to choose appropriate abstract domains such as the domains of Intervals, Octagons, Polyhedra, etc.

4.4 Conclusions

In this chapter, we define concrete and abstract semantics of HQL in terms of the semantics of `Session` methods, objects and `Session` class. We show that the state invariant property of `Session` class in a suitable abstract domain verifies the database consistency w.r.t. a given set of integrity constraints.

CHAPTER 5

Semantic-based Dependency Computation of Database Applications

Preface

Syntax-based dependency computation often fails to generate an optimal set of dependencies, which increases the susceptibility of false alarms in many software engineering activities. This demands the need of semantics-based dependency computation taking into account variables values rather than their syntactic structures. This chapter is dedicated for this purpose. In particular, considering the previously defined abstract semantics of database applications as the underlying basis, we instantiate semantics-based dependency computation in various relational and non-relational abstract domains tunable with respect to the precision and efficiency. We develop a prototype semDDA, a semantics-based Database Dependency Analyzer, and we present experimental evaluation results in various abstract domains 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.

5.1 Introduction

Syntax-based construction of Dependency Graph depends on the computations of (*i*) data-dependency among variables and statements based on the syntactic variables presence in the statements and (*ii*) control-dependency based on the syntactic structure of programs [100]. Interestingly, authors in [105], [63], [89] noticed that syntax-based approach often fail to compute optimal set of dependencies, particularly when syntactic presence of variables is not enough to represent relevancy. For instance, consider an expression "$e = x + 2 \times w \ mod \ 2$" where e is syntactically dependent on w but semantically there is no such dependency as the evaluation of the expression "$2 \times w \ mod \ 2$" is always zero. Therefore, computation of such false dependencies focusing variables value motivate researchers to refine syntax-based dependency graph into more refined semantics-based dependency graph.

As we already stated in chapter 1 that the impact of precision of dependency information may influences various applications like database code slicing, database leakage analysis, data provenance, materialization view creation, etc. [23, 53, 55]. Let us again highlight this using the following two database statements Q_1 and Q_2:

Q_1 : `UPDATE` *emp* `SET` $age := age + 1$ `WHERE` $age \geqslant 35$ `AND` $sal \leqslant 2000$

Q_2 : `SELECT MAX(`*sal*`), AVG(`*age*`) FROM` *emp* `WHERE` $age \leqslant 30$

Observe that Q_2 is syntactically DD-Dependent on Q_1 for '*age*' as it is a defined-variable in Q_1 and a used-variable in Q_2. However, if we focus on the values of '*age*' in the database, the part of *age*-values defined by Q_1 is not overlapping with the *age*-values subsequently used by Q_2. Therefore, there exist no semantics-based dependency between Q_1 and Q_2.

Although [121] defines DD-dependency in terms of defined- and used-values of databases, but their definition on PD-dependency rely on syntactic presence of variables and attributes in the statements. They refer Action-Condition rules to compute overlapping of database-parts, however this fails to capture semantic independencies when the application contains more than one defining database statements (in sequence) for an attribute x which is subsequently used by another statement. The main reason behind this is the flow-insensitivity of the Condition-Action rules, resulting into a set of false dependencies. Since then no significant contribution is found in this research direction.

In this chapter, we put forward the notion of semantics-based dependency computation of database applications as a way to refine DOPDG by removing false dependencies. To summarize, our contributions in this chapter are:

- Adapting the abstract semantics to define computable abstract semantics towards independency computation of database applications, even in an undecidable scenario when the input database instance is unknown.
- Design of an algorithm to compute semantics-based independencies among database statements based on the abstract semantics.
- A detailed analytical study on precision vs. efficiency when computing dependency in various well-suited non-relational and relational abstract domains, e.g. Interval, Octagon, Polyhedra, Powerset domain.
- Development of a prototype semDDA, **sem**antics-based **D**atabase **D**ependency **A**nalyzer, integrated with various abstract domains which enables users to perform precise dependency computation in various abstract domains of interest.
- Experimental evaluation on a set of open-source database-driven JSP web applications as part of the GotoCode project [1] using our semDDA tool[1]. Experiments demonstrate the results in different abstract domains with a detailed comparison on precision and efficiency. This clearly shows that our technique improves precision w.r.t. the proposal by Willmor et al. [121].

5.2 Related Works

Ferrante et al. [45] first introduced the notion of Program Dependency Graph (PDG) aiming program optimization. Since then, PDG is playing crucial roles in a wide range

[1]Available at: `https://github.com/angshumanjana/SemDDA.`

of software-engineering activities, e.g. program slicing [116], code-reuse [72], language-based information flow security analysis [55,59,79], code-understanding [102]. Over the time, various forms of dependency graphs for various programming languages are proposed in order to address several language-specific problems. In [124], 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 [24] proposed a PDG for parallel and distributed programs. In [49], 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. [64] introduced System Dependency Graph (SDG) in case of inter-procedural programs. Class Dependency Graph (ClDG) is introduced for Object Oriented Programming (OOP) languages in [83]. Willmor et.al. [121] introduced a variant of program dependency graph, known as Database-Oriented Program Dependency 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 [10]. 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 [89] 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. [2] applied predicate transformer (weakest precondition) to apply on dependency tree among a series of attribute-defining statements, whereas [53] formalized the semantics for dependency refinement in a simple setting following the Abstract Interpretation as an initial attempt.

The authors in [92] and [84] 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 $Q_1 \sqsubseteq Q_2 \iff \forall D\ Q_1(D) \subseteq Q_2(D)$ and $Q_1 \equiv Q_2 \iff Q_1 \sqsubseteq Q_2 \land Q_2 \sqsubseteq Q_1$, where $Q(D)$ represents the result of query Q on database D. The computational complexity of conjunctive query containment is NP-complete [92]. 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 [84,92] 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 [26] 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 [37] 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. [119] 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 [80] 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 [90], 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

```
0. public class saleOffer {
1.   public static void main(String[] args) throws SQLException {
2.    float x = 0.1;
3.    float y = 0.05;
4.    try { Statement con = DriverManager.getConnection("jdbc mysql: . . . ", "scott", "tiger").createStatement();
          /∗ 5% discount offer based on the purchase amount. ∗/
5.        con
       .executeQuery("UPDATE Sales SET purchase_amt = purchase_amt − y ∗ purchase_amt WHERE purchase_amt  BETWEEN 1000 AND 3000 " );
          /∗ 10% discount offer based on the purchase amount. ∗/
6.           con.executeQuery("UPDATE Sales SET purchase_amt = purchase_amt − x ∗ purchase_amt WHERE purchase_amt > 3000 ");
          /∗ Free delivery offer. ∗/
7.            con.executeQuery("UPDATE Sales SET purchase_amt = purchase_amt − delivery_charge ");
             . . .
             . . .
             . . .

11.           ResultSet rs=con.executeQuery("SELECT cust_name, purchase_amt FROM Sales WHERE purchase_amt ⩾ 200 ");
             . . .
             . . .
             . . .
          /∗ Points increment based on the purchase amount and wallet balance. ∗/
15.           con
       .executeUpdate("UPDATE Sales SET point = point + 2 WHERE (purchase_amt + wallet_bal) ⩾ 5000 AND (purchase_amt + wallet_bal) < 10000 ")
       ;

16.           con.executeUpdate("UPDATE Sales SET point = point + 4 WHERE (purchase_amt + wallet_bal) ⩾ 10000 "); } catch
       (Exception e) { . . . } }}
```

Figure 5.1: Database Code Snippet `Prog`

the column coverage for checking integrity constraints.

5.3 A Running Example

Consider the database code snippet `Prog` depicted in Figure 5.1. The code implements a module which provides a set of offers on various purchases made on an online shopping system.

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

Observing the code carefully, we can identify a number of dependencies among the statements in `Prog`. Some of them, although exist syntactically, may not be valid dependencies when we consider semantics of the program. For example, although statement 6 is syntactically DD-dependent on statement 5, but they are semantically independent as the values of the attribute *purchase_amt* defined by statement 5 can never be used by statement 6. In the subsequent sections, we pursue various existing approaches to refine dependency information, and finally we propose an abstract interpretation-based approach to approximate defined and used database parts by database statements (at various levels of abstractions) and hence to compute semantics-based dependencies among them based on the overlapping. We show, in section 5.7, how the proposed approach effectively identifies false DD-dependencies in `Prog`.

5.4 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.

5.4.1 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) [100] 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 and an imperative statement, and (3) Database-Database dependency (DD-dependency) which represents a dependency between two SQL statements. This is to observe that syntax-based PP-dependencies and control dependencies in DOPDGs are the same as syntax-based data-dependencies and control-dependencies in PDGs respectively[2]. Let us define them below:

Definition 5.1 (Program-Program (PP) dependency [100]) *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 5.2 (Program-Database (PD) dependency [121]) *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 5.3 (Database-Database (DD) dependency) *A database statement Q_2 is DD-dependent on another database statement Q_1 for an attribute a (denoted $Q_1 \xrightarrow{a} Q_2$) if the following conditions hold:* (*i*) *a is defined by Q_1,* (*ii*) *a is used by Q_2, and* (*iii*) *there is no rollback operation in between them, which undoes the effect of Q_1 on a.*

The syntax-based dependency computation depends on the syntactic presence of one variable in the definition of another variable or on the control structure of the program. Let $\mathbb{C}$, $\mathbb{V}_a$ and $\mathbb{V}_d$ be the sets of statements, application-variables and database-attributes in database programs. Let $\mathbb{V} = \mathbb{V}_a \cup \mathbb{V}_d$ where $\mathbb{V}_a \cap \mathbb{V}_d = \emptyset$. The construction of syntax-based DOPDG can be formalized based on the two following functions:

$$\mathsf{USE}: \ \mathbb{C} \rightarrow \wp(\mathbb{V}) \qquad (5.1)$$

$$\mathsf{DEF}: \ \mathbb{C} \rightarrow \wp(\mathbb{V}) \qquad (5.2)$$

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

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

[2]In the rest of the paper, we represent DD-, PD-, control-dependencies by blue dashed-line, red dotted-line and black line respectively.

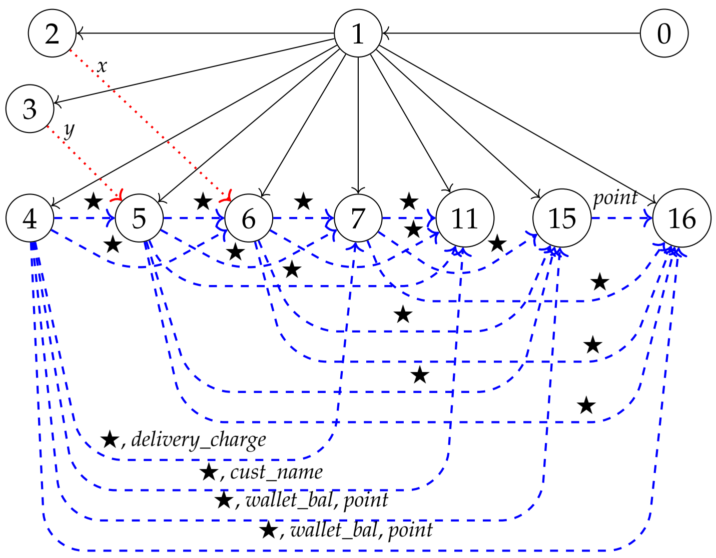


Figure 5.2: Pure Syntax-based DOPDG (★ denotes attribute *purchase_amt*) of `Prog`

Example 22 *Consider our running example* `Prog` *depicted in Figures 5.1. The control dependencies 1→2, 1→3, 1→4, etc. are computed in similar way as in the case of traditional PDG. The used and defined variables at each program point of* `Prog` *are computed as follows:*

$DEF(2) = \{x\}$ $\qquad DEF(3) = \{y\}$

$DEF(4) = \{purchase\_amt, delivery\_charge, cust\_name,$
$\qquad wallet\_bal, point\}$

$DEF(5) = \{purchase\_amt\}$ $\qquad USE(5) = \{purchase\_amt,\ y\}$

$DEF(6) = \{purchase\_amt\}$ $\qquad USE(6) = \{purchase\_amt,\ x\}$

$DEF(7) = \{purchase\_amt\}$

$USE(7) = \{purchase\_amt,\ delivery\_charge\}$

$USE(11) = \{purchase\_amt,\ cust\_name\}$

$DEF(15) = \{point\}$

$USE(15) = \{purchase\_amt,\ wallet\_bal,\ point\}$

$DEF(16) = \{point\}$

$USE(16) = \{purchase\_amt,\ wallet\_bal,\ point\}$

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

- *DD-dependencies for purchase_amt:* $4 \dashrightarrow 5$, $4 \dashrightarrow 6$, $4 \dashrightarrow 7$, $4 \dashrightarrow 11$,

$4 \dashrightarrow 15$, $4 \dashrightarrow 16$, $5 \dashrightarrow 6$, $5 \dashrightarrow 7$, $5 \dashrightarrow 11$, $5 \dashrightarrow 15$, $5 \dashrightarrow 16$, $6 \dashrightarrow 7$, $6 \dashrightarrow 11$, $6 \dashrightarrow 15$, $6 \dashrightarrow 16$, $7 \dashrightarrow 11$, $7 \dashrightarrow 15$, $7 \dashrightarrow 16$,

- *DD-dependencies for other attributes:* $4 \dashrightarrow 7$, $4 \dashrightarrow 11$, $4 \dashrightarrow 15$, $4 \dashrightarrow 16$, $15 \dashrightarrow 16$

- *PD-dependencies for x and y:* $2 \dashrightarrow 6$, $3 \dashrightarrow 5$

The syntax-based DOPDG of `Prog` *is depicted in Figure 5.2.*

Limitations.

Syntax-based dependency computation often introduces false dependencies, leading to an imprecise analysis. For instance, in Example 22, 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 $15 \dashrightarrow 16$. 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.

5.4.2 An Improved Syntax-driven Construction of DOPDGs

The authors in [2] proposed 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:

$$\mathsf{USE}: \ \mathbb{C} \rightarrow \wp(\mathbb{V} \times \mathbb{L}) \tag{5.3}$$

$$\mathsf{DEF}: \ \mathbb{C} \rightarrow \wp(\mathbb{V} \times \mathbb{L}) \tag{5.4}$$

where $\mathbb{L} = \{$●,◒$\}$ is a set of labels. The label ● 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 other hand, the label ◒ 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 ● (i.e., *fully-defined*). Let us illustrate this on our running example.

Example 23 *Applying equations 5.3 and 5.4 on all statements in `Prog` of the running example, we get the following information:*

DEF(2) =$\{(x,$ ●$)\}$ DEF(3) = $\{(y,$ ●$)\}$
DEF(4) =$\{(purchase\_amt,$●$),$ $(cust\_name,$●$),$ $(point,$●$),$
$(wallet\_bal,$●$),$ $(delivery\_charge,$●$)\}$
DEF(5) =$\{(purchase\_amt,$◒$)\}$
USE(5) =$\{(purchase\_amt,$◒$), (y,$●$)\}$
DEF(6) =$\{(purchase\_amt,$◒$)\}$
USE(6) =$\{(purchase\_amt,$◒$), (x,$●$)\}$
DEF(7) =$\{(purchase\_amt,$●$)\}$
USE(7) =$\{(purchase\_amt,$●$),$ $(delivery\_charge,$●$)\}$
USE(11)=$\{(purchase\_amt,$ ◒$),$ $(cust\_name,$ ◒$)\}$
DEF(15)=$\{(point,$ ◒$)\}$
USE(15)=$\{(purchase\_amt,$ ◒$), (wallet\_bal,$ ◒$), (point,$ ◒$)\}$
DEF(16)=$\{(point,$ ◒$)\}$
USE(16)=$\{(purchase\_amt,$ ◒$), (wallet\_bal,$ ◒$), (point,$ ◒$)\}$

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

- *DD-dependencies for purchase_amt:* 4 $\dashrightarrow$ 5 , 4 $\dashrightarrow$ 6 , 4 $\dashrightarrow$ 7 , 5 $\dashrightarrow$ 6 , 5 $\dashrightarrow$ 7 , 6 $\dashrightarrow$ 7 , 7 $\dashrightarrow$ 11 , 7 $\dashrightarrow$ 15 , 7 $\dashrightarrow$ 16 ,

- *DD-dependencies for other attributes:* 4 $\dashrightarrow$ 7 , 4 $\dashrightarrow$ 11 , 4 $\dashrightarrow$ 15 , 4 $\dashrightarrow$ 16 , 15 $\dashrightarrow$ 16

- *PD-dependencies for x and y:* 2 $\dashrightarrow$ 6 , 3 $\dashrightarrow$ 5

The label ● 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* $4 \dashrightarrow 11$ *,* $5 \dashrightarrow 11$ *,* $6 \dashrightarrow 11$ *,* $4 \dashrightarrow 15$ *,* $5 \dashrightarrow 15$ *,* $6 \dashrightarrow 15$ *,* $4 \dashrightarrow 16$ *,* $5 \dashrightarrow 16$ *and* $6 \dashrightarrow 16$ *for purchase_amt. Observe that the DD-dependency* $4 \dashrightarrow 11$ *exists for cust_name and dependencies* $4 \dashrightarrow 15$ *,* $4 \dashrightarrow 16$ *exist for both wallet_bal and point. The improved syntax-based DOPDG of* `Prog` *is depicted in Figures 5.3.*

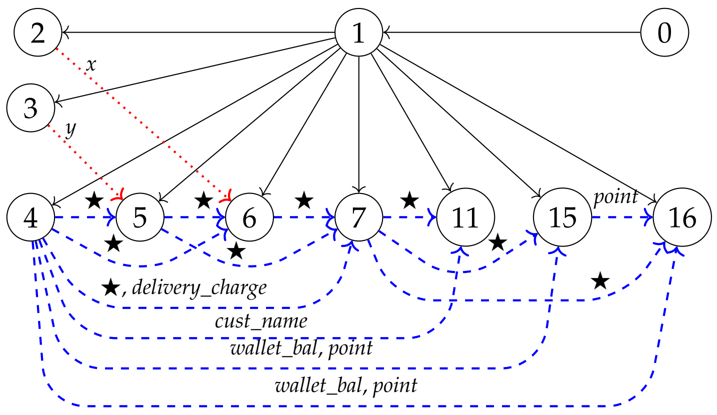


Figure 5.3: Improved Syntax-based DOPDG of `Prog` (★ denotes attribute *purchase_amt*)

Limitations.

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

5.4.3 DOPDG Construction on Condition-Action Rules

Although Willmor et al. [121] defined PD-dependency (Definition 5.2) in terms of syntax, however interestingly they defined DD-dependency in terms of *defined* and *used* values (see Definition 5.4). This leads to an improvement in the precision of DD-dependency computation. However, the preciseness depends on how precisely one can identify the overlapping of database-parts by various database operations.

Definition 5.4 (Database-Database (DD) dependency [121]) *Let Q.SEL, Q.INS, Q.UPD and Q.DEL denote the parts of database state which are selected, inserted, updated, and deleted respectively by Q. A database statement Q_1 is DD-dependent on another database statement*

Q_2 iff (i) the database-part defined by Q_2 overlaps the database-part used by Q_1, i.e. $Q_1.SEL \cap (Q_2.INS \cup Q_2.UPD \cup Q_2.DEL) \neq \emptyset$, and (ii) there is no roll-back operation in the execution path p between Q_2 and Q_1 (exclusive) which reverses back the effect of Q_2.

As a solution to compute this overlapping, Willmor et al. refer to the propagation algorithm in [35] designed for the static analysis of Condition-Action rules in expert database systems. The Condition-Action rules defined in an expert database system enable it to react automatically in some situations without the need of user access. These rules are, in general, expressed in the form $\mathsf{E}_{cond} \longrightarrow \mathsf{E}_{act}$, where E_{act} represents an action as data modification operation (e.g. `INSERT`, `UPDATE` and `DELETE`) and E_{cond} represents a condition. Formally, [10] considers an extended version of the relational algebra by introducing an additional operator ε, known as attribute extension operator, in case of database update. This operator is defined as $\varepsilon[x = expr]e$, where the expression $expr$ is evaluated over each tuple t of e and the resulting value is entered into the new attribute x for t under the new schema $schema(e) \cup \{x\}$. Let us illustrate this with running example.

Example 24 *Consider our running example in Section 5.3. Following the extended relational algebra, we get the following Condition-Action rules at program points 5 and 6:*

$$
\begin{aligned}
&E^5_{cond} \rightarrow \pi_{purchase\_amt}(\sigma_{purchase\_amt \geqslant 1000 \wedge purchase\_amt \leqslant 3000}\ Sales)\\
&E^5_{act} \rightarrow \varepsilon[purchase\_amt' = purchase\_amt\ -\ 0.05 \times\\
&\qquad purchase\_amt](\sigma_{purchase\_amt \geqslant 1000 \wedge purchase\_amt \leqslant 3000}\ Sales)\\
&E^6_{cond} \rightarrow \pi_{purchase\_amt}(\sigma_{purchase\_amt > 3000}\ Sales)\\
&E^6_{act} \rightarrow \varepsilon[purchase\_amt' = purchase\_amt\ -\ 0.1 \times\\
&\qquad\qquad\qquad purchase\_amt](\sigma_{purchase\_amt > 3000}\ Sales)
\end{aligned}
$$

where π and σ 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_{act}) and the post-defined part (i.e., database-part obtained after performing the action E_{act}) are in use by the condition E_{cond}; (*ii*) the pre-defined part is not in use by E_{cond} whereas the post-defined part is in use by E_{cond}, (*iii*) the pre-defined part is in use by E_{cond} whereas the post-defined part is not in use by E_{cond}. Let us illustrate this by recalling the rules already defined in example 24. 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 25 *Consider the Condition-Action rules at program points 5 and 6 of our running example expressed in Example 24. Observe that the predicates* ($1000 \leqslant purchase\_amt \leqslant 3000$) *in* E^5_{act} *and* ($purchase\_amt > 3000$) *in* E^6_{cond} *are contradictory – meaning that* E^5_{act} *operates on a part of data which is not accessed by* E^6_{cond}*. In other words, the action* E^5_{act} *does not affect the condition* E^6_{cond}*. Therefore, DD-dependency* $5 \dashrightarrow 6$ *is false. Similarly we can also identify another false DD-dependency* $15 \dashrightarrow 16$ *. The refined set of data dependencies are:*

- *DD-dependencies for purchase_amt:* $4 \dashrightarrow 5$, $4 \dashrightarrow 6$, $4 \dashrightarrow 7$, $4 \dashrightarrow 11$, $4 \dashrightarrow 15$, $4 \dashrightarrow 16$, $5 \dashrightarrow 7$, $5 \dashrightarrow 11$, $5 \dashrightarrow 15$, $5 \dashrightarrow 16$, $6 \dashrightarrow 7$, $6 \dashrightarrow 11$, $6 \dashrightarrow 15$, $6 \dashrightarrow 16$, $7 \dashrightarrow 11$, $7 \dashrightarrow 15$, $7 \dashrightarrow 16$

- *DD-dependencies for other attributes:* $4 \dashrightarrow 7$, $4 \dashrightarrow 11$, $4 \dashrightarrow 15$, $4 \dashrightarrow 16$

- *PD-dependencies for x and y:* $2 \cdots\!\rightarrow 6$, $3 \cdots\!\rightarrow 5$

Figure 5.4 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 $4 \dashrightarrow 11$, $4 \dashrightarrow 15$, $4 \dashrightarrow 16$, $5 \dashrightarrow 11$, $5 \dashrightarrow 15$, $5 \dashrightarrow 16$, $6 \dashrightarrow 11$, $6 \dashrightarrow 15$ and $6 \dashrightarrow 16$ in `Prog` due to the presence of multiple definitions of *purchase_amt* 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

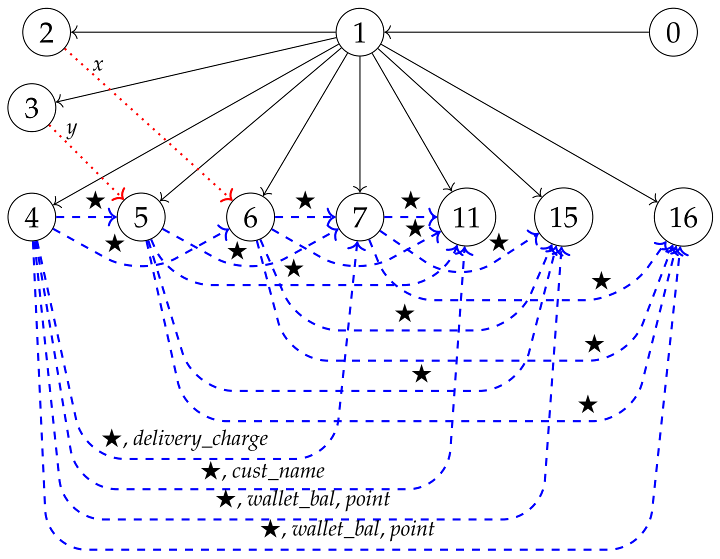


Figure 5.4: Condition-Action Rules-based DOPDG of `Prog` (★ denotes attribute *purchase_amt*)

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.

5.5 Semantics-based Dependency: A Formalization in Concrete Domain

As witnessed in section 5.4, 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 [89]. For instance, consider an arithmetic expression "$e = x^2 + 4w \bmod 2 + z$". Although in this expression e syntactically depends on w, semantically there is no dependency as the evaluation of "$4w \bmod 2$" is always zero.

Given a SQL statement $Q = \langle A, \phi \rangle$ and its target table t. Suppose $\vec{x} = \texttt{USE}(A)$, $\vec{y} = \texttt{USE}(\phi)$ and $\vec{z} = \texttt{DEF}(\text{Q})$. According to the concrete semantics, suppose $\mathscr{T}_{dba}[\![Q]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_a)$.

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

$$\mathbf{A}_{def}(Q, t) = \Delta(\rho_{t'}(\vec{z}), \rho_t(\vec{z})) \tag{5.5}$$

$$\mathbf{A}_{use}(Q, t) = \rho_{t\downarrow\phi}(\vec{x}) \cup \rho_{t\downarrow\phi}(\vec{y}) \tag{5.6}$$

where

$t \downarrow \phi$	:	Set of tuples in table t which satisfies the condition-part ϕ.
$\rho_{t\downarrow\phi}(\vec{x})$	:	Values of $\vec{x}$ in $(t \downarrow \phi)$.
$\rho_{t\downarrow\phi}(\vec{y})$	:	Values of $\vec{y}$ in $(t \downarrow \phi)$.
Δ	:	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 $\mathbf{A}_{use}$ maps a query Q to the part of the database information used by it, whereas the function $\mathbf{A}_{def}$ defines the changes occurred in the database states when data is updated or deleted or inserted by Q. The following example illustrates this.

Example 26 *Let us consider the concrete database table t shown in Table 3.2(a) and the following update statement:*

$$Q_{upd} : \texttt{UPDATE}\ t\ \texttt{SET}\ sal := sal + 100\ \texttt{WHERE}\ age \geqslant 35$$

where $A = \texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle)$ *and* $\phi = age \geqslant 35$. *According to equations 5.5 and 5.6, the* used-*part and defined-part are as follows:*

$$\mathbf{A}_{use}(Q_{upd}, t) = \rho_{t\downarrow(age\geqslant 35)}(sal) \cup \rho_{t\downarrow(age\geqslant 35)}(age)$$

$$\mathbf{A}_{\mathrm{def}}(Q_{upd}, t) = \Delta(\rho_{t'}(sal), \rho_t(sal))$$

These are depicted in Tables 5.1(a) and 5.1(b) respectively where we have denoted $\mathbf{A}_{use}(Q_{upd}, t)$ *and* $\mathbf{A}_{\mathrm{def}}(Q_{upd}, t)$ *by red color.*

(a) $\mathbf{A}_{use}(Q_{upd}, t)$

eid	*sal*	*age*	*dno*
1	1500	35	10
2	800	28	20
3	2500	50	10
4	3000	62	10

(b) $\mathbf{A}_{def}(Q_{upd}, t)$

eid	*sal*	*age*	*dno*
1	1600	35	10
2	800	28	20
3	2600	50	10
4	3100	62	10

Table 5.1: The *used* and *defined* part of t by Q_{upd} (marked with 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 $target(Q_1) = t$ and $\mathscr{T}_{dba}[\![Q_1]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_a)$ and $target(Q_2) = t'$. Following the equations 5.5 and

5.6, we can compute the *defined* part of t by Q_1 as $\mathbf{A}_{def}(Q_1, t)$ and *used*-part of t' by Q_2 as $\mathbf{A}_{use}(Q_2, t')$. Therefore, we can say Q_2 is DD-dependent on Q_1 when $\mathbf{A}_{def}(Q_1, t)$ and $\mathbf{A}_{use}(Q_2, t')$ overlap with each other, i.e. $\mathbf{A}_{use}(Q_2, t') \cap \mathbf{A}_{def}(Q_1, t) \neq \emptyset$. Observe that Q_1 is either `UPDATE`, `INSERT` and `DELETE` statement which defines the database. This is defined in Definition 5.5.

Definition 5.5 (Semantics-based DD-dependency [53]) *A SQL statement $Q_2 = \langle A_2, \phi_2 \rangle$ with target$(Q_2) = t'$ is DD-dependent for Υ on another SQL statement $Q_1 = \langle A_1, \phi_1 \rangle$ with target$(Q_1) = t$ (denoted $Q_1 \xrightarrow{\Upsilon} Q_2$) if $Q_1 \in \{Q_{upd}, Q_{ins}, Q_{del}\}$ and $\mathscr{T}_{dba}[\![Q_1]\!](\rho_t, \rho_a) = (\rho_{t'}, \rho_a)$ and the overlapping-part $\Upsilon = \boldsymbol{A}_{use}(Q_2, t') \cap \boldsymbol{A}_{\text{def}}(Q_1, t) \neq \emptyset$.*

When an initial database instance is unknown, due to infiniteness of the concrete domains, the computation of concrete semantics of database programs and hence $\mathbf{A}_{use}$, $\mathbf{A}_{def}$ and Υ 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 [30] to compute abstract semantics of database languages, in a decidable way, as a sound approximation of its concrete counterparts.

5.6 Semantics-based Abstract dependency: A Sound Approximation

In this section, we define abstract semantics of database statements for independency computation 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.

5.6.1 Defining Abstract Semantics of Database Statements towards Independency Computation

Since our objective is to compute semantics-based DD-independencies, it is important to identify database-parts (identified by the condition ϕ) before and after performing the action A. With this objective, unlike equation 3.3 which results in a single abstract

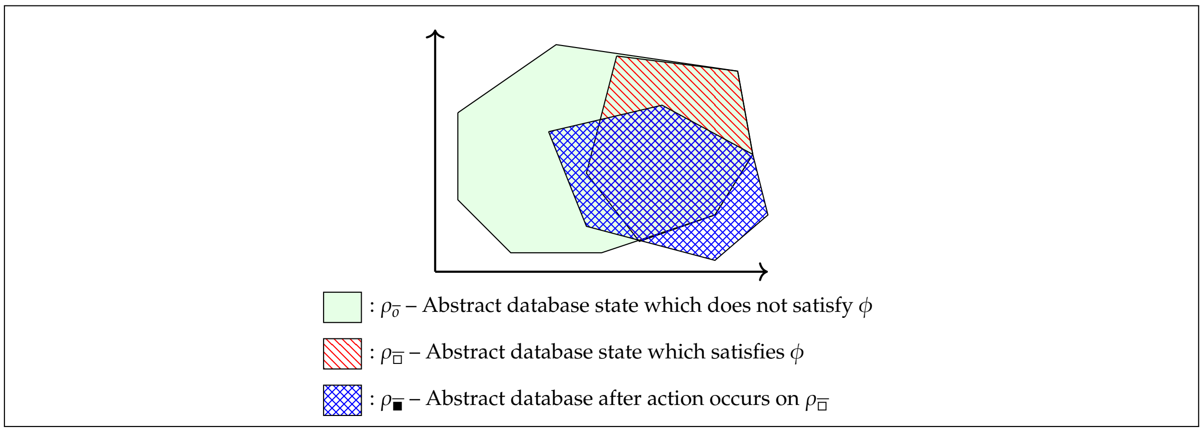

$\rho_{\overline{o}}$ – Abstract database state which does not satisfy ϕ
$\rho_{\overline{\square}}$ – Abstract database state which satisfies ϕ
$\rho_{\overline{\blacksquare}}$ – Abstract database after action occurs on $\rho_{\overline{\square}}$

Figure 5.5: Generic visual representation of $\langle \rho_{\overline{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$

state $\overline{\rho}$, we define a variant of the abstract transition relation as follows:

$$\overline{\mathscr{T}}_{dep} : \mathbb{C} \times \overline{\Sigma}_{dba} \mapsto (\overline{\mathfrak{E}}_{dbs} \times \overline{\mathfrak{E}}_{dbs} \times \overline{\mathfrak{E}}_{dbs}) \tag{5.7}$$

which results in a three-tuple $\langle \rho_{\overline{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$ of abstract database states, where $\rho_{\overline{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \in \overline{\mathfrak{E}}_{dbs}$. The first component $\rho_{\overline{o}}$ represents an abstract database state which does not satisfy ϕ, whereas the second component $\rho_{\overline{\square}}$ represents an abstract database state which satisfies ϕ. Observe that an abstract database-part which may or may not satisfy ϕ (due to abstraction) will be included in both $\rho_{\overline{o}}$ and $\rho_{\overline{\square}}$. The third component $\rho_{\overline{\blacksquare}}$ is obtained after performing an action A on $\rho_{\overline{\square}}$. These are depicted in Figure 5.5.

The abstract semantics of database statements for independency computation in various abstract domains following equation 5.7 is defined as follows:

5.6.1.1 Domain of Intervals

In order to compute semantics-based DD-independencies, we define $\overline{\mathscr{T}}_{dep}$, according to equations 5.7, in $\mathbb{I}$ for database statements as follows:

$$\begin{aligned}
&\overline{\mathscr{T}}_{dep}[\![\langle A, \phi\rangle]\!]\overline{\rho}\\
&\quad = \overline{\mathscr{T}}_{dep}[\![\langle A, \phi\rangle]\!](\rho_{\overline{d}}\,,\ \rho_{\overline{a}})\\
&\quad = \overline{\mathscr{T}}_{dep}[\![\langle A, \phi\rangle]\!](\rho_{\overline{t}}\,,\ \rho_{\overline{a}})\\
&\qquad\quad \text{where } t = \mathit{target}(\langle A, \phi\rangle) \text{ and } \exists \overline{t} \in \overline{d} :\ t \in \gamma(\overline{t})\\
&\quad = \Big\langle \rho_{\overline{FM}}\,,\ \rho_{\overline{TM}}\,,\ \rho_{\overline{TM'}} \Big\rangle
\end{aligned} \tag{5.8}$$

where

- $\overline{\mathscr{T}_f}[\![\phi]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) = (\rho_{\overline{TM}}\,,\ \rho_{\bar{a}})$
- $\overline{\mathscr{T}_f}[\![\neg\phi]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) = (\rho_{\overline{FM}}\,,\ \rho_{\bar{a}})$
- $\overline{\mathscr{T}_c}[\![A]\!](\rho_{\overline{TM}}\,,\ \rho_{\bar{a}}) = (\rho_{\overline{TM'}}\,,\ \rho_{\bar{a}})$

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

`UPDATE` statement:

$$\overline{\mathscr{T}}_{dep}[\![\left\langle \texttt{UPDATE}(\vec{v}_d,\ \vec{e}),\ \ \phi\right\rangle]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) \ = \ \left\langle \rho_{\overline{FM}}\,,\ \ \rho_{\overline{TM}}\,,\ \ \rho_{\overline{TM'}}\right\rangle$$

$$\begin{aligned}
\text{where } & \overline{\mathscr{T}_f}[\![\neg\phi]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) = (\rho_{\overline{FM}}\,,\ \rho_{\bar{a}})\\
& \overline{\mathscr{T}_f}[\![\phi]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) = (\rho_{\overline{TM}}\,,\ \rho_{\bar{a}})\\
& \overline{\mathscr{T}_c}[\![\texttt{UPDATE}(\vec{v}_d,\vec{e})]\!](\rho_{\overline{TM}}\,,\ \rho_{\bar{a}})\\
& = (\rho_{\overline{TM}}[\vec{v}_d\ \leftarrow \text{-}\mathscr{T}_e[\![\vec{e}]\!](\rho_{\overline{TM}}\ \ ,\ \ \rho_{\bar{a}})],\ \ \rho_{\bar{a}})\\
& = (\rho_{\overline{TM'}}\,,\ \rho_{\bar{a}})
\end{aligned}$$

`INSERT` statement:

$$\overline{\mathscr{T}}_{dep}[\![\left\langle \texttt{INSERT}(\vec{v}_d,\ \vec{e}),\ \ false\right\rangle]\!](\rho_{\bar{t}}\ \ ,\ \ \rho_{\bar{a}}) = \left\langle \rho_{\bar{t}}\ \ ,\ \ \rho_{\perp}\ \ ,\ \ \rho_{\overline{new}}\right\rangle$$

$$\begin{aligned}
\text{where } & \overline{\mathscr{T}_f}[\![\neg false]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) = (\rho_{\bar{t}}\,,\ \rho_{\bar{a}})\\
& \overline{\mathscr{T}_f}[\![false]\!](\rho_{\bar{t}}\,,\ \rho_{\bar{a}}) = (\rho_{\perp}\,,\ \rho_{\bar{a}})\\
& \overline{\mathscr{T}_c}[\![\texttt{INSERT}(\vec{v}_d\,,\ \vec{e})]\!](\rho_{\perp}\,,\ \rho_{\bar{a}})\\
& = (\rho_{\perp}[\vec{v}_d\ \leftarrow\ \overline{\mathscr{T}_e}[\![\vec{e}]\!](\rho_{\perp}\,,\ \rho_{\bar{a}})]\,,\ \rho_{\bar{a}})\\
& = (\rho_{\overline{new}}\ \ ,\ \ \rho_{\bar{a}})
\end{aligned}$$

where $\rho_{\perp}$ maps the attributes to the bottom element in the abstract domain which represents *"undefined"* values.

DELETE statement:

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{DELETE}(\vec{v_d}),\ \phi\Big\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}})\ =\ \Big\langle \rho_{\overline{FM}}\ ,\ \rho_{\overline{TM}}\ ,\ \rho_{\perp}\Big\rangle$$

SELECT statement:

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{SELECT}\Big(v_a, f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\Big),\ \ \phi_1\Big\rangle]\!](\rho_{\bar{t}}\ ,\ \ \rho_{\bar{a}})$$
$$=\Big\langle \rho_{\overline{FM}}\ ,\ \rho_{\overline{TM}}\ ,\ \rho_{\overline{TM}}\Big\rangle$$

Observe that the select operation does not change any information.

Example 27 *Consider the abstract state $\overline{\rho} = \langle \rho_{\bar{t}}\ ,\ \rho_{\bar{a}}\rangle$ and $\rho_{\bar{a}} = \langle x \mapsto [100, 100]\rangle$ where $\bar{t}$ is depicted in Table 3.3(b), as defined in Example 16. Consider the following statements:*

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

The abstract syntax of the statements are:

$$Q_{upd} = \langle \textit{UPDATE}(\langle sal\rangle, \langle sal + x\rangle),\ sal \geqslant 1500\rangle$$
$$Q_{ins} = \langle \textit{INSERT}(\langle eid,\ sal,\ age,\ dno\rangle,\ \langle 5, 2700, 52, 20\rangle),\ false\rangle$$
$$Q_{del} = \langle \textit{DELETE}(\langle eid,\ sal,\ age,\ dno\rangle),\ age\ \geqslant\ 61\rangle$$
$$Q_{sel} = \langle \textit{SELECT}(\langle age\rangle),\ age \leqslant 50\rangle$$

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

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \textit{UPDATE}(\langle sal\rangle,\ \ \langle sal\ +\ x\rangle),\ \ sal\ \geqslant\ 1500\Big\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}})$$
$$=\Big\langle \rho_{\overline{FM}}\ ,\ \rho_{\overline{TM}}\ ,\ \rho_{\overline{TM'}}\Big\rangle$$

where $\rho_{\overline{FM}}$, $\rho_{\overline{TM}}$ and $\rho_{\overline{TM'}}$ are shown in Table 3.5 of Example 16.

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

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{INSERT}(\langle eid, sal, age, dno\rangle, \langle 5, 2700, 52, 20\rangle), false\Big\rangle]\!](\rho_{\bar{t}}, \rho_{\bar{a}})$$
$$= \Big\langle \rho_{\bar{t}} \,,\, \rho_{\perp} \,,\, \rho_{\overline{new}} \Big\rangle \qquad where$$

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

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

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{DELETE}(\langle eid,\ sal,\ age,\ dno\rangle),\ age \geqslant 61\Big\rangle]\!](\rho_{\bar{t}} \,,\, \rho_{\bar{a}})$$
$$= \Big\langle \rho_{\overline{FM}} \,,\, \rho_{\overline{TM}} \,,\, \rho_{\perp} \Big\rangle \qquad where$$

$$\rho_{\overline{TM}} = \overline{\mathscr{T}_f}[\![\ age \geqslant 61\ ]\!](\rho_{\bar{t}}) = \rho_{\bar{t}}\Big[age \leftarrow [61, 62]\Big]$$
$$\rho_{\overline{FM}} = \overline{\mathscr{T}_f}[\![\neg(age \geqslant 61)\ ]\!](\rho_{\bar{t}}) = \rho_{\bar{t}}\Big[age \leftarrow [28, 60]\Big]$$

The abstract semantics of Q_{sel} w.r.t. $\overline{\rho}$ is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{SELECT}(\langle age\rangle), age \leqslant 50\Big\rangle]\!](\rho_{\bar{t}} \,,\, \rho_{\bar{a}}) = \Big\langle \rho_{\overline{FM}} \,,\, \rho_{\overline{TM}} \,,\, \rho_{\overline{TM}} \Big\rangle \quad where$$

$$\rho_{\overline{TM}} = \overline{\mathscr{T}_f}[\![\ age \leqslant 50\ ]\!](\rho_{\bar{t}}) = \rho_{\bar{t}}\Big[age \leftarrow [28, 50]\Big]$$
$$\rho_{\overline{FM}} = \overline{\mathscr{T}_f}[\![\neg(\ age \leqslant 50)\ ]\!](\rho_{\bar{t}}) = \rho_{\bar{t}}\Big[age \leftarrow [51, 62]\Big]$$

5.6.1.2 Relational Abstract Domain of Octagons

To yield more precise analysis as compared to the interval abstract domain, Antoine Miné [94] proposed a weakly 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.

Like for the interval domain, the following transition relation is defined, according to equation 5.7, to compute semantics-based DD-independency in the domain of octagons:

$$\overline{\mathscr{T}}_{dep} : \mathbb{C} \times \mathbb{M}_{\perp} \mapsto (\mathbb{M}_{\perp} \times \mathbb{M}_{\perp} \times \mathbb{M}_{\perp})$$

Below is the definition of $\overline{\mathscr{T}}_{dep}$ for various database statements in octagon domain.

1. UPDATE:

$$\overline{\mathscr{T}}_{dep}[\![\langle \mathtt{UPDATE}(\vec{v}_d,\ \vec{e}),\ \phi\rangle]\!]\mathsf{m} = \begin{cases} \Big\langle \mathsf{m}_F,\ \mathsf{m}_T,\ \mathsf{m}_{T'}\Big\rangle \text{ if } \phi \in \Big\{k_i x_i + k_j x_j \leqslant k\Big\} \text{ where } x_i, x_j \in \mathbb{V} \text{ and } k_i, k_j \in [-1,0,1] \text{ and } k \in \mathbb{R} \\ \langle \mathsf{m}, \mathsf{m}, \mathsf{m}'\rangle \quad \text{otherwise} \end{cases}$$

where

$$\overline{\mathscr{T}_f}[\![\neg\phi]\!]\mathsf{m} = \mathsf{m}_F \text{ and } \overline{\mathscr{T}_f}[\![\phi]\!]\mathsf{m} = \mathsf{m}_T$$

$$\overline{\mathscr{T}_c}[\![\mathtt{UPDATE}(\vec{v}_d,\ \vec{e})]\!]\mathsf{m}_T = \mathsf{m}_T\Big[\vec{v}_d \leftarrow \overline{\mathscr{T}_e}[\![\vec{e}]\!]\mathsf{m}_T\Big] = \mathsf{m}_{T'}$$

$$\overline{\mathscr{T}_c}[\![\mathtt{UPDATE}(\vec{v}_d, \vec{e})]\!]\mathsf{m} = \mathsf{m}'$$

2. INSERT: $\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{INSERT}(\vec{v}_d, \vec{e}), \mathit{false}\Big\rangle]\!]\mathsf{m} = \Big\langle \mathsf{m},\ \mathsf{m}_{\perp},\ \mathsf{m}_{new}\Big\rangle$ where

$$\overline{\mathscr{T}_f}[\![\neg \mathit{false}]\!]\mathsf{m} = \mathsf{m} \text{ and } \overline{\mathscr{T}_f}[\![\mathit{false}]\!]\mathsf{m} = \mathsf{m}_{\perp} \text{ and}$$

$$\overline{\mathscr{T}_c}[\![\mathtt{INSERT}(\vec{v}_d, \vec{e})]\!]\mathsf{m}_{\perp} = \mathsf{m}_{\perp}[\vec{v}_d \leftarrow \overline{\mathscr{T}_e}[\![\vec{e}]\!]\mathsf{m}_{\perp}] = \mathsf{m}_{new}$$

where $\mathsf{m}_{\perp}$ represents bottom element that contains an unsatisfiable set of constraints.

3. DELETE:

$$\overline{\mathscr{T}}_{dep}[\![\langle \mathtt{DELETE}(\vec{v}_d),\ \phi\rangle]\!]\mathsf{m} = \begin{cases} \Big\langle \mathsf{m}_F,\ \mathsf{m}_T,\ \mathsf{m}_{\perp}\Big\rangle \text{ if } \phi \in \Big\{k_i x_i + k_j x_j \leqslant k\Big\} \text{ where } x_i, x_j \in \mathbb{V} \text{ and } k_i, k_j \in [-1,0,1] \text{ and } k \in \mathbb{R} \\ \langle \mathsf{m}, \mathsf{m}, \mathsf{m}_{\perp}\rangle \quad \text{otherwise} \end{cases}$$

4. SELECT:

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathsf{SELECT}\Big(v_a, f(\vec{e'}), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})\Big), \phi_1\Big\rangle]\!]\mathsf{m} = \begin{cases} \Big\langle \mathsf{m}_F,\ \mathsf{m}_T,\ \mathsf{m}_T\Big\rangle \text{ if } \phi \in \Big\{k_i x_i + k_j x_j \leqslant k\Big\} \text{ where} \\ x_i, x_j \in \mathbb{V} \text{ and } k_i, k_j \in [-1, 0, 1] \text{ and } k \in \mathbb{R} \\ \\ \langle \mathsf{m}, \mathsf{m}, \mathsf{m}\rangle \quad \text{otherwise} \end{cases}$$

Observe that $\phi \in \{k_i x_i + k_j x_j \leqslant k\}$ checks whether the condition in `WHERE` clause of database statement respects the form of octagonal constraints.

Example 28 *Consider the concrete database table t shown in Table 3.3(a), and its corresponding abstract representation in the form of CDBM m_t in the domain of octagons as*

$$\mathsf{m}_t \overset{rep}{=} \Big\{ -eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000, \\ -age \leqslant -28, age \leqslant 62, -dno \leqslant -10, dno \leqslant 20\Big\}.$$

Consider the following statements:

$$Q_{upd} = \texttt{UPDATE}\ t\ \texttt{SET}\ sal = sal + x\ \texttt{WHERE}\ age \geqslant 35$$
$$Q_{ins} = \texttt{INSERT INTO}\ t(eid, sal, age, dno)\texttt{VALUES}(5, 2700, 52, 20)$$
$$Q_{del} = \texttt{DELETE FROM}\ t\ \texttt{WHERE}\ age \geqslant 61$$
$$Q_{sel} = \texttt{SELECT}\ sal\ \texttt{FROM}\ t\ \texttt{WHERE}\ age \leqslant 50$$

The abstract syntax are

$$Q_{upd} = \langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle), age \geqslant 35\rangle$$
$$Q_{ins} = \langle \texttt{INSERT}(\langle eid, sal, age, dno\rangle, \langle 5, 2700, 52, 20\rangle), false\rangle$$
$$Q_{del} = \langle \texttt{DELETE}(\langle eid, sal, age, dno\rangle), age \geqslant 61\rangle$$
$$Q_{sel} = \langle \texttt{SELECT}(\langle sal\rangle), age \leqslant 50\rangle$$

The abstract semantics of the Q_{upd} with respect to m_t is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle, \langle sal + 100\rangle), age \geqslant 35\Big\rangle]\!]\mathsf{m}_t = \Big\langle \mathsf{m}_F, \mathsf{m}_T, \mathsf{m}_{T'}\Big\rangle$$

where m_T, m_F and $\mathsf{m}_{T'}$ are depicted in example 18.

The abstract semantics of the Q_{ins} w.r.t. m_t is

$$\begin{aligned}&\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{INSERT}(\langle eid, sal, age, dno\rangle, \langle 5, 2700, 52, 20\rangle),\ false\Big\rangle]\!]\mathsf{m}_t\\ &= \Big\langle \mathsf{m}_t,\ \mathsf{m}_\perp,\ \mathsf{m}_{new}\Big\rangle \qquad \textit{where}\end{aligned}$$

$$\begin{aligned}\mathsf{m}_{new} \overset{rep}{=} \Big\{ &-eid \leqslant -5, eid \leqslant 5, -sal \leqslant -2700, sal \leqslant 2700,\\ &-age \leqslant -52,\ age \leqslant 52, -dno \leqslant -20,\ dno \leqslant 20\Big\}\end{aligned}$$

The abstract semantics of the Q_{del} w.r.t. m_t is

$$\begin{aligned}&\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{DELETE}(\langle eid,\ sal,\ age,\ dno\rangle),\ age \geqslant 61\Big\rangle]\!]\mathsf{m}_t\\ &= \Big\langle \mathsf{m}_F,\ \mathsf{m}_T,\ \mathsf{m}_\perp\Big\rangle \qquad \textit{where}\end{aligned}$$

$$\begin{aligned}\mathsf{m}_T \overset{rep}{=} \Big\{ &-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000,\ age \leqslant 62,\\ &-age \leqslant -61, -dno \leqslant -10, dno \leqslant 20\Big\}\\ \mathsf{m}_F \overset{rep}{=} \Big\{ &-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000,\ age \leqslant 60,\\ &-age \leqslant -28, -dno \leqslant -10, dno \leqslant 20\Big\}\end{aligned}$$

The abstract semantics of the Q_{sel} w.r.t. m_t is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{SELECT}(\langle age\rangle),\ age \leqslant 50\Big\rangle]\!]\mathsf{m}_t = \Big\langle \mathsf{m}_F,\ \mathsf{m}_T,\ \mathsf{m}_T\Big\rangle \quad \textit{where}$$

$$\begin{aligned}\mathsf{m}_T \overset{rep}{=} \Big\{ &-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000,\ age \leqslant 50,\\ &-age \leqslant -28, -dno \leqslant -10, dno \leqslant 20\Big\}\\ \mathsf{m}_F \overset{rep}{=} \Big\{ &-eid \leqslant -1, eid \leqslant 4, -sal \leqslant -800, sal \leqslant 3000,\ age \leqslant 62,\\ &-age \leqslant -51, -dno \leqslant -10, dno \leqslant 20\Big\}\end{aligned}$$

5.6.1.3 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 [36] 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) [66], Information flow analysis to detect possible information leakages combining symbolic propositional formulas domain and numerical polyhedra domain [122], Hybrid systems verification tool SpaceEx [46], etc.

Let us define the transition relation $\overline{\mathscr{T}}_{dep} : \mathbb{C}\times\mathbb{P} \mapsto (\mathbb{P}\times\mathbb{P}\times\mathbb{P})$ to compute semantics-based DD-independency in the domain of polyhedra for database statements:

1. UPDATE: $\overline{\mathscr{T}}_{dep}[\![\langle \mathtt{UPDATE}(\vec{v}_d, \vec{e}),\ \phi\rangle]\!]\mathsf{P} = \Big\langle \mathsf{P}_F, \mathsf{P}_T, \mathsf{P}_{T'}\Big\rangle$ where

$$\overline{\mathscr{T}_f}[\![\neg\phi]\!]\mathsf{P} = \mathsf{P}_F.$$
$$\overline{\mathscr{T}_f}[\![\phi]\!]\mathsf{P} = \mathsf{P}_T$$
$$\overline{\mathscr{T}_c}[\![\mathtt{UPDATE}(\vec{v}_d, \vec{e})]\!]\mathsf{P}_T = \overline{\mathscr{T}_c}[\![\vec{v}_d = \vec{e}]\!]\mathsf{P}_T = \mathsf{P}_{T'}$$

We denote by the notation $\vec{v}_d{=}\,\vec{e}$ a series of assignments $\langle v_1{=}\,e_1, v_2{=}\,e_2, \ldots, v_n{=}\,e_n\rangle$ where $\vec{v}_d = \langle v_1, v_2, \ldots, v_n\rangle$ and $\vec{e}{=}\langle e_1, e_2, \ldots, e_n\rangle$, which follow the transition semantic definition for the assignment statement.

2. INSERT: $\overline{\mathscr{T}}_{dep}[\![\langle \mathtt{INSERT}(\vec{v}_d, \vec{e}), false\rangle]\!]\mathsf{P} = \Big\langle \mathsf{P}, \mathsf{P}_\perp, \mathsf{P}_{new}\Big\rangle$ where

$$\overline{\mathscr{T}_f}[\![\neg false]\!]\mathsf{P} = \mathsf{P}$$
$$\overline{\mathscr{T}_f}[\![false]\!]\mathsf{P} = \mathsf{P}_\perp$$
$$\overline{\mathscr{T}_c}[\![\mathtt{INSERT}(\vec{v}_d, \vec{e})]\!]\mathsf{P}_\perp = \mathsf{P}_\perp\Big[\vec{v}_d \leftarrow \overline{\mathscr{T}_e}[\![\vec{e}]\!]\mathsf{P}_\perp\Big] = \mathsf{P}_{new}$$

P_{new} represents a polyhedron corresponding to the new inserted tuple values.

3. DELETE: $\overline{\mathscr{T}}_{dep}[\![\langle \mathtt{DELETE}(\vec{v}_d),\ \phi\rangle]\!]\mathsf{P} = \Big\langle \mathsf{P}_F,\ \mathsf{P}_T,\ \mathsf{P}_\perp\Big\rangle$

4. SELECT: $\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{SELECT}\Big(v_a, f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\Big),\ \phi_1\Big\rangle]\!]\mathsf{P} = \Big\langle \mathsf{P}_F, \mathsf{P}_T, \mathsf{P}_T\Big\rangle$

Example 29 *Consider the database table t in Table 3.3(a) and its corresponding abstract representation $\mathtt{P}_t = (\Theta, n)$ in the form of polyhedron, where*

$$\mathtt{P}_t = \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, \\ age \geqslant 28, -age \geqslant -62, dno \geqslant 10, -dno \geqslant -20\Big\}$$

Consider the following statements:

$$\begin{aligned}
Q_{upd} =& \texttt{UPDATE}\ t\ \texttt{SET}\ sal = sal + sal \times 0.2\ \texttt{WHERE}\ dno + age \geqslant 60\\
Q_{ins} =& \texttt{INSERT INTO}\ t\ (eid, sal, age, dno) \texttt{VALUES}(5, 2700, 52, 20)\\
Q_{del} =& \texttt{DELETE FROM}\ t\ \texttt{WHERE}\ age \geqslant 61\\
Q_{sel} =& \texttt{SELECT}\ age\ \texttt{FROM}\ t\ \texttt{WHERE}\ age + dno \leqslant 60
\end{aligned}$$

The equivalent abstract syntax are:

$$\begin{aligned}
Q_{upd} =& \langle \texttt{UPDATE}(\langle sal\rangle,\ \langle sal + sal \times 0.2\rangle),\ dno + age \geqslant 60\rangle\\
Q_{ins} =& \langle \texttt{INSERT}(\langle eid,\ sal,\ age,\ dno\rangle, \langle 5, 2700, 52, 20\rangle),\ false\rangle\\
Q_{del} =& \langle \texttt{DELETE}(\langle eid,\ sal,\ age,\ dno\rangle),\ age \geqslant 61\rangle\\
Q_{sel} =& \langle \texttt{SELECT}(\langle age\rangle),\ age + dno \leqslant 60\rangle
\end{aligned}$$

The abstract semantics of Q_{upd} w.r.t. $\mathtt{P}_t$ is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \texttt{UPDATE}(\langle sal\rangle,\ \langle sal + sal \times 0.2\rangle),\ dno + age \geqslant 60\Big\rangle]\!]\mathtt{P}_t = \Big\langle \mathtt{P}_F,\ \mathtt{P}_T,\ \mathtt{P}_{T'}\Big\rangle \quad where$$

$$\mathtt{P}_T = \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 40, \\ -age \geqslant -62, dno \geqslant 10, -dno \geqslant -20, dno + age \geqslant 60\Big\}$$

$$\mathtt{P}_F = \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 28, \\ -age \geqslant -49, dno \geqslant 10, -dno \geqslant -20, -dno - age \geqslant -59\Big\}$$

$$\mathtt{P}_{T'} = \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 960, -sal \geqslant -3600, age \geqslant 40,$$

$$-age \geqslant -62, dno \geqslant 10, -dno \geqslant -20, dno + age \geqslant 60\Big\}$$

The abstract semantics of Q_{ins} w.r.t. P_t is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{INSERT}(\langle eid, sal, age, dno\rangle, \langle 5, 2700, 52, 20\rangle),\ false\Big\rangle]\!]\mathsf{P}_t = \Big\langle \mathsf{P}_t,\ \mathsf{P}_\perp,\ \mathsf{P}_{new}\Big\rangle \quad where$$

$$\begin{aligned}\mathsf{P}_{new} =& \Big\{eid \geqslant 5, -eid \geqslant -5, sal \geqslant 2700, -sal \geqslant -2700, age \geqslant 52,\\ & -age \geqslant -52, dno \geqslant 20, -dno \geqslant -20\Big\}\end{aligned}$$

The abstract semantics of Q_{del} w.r.t. P_t is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{DELETE}(\langle eid,\ sal,\ age,\ dno\rangle),\ age\ \geqslant\ 61\Big\rangle]\!]\mathsf{P}_t = \Big\langle \mathsf{P}_F,\ \mathsf{P}_T,\ \mathsf{P}_\perp\Big\rangle \qquad where$$

$$\begin{aligned}\mathsf{P}_T =& \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 61,\\ & -age \geqslant -62, dno \geqslant 10, -dno \geqslant -20\Big\}\\ \mathsf{P}_F =& \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 28,\\ & -age \geqslant -60, dno \geqslant 10, -dno \geqslant -20\Big\}\end{aligned}$$

The abstract semantics of Q_{sel} w.r.t. P_t is

$$\overline{\mathscr{T}}_{dep}[\![\Big\langle \mathtt{SELECT}(\langle age\rangle),\ age + dno \leqslant 60\Big\rangle]\!]\mathsf{P}_t = \Big\langle \mathsf{P}_F,\ \mathsf{P}_T,\ \mathsf{P}_T\Big\rangle \qquad where$$

$$\begin{aligned}\mathsf{P}_T =& \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 800, -sal \geqslant -3000, age \geqslant 28,\\ & -age \geqslant -50, dno \geqslant 10, -dno \geqslant -20, -dno - age \geqslant -60\Big\}\end{aligned}$$

$$\begin{aligned}\mathsf{P}_F =& \Big\{eid \geqslant 1, -eid \geqslant -4, sal \geqslant 960, -sal \geqslant -3600, age \geqslant 51,\\ & -age \geqslant -62, dno \geqslant 10, -dno \geqslant -20, dno + age \geqslant 61\Big\}\end{aligned}$$

5.6.1.4 Powerset of Interval Domain

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.

To compute semantics-based dependencies, we adapt equation 5.7 for identifying *used* and *defined* database-parts in powerset abstract domain. Given the semantic domain $\wp(\overline{\mathbb{I}})$, the abstract state is defined as $\overline{\rho} = (\rho_{\bar{t}},\ \rho_{\bar{a}})$ where $\rho_{\bar{t}} : \mathtt{attr}(\bar{t}) \rightarrow \wp(\overline{\mathbb{I}})$ and $\rho_{\bar{a}} : \mathbb{V}_a \rightarrow \wp(\overline{\mathbb{I}})$.

Like other domains, according to equation 5.7, the abstract semantics in the powerset abstract domain for database statements are similarly defined below:

$$\texttt{UPDATE:}\quad \overline{\mathscr{T}}_{dep}[\![\left\langle \texttt{UPDATE}(\vec{v}_d,\ \vec{e})\ ,\ \phi \right\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) \ = \ \left\langle \rho_{\overline{FM}}\ ,\ \rho_{\overline{TM}}\ ,\ \rho_{\overline{TM'}} \right\rangle$$

$$\begin{aligned} \text{where } & \overline{\mathscr{T}_f}[\![\neg\phi]\!](\rho_{\bar{t}}, \rho_{\bar{a}}) = (\rho_{\overline{FM}}\ ,\ \rho_{\bar{a}}) \\ & \overline{\mathscr{T}_f}[\![\phi]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) = (\rho_{\overline{TM}}\ ,\ \rho_{\bar{a}}) \\ & \overline{\mathscr{T}_c}[\![\texttt{UPDATE}(\vec{v}_d, \vec{e})]\!](\rho_{\overline{TM}}\ ,\ \rho_{\bar{a}}) \\ & = (\rho_{\overline{TM}}[\vec{v}_d \ \leftarrow\ \overline{\mathscr{T}_e}[\![\vec{e}]\!](\rho_{\overline{TM}}\ ,\ \rho_{\bar{a}})]\ ,\ \rho_{\bar{a}}) \\ & = (\rho_{\overline{TM'}}\ ,\ \rho_{\bar{a}}) \end{aligned}$$

$$\texttt{INSERT:}\quad \overline{\mathscr{T}}_{dep}[\![\left\langle \texttt{INSERT}(\vec{v}_d,\ \vec{e}),\ false \right\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) = \left\langle \rho_{\bar{t}}\ ,\ \rho_{\perp}, \rho_{\overline{new}} \right\rangle$$

$$\begin{aligned} \text{where } & \overline{\mathscr{T}_f}[\![\neg false]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) = (\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) \\ & \overline{\mathscr{T}_f}[\![false]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) = (\rho_{\perp}\ ,\ \rho_{\bar{a}}) \text{ where } \rho_{\perp} : \mathtt{attr}(t) \rightarrow \emptyset \\ & \overline{\mathscr{T}_c}[\![\texttt{INSERT}(\vec{v}_d, \vec{e})]\!](\rho_{\perp}\ ,\ \rho_{\bar{a}}) \\ & = (\rho_{\perp}[\vec{v}_d \leftarrow \overline{\mathscr{T}_e}[\![\vec{e}]\!](\rho_{\perp}\ ,\ \rho_{\bar{a}})]\ ,\ \rho_{\bar{a}}) \\ & = (\rho_{\overline{new}}\ ,\ \rho_{\bar{a}}) \end{aligned}$$

$$\texttt{DELETE:}\quad \overline{\mathscr{T}}_{dep}[\![\left\langle \texttt{DELETE}(\vec{v}_d),\ \ \phi \right\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) = \left\langle \rho_{\overline{FM}}\ ,\ \rho_{\overline{TM}}\ ,\ \rho_{\perp} \right\rangle$$

SELECT: $\overline{\mathscr{T}}_{dep}[\![\left\langle \mathsf{SELECT}\left(v_a,\ f(\vec{e'}),\ r(\vec{h}(\vec{x})),\ \phi_2,\ g(\vec{e})\right),\ \phi_1 \right\rangle]\!](\rho_{\bar{t}}\ ,\ \rho_{\bar{a}}) = \left\langle \rho_{\overline{FM}}\ ,\ \rho_{\overline{TM}}\ ,\ \rho_{\overline{TM}} \right\rangle$

5.6.2 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 $\overline{\mathscr{T}}_{dba}$ and computes abstract states w.r.t. abstract domain $\overline{D}$ 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) until least fixed point solution is reached. After obtaining the abstract state at each program point in the form of collecting semantics, step 26 applies $\overline{\mathscr{T}}_{dep}$ in order to get state-representation in the form of three-tuples $\langle \rho_{\bar{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$ (as defined in equation 5.7). This abstract semantics is used to compute *used*- and *defined*-parts and hence the semantics-based independencies (described next). Observe that if the initial database is unknown then the domain range of each attribute and other integrity constraints are considered to represent the initial abstraction of database as an overapproximation of all possible initial database states, as defined in steps 1 and 9.

5.6.3 Approximating *used*- and *defined* Database Parts in Various Abstract Domains

Given a database statement Q, let $\overline{\rho} = \langle \rho_{\bar{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$ be an abstract state at Q obtained by following Algorithm 1. In order to determine abstract DD-dependency between two database statements, we need to identify abstract database-parts to be *defined* or *used* by Q. To this aim, let us define sound abstract functions $\overline{\mathbf{A}}_{def}$ and $\overline{\mathbf{A}}_{use}$ w.r.t. their concrete counterparts already defined in equations 5.5 and 5.6 respectively. Suppose D^{Q} and U^{Q} denote the *defined* and the *used* abstract database-parts by Q respectively. Therefore,

$$\mathsf{D}^{\mathsf{Q}} = \overline{\mathbf{A}}_{def}(Q, \overline{\rho}) = \langle \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle \tag{5.9}$$

Algorithm 1: CompAbsSem

```
Input: Database program $\mathcal{P}$ containing $n$ statements, Initial database $dB$, and
       Abstract domain $\overline{D}$.
Output: Abstract state at each program point of $\mathcal{P}$

Compute $\overline{\rho} = \langle \rho_{\overline{dB}}, \rho_{\overline{a}} \rangle$ using the abstraction function $\alpha$ following the Galois
connection $\Big\langle (\wp(D), \subseteq), \alpha, \gamma, (\overline{D}, \sqsubseteq) \Big\rangle$ as formalized in Definition 3.1.
$\forall i \in [1, \ldots, n], AS(c_i) := \bot.$ ;     // Initializing $AS(c_i)$ as initial abstract collecting
semantics.
Apply data flow equations defined in steps 4 - 25 until least fix-point is reached.
for $i$ =1 to $n$ do
    ; // Defining data flow equation for CFG-node corresponding to a statement $c_i$ in $\mathcal{P}$.
    switch ($c_i$)
        case start:
            $AS(c_i)=\bot$
        case DB-connect:
            $AS(c_i)=\langle \rho_{\overline{dB}}, \rho_{\overline{a}} \rangle$
        case assignment:
            $AS(c_i) = \bigsqcup_{c_j \in pred(c_i)} \Big\{ \overline{\mathcal{T}}_{dba}[\![x = e]\!](\overline{\rho}) \mid \overline{\rho} \in AS(c_j) \Big\}$
        case test:
            $AS(c_i) = \bigsqcup_{c_j \in pred(c_i)} \Big\{ \overline{\mathcal{T}}_{dba}[\![b]\!](\overline{\rho}) \mid \overline{\rho} \in AS(c_j) \Big\}$
        case update:
            $AS(c_i) = \bigsqcup_{c_j \in pred(c_i)} \Big\{ \overline{\mathcal{T}}_{dba}[\![\texttt{UPDATE}(\vec{v}_d, \vec{e}), \phi \rangle]\!](\overline{\rho}) \mid \overline{\rho} \in AS(c_j) \Big\}$
        case delete:
            $AS(c_i) = \bigsqcup_{c_j \in pred(c_i)} \Big\{ \overline{\mathcal{T}}_{dba}[\![\texttt{DELETE}(\vec{v}_d), \phi \rangle]\!](\overline{\rho}) \mid \overline{\rho} \in AS(c_j) \Big\}$
        case insert:
            $AS(c_i) = \bigsqcup_{c_j \in pred(c_i)} \Big\{ \overline{\mathcal{T}}_{dba}[\![\texttt{INSERT}(\vec{v}_d, \vec{e}), false \rangle]\!](\overline{\rho}) \mid \overline{\rho} \in AS(c_j) \Big\}$
        case select:
            $AS(c_i) = \bigsqcup_{c_j \in pred(c_i)} \Big\{ \overline{\mathcal{T}}_{dba}[\![\langle \texttt{SELECT}\big(v_a, f(\vec{e}), r(\vec{h}(\vec{x})), \phi_2, g(\vec{e})\big), \phi_1 \rangle]\!](\overline{\rho})$
            $\mid \overline{\rho} \in AS(c_j) \Big\}$
        case join:
            $AS(c_i)= \bigsqcup_{c_j \in pred(c_i)} AS(c_j)$
        case end:
            $AS(c_i)=\bigsqcup_{c_j \in pred(c_i)} AS(c_j)$
Apply the abstract transition relation $\overline{\mathcal{T}}_{dep}$ on the abstract state $AS(c_j)$ obtained at
each program point.
End
```

SQL	Abstract Domain of Intervals $\overline{\rho}$		Abstract Domain of Octagons $\overline{\rho}$		Abstract Domain of Polyhedra $\overline{\rho}$		Powerset of Interval $\overline{\rho}_{\mathbb{I}}$
	Abstract state	*defined-/ used*-part	Abstract state	*defined-/ used*-part	Abstract state	*defined-/ used*-part	*defined-/ used*-part
Update Q_{upd}	$\left\langle \rho_{\overline{FM}}, \rho_{\overline{TM}}, \rho_{\overline{TM'}} \right\rangle$	$\mathbf{A}_{def}(Q_{upd}, \overline{\rho}) = \langle \rho_{\overline{TM}}, \rho_{\overline{TM'}} \rangle$	$\left\langle \mathsf{m}_F, \mathsf{m}_T, \mathsf{m}'_T \right\rangle$	$\mathbf{A}_{def}(Q_{upd}, \overline{\rho}) = \langle \mathsf{m}_T, \mathsf{m}'_T \rangle$	$\left\langle \mathsf{P}_F, \mathsf{P}_T, \mathsf{P}'_T \right\rangle$	$\mathbf{A}_{def}(Q_{upd}, \overline{\rho}) = \langle \mathsf{P}_T, \mathsf{P}'_T \rangle$	$\mathbf{A}_{def}(Q_{upd}, \overline{\rho}_{\mathbb{I}}) = \langle \rho_{\overline{TM}}, \rho_{\overline{TM'}} \rangle$
		$\mathbf{A}_{use}(Q_{upd}, \overline{\rho}) = \langle \rho_{\overline{TM}} \rangle$		$\mathbf{A}_{use}(Q_{upd}, \overline{\rho}) = \langle \mathsf{m}_T \rangle$		$\mathbf{A}_{use}(Q_{upd}, \overline{\rho}) = \langle \mathsf{P}_T \rangle$	$\mathbf{A}_{use}(Q_{upd}, \overline{\rho}_{\mathbb{I}}) = \langle \rho_{\overline{TM}} \rangle$
Delete Q_{del}	$\left\langle \rho_{\overline{FM}}, \rho_{\overline{TM}}, \rho_{\perp} \right\rangle$	$\mathbf{A}_{def}(Q_{del}, \overline{\rho}) = \langle \rho_{\overline{TM}}, \emptyset \rangle$	$\left\langle \mathsf{m}_F, \mathsf{m}_T, \mathsf{m}_{\perp} \right\rangle$	$\mathbf{A}_{def}(Q_{del}, \overline{\rho}) = \langle \mathsf{m}_T, \emptyset \rangle$	$\left\langle \mathsf{P}_F, \mathsf{P}_T, \mathsf{P}_{\perp} \right\rangle$	$\mathbf{A}_{def}(Q_{del}, \overline{\rho}) = \langle \mathsf{P}_T, \emptyset \rangle$	$\mathbf{A}_{def}(Q_{del}, \overline{\rho}_{\mathbb{I}}) = \langle \rho_{\overline{TM}}, \emptyset \rangle$
		$\mathbf{A}_{use}(Q_{del}, \overline{\rho}) = \langle \rho_{\overline{TM}} \rangle$		$\mathbf{A}_{use}(Q_{del}, \overline{\rho}) = \langle \mathsf{m}_T \rangle$		$\mathbf{A}_{use}(Q_{del}, \overline{\rho}) = \langle \mathsf{P}_T \rangle$	$\mathbf{A}_{use}(Q_{del}, \overline{\rho}_{\mathbb{I}}) = \langle \rho_{\overline{TM}} \rangle$
Insert Q_{ins}	$\left\langle \rho_{\overline{t}}, \rho_{\perp}, \rho_{\overline{new}} \right\rangle$	$\mathbf{A}_{def}(Q_{ins}, \overline{\rho}) = \langle \emptyset, \rho_{\overline{new}} \rangle$	$\left\langle \mathsf{m}_t, \mathsf{m}_{\perp}, \mathsf{m}_{new} \right\rangle$	$\mathbf{A}_{def}(Q_{ins}, \overline{\rho}) = \langle \emptyset, \mathsf{m}_{new} \rangle$	$\left\langle \mathsf{P}_t, \mathsf{P}_{\perp}, \mathsf{P}_{new} \right\rangle$	$\mathbf{A}_{def}(Q_{ins}, \overline{\rho}) = \langle \emptyset, \mathsf{P}_{new} \rangle$	$\mathbf{A}_{def}(Q_{ins}, \overline{\rho}_{\mathbb{I}}) = \langle \emptyset, \rho_{\overline{new}} \rangle$
		$\mathbf{A}_{use}(Q_{ins}, \overline{\rho}) = \langle \emptyset \rangle$		$\mathbf{A}_{use}(Q_{ins}, \overline{\rho}) = \langle \emptyset \rangle$		$\mathbf{A}_{use}(Q_{ins}, \overline{\rho}) = \langle \emptyset \rangle$	$\mathbf{A}_{use}(Q_{ins}, \overline{\rho}_{\mathbb{I}}) = \langle \emptyset \rangle$
Select Q_{sel}	$\left\langle \rho_{\overline{FM}}, \rho_{\overline{TM}}, \rho_{\overline{TM}} \right\rangle$	$\mathbf{A}_{def}(Q_{sel}, \overline{\rho}) = \langle \emptyset, \emptyset \rangle$	$\left\langle \mathsf{m}_F, \mathsf{m}_T, \mathsf{m}_T \right\rangle$	$\mathbf{A}_{def}(Q_{sel}, \overline{\rho}) = \langle \emptyset, \emptyset \rangle$	$\left\langle \mathsf{P}_F, \mathsf{P}_T, \mathsf{P}_T \right\rangle$	$\mathbf{A}_{def}(Q_{sel}, \overline{\rho}) = \langle \emptyset, \emptyset \rangle$	$\mathbf{A}_{def}(Q_{sel}, \overline{\rho}_{\mathbb{I}}) = \langle \emptyset, \emptyset \rangle$
		$\mathbf{A}_{use}(Q_{sel}, \overline{\rho}) = \langle \rho_{\overline{TM}} \rangle$		$\mathbf{A}_{use}(Q_{sel}, \overline{\rho}) = \langle \mathsf{m}_T \rangle$		$\mathbf{A}_{use}(Q_{sel}, \overline{\rho}) = \langle \mathsf{P}_T \rangle$	$\mathbf{A}_{use}(Q_{sel}, \overline{\rho}_{\mathbb{I}}) = \langle \rho_{\overline{TM}} \rangle$

Table 5.2: Abstract *defined*- and *used*-part of database by SQL statements in various abstract domains

$$\mathsf{U}^{Q} = \overline{\mathbf{A}}_{use}(Q, \overline{\rho}) = \langle \rho_{\overline{\square}} \rangle \tag{5.10}$$

Observe that $\overline{\mathbf{A}}_{use}$ maps a query Q to the abstract database-part used by it, whereas $\overline{\mathbf{A}}_{def}$ 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 $\rho_{\overline{\square}}$ and $\rho_{\overline{\blacksquare}}$ respectively represent the true-part before and the updated-part after executing Q on the abstract database. Note that although the *defined*-part can be computed by following the abstract difference operation $\overline{\Delta}$ (corresponding to Δ defined in equation 5.5), however to avoid computational complexity in dependency computation, we keep both of these separated. Table 5.2 depicts *defined* and *used* parts by different database statements in various abstract domains.

5.6.4 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.

Let $\overline{\rho}^{Q_1} = \langle \rho^{Q_1}_{\overline{o}}, \rho^{Q_1}_{\overline{\square}}, \rho^{Q_1}_{\overline{\blacksquare}} \rangle$ and $\overline{\rho}^{Q_2} = \langle \rho^{Q_2}_{\overline{o}}, \rho^{Q_2}_{\overline{\square}}, \rho^{Q_2}_{\overline{\blacksquare}} \rangle$ be the abstract states at Q_1 and Q_2 respectively. The *defined*-part by Q_1 and the *used*-part by Q_2 are :

$$\mathsf{D}^{Q_1} = \overline{\mathbf{A}}_{def}(Q_1, \overline{\rho}^{Q_1}) = \langle \rho^{Q_1}_{\overline{\square}}, \rho^{Q_1}_{\overline{\blacksquare}} \rangle$$
$$\mathsf{U}^{Q_2} = \overline{\mathbf{A}}_{use}(Q_2, \overline{\rho}^{Q_2}) = \langle \rho^{Q_2}_{\overline{\square}} \rangle$$

The semantic dependence and independence of Q_2 on Q_1 are determined based on the following four cases:

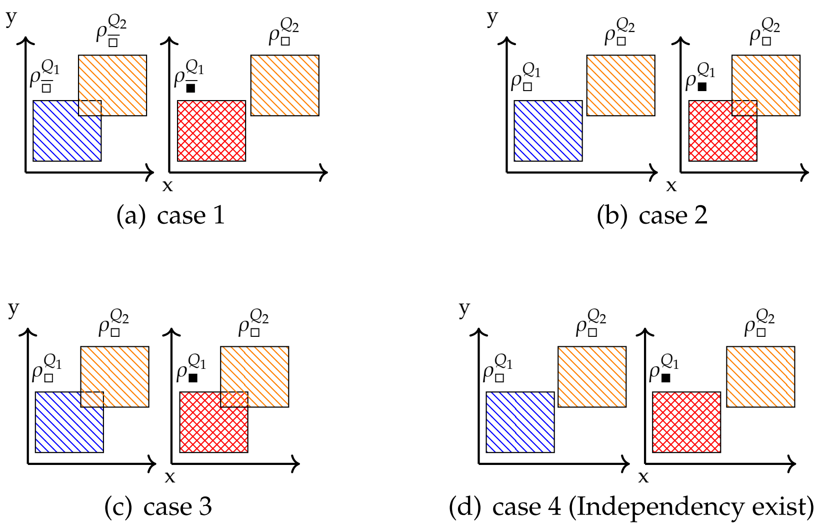
$$\textbf{Case} - \textbf{1}.\ \rho^{Q_1}_{\overline{\square}} \sqcap \rho^{Q_2}_{\overline{\square}} \neq \emptyset \wedge \rho^{Q_1}_{\overline{\blacksquare}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset$$

Case − 2. $\rho^{Q_1}_{\overline{\square}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset \wedge \rho^{Q_1}_{\overline{\blacksquare}} \sqcap \rho^{Q_2}_{\overline{\square}} \neq \emptyset$

Case − 3. $\rho^{Q_1}_{\overline{\square}} \sqcap \rho^{Q_2}_{\overline{\square}} \neq \emptyset \wedge \rho^{Q_1}_{\overline{\blacksquare}} \sqcap \rho^{Q_2}_{\overline{\square}} \neq \emptyset$

Case − 4. $\rho^{Q_1}_{\overline{\square}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset \wedge \rho^{Q_1}_{\overline{\blacksquare}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset$

The pictorial representation of the above cases are depicted in Figure 5.6. Observe that only case 4 indicates a semantic independency between Q_1 and Q_2 whereas all other cases indicate a semantic dependency between them. Therefore, Q_2 is DD-independent


y
x
$\rho^{Q_1}_{\overline{\square}}$
$\rho^{Q_2}_{\overline{\square}}$
$\rho^{Q_1}_{\overline{\blacksquare}}$

(a) case 1 (b) case 2 (c) case 3 (d) case 4 (Independency exist)

Figure 5.6: Representations of independence and dependencies

on Q_1 iff $\mathsf{D}^{Q_1} \sqcap \mathsf{U}^{Q_2} = \emptyset$; that is

$$\rho^{Q_1}_{\overline{\square}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset \wedge \rho^{Q_1}_{\overline{\blacksquare}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset \tag{5.11}$$

Theorem 5.1 states that, given an abstract domain, equation 5.11 is necessary and sufficient condition for abstract DD-independency. Observe that this theorem does not establish anything about its soundness w.r.t. its concrete counterpart.

Theorem 5.1 *Given an abstract domain, the necessary and sufficient condition for a database statement Q_2 to be abstract DD-independent on another statement Q_1 is $\rho^{Q_1}_{\overline{\square}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset \wedge \rho^{Q_1}_{\overline{\blacksquare}} \sqcap \rho^{Q_2}_{\overline{\square}} = \emptyset$.*

Proof 2 *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 $\overline{\rho}$ and $\overline{\rho}'$ at Q_1 and Q_2 respectively which are obtained in step 25 of Algorithm*

1. Let the abstract semantics applying $\overline{\mathscr{T}}_{dep}$ in step 26 be $\overline{\mathscr{T}}_{dep}[\![Q_1]\!]\overline{\rho} = \langle \rho_{\overline{o}}^{Q_1}, \rho_{\overline{\square}}^{Q_1}, \rho_{\overline{\blacksquare}}^{Q_1} \rangle$ and $\overline{\mathscr{T}}_{dep}[\![Q_2]\!]\overline{\rho}' = \langle \rho_{\overline{o}}^{Q_2}, \rho_{\overline{\square}}^{Q_2}, \rho_{\overline{\blacksquare}}^{Q_2} \rangle$. 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 ϕ_2. This is captured in* **Case-2***.*

2. *Removal of existing information: As a result of the modification done by Q_1 some information (which was previously accessed by Q_2) now can not be accessed by Q_2 due to the unsatisfiability of ϕ_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 $\rho_{\overline{\square}}^{Q_2}$ accessed by Q_2 overlaps with the parts $\rho_{\overline{\square}}^{Q_1}$ and $\rho_{\overline{\blacksquare}}^{Q_1}$ 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 semantic-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_i overlaps with the *used*-part by Q_j, 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_i and Q_j and *flag* is *true*, then in the next step 11 the algorithm checks the condition $\mathsf{D}^{Q_i} \sqsubseteq \mathsf{D}^{Q_j}$ in order to verify whether *defined*-part at program point Q_i is fully covered by the *defined*-part at program point Q_j. If yes, the execution immediately breaks the inner loop and does not check for dependency of the subsequent database statements (after Q_j) on Q_i, and hence disregards the false dependencies which may occur due to redefinition of attributes values by Q_j.

Algorithm 2: **semDOPDG**

```
Input: used- and defined-parts at each program point in the database program $\mathcal{P}$.
Output: Semantic-based DD-dependency

Set flag=true
Identify database statements present in $\mathcal{P}$. Let $m$ be the number of database
statements.
for $i$ =1 to m-1 do
    for $j$=$i$+1 to $m$ do
        if $D^{Q_i} \sqcap U^{Q_j} \neq \emptyset$ then
            Add edges from $i^{th}$ statement to $j^{th}$ statement ($i \rightarrow j$)
            Set flag = true;
        else
            Set flag = false;
        if flag==true then
            if $D^{Q_i} \sqsubseteq D^{Q_j}$ then
                break;
End
```

5.7 Illustration on the Running Example

Now we illustrate our approach on the running example `Prog` in section 5.3. 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 `Prog`.
- Compute *defined*- and *used*-parts based on the abstract semantics.
- Refinement of syntactic dependencies in `Prog` based on the semantics-based independencies using Algorithm 2.

A comparative result of the analysis in various abstract domains is depicted in Table 5.3. 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 $\overline{\rho}$ taking those variables into account with an assumption that *purchase_amt* is typed with unsigned smallint. Therefore, $\overline{\rho} = (\rho_{\overline{dB}}\ ,\ \rho_{\overline{a}})$ and $\overline{\mathscr{T}}_{dba}[\![4]\!](\overline{\rho}) = \overline{\rho}^4$ where $\rho^4_{\overline{dB}} = \langle purchase\_amt \mapsto [0,\ 65000]\rangle$. and $\rho_{\overline{a}} = \langle x \mapsto [0.1,\ 0.1],\ y \mapsto [0.05,\ 0.05]\rangle$.

Dependency	pure syntax-based	Improved syntax-based	Condition-Action rule-based	Interval domain	Octagon domain	Polyhedra domain
DD-dependency	$4 \dashrightarrow \{5,6,7,11,15,16\}$ $5 \dashrightarrow \{6,7,11,15,16\}$ $6 \dashrightarrow \{7,11,15,16\}$ $7 \dashrightarrow \{11,15,16\}$ $15 \dashrightarrow 16$	$4 \dashrightarrow \{5,6,7,11,15,16\}$ $5 \dashrightarrow \{6,7\}$ $6 \dashrightarrow 7$ $7 \dashrightarrow \{11,15,16\}$ $15 \dashrightarrow 16$	$4 \dashrightarrow \{5,6,7,11,15,16\}$ $5 \dashrightarrow \{7,11,15,16\}$ $6 \dashrightarrow \{7,11,15,16\}$ $7 \dashrightarrow \{11,15,16\}$	$4 \dashrightarrow \{5,6,7,11,15,16\}$ $5 \dashrightarrow 7$ $6 \dashrightarrow 7$ $7 \dashrightarrow \{11,15,16\}$ $15 \dashrightarrow 16$	$4 \dashrightarrow \{5,6,7,11,15,16\}$ $5 \dashrightarrow 7$ $6 \dashrightarrow 7$ $7 \dashrightarrow \{11,15,16\}$	$4 \dashrightarrow \{5,6,7,11,15,16\}$ $5 \dashrightarrow 7$ $6 \dashrightarrow 7$ $7 \dashrightarrow \{11,15,16\}$
PD-dependency	$2 \dashrightarrow 6$, $3 \dashrightarrow 5$	$2 \dashrightarrow 6$, $3 \dashrightarrow 5$	$2 \dashrightarrow 6$, $3 \dashrightarrow 5$	$2 \dashrightarrow 6$, $3 \dashrightarrow 5$	$2 \dashrightarrow 6$, $3 \dashrightarrow 5$	$2 \dashrightarrow 6$, $3 \dashrightarrow 5$

Table 5.3: Representation of dependency results on `Prog` in various approaches

The abstract semantics of statement 5 is

$$\overline{\mathscr{T}}_{dba}[\![5]\!](\overline{\rho}^4) = \overline{\rho}^5 \quad \text{and} \quad \overline{\mathscr{T}}_{dep}[\![5]\!](\overline{\rho}^4) = \left\langle \rho^5_{\overline{FM}}\,,\ \rho^5_{\overline{TM}}\,,\ \rho^5_{\overline{TM'}} \right\rangle$$

where $\rho^5_{\overline{FM}}$ is obtained from $\rho^4_{\overline{dB}}$ where the condition is not satisfied. This creates two intervals (*purchase_amt* $\leftarrow$ [0, 999]) and (*purchase_amt* $\leftarrow$ [3001, 65000]). Therefore, $\rho^5_{\overline{FM}}$ is represented using two abstract tuples l_1 and l_2 such that $\rho^5_{\overline{FM}}$=$\rho^4_{\overline{dB}}\Big[\ l_1(purchase\_amt \leftarrow [0,\ 999]),\ l_2(purchase\_amt \leftarrow [3001,\ 65000])\ \Big]$. The part for which the condition evaluates to true is $\rho^5_{\overline{TM}} = \rho^4_{\overline{dB}}\Big[purchase\_amt \leftarrow [1000,\ 3000]\Big]$ and therefore $\rho^5_{\overline{TM'}} = \rho_{\overline{TM}}\Big[purchase\_amt \leftarrow [950,\ 2850]\Big]$.

Similarly, abstract semantics of the statement 6 is

$$\overline{\mathscr{T}}_{dba}[\![6]\!](\overline{\rho}^5) = \overline{\rho}^6 \quad \text{and} \quad \overline{\mathscr{T}}_{dep}[\![6]\!](\overline{\rho}^5) = \left\langle \rho^6_{\overline{FM}}\,,\ \rho^6_{\overline{TM}}\,,\ \rho^6_{\overline{TM'}} \right\rangle$$

where $\rho^6_{\overline{FM}}$ =$\rho^5_{\overline{dB}}\Big[\ purchase\_amt \leftarrow [0,3000]\ \Big]$, $\rho^6_{\overline{TM}} = \rho^5_{\overline{dB}}\Big[\ purchase\_amt \leftarrow [3001,65000]\ \Big]$ and $\rho^6_{\overline{TM'}} = \rho_{\overline{TM}}\Big[\ purchase\_amt \leftarrow [2701,58500]\ \Big]$.

The *defined*-part by statement 5 and the *used*-part by statement 6 are defined as follows:

$$\mathsf{D}^5 = \overline{\mathbf{A}}_{def}(\overline{\rho}^5, 5) = \langle \rho^5_{\overline{TM}}\,,\ \rho^5_{\overline{TM'}} \rangle \text{ and } \mathsf{U}^6 = \overline{\mathbf{A}}_{use}(\overline{\rho}^6, 6) = \langle \rho^6_{\overline{TM}} \rangle$$

Therefore, the dependency $5 \dashrightarrow 6$ does not exist semantically as

$$\mathsf{D}^5 \sqcap \mathsf{U}^6 = \emptyset \Longrightarrow \rho^5_{\overline{TM}} \sqcap \rho^6_{\overline{TM}} = \emptyset \wedge \rho^5_{\overline{TM'}} \sqcap \rho^6_{\overline{TM}} = \emptyset$$

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 $15 \dashrightarrow\!\!\!\!\!/\ 16$.

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 $0 \leqslant point \leqslant 100$ and $100 \leqslant wallet\_bal \leqslant 90000$ are defined on '*point*' and '*wallet_bal*'. Therefore, the abstract state at program point 4 is:

$$\begin{aligned}\mathsf{P}^{11}_{dB} = \Big\{ & purchase\_amt \geqslant 0, -purchase\_amt \geqslant -65000, point \geqslant 0,\\ & -point \geqslant -100, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -90000 \Big\}\end{aligned}$$

The abstract semantics of statement 15 is defined as:

$$\overline{\mathscr{T}}_{dba}[\![15]\!](\mathsf{P}^{11}_{dB}) = \mathsf{P}^{15}_{dB} \text{ and } \overline{\mathscr{T}}_{dep}[\![15]\!](\mathsf{P}^{11}_{dB}) = \left\langle \mathsf{P}^{15}_{F}, \mathsf{P}^{15}_{T}, \mathsf{P}^{15}_{T'} \right\rangle \text{ where}$$

$$\begin{aligned}\mathsf{P}^{15}_{F} = \Big\{ & purchase\_amt \geqslant 0, -purchase\_amt \geqslant -65000, point \geqslant 0,\\ & -point \geqslant -100, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -90000 \Big\}\\ \mathsf{P}^{15}_{T} = \Big\{ & purchase\_amt \geqslant 0, -purchase\_amt \geqslant -9899, point \geqslant 0,\\ & -point \geqslant -100, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -9999,\\ & \ 5000 \leqslant purchase\_amt + wallet\_bal \leqslant 9999 \Big\}\\ \mathsf{P}^{15}_{T'} = \Big\{ & purchase\_amt \geqslant 0, -purchase\_amt \geqslant -9899, point \geqslant 2,\\ & -point \geqslant -102, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -9999,\\ & \ 5000 \leqslant purchase\_amt + wallet\_bal \leqslant 9999 \Big\}\end{aligned}$$

Similarly, abstract semantics of statement 16 is:

$$\overline{\mathscr{T}}_{dba}[\![16]\!](\mathsf{P}^{15}_{dB}) = \mathsf{P}^{16}_{dB} \text{ and } \overline{\mathscr{T}}_{dep}[\![16]\!](\mathsf{P}^{15}_{dB}) = \left\langle \mathsf{P}^{16}_{F}, \mathsf{P}^{16}_{T}, \mathsf{P}^{16}_{T'} \right\rangle \text{ where}$$

$$\begin{aligned}\mathsf{P}^{16}_{F} = \Big\{ & purchase\_amt \geqslant 0, -purchase\_amt \geqslant -9899, point \geqslant 0,\\ & -point \geqslant -100, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -9999\end{aligned}$$

$$\begin{aligned}
&\quad purchase\_amt + wallet\_bal \leqslant 9999\Big\} \\
\mathsf{P}^{16}_{T} =&\Big\{purchase\_amt \geqslant 0, -purchase\_amt \geqslant -65000, point \geqslant 0, \\
&\quad -point \geqslant -100, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -90000 \\
&\quad purchase\_amt + wallet\_bal \geqslant 10000\Big\} \\
\mathsf{P}^{16}_{T'} =&\Big\{purchase\_amt \geqslant 0, -purchase\_amt \geqslant -65000, point \geqslant 4, \\
&\quad -point \geqslant -104, wallet\_bal \geqslant 100, -wallet\_bal \geqslant -90000 \\
&\quad purchase\_amt + wallet\_bal \geqslant 10000\Big\}
\end{aligned}$$

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

$$\mathsf{D}^{15} = \overline{\mathbf{A}}_{def}(\mathsf{P}^{15}, 15) = \langle \mathsf{P}^{15}_{T}\ ,\ \mathsf{P}^{15}_{T'} \rangle \text{ and } \mathsf{U}^{16} = \overline{\mathbf{A}}_{use}(\mathsf{P}^{6}, 16) = \langle \mathsf{P}^{16}_{T} \rangle$$

Therefore the dependency $15 \dashrightarrow 16$ does not exist semantically, as

$$\mathsf{D}^{15} \sqcap \mathsf{U}^{16} = \emptyset \Longrightarrow \mathsf{P}^{15}_{T} \sqcap \mathsf{P}^{16}_{T} = \emptyset \land \mathsf{P}^{15}_{T'} \sqcap \mathsf{P}^{16}_{T} = \emptyset$$

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

5.8 Soundness of the Analysis

Lemma 5.1 *Let $\overline{\rho}$ be an abstract state. The abstract semantic function $\overline{\mathscr{T}}_{dep}$ is sound w.r.t. γ if $\forall Q \in \mathbb{Q}$, $\forall \rho \in \gamma(\overline{\rho})$: $\mathscr{T}_{dep}[\![Q]\!]\rho \subseteq \gamma(\overline{\mathscr{T}}_{dep}[\![Q]\!]\overline{\rho})$.*

Proof 3 *Given an abstract state $\overline{\rho}$ and a database statement $Q = \langle A,\ \phi \rangle \in \mathbb{Q}$, the abstract semantic function $\overline{\mathscr{T}}_{dep}$ on $\overline{\rho}$ computes abstract database state in the form of three-tuple as follows:*

$$\overline{\mathscr{T}}_{dep}[\![Q]\!]\overline{\rho} = \overline{\mathscr{T}}_{dep}[\![\langle A,\ \phi \rangle]\!]\overline{\rho} = \langle \rho_{\overline{\mathrm{o}}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$$

where $\rho_{\overline{\mathrm{o}}}$ represents abstract database state which must (or may) not satisfy ϕ, whereas $\rho_{\overline{\square}}$ represents abstract database state which must (or may) satisfy ϕ. $\rho_{\overline{\blacksquare}}$ is obtained after performing an action A on $\rho_{\overline{\square}}$. Now let ρ be a concrete state such that $\rho \in \gamma(\overline{\rho})$, the concrete semantics *similarly is defined as*

$$\mathscr{T}_{dep}[\![Q]\!]\rho = \mathscr{T}_{dep}[\![\langle A,\ \phi\rangle]\!]\rho = \langle \rho_o, \rho_\square, \rho_\blacksquare\rangle$$

where ρ_o, $\rho_\square$ represent concrete database state based on the satisfaction and dissatisfaction of ϕ respectively, and $\rho_\blacksquare$ is obtained after performing A on $\rho_\square$. Like in lemma 3.1, because of three-valued logic of ϕ due to the imprecision introduced in the abstract domain and the local correctness of the operations in A, we get $\rho_o \in \gamma(\rho_{\overline{o}})$, $\rho_\square \subseteq \gamma(\rho_{\overline{\square}})$ and $\rho_\blacksquare \subseteq \gamma(\rho_{\overline{\blacksquare}})$, which implies that $\mathscr{T}_{dep}[\![Q]\!]\rho \subseteq \gamma(\overline{\mathscr{T}}_{dep}[\![Q]\!]\overline{\rho})$.

Lemma 5.2 *Let $\overline{\rho}$ be an abstract state. The abstract function $\overline{\boldsymbol{A}}_{\text{def}}$ is sound w.r.t. γ if $\forall\rho \in \gamma(\overline{\rho})$, $\forall Q \in \mathbb{Q}$: $\gamma(\overline{\boldsymbol{A}}_{\text{def}}(Q,\ \overline{\rho})) \supseteq \boldsymbol{A}_{\text{def}}(Q,\ \rho)$*

Proof 4 *Given a database statement $Q = \langle A,\ \phi\rangle \in \mathbb{Q}$ and an abstract state $\overline{\rho}$, the abstract semantics (based on equation 5.7) is defined as $\overline{\mathscr{T}}_{dep}[\![Q]\!]\overline{\rho} = \overline{\mathscr{T}}_{dep}[\![\langle A,\ \phi\rangle]\!]\overline{\rho} = \langle \rho_{\overline{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}}\rangle$. As per the equation 5.9, the abstract* defined*-part is*

$$\overline{\boldsymbol{A}}_{\text{def}}(Q,\ \overline{\rho}) = \langle \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}}\rangle$$

Now given a concrete state $\rho = (\rho_t\ ,\ \rho_a) \in \gamma(\overline{\rho})$, we get the concrete semantics of Q, according to equation 3.1, as $\mathscr{T}_{dba}[\![Q]\!]\rho = \mathscr{T}_{dba}[\![\langle A,\ \phi\rangle]\!](\rho_t\ ,\ \rho_a) = (\rho_{t'}\ ,\ \rho_a)$. Alternatively, $\mathscr{T}_{dep}$ on ρ computes concrete semantics of Q as $\mathscr{T}_{dep}[\![Q]\!]\rho = \mathscr{T}_{dep}[\![\langle A,\ \phi\rangle]\!]\rho = \langle \rho_o, \rho_\square, \rho_\blacksquare\rangle$. As per the equation 5.5, we can define the defined*-part in the concrete domain by defining Δ, which computes the difference between database states before and after applying Q, in the form below:*

$$\boldsymbol{A}_{\text{def}}(Q,\ \rho_t) = \Delta(\rho_{t'}\ ,\ \rho_t) = \langle \rho_\square, \rho_\blacksquare\rangle$$

Assuming the local correctness of ϕ and A, we get $\rho_\square \subseteq \gamma(\rho_{\overline{\square}})$ and $\rho_\blacksquare \subseteq \gamma(\rho_{\overline{\blacksquare}})$ respectively. Therefore, $\gamma(\overline{\boldsymbol{A}}_{\text{def}}(Q,\ \overline{\rho})) \supseteq \boldsymbol{A}_{\text{def}}(Q,\ \rho)$.

Lemma 5.3 *Let $\overline{\rho}$ be an abstract state. The abstract function $\overline{\boldsymbol{A}}_{\text{use}}$ is sound w.r.t. γ if $\forall\rho \in \gamma(\overline{\rho})$, $\forall Q \in \mathbb{Q}$: $\gamma(\overline{\boldsymbol{A}}_{\text{use}}(Q,\ \overline{\rho})) \supseteq \boldsymbol{A}_{\text{use}}(Q,\ \rho)$*

Proof 5 *Proof is same as lemma 5.2.*

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

$$\forall X \in \gamma(\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho})), \forall Y \in \gamma(\overline{\boldsymbol{A}}_{use}(Q_2, \overline{\rho}')) : \\ X \cap Y \subseteq \gamma(\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho}) \sqcap \overline{\boldsymbol{A}}_{use}(Q_2, \overline{\rho}'))$$

which implies $\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho}) \sqcap \overline{\boldsymbol{A}}_{use}(Q_2, \overline{\rho}') = \emptyset \Rightarrow X \cap Y = \emptyset$.

Proof 6 *Consider two database statements Q_1 and Q_2. Let $\overline{\rho} = \langle \rho_{\overline{o}}, \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$ and $\overline{\rho}' = \langle \rho'_{\overline{o}}, \rho'_{\overline{\square}}, \rho'_{\overline{\blacksquare}} \rangle$ be the abstract states at Q_1 and Q_2 respectively, which are obtained by applying Algorithm 1. According to equations 5.9 and 5.10, we get the* defined*-part by Q_1 and the* used*-part by Q_2 as $\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho}) = \langle \rho_{\overline{\square}}, \rho_{\overline{\blacksquare}} \rangle$ and $\overline{\boldsymbol{A}}_{use}(Q_2, \overline{\rho}') = \langle \rho'_{\overline{\square}} \rangle$ respectively. Now, the semantics independency in abstract domain can be defined as $\rho_{\overline{\square}} \sqcap \rho'_{\overline{\square}} = \emptyset \wedge \rho_{\overline{\blacksquare}} \sqcap \rho'_{\overline{\square}} = \emptyset$. Given the concrete states $\rho = \langle \rho_o, \rho_\square, \rho_\blacksquare \rangle$ and $\rho' = \langle \rho'_o, \rho'_\square, \rho'_\blacksquare \rangle$ where $\rho \in \gamma(\overline{\rho})$ and $\rho' \in \gamma(\overline{\rho}')$, the semantics independency between Q_1 and Q_2 in the concrete domain is defined as $\rho_\square \cap \rho'_\square = \emptyset \wedge \rho_\blacksquare \cap \rho'_\square = \emptyset$. From lemma 5.2 and 5.3, we get $\gamma(\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho})) \supseteq \boldsymbol{A}_{\mathrm{def}}(Q_1, \rho)$ and $\gamma(\overline{\boldsymbol{A}}_{\mathrm{use}}(Q_2, \overline{\rho}')) \supseteq \boldsymbol{A}_{\mathrm{use}}(Q_2, \rho')$ respectively. This implies that $\gamma(\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho}) \sqcap \overline{\boldsymbol{A}}_{\mathrm{use}}(Q_2, \overline{\rho}')) \supseteq \boldsymbol{A}_{\mathrm{def}}(Q_1, \rho) \cap \boldsymbol{A}_{\mathrm{use}}(Q_2, \rho')$. Therefore, $\overline{\boldsymbol{A}}_{def}(Q_1, \overline{\rho}) \sqcap \overline{\boldsymbol{A}}_{use}(Q_2, \overline{\rho}') = \emptyset \Rightarrow X \cap Y = \emptyset$ where $X = \boldsymbol{A}_{\mathrm{def}}(Q_1, \rho) \in \gamma(\overline{\boldsymbol{A}}_{\mathrm{def}}(Q_1, \overline{\rho}))$ and $Y = \boldsymbol{A}_{\mathrm{use}}(Q_2, \rho') \in \gamma(\overline{\boldsymbol{A}}_{use}(Q_2, \overline{\rho}'))$.*

5.9 Extension of Dependency Analysis to HQL

As dependency analysis of HQL follows similar approach as in the case of SQL, let us now provide a brief description on how to achieve this in case of HQL. Recall the equations 4.1 and 4.2 in chapter 4 and the equation 5.7 in section 5.6.1. The equations 4.1 establishes the Galois Connection showing the abstraction of concrete interaction states. Based on this, the equation 4.2 defines a sound approximation of session methods semantics w.r.t. their concrete counterparts. Since an abstract interaction state

$\langle \bar{e}, \bar{s}, \rho_{\bar{d}}, \overline{\texttt{Esc}} \rangle$ involves an abstract database state $\rho_{\bar{d}}$, the abstract semantics of session methods can easily be defined in terms of three components by following equation 5.7.

5.10 Implementation and Experimental Evaluation

We have implemented a prototype tool `SemDDA`[3] – 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 [1][4, 5].

5.10.1 The SemDDA Tool

The aim of designing `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 5.7 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 `semDDA`.

(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

[3]The source code is available on github: `https://github.com/angshumanjana/SemDDA`.

[4]The original website `''http://www.gotocode.com''` does no longer exist at this moment. We have archived the benchmark codes at `''https://github.com/angshumanjana/GotoCode''`.

[5] These benchmark codes are used by many authors in their experiments, such as [16,56,57,106].

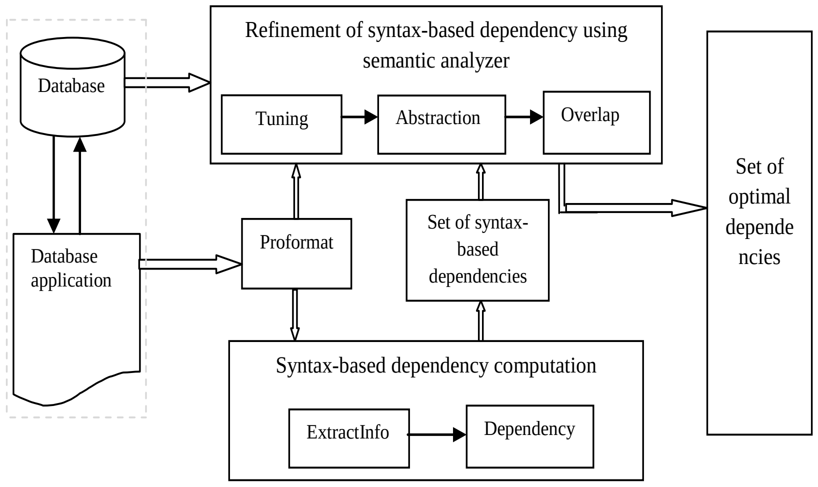

Refinement of syntax-based dependency using semantic analyzer
Database
Tuning
Abstraction
Overlap
Set of optimal dependencies
Database application
Proformat
Set of syntax-based dependencies
Syntax-based dependency computation
ExtractInfo
Dependency

Figure 5.7: Architecture of `SemDDA`

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.

Limitations of `SemDDA`: The current implementation of `SemDDA` consider only database code written in JSP embedding SQL. At this stage, the tool does not support the followings: (i) dynamically generated queries, (ii) HQL queries, (iii) nested queries and (iv) string data-type.

5.10.2 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 [1]. A brief description of these benchmark codes are mentioned in Table 5.4. 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.

5.10.2.1 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 5^{th} column of Table 5.5. 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.

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

We implemented Condition-Action rules using Satisfiability Modulo Theories (SMT). In particular, we used Z3 [39], 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

Applications Names	Number of Files Tested	Descriptions
Events	1	It is a basic online event management system. It includes many features like event information (event name, year, presenter, etc.), users administration, etc.
Ledger	1	It is an example implementation of a web-based ledger which allows a user to track bank deposits, withdrawals, commission and view current balance.
Portal	2	It is a fully functional online web-based Portal which is useful for small organizations, clubs, user groups, and schools. It provides several functionalities like user registration, news section, list of club officers and etc. The considering files mainly work on the administration of club officers and members.
EmplDir	2	It is a basic employee directory that may use as an online system for small companies. It serves deferent searching facilities (e.g. by name, email) to the user. The selected files are dealing to store the employee and departmental information.
Bookstore	2	It is an online store system that keeps various books information, articles and other items. It has many features like user registrations, shopping cart, administration of credit card types and etc. It utilizes VeriSign's payflow link system to verify and charge credit cards.
BugTrack	3	It is a basic fully functional web-based bug tracking system which may useful for small teams working on software projects. It keeps projects information and its associated employee's detail (consider files work for this purpose), also provides many searching options.

Table 5.4: Description of the benchmark programs [1]

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

Table 5.5: DD-dependency results in various approaches (NCLOC denotes Non-Comment Lines of Code)

(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 30 *Consider the following pair of database statements:*

$$Q_1 : \texttt{UPDATE}\ emp\ \texttt{SET}\ hra = hra + 100\ \texttt{WHERE}\ da + hra \geqslant 1000$$
$$Q_2 : \texttt{SELECT}\ hra\ \texttt{FROM}\ emp\ \texttt{WHERE}\ da + hra \leqslant 5000$$

The equivalent SSA form of these statements are:

$$Q_1 : \texttt{UPDATE}\ emp\ \texttt{SET}\ hra2 = hra1 + 100\ \ \texttt{WHERE}\ da1 + hra1 \geqslant 1000$$
$$Q_2 : \texttt{SELECT}\ hra2\ \texttt{FROM}\ emp\ \texttt{WHERE}\ da1 + hra2 \leqslant 5000$$

The VC of this pair of statements is:

$$V_c = (hra2 == hra1 + 100) \wedge (da1 + hra1 \geqslant 1000)\ \wedge (da1 + hra2 \leqslant 5000)$$

The encoded version of V_c in Z3 is:

1. (*declare* − *const hra*1 *Int*)
2. (*declare* − *const hra*2 *Int*)
3. (*declare* − *const da*1 *Int*)
4. (*push*)
5. (*assert* (= (+ *hra*1 100) *hra*2))
6. (*assert* (>= (+ *hra*1 *da*1) 1000))
7. (*assert* (<= (+ *hra*2 *da*1) 5000))

8. $(check - sat)$

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 6^{th} column of Table 5.5. 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.

5.10.2.3 Results based on the Abstract Semantics

Columns 7^{th}, 8^{th}, 9^{th} and 10^{th} of Table 5.5 depict DD-dependency results in the domains of intervals, octagons, polyhedra and powerset of intervals respectively. It is worthwhile to note that the analysis-results for five benchmark codes ('EditOfficer', 'EditMember', 'BookMaint', 'EditorialsRecord', 'BugRecord') in the interval domain improves w.r.t. their syntax-based results. On the other hand, analysis in the domain of octagons for 'EmployeeMaint', 'ProjectMaint', 'EventNew' and 'EmployeeMaint' results in more precise dependency information compared to that in the interval domain, due to the presence of restricted attributes relationship (which involves at most two attributes) in SQL statements. Similarly, polyhedra domain analysis captures more precise DD-dependency results, shown in the case of 'LedgerRecord' and 'EmpsRecord', compared to their interval and octagon counterparts, as they allow unrestricted relationship among attributes. We obtain an 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 5.8 compares all DD-dependency results.

Table 5.6 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

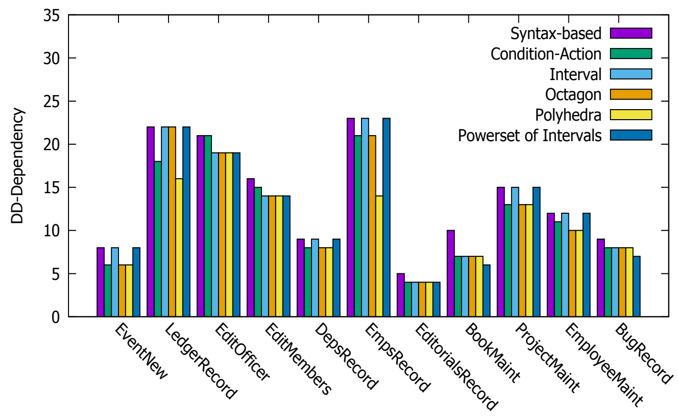


Figure 5.8: Comparative analysis of DD-dependency results in various approaches.

5.9. 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 5.5).

File Names(.jsp)	Abstract Domains			
	Interval	Octagon	Polyhedra	Powerset of Intervals
EventNew	167	194	345	171
LedgerRecord	204	312	7943737	211
EditOfficer	86	91	201	87
EditMembers	116	178	354	119
DepsRecord	96	110	163	98
EmpsRecord	192	254	366314	197
EditorialsRecord	76	103	160	77
BookMaint	110	162	432	113
ProjectMaint	80	81	189	81
EmployeeMaint	126	169	986	129
BugRecord	94	97	157	97

Table 5.6: Execution time (in milliseconds) in various Abstract Domains.

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.

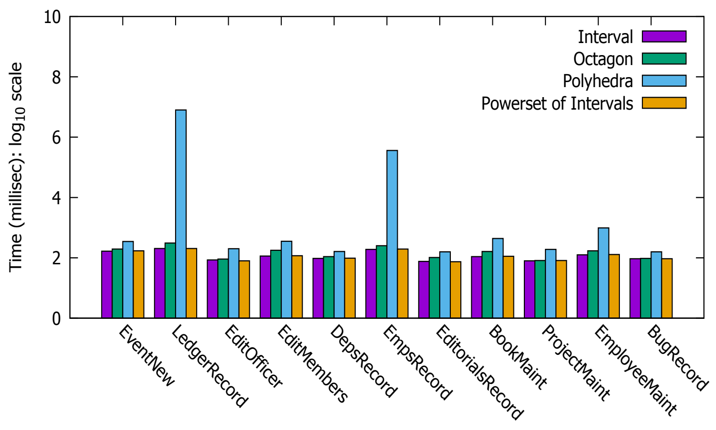


Figure 5.9: Analysis Time (in $\log_{10}$ scale) in various Abstract Domains

5.11 Conclusions

In this chapter, we propose a novel approach to compute semantics-based independencies among database program statements, based on the Abstract Interpretation theory. This steers construction of semantic-based dependency graphs with a 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.

CHAPTER 6

Policy-based Database Code Slicing

Preface

Program slicing is a static analysis technique which is widely used in various software engineering activities, e.g. debugging, testing, code-understanding, code-optimization, etc. It extracts from programs a subset of statements which is relevant to a given behavior. In this chapter, we introduce a new form of code slicing, known as policy-based slicing, of database applications based on the refined notion of dependency graph. We show how the use of semantics-based dependency, together with semantic relevancy of statements, may improve the precision of the slice w.r.t. a given policy.

6.1 Introduction

Program slicing [116] is a static analysis technique that extracts from programs the statements which are relevant to a given behavior. It allows engineers to address several software-related problems, including program understanding, debugging, maintenance, parallelization, integration, software measurement, etc. [15,53,62,78].

Works on program slicing have been starting with the pioneering work of Mark Weiser in the 1981 [120]. He presented in his seminal paper an iterative, static and backward slicing technique which is based on data flow analysis and on the influence of predicates on statement execution. The slice is computed as the set of all statements of the program that might affect directly or indirectly the value of the variable in the set V just before the execution of the statement p for all inputs. In contrast, in case of dynamic slicing [78], programmers are more interested in a slice that preserves the program's behavior for a specific program input rather than for all program inputs.

Over the past, various forms of slicing are introduced [71, 78, 120]. A forward slice [14] contains those statements of the program which are affected by the slicing criterion, whereas program chopping [71] is a kind of "filtered" slice that extracts all the program statements that serve to transmit effects from a given source element s to a given target element t. The authors in [88] defined the base vocabulary and slicing criteria for static [120], dynamic [78], quasi static [117], simultaneous dynamic [58], and conditioned slicing [18] using formal notation. They defined the conditioned slice as a general purpose slice, whereas other slice types are specializations of conditioned slicing.

The existing slicing techniques in the literature mostly use dependency graphs for the efficient computation of program slices. For instance, PDG, SDG, ClDG and DOPDG are well suited for intra-, inter-, object-oriented and database code slicing respectively.

As industries and organizations often introduce new policies or modify their existing policies, maintenance of associated large-scale complex database programs to reflect these changes becomes a tedious task. This forces software engineers to continuously examine codes in order to identify only the relevant part which should participates in this process. To accelerate this, in this chapter, we introduce a new form of program slicing, known as policy-based slicing, of database programs. We restrict ourselves to the policies defined only on the underlying database. For example, let us suppose that the company decides to introduce a new policy which respects the consistency of employees salary structure, defined below:

The salary of employees of age less or equal to 40 cannot have a salary greater than 75% of the maximum salary of the same category.

Formally, we can write it as:
$\forall t_e \in \texttt{temp}, \exists t_j \in \texttt{tjob}.\Big(t_e.tjid = t_j.tjobid \wedge t_e.tsal \leqslant \frac{(75 \times t_j.tmaxsl)}{100} \wedge t_e.tage \leqslant 40\Big)$ where `temp` and `tjob` represent the underlying database tables.

In particular, our main contributions in this chapter are:

- We propose the construction of syntax-based dependency graph for HQL, by extending its variant for object-oriented languages.
- We show how the use of semantics-based dependency, together with semantic relevancy of statements, may improve the precision of the slice w.r.t. a given policy.

As usual, the primary steps involved in the computation of slice w.r.t. a policy ψ are:

1. Construction of syntax-based dependency graph.
2. Refinement of the graph by computing semantics-based dependencies.
3. Computation of slice by traversing the graph either backward (in case of backward slicing) or forward (in case of forward slicing) direction from the node of reference w.r.t. all variables involved in the policy.

Let us consider two cases, one for SQL and another for HQL, to demonstrate these.

6.2 Slicing of SQL

Recall the database code `Prog` already depicted in Figure 5.1 of chapter 5. Let us assume that, according to the company policy ψ, the module increments '*point*' by 2 and 4 when ($5000 \leqslant purchase\_amt + wallet\_bal < 10000$) and ($purchase\_amt + wallet\_bal \geqslant 10000$) respectively. Let us suppose that the company decides to revise this existing policy ψ into a new one ψ_1 which is defined below:

For the customer of purchase amount and wallet balance summation is more than or equal to 10000 should have point incremented by 5.

In order to respect ψ_1, some specific statements in `Prog` need to be changed. Therefore, we need to perform a slicing of `Prog` w.r.t. ψ_1 to extract only those relevant part.
Let us follow the steps mentioned before.

1. Construction of syntax-based dependency graph.
We recall the construction of syntax-based DOPDG of `Prog` from chapter 5 and let us show the graph again below:

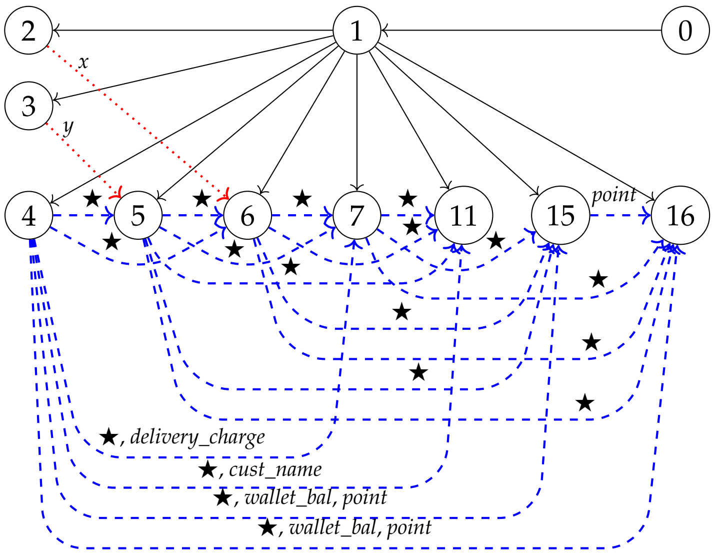


Figure 6.1: Syntax-based DOPDG (★ denotes attribute *purchase_amt*) of `Prog`

2. Refinement of the graph by computing semantics-based dependencies.
As already illustrated in chapter 5, after refining false dependencies $5 \dashrightarrow 6$, $4 \dashrightarrow 11$, $5 \dashrightarrow 11$, $6 \dashrightarrow 11$, $4 \dashrightarrow 15$, $5 \dashrightarrow 15$, $6 \dashrightarrow 15$, $4 \dashrightarrow 16$, $5 \dashrightarrow 16$, $6 \dashrightarrow 16$ and $15 \dashrightarrow 16$, we obtained a refined version of semantics-based DOPDG which is depicted in Figure 6.2.

3. Computation of backward slice w.r.t. ψ_1.
Since our objective is to extract all statements relevant to ψ_1, we consider the node 16 corresponding to the last statements of `Prog` as the point of reference and we take into consideration the attributes *point*, *purchase_amt*, *wallet_bal* involved in ψ_1. Therefore, the slicing criterion is denoted as $\langle 16, \{point, purchase\_amt, wallet\_bal\}\rangle$. The backward

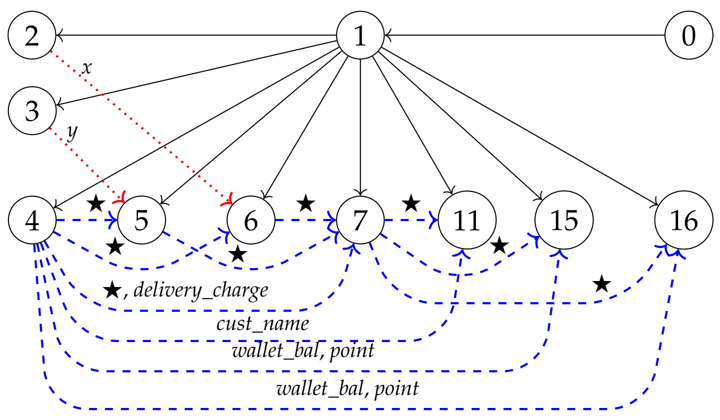


Figure 6.2: Refined semantics-based DOPDG (★ denotes attribute *purchase_amt*) of `Prog`

```
0. public class saleOffer {
1.    public static void main(String[] args) throws SQLException {
2.      float x = 0.1;
3.      float y = 0.05;
4.      try { Statement con = DriverManager.getConnection("jdbc mysql: . . . ", "scott", "tiger").createStatement();

5.            con
      .executeQuery("UPDATE Sales SET purchase_amt = purchase_amt − y ∗ purchase_amt WHERE purchase_amt BETWEEN 1000 AND 3000 ");

6.            con.executeQuery("UPDATE Sales SET purchase_amt = purchase_amt − x ∗ purchase_amt WHERE purchase_amt > 3000 ");

7.            con.executeQuery("UPDATE Sales SET purchase_amt = purchase_amt − delivery_charge ");

16.           con.executeUpdate("UPDATE Sales SET point = point + 4 WHERE (purchase_amt + wallet_bal) ⩾ 10000 "); } catch
      (Exception e) { . . . } }}
```

Figure 6.3: Slice of `Prog` w.r.t. ψ_1

slice after traversing the refined DOPDG w.r.t. the slicing criteria is shown in Figure 6.3.

Observe that the preciseness is improved over its syntactic slice after disregarding the statement 15.

6.3 Slicing of HQL

We observe that none of the existing syntax-based graph construction approaches are directly applicable to the case of HQL due to the correspondence between high-level application variables with the low-level database attributes through hibernate (`session`) interface. Moreover, we have to treat transient objects and persistent objects differently during the construction of dependency graph.

```
class Serv {
    public static void main(String arg[]){
        Configuration cfg =new Configuration();
        cfg.Configure("hibernate.cfg.xml");
        Session ses =(cfg.buildSessionFactory()).openSession();
        Transaction tr = ses.beginTransaction();
        int i = (new Scanner(System.in)).nextInt();
        if(i==1){
            int id_v = getparam(...);
            List ls = ( ses.createQuery("SELECT a.jname FROM emp e INNER JOIN e.Job a WHERE
              e.eid = :xid").setParameter("xid", id_v)).list();
            Object obj = (Object)ls;
            System.out.println((String)obj);}
        if(i==2){
            int id_v = getparam(...);
            int sal_v = getparam(...);
            int r1 = (ses.createQuery("UPDATE emp e SET e.sal= e.sal+:xsal WHERE e.sal⩾1500")
            .setParameter("xsal",sal_v)).executeUpdate();}
        if(i==3){
            int id_v = getparam(...);
            int r2 = (ses.createQuery("DELETE FROM emp e WHERE e.sal⩽1000"))
            .executeUpdate();}
        tr.commit();
        ses.close(); } }
```

(a) Class Serv

teid	*tjid*	*tsal*	*tage*
1	3	1200	35
2	2	600	28
3	4	1000	30
4	1	2500	45
5	1	1600	20

(b) Database dB: Table temp

tjobid	*tjname*	*tjcat*	*tmaxsl*	*tminsl*
1	Asst.Prof	A	3000	1000
2	HR	C	1000	500
3	Asso.Prof	A	3000	1000
4	Registrar	B	2000	800
5	Prof.	A	3000	1000

(c) Database dB: Table tjob

Figure 6.4: An example of HQL code and underlying database

Let us first demonstrate the construction of syntax-based dependency graph of HQL codes using a suitable example and then the computation of slice on its refined version w.r.t. a given policy.

Example 31 *Consider an enterprise information system depicted in Figure 6.4 where the HQL program* `Serv` *performs three different operations (select, update, delete) on employees information stored in the database dB (Tables 6.4(b) and 6.4(c)) based on the user choice. Observe that the fields eid, jid, sal, age of POJO class* `emp` *in* `Serv` *correspond to the attributes teid, tjid, tsal, tage of the database table* `temp` *respectively. Similarly the fields jobid, jname of POJO class* `Job` *in* `Serv` *correspond to the attributes tjobid, tjname of the database table* `tjob` *respectively. This mapping is defined in a mapping file of hibernate framework. Let us consider the policy ψ_2 below:*

The salary of employees of age less or equal to 40 cannot have a salary greater than 75% of the maximum salary of the same category.

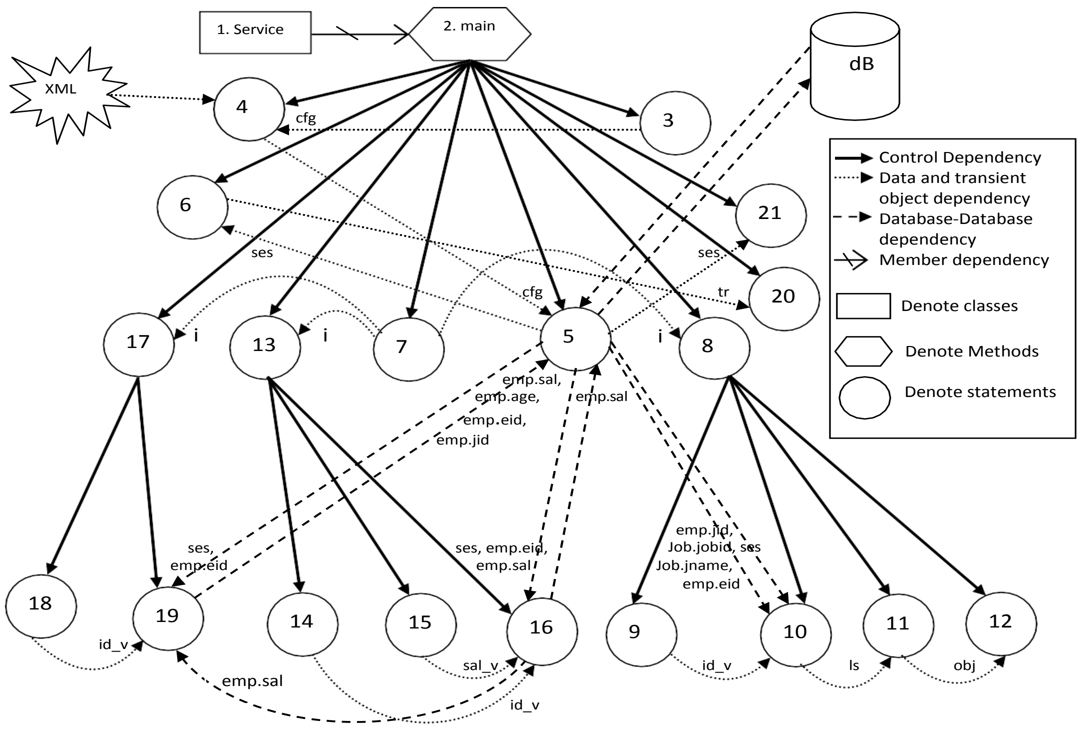


Figure 6.5: Syntax-based dependency graph of Serv

We now follow the same steps, mentioned before, to compute slice of Serv w.r.t. ψ_2:

1. Construction of syntax-based dependency graph.

The syntax-based dependency of Serv is depicted in Figure 6.5. We consider the following three types of dependencies in HQL programs:

(a) Intra-class Intra-method Dependencies: These represent the dependencies within the same method of a class, and it follows the Program Dependency Graph-based approach [100]. We denote these dependencies by dotted edges. Edges 7 - 13, 7 - 17, 7 - 8, 18 - 19, etc., are of this type.

(b) Intra-class Inter-method Dependencies: These represent the dependencies between statements of two different methods within the same class and are constructed by following System Dependency Graph-based approach [64]. We denote these dependencies by long-dash-dotted edges. There is no edge of this type in our example.

(c) Inter-class Inter-method Dependencies:

Through transient objects. Inter-class Inter-method dependencies occur in OOP when a method in one class calls another method in other class. This is done by calling the method through an object of the called-class. Therefore, additional in-parameters corresponding to the object-fields through which the method is called, must be considered [83]. Note that, in this scenario, a constructor-call during object creation is also a part of the graph which follows the same representation as of other inter-class inter-method calls. We denote these dependencies by long-dash-dotted edges (as in the case of Intra-class Inter-method Dependencies). Edges 3 - 4, 5 - 21, 6 - 20, 5 - 6, 4 - 5, etc., are of this type. Observe that node 4 calls "`configure()`" method on the object "cfg" which is received from node 3. It configures the "cfg" object using "XML" file and acts as a source for newly-configured "cfg" object. For the sake of simplicity, we do not include here the details of the calling scenario by node 4. Similarly, we hide the details of the calling-scenarios by the nodes 5 (which creates the session object by calling `openSession()`), 6 (which creates the transaction object by calling `beginTransaction()`), 20 (when calling `commit()`) and 21 (when calling `close()`) respectively.

Through session objects. Various `Session` methods are used to convert objects from transient state to persistent state and to perform various operations, like select, update, delete on the persistent objects in the database. In other words, Hibernate `Session` serves as an intermediate way for the interaction between high-level HQL variables and the database attributes. As creation of a `Session` object implicitly establishes connection with the database, we consider the nodes which create `Session` objects as the sources of the database (hence database-attributes). For instance, the node 5 acts as a source of `dB`. When `Session` methods (`save(), creatQuery()`) are called through `Session` objects, either a transient object (in case of `save()`) or an object-oriented variant of SQL statement (in case of `createQuery()`) are passed as a parameter. For instance, see the nodes 10, 16, 19. The presence of HQL variables in the parameter which have a mapping with the database attributes leads to a number of dependencies shown by dash-lines between 5 - 10, 5 - 16 and 5 - 19. We call such dependencies as session-database dependencies. These edges are labeled with used- and defined- HQL variables present in the parameter which have a correspondence with database attributes. For

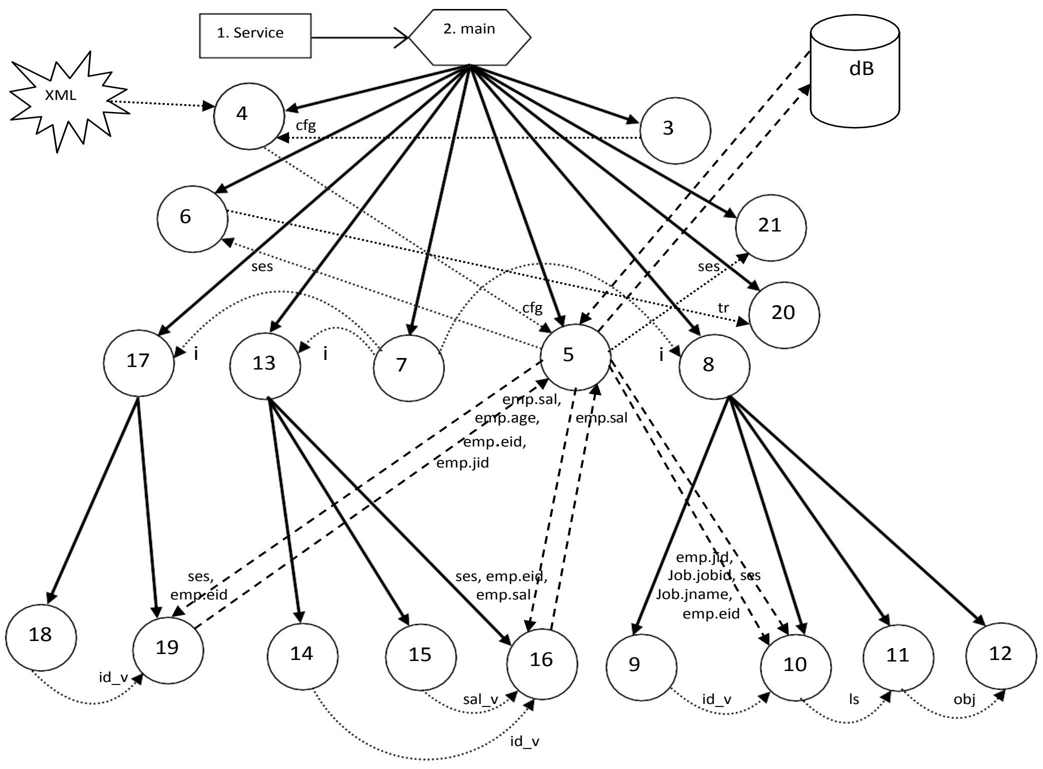


Figure 6.6: Refined semantics-based dependency graph of `Serv`

instance, in case of `obj` passed to `save()`, all database attributes corresponding to object fields act as defined variables, whereas in case of update and delete as parameters in the `createQuery()`, the variables in the `WHERE` clause act as used variables and the variables in the action part act as defined variables. For `SELECT`, all variables in the parameter act as used variables.

Observe that for the sake of simplified representation, we hide the detail calling scenario of `createQuery()` by nodes 10, 16, 19. The edges connecting the node 5 and dB indicates propagation and synchronization of memory and database states.

2. Refinement of the graph by computing semantics-based dependencies.
The semantics-based dependency analysis in the domain of polyhedra detects a false dependency 16 → 19. The refined form of the graph is depicted in Figure 6.6.

3. Computation of backward slice w.r.t. ψ_2.
We consider the node 21 corresponding to the last statements of `Serv` and the attributes *tsal*, *tage*, *tmaxsl* involved in ψ_2. The backward slice w.r.t. the slicing criterion

$\langle 21, \{tsal, tage, tmaxsl\}\rangle$ is shown in Figure 6.7.

Observe that the slice is more precise w.r.t. its syntactic varsion, as it does not contain the statement 10.

```
1.    class service {
2.        public static void main(String arg[]){
3.            Configuration cfg =new Configuration();
4.            cfg.Configuration("hibernate.cfg.xml");
5.            Session ses =(cfg.buildSessionFactory()).openSession();
7.            int i = (new Scanner(System.in)).nextInt();
13.           if(i==2){
14.               int id_v = getparam(...);
15.               int sal_v = getparam(...);
16.               int r1 = (ses.createQuery("UPDATE emp e SET e.sal=
                  e.sal+:xsal WHERE e.sal⩾1500").setParameter("xsal",sal_v))
                  .executeUpdate();}
17.           if(i==3){
18.               int id_v = getparam(...);
19.               int r2 = (ses.createQuery("DELETE FROM emp e WHERE
                  e.sal⩽1000")).executeUpdate();}
21.           ses.close(); } }
```

Figure 6.7: Slice of `Serv` w.r.t ψ_2

6.4 Precision Improvement using Semantic Relevancy of Database Statements w.r.t. Policy

Let us recall the formal definition of semantic relevancy from [53].

Definition 6.1 (Semantic Relevancy [53]) *Let Σ_ℓ be the set of states possibly occurring at program point ℓ. A statement c at program point ℓ is semantically irrelevant w.r.t. a concrete property ω if $\forall \rho \in \Sigma_\ell.\ \omega(S[\![c]\!]\rho) = \omega(\rho)$, where $S[\![.]\!]$ is the semantic function.*

Because of the computational complexity (in case of large database) and undecidability in the concrete domain, by lifting the semantics to an abstract domain $\overline{\mathbb{D}}$, we can define the semantic relevancy w.r.t. an abstract property $\overline{\omega}$ in $\overline{\mathbb{D}}$ as follows: $\forall \overline{\rho} \in \overline{\Sigma_\ell}.\ \overline{\omega}(\overline{S}[\![c]\!]\overline{\rho}) = \overline{\omega}(\overline{\rho})$. The semantic irrelevancy in an abstract domain is sound by its constructions [30, 53].

Consider the following database statement and the underlying database table `Tab` in Figure 6.8(a).

$$Q : \texttt{UPDATE Tab SET } age = age\texttt{+1 WHERE } age \leqslant\texttt{60}$$

teid	*tsal*	*tage*
1	1200	55
2	1600	62
3	2000	45
4	800	18

(a) Table `Tab`

teid	*tsal*	*tage*
1	1200	56
2	1600	62
3	2000	46
4	800	19

(b) Result after execution of Q on `Tab`

$\overline{teid}$	$\overline{tsal}$	$\overline{tage}$
$[1,4]$	$[800, 2000]$	$[18,62]$

(c) Abstract table $\overline{\texttt{Tab}}$ corresponding to `Tab`

$\overline{teid}$	$\overline{tsal}$	$\overline{tage}$
$[1,4]$	$[800, 2000]$	$[19,62]$

(d) Result after execution of $\overline{Q}$ on $\overline{\texttt{Tab}}$

Figure 6.8: Concrete and Abstract Query Semantics.

Suppose the company policy ψ_3 (defined on `Tab`) says that employee's ages must belong to the range 18 and 62 (i.e. $18 \leqslant tage \leqslant 62$). We denote by ρ_D the state of the database D which includes the state of `Tab`. The semanticsof Q in σ_D, i.e. $S[\![Q]\!]\sigma_D$ yields the result shown in Figure 6.8(b). We observe that the policy ψ_3 is satisfied before and after the execution of Q, i.e. $\psi_3(\rho_D) = \psi_3(S[\![Q]\!]\rho_D)$. Therefore, Q is *irrelevant* w.r.t. ψ_3, assuming ρ_D is the only state that occurs at the program point of Q.

Although this example is trivial to compute the irrelevancy of Q in concrete domain, in case of very large database (or even when database state depends on run-time inputs) the irrelevancy can be computed in an abstract domain of interest. For instance, consider the abstract domain of intervals. The abstract table $\overline{\texttt{Tab}}$ corresponding to `Tab` in the abstract domain is shown in Table 6.8(c).

The corresponding abstract state which include $\overline{\texttt{Tab}}$ is denote by $\overline{\rho}_D$. The abstract semantics $\overline{S}[\![\overline{Q}]\!]\overline{\rho}_D$ where $\overline{Q}$ is

$$Q : \texttt{UPDATE Tab SET } age = age + [1,1] \texttt{ WHERE } age \leqslant [60,60]$$

yields the abstract result depicted in Figure 6.8(d). We observe that $\psi_3(\overline{\rho}_D) = \psi_3(\overline{S}[\![\overline{Q}]\!]\overline{\rho}_D)$. By following [53], we can prove the soundness, i.e.

$$(\psi_3(\overline{\rho}_D) = \psi_3(\overline{S}[\![\overline{Q}]\!]\overline{\rho}_D)) \implies \forall Q \in \gamma(\overline{Q}), \forall \rho_D \in \gamma(\overline{\rho}_D) : \ \psi_3(\rho_D) = \psi_3(S[\![Q]\!]\rho_D)$$

where γ is a concretization function [30].

Let us illustrate the precision improvement of the slice of HQL code serv (in Figure 6.4) w.r.t. ψ_2 using the notion of semantic relevancy in the domain of polyhedra.

Example 32 *The initial polyhedron*[1] *corresponding to the database* `dB` *(Figure 6.4(b)) is* $\mathtt{P}_{dB} = \langle\{teid \geqslant 1, -teid \geqslant -5, tsal \geqslant 600, -tsal \geqslant -2500, tage \geqslant 20, -tage \geqslant -45\}, 3\rangle$. *The pictorial representation is shown in Figure 6.9(a).*

The abstract syntax of `Session` *methods* m_{sel}, m_{upd} *and* m_{del} *at program points 10, 16, 19 respectively in* `Serv` *are:*

$$m_{sel} ::= \langle \mathtt{C}, \phi, \mathtt{OP}\rangle \text{ where } \mathtt{C} = \{\mathtt{emp}, \mathtt{Job}\},\ \phi = \{eid = jobid,\ eid = id\_v\},$$

$$\mathtt{OP} = \mathtt{SEL}(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi',\ g(\vec{exp}))\rangle\ = \mathtt{SEL}(id, \mathtt{ALL}(id(jname)), true, id)\rangle,$$

and id denotes identity function.

$$m_{upd} ::= \langle\{\mathtt{emp}\}, \{e.sal \geqslant 1500\}, \mathtt{UPD}(\langle sal\rangle, \langle sal + sal\_v\rangle)\rangle\rangle$$

$$m_{del} ::= \langle\{\mathtt{emp}\}, \{e.sal \leqslant 1000\}, \mathtt{DEL}()\rangle$$

The transition semantics on $\mathtt{P}_{dB}$ *are:*

$$\begin{aligned}
&\mathscr{T}_{hql}[\![m_{sel}]\!]\mathtt{P}_{dB}\\
&=\mathscr{T}_{hql}[\![\langle\{\mathtt{emp},\mathtt{Job}\}, \{eid = jobid, eid = id\_v\}, \mathtt{SEL}(id, \mathtt{ALL}(id(jname)), true, id)\rangle]\!]\mathtt{P}_{dB}\\
&=\mathscr{T}_{sql}[\![\langle\{\mathtt{temp},\mathtt{tjob}\}, \{teid = tjobid, teid = tid\_v\}, \mathtt{SELECT}(id, \mathtt{ALL}(id(tjname)), true, id)\rangle]\!]\mathtt{P}_{dB}\\
&=\{\mathtt{P}_{dB}\}
\end{aligned}$$

Note that, select operation does not change the database, hence the polyhedron remains unchanged (see Figure 6.9(b)).

$$\begin{aligned}
\mathscr{T}_{hql}[\![m_{upd}]\!] &= \mathscr{T}_{hql}[\![\langle\{\mathtt{emp}\}, \{e.sal \geqslant 1500\}, \mathtt{UPD}(\langle sal\rangle, \langle sal + sal\_v\rangle)\rangle\rangle]\!]\mathtt{P}_{dB}\\
&= \mathscr{T}_{sql}[\![\langle\{\mathtt{temp}\}, \{tsal \geqslant 1500\}, \mathtt{UPDATE}(\langle tsal\rangle, \langle tsal + sal\_v\rangle)\rangle\rangle]\!]\mathtt{P}_{dB}\\
&= \{\mathtt{P}'_T, \mathtt{P}_F\} \quad \text{where}
\end{aligned}$$

$$\begin{aligned}
\mathtt{P}_T &= \mathtt{P}_{dB} \sqcap \{tsal \geqslant 1500\}\\
\mathtt{P}'_T &= \mathscr{T}_{sql}[\![tsal := tsal + sal\_v]\!](\mathtt{P}_T)\\
&= \langle\{teid \geqslant 1,\ -teid \geqslant -5,\ tage \geqslant 20,\ -tage \geqslant -45,\ tsal \geqslant 600\},\ 3\rangle.\\
\mathtt{P}_F &= \{\mathtt{P}_{dB} \sqcap \neg(tsal \geqslant 1500)\}
\end{aligned}$$

[1]For the sake of simplicity, we consider the polyhedron in the space involving only three attributes *teid*, *tsal* and *tage*.

$$= \langle\{teid \geqslant 1, -teid \geqslant -5, tsal \geqslant 600, -tsal \geqslant -1499, tage \geqslant 20, -tage \geqslant -45\}, 3\rangle$$

Note that, since the updation of tsal depends on run-time input, we project-out the upper bound of the attribute from the set of restraints in P'_T in order to guarantee the soundness. The polyhedron representation of P'_T is shown in Figure 6.9(c).

$$\mathscr{T}_{hql}[\![m_{del}]\!] = \mathscr{T}_{hql}[\![\langle\{\texttt{emp}\}, \{e.sal \leqslant 1000\}, \mathit{DEL}()]\!]\mathsf{P}_{dB}$$
$$= \mathscr{T}_{sql}[\![\langle\{\texttt{temp}\}, \{tsal \leqslant 1000\}, \mathit{DELETE}()]\!]\mathsf{P}_{dB} = \{\mathsf{P}_T, \mathsf{P}_F\} \quad where$$

$$\mathsf{P}_T = \mathsf{P}_{dB} \sqcap \{tsal \leqslant 1000\}$$
$$= \langle\{teid \geqslant 1, -teid \geqslant -5, tage \geqslant 20, -tage \geqslant -45, tsal \geqslant 600, -tsal \geqslant -1000\}, 3\rangle.$$
$$\mathsf{P}_F = \{\mathsf{P}_{dB} \sqcap \neg(tsal \leqslant 1000)\}$$
$$= \langle\{teid \geqslant 1, -teid \geqslant -5, tsal \geqslant 1001, -tsal \geqslant -2500, tage \geqslant 20, -tage \geqslant -45\}, 3\rangle$$

Observe that the initial polyhedron of dB is covered by the polyhedron representing ψ_2. This can

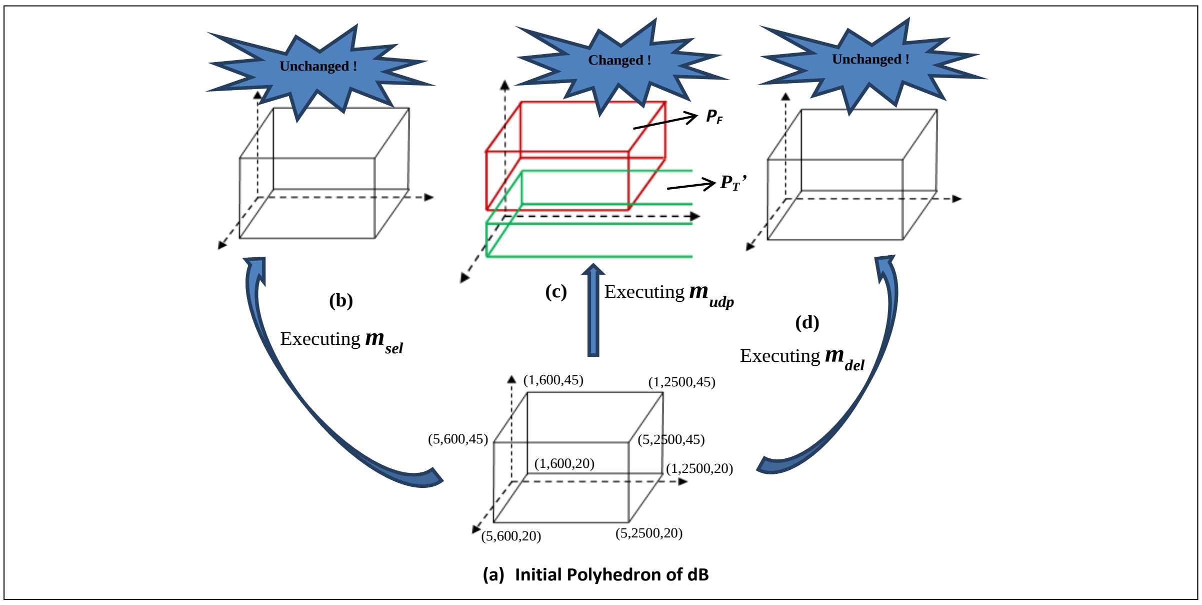


Figure 6.9: Polyhedra representation of `temp` on various operation (m_{sel},m_{udp},m_{del}).

verified by performing the operations, e.g. interaction, emptiness checking, etc. on the polyhedra domain. Therefore, initially ψ_2 is satisfied by dB. The abstract semantics-based analysis proves that m_{del} does not change the initial polyhedron P_{dB} (see Figure 6.9). Therefore it is semantically irrelevant w.r.t. ψ_2. Therefore, the semantics-based slice, disregarding this irrelevant statement,

```
 1.   class service {
 2.       public static void main(String arg[]){
 3.           Configuration cfg =new Configuration();
 4.           cfg.Configuration("hibernate.cfg.xml");
 5.           Session ses =(cfg.buildSessionFactory()).openSession();
 7.           int i = (new Scanner(System.in)).nextInt();
13.           if(i==2){
14.               int id_v = getparam(...);
15.               int sal_v = getparam(...);
16.               int r1 = (ses.createQuery("UPDATE emp e SET e.sal=e.sal+:xsal WHERE e.sal≥1500")
                  .setParameter("xsal",sal_v)).executeUpdate();}
21.           ses.close(); } }
```

Figure 6.10: Slice of Serv considering semantics relevancy w.r.t. ψ_2

is shown in Figure 6.10. This indeed improves the precision of the slice.

6.5 Discussions and Conclusions

Policy-based slicing is comparable with conditioned slicing and specification-based slicing (several variants exist, *e.g.* precondition-, postcondition-, contract-, assertion-based, etc.) [19]. The method defined for finding a conditioned slice is to use symbolic execution and reject infeasible paths based on the constraints defined by the first order logic equations. Specification-based slicing approach is proposed based on the axiomatic semantics (weakest precondition or strongest postcondition computations) in program verification. All these approaches use SMT solver which has exponential complexity [77]. Our proposal is based on the semantic analysis in various non-relational and relational abstract domains in the Abstract Interpretation framework. We experience that semantics-based dependency, together with semantic relevancy of statements triggers a generation of more precise slices w.r.t. their syntax-based results.

CHAPTER 7

Data Leakage Analysis of Database Applications

Preface

Language-based information-flow security analysis has emerged as a promising technique to detect possible information leakage in any software systems. Confidential data stored in an underlying database may be leaked to an unauthorized user due to improper coding of database applications. In this chapter, we extend the full power of the proposed model in [55] to the case of HQL, particularly by focussing on the `session` methods. We define the abstract semantics of HQL over the domain of propositional formulae by considering variables dependencies at each program point. This allows us to identify illegitimate information flow by checking the satisfiability of propositional formulae with respect to a truth value assignment based on their security levels. Finally we explain how the reduced product of the analysis-results obtained from symbolic propositional formulae domain and numerical abstract domain may further improve the precision.

7.1 Introduction

Information is considered as most valuable assets for any organization or enterprise. Sensitive data may be leaked maliciously or even accidentally through a bug in the program. For example, any health information processing system may release patients' data, or any online transaction system may release customers' credit card information through covert channels, while processing. Confidentiality of sensitive information, that refers to preventing information from being leaked to unauthorized users, is one of the prime factors that needs to be maintained by information systems. Unauthorized disclosure of sensitive information may put the whole system into risks. Therefore, protecting private data in computer systems is a promising field of research. While access control and encryption prevent confidential information from being read or modified by unauthorized users at source level, they do not regulate the information propagation after it has been released for execution. Confidentiality may be compromised during the flow of information along the control structure of any software systems [108]. Assuming variables 'h' and 'l' are private and public respectively, the following code fragments depict two different scenarios (explicit/direct flow and implicit/indirect flow) of information leakage:

```
l := h                                Explicit/Direct flow
if(h=0) l:=5; else l:=10;             Implicit/Indirect flow
```

Observe that confidential value in 'h' can be deduced by attackers observing 'l' on the output channel.

Secure information flow is comprised of two related aspects: information confidentiality and information integrity. Confidentiality refers to limiting the access and disclosure of sensitive information to authorized users only. For instance, when we purchase something online, our private data, e.g. credit card number, must be sent only to the merchant without disclosing to any third person during the transmission. Dually, the notion of integrity indicates that data or messages cannot be modified undetectably by any unauthorized person [122].

A wide range of language-based techniques are proposed in the past decades to analyze this illegitimate flow in software products [51, 61, 85, 103, 108, 113]. Works in this direction have been starting with the pioneering work of Dennings in the 1970s [40]. As a starting point, the analysis classifies the program variables into various security classes. The simplest one is to consider two: Public/Low (denoted L) and Private/High (denoted H). Considering a mathematical lattice-model of security classes with order $L \leq H$, the secure information flow policy is defined on the lattice: an upward-flow in the lattice is only permissible to preserve confidentiality. Dually, in case of integrity, the lattice-model labels the variables as Tainted (denoted T) and Untainted (denoted U), and follows a dual flow-policy.

The correctness is guaranteed by respecting the non-interference principle [85] that says "a variation of confidential data does not cause any variation to public data": *Given a program $\mathcal{P}$ and set of states Σ. The non-interference policy states that $\forall \sigma_1, \sigma_2 \in \Sigma$. $\sigma_1 \equiv_L \sigma_2 \implies [\![\mathcal{P}]\!]\sigma_1 \equiv_L [\![\mathcal{P}]\!]\sigma_2$, where $[\![.]\!]$ is semantic function and $\equiv_L$ represents low-equivalence relation between states.*

In practice, non-interference principle is restrictive and deemed to be impractical. For instance, in password validation program all stored sensitive passwords are compared with user-given text and a boolean value is transmitted on the public channel as the result. To allow such intensional leakage, the notion of declassification [109] is introduced, where controlled information release is permitted.

Most of the notable works are proposed for imperative, object-oriented, functional, etc. [61, 103, 108]. In [55], authors proposed a framework to identify possible leakage of database applications that covers only SQL statements. In this chapter, we extend the full power of the proposed model in [55] to the case of HQL, focussing on Session methods which act as persistent manager. This allows us to perform leakage analysis of sensitive database information when is accessed through high-level HQL code. In particular,

- We define the abstract transition semantics of HQL over the domain of propositional formulae, by considering variables' dependencies at each program point.
- We use a truth assignment function that assigns each of the database and application variables a truth value based on its sensitivity level, and checks the satisfiability of propositional formula in order to identify any possible information leakage.
- We show how the use of abstract semantics at various levels of abstraction of numerical domain can improve the analysis precision when combined with the results in the propositional formula domain. To this aim, we define a reduce

product operation among the results obtained in various abstract domains, in order to exclude pointless dependency for the variables which have same value during the execution.

7.2 Related Works

A comprehensive survey on language-based information-flow analysis is reported in [108]. Most popular static analysis techniques are based on type systems [108,113,118], dependency graphs [20,60,85], formal approaches [4,41,74,122,123], etc. Besides the conservative nature of static analysis, the run-time monitoring systems detect unauthorized information flow dynamically; however, precision of the analysis completely depends on the execution overload, and of course, it is very prone to false negative [9,112].

The security type system considers various security types (*e.g.*, *low* and *high*) and a collection of typing rules which determine the type of expressions/commands to guarantee a secure information flow [108,113,118]. Some of the typing rules from [108] are mentioned below:

- Expression Type: $\frac{}{\vdash exp\colon high}$ $\quad$ $\frac{h \notin \mathtt{Var}(exp)}{\vdash exp\colon low}$
- Explicit-flow Rules: $\frac{}{[pc] \vdash h := exp}$ $\quad$ $\frac{\vdash exp\colon low}{[low] \vdash l := exp}$
- Implicit-flow Rules: $\frac{\vdash exp\colon pc \quad [pc] \vdash c_1 \quad [pc] \vdash c_2}{[pc] \vdash if\ exp\ then\ c_1\ else\ c_2}$ $\quad$ $\frac{\vdash exp\colon pc \quad [pc] \vdash c}{[pc] \vdash while\ exp\ do\ c}$
- Subsumption Rule: $\frac{[high] \vdash c}{[low] \vdash c}$

The notation [*pc*] denotes the security context which can be either [*low*] or [*high*]. According to the subsumption rule, if a program is typable in a high context then it is also typable in a low context. This allows to reset the program security context to low after a high conditional or a loop.

Although type-based approach is provably sound, but a major drawback is the lack of expressiveness. Moreover, it is not flow-sensitive which may produce false alarm. For instance, consider the following code:

```
1 if(h=1) then
2      l:= 10;
3 else l:= 5;
    ......
7 l:= 0;
8 output l;
```

Although the program is secure with respect to the classical noninterference principle as the output is always zero, but the type-based approach produces false alarm according to the implicit-flow rule.

As information flow is closely related to the dependency information of programs, the notion of Program Dependency Graph (PDG) is used widely to capture illegitimate flow in programs [20, 60, 61, 85]. For instance, in PDG-based approaches, the above code is secure as there is no path $1 \overset{*}{\rightarrow} 8$ in the corresponding PDG. Various extensions of PDG exist, for example System Dependency Graph (SDG) in case of inter-procedural call to capture context-sensitivity, Class Dependency Graph (ClDG) in case of Object-Oriented Languages to capture object-sensitivity on dynamic dispatch, etc [60]. Once the dependency graph of a program is constructed, static analysis is performed on the graph to identify the presence of possible insecure flow. An worth mentioning approach is backward slicing which collects all possible paths (or source-nodes) influencing (directly/indirectly) the observable nodes: to be secure, the levels of variables in a path must not exceed the levels of observable variables in the output-node of that path. In other words, slicing helps to partition any insecure program (as a whole) in to secure and insecure part [20]. Semantics-based improvement (*e.g*. path-conditions) is also proposed to disregard semantically unreachable paths [60].

Approaches based on formal techniques, *e.g*. Abstract Interpretation theory, Hoare Logic, Model Checking, etc. are proposed in [4,41,74,122,123] to analyze secure information flow in software products. Leino and Joshi [74] first introduced a semantics-based approach to analyzing secure information flow based on the semantic equivalence of programs. [122,123] defined the concrete semantics of programs and lift it to an abstract domain suitable for flow analysis. In particular, they consider the domain of propositional formula representing variables' dependency. The abstract semantics is further refined by combining with numerical abstract domain which improves the precision of the analysis. A variety of logical forms are proposed to characterize information flow security. Amtoft and Banerjee [4] defined prelude semantics by treating program commands as prelude transformer. They introduced a logic based on the Abstract Interpretation of prelude semantics that makes independency between program variables explicit. They used Hoare logic and applied this logic to forward program slicing: forward *l*-slice is independent of *h* variables and is secure from information leakage. Authors in [5] defines a set of proof rules for secure information flow based on axiomatic approach. Recently, [41] proposed a model checking-based approach for reactive systems. The authors in [110] proposed a framework for information flow control in a functional language with language-integrated queries (with Microsoft's LINQ on the backend). They developed a security type system with a treatment of algebraic data types and pattern matching by reusing the existing type systems. A major drawback of this approach is the flow-insensitivity which may produce false alarm.

7.3 Information Leakage Analysis of SQL

In this section, we recall from [55] the abstract transition semantics of SQL statements over the domain of propositional formulae and assignment of truth values to each variable considering its sensitivity to detect possible information leakage.

7.3.1 Abstract Semantics of SQL

In [55] authors used the Abstract Interpretation theory to define an abstract semantics of SQL embedded program using symbolic domain of positive propositional formulae in the form

$$\bigwedge_{0\leq i\leq n,\ \ 0\leq j\leq m} \{y_i \rightarrow z_j\}$$

which means that the values of variable z_j possibly depend on the values of variable y_i. The computation of abstract semantics of a program in the propositional formulae domain provides a sound approximation of variables dependency, which allows to capture possible information flow in the program. The information leakage analysis is then performed by checking the satisfiability of formulae after assigning truth values to variables based on their sensitivity levels.

Let Pos and $\mathbb{L}$ be the domain of propositional formulae and the set of program points respectively. Let $in : \mathbb{C} \longmapsto \mathbb{L}$ and $fin : \mathbb{C} \longmapsto \mathbb{L}$ be two functions where $in[\![c]\!]$ and $fin[\![c]\!]$ denote the sets of initial and final labels of statement $c \in \mathbb{C}$ respectively. The set of variables appearing in a statement is determined by the function $\mathscr{V} : \mathbb{C} \longmapsto \mathbb{V}$, where $\mathbb{V} = \mathbb{V}_a \cup \mathbb{V}_d$.

An abstract state $\overline{\rho} \in \overline{\Sigma} \equiv \mathbb{L} \times \mathtt{Pos}$ is a pair $\langle \ell, \psi \rangle$ where $\psi \in \mathtt{Pos}$ represents the variables dependency, in the form of propositional formulae, among program variables up to the program label $\ell \in \mathbb{L}$.

The abstract labeled transition semantics $\overline{\mathscr{T}}[\![c]\!]$ of a statement c is a set of transitions between abstract states $\overline{\rho}_1$ and $\overline{\rho}_2$, denoted by $\overline{\rho}_1 \xrightarrow{c} \overline{\rho}_2$. Table 7.1 depicts abstract labeled transition semantics of various statements in database applications, where the function $\mathtt{BV}(c)$ denotes the "defined variables" in statement c. Let $\mathtt{SF}(\psi)$ denotes the set of subformulas in ψ, and the operator $\ominus$ is defined by $\psi_1 \ominus \psi_2 = \bigwedge\big(\mathtt{SF}(\psi_1) \backslash \mathtt{SF}(\psi_2)\big)$.

7.3.2 Assigning Truth Values

Let $\Gamma : \mathbb{V} \rightarrow \{L, H\}$ be a function that assigns to each of the database and application variables in a program $\mathcal{P}$ a security class, either public (L) or private (H). We say that program $\mathcal{P}$ respects the confidentiality property, if and only if there is no information flow from private to public variables. To verify this property, a corresponding truth assignment function $\hat{\Gamma}$ is used:

$$\hat{\Gamma}(x) = \begin{cases} T & \text{if } \Gamma(x) = H \\ F & \text{if } \Gamma(x) = L \end{cases}$$

If $\hat{\Gamma}$ does not satisfy any propositional formula in ψ of an abstract state, the analysis will report a possible information leakage.

7.4 Information Leakage Analysis of HQL

The proposed abstract transition semantics of SQL can not be applied directly to the case of HQL due to the presence and interaction of high-level HQL variables and database attributes through `Session` methods. The new challenge in this scenario *w.r.t.* state-of-the-art of information leakage detection is that we need to consider both application variables and HQL variables (corresponding to the database attributes). Moreover, as

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket \text{SELECT} \rrbracket\\
&\overset{def}{=} \quad \overline{\mathscr{T}}\llbracket \langle {}^{\ell_5}assign(v_a),\ {}^{\ell_4}f(\vec{e'}),\ {}^{\ell_3}r(\vec{h}(\vec{x})),\ {}^{\ell_2}\phi',\ {}^{\ell_1}g(\vec{e}),\ {}^{\ell_0}\phi \rangle \rrbracket\\
&\overset{def}{=} \quad \{\langle \ell_0, \psi\rangle \xrightarrow{\text{SELECT}} \langle fin\llbracket \text{SELECT} \rrbracket, \psi'\rangle\}\\
&\text{where } \psi' = \bigwedge \left\{ y \to v_a.w_i \mid y \in (\mathscr{V}\llbracket \phi \rrbracket \cup \mathscr{V}\llbracket \vec{e} \rrbracket \cup \mathscr{V}\llbracket \phi' \rrbracket \cup \mathscr{V}\llbracket \vec{e'} \rrbracket) \wedge v_a.w_i \in v_a.\vec{w} \wedge y \neq v_a.w_i \right\} \wedge\\
&\bigwedge \left\{ z_i \to v_a.w_i \mid z_i \in \mathscr{V}\llbracket x_i \rrbracket \wedge\ x_i \in \vec{x} \wedge v_a.w_i \in v_a.\vec{w} \right\} \wedge \left(\psi \ominus \bigwedge \{ u \to v_a.w_i \mid u \in \mathbb{V} \wedge v_a.w_i \in v_a.\vec{w} \} \right)
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket \text{UPDATE} \rrbracket\\
&\overset{def}{=} \quad \overline{\mathscr{T}}\llbracket \langle {}^{\ell'}\vec{v_d} \overset{upd}{=} \vec{e},\ {}^{\ell}\phi \rangle \rrbracket\\
&\overset{def}{=} \quad \{\langle \ell, \psi\rangle \xrightarrow{\text{UPDATE}} \langle fin\llbracket \text{UPDATE} \rrbracket, \psi'\rangle\}\\
&\qquad\qquad\qquad \text{where } \psi' = \bigwedge \left\{ y \to z \mid y \in \mathscr{V}\llbracket \phi \rrbracket \wedge z \in \vec{v_d} \right\} \wedge\\
&\qquad\qquad\qquad \bigwedge \left\{ y_i \to z_i \mid y_i \in \mathscr{V}\llbracket e_i \rrbracket \wedge\ e_i \in \vec{e} \wedge z_i \in \vec{v_d} \right\} \wedge \psi
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket \text{INSERT} \rrbracket\\
&\overset{def}{=} \quad \overline{\mathscr{T}}\llbracket \langle {}^{\ell'}\vec{v_d} \overset{new}{=} \vec{e},\ {}^{\ell}true \rangle \rrbracket\\
&\overset{def}{=} \quad \{\langle \ell, \psi\rangle \xrightarrow{\text{INSERT}} \langle fin\llbracket \text{INSERT} \rrbracket, \psi'\rangle\}\\
&\qquad\qquad\qquad \text{where } \psi' = \bigwedge \left\{ y_i \to z_i \mid y_i \in \mathscr{V}\llbracket e_i \rrbracket \wedge e_i \in \vec{e} \wedge z_i \in \vec{v_d} \right\} \wedge \psi
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket \text{DELETE} \rrbracket\\
&\overset{def}{=} \quad \overline{\mathscr{T}}\llbracket \langle {}^{\ell'}del(\vec{v_d}),\ {}^{\ell}\phi \rangle \rrbracket\\
&\overset{def}{=} \quad \{\langle \ell, \psi\rangle \xrightarrow{\text{DELETE}} \langle fin\llbracket \text{DELETE} \rrbracket, \psi'\rangle\}\\
&\qquad\qquad\qquad \text{where } \psi' = \bigwedge \left\{ y \to z \mid y \in \mathscr{V}\llbracket \phi \rrbracket \wedge z \in \vec{v_d} \right\} \wedge \psi
\end{aligned}
$$

$$
\overline{\mathscr{T}}\llbracket {}^{\ell}skip \rrbracket \overset{def}{=} \{\langle \ell, \psi\rangle \to \langle fin\llbracket {}^{\ell}skip \rrbracket, \psi\rangle\}
$$

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket {}^{\ell}v_a = e \rrbracket \overset{def}{=} \{\langle \ell, \psi\rangle \to \langle fin\llbracket {}^{\ell}v_a = e \rrbracket, \psi'\rangle\}\\
&\qquad\qquad \text{where } \psi' = \bigwedge \left\{ y \to v_a \mid y \in \mathscr{V}\llbracket e \rrbracket \wedge y \neq v_a \right\} \wedge \left(\psi \ominus \bigwedge \{ u \to v_a \mid u \in \mathbb{V} \} \right)
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket if\ {}^{\ell}b\ then\ c_1\ else\ c_2\ {}^{\ell'}endif \rrbracket\\
&\overset{def}{=} \quad \overline{\mathscr{T}}\llbracket c_1 \rrbracket \cup \overline{\mathscr{T}}\llbracket c_2 \rrbracket\\
&\qquad\qquad \bigcup \{\langle \ell, \psi\rangle \to \langle in\llbracket c_1 \rrbracket, \psi\rangle\} \bigcup \{\langle \ell, \psi\rangle \to \langle in\llbracket c_2 \rrbracket, \psi\rangle\}\\
&\qquad\qquad \bigcup \{\langle \ell', \psi\rangle \to \langle fin\llbracket if\ {}^{\ell}b\ then\ c_1\ else\ c_2\ {}^{\ell'}endif \rrbracket, \psi'\rangle\}\\
&\qquad\qquad \bigcup \{\langle \ell', \psi\rangle \to \langle fin\llbracket if\ {}^{\ell}b\ then\ c_1\ else\ c_2\ {}^{\ell'}endif \rrbracket, \psi''\rangle\}\\
&\qquad\qquad\qquad \text{where } \psi' = \bigwedge \left\{ y \to z \mid y \in \mathscr{V}\llbracket b \rrbracket \wedge z \in \mathsf{BV}(c_1) \wedge y \neq z \right\} \wedge \psi\\
&\qquad\qquad\qquad \text{where } \psi'' = \bigwedge \left\{ y \to z \mid y \in \mathscr{V}\llbracket b \rrbracket \wedge z \in \mathsf{BV}(c_2) \wedge y \neq z \right\} \wedge \psi
\end{aligned}
$$

$$
\begin{aligned}
&\overline{\mathscr{T}}\llbracket while\ {}^{\ell}b\ do\ c\ {}^{\ell'}done \rrbracket\\
&\overset{def}{=} \quad \overline{\mathscr{T}}\llbracket c \rrbracket \bigcup \{\langle \ell, \psi\rangle \to \langle in\llbracket c \rrbracket, \psi\rangle\} \bigcup \{\langle \ell', \psi\rangle \to \langle fin\llbracket while\ {}^{\ell}b\ do\ c\ {}^{\ell'}done \rrbracket, \psi'\rangle\}\\
&\qquad\qquad\qquad \text{where } \psi' = \bigwedge \left\{ y \to z \mid y \in \mathscr{V}\llbracket b \rrbracket \wedge z \in \mathsf{BV}(c) \wedge y \neq z \right\} \wedge \psi
\end{aligned}
$$

$$
\overline{\mathscr{T}}\llbracket c_1;\ c_2 \rrbracket \overset{def}{=} \overline{\mathscr{T}}\llbracket c_1 \rrbracket \cup \overline{\mathscr{T}}\llbracket c_2 \rrbracket
$$

Table 7.1: Definition of Abstract Transition Function $\overline{\mathscr{T}}$

we are interested on persistent data, analyzing object-oriented features of HQL does not meet our objectives neither. In this section, we extend information leakage analysis to the case of HQL. The analysis is performed by (*i*) computing an overapproximation of variables' dependency at each program point, in the form of propositional formula, (*ii*) checking the satisfiability on assigning truth values to variables, in order to identify possible leakage of HQL.

7.4.1 Abstract Semantics of HQL

Methods in HQL include a set of imperative statements[1]. We assume, for the sake of the simplicity, that attackers are able to observe public variables inside of the main method only. Therefore, our analysis only aims at identifying variable dependency at input-output labels of methods.

The abstract transition semantics of constructors and conventional methods are defined below.

Definition 7.1 (Abstract Transition Semantics of Constructor) *Consider a class $c = \langle \mathtt{init}, \mathtt{F}, \mathtt{M} \rangle$ where* `init` *is a default constructor. Let ℓ be the label of* `init`*. The abstract transition semantics of* `init` *is defined as*

$$\overline{\mathscr{T}}[\![{}^{\ell}\mathtt{init}]\!] = \{(\ell, \psi) \to (\mathtt{Succ}({}^{\ell}\mathtt{init}), \psi)\}$$

where $\mathtt{Succ}({}^{\ell}\mathtt{init})$ denotes the successor label of `init`*. Observe that the default constructor is used to initialize the objects-fields only, and it does not add any new dependency.*

The abstract transition semantics of parameterized constructors are defined in the same way as of class methods $m \in \mathtt{M}$.

Definition 7.2 (Abstract Transition Semantics of Methods) *Let $U \in \wp(\mathtt{Var})$ be the set of variables which are passed as actual parameters when invoked a method $m \in \mathtt{M}$ on an abstract state (ℓ, ψ) at program label ℓ. Let $V \in \wp(\mathtt{Var})$ be the formal arguments in the definition of m. We assume that $U \cap V = \emptyset$. Let a and b be the variable returned by m and the variable to which the value returned by m is assigned. The abstract transition semantics is defined as*

$$\overline{\mathscr{T}}[\![{}^{\ell}m]\!] = \{(\ell, \psi) \to (\mathtt{Succ}({}^{\ell}m), \psi')\}$$

[1]For a detailed abstract transition semantics of imperative statements, see chapter 2.

where $\psi' = \{x_i \to y_i \mid x_i \in U, y_i \in V\} \cup \{a \to b\} \cup \psi$ *and* $\texttt{Succ}({}^{\ell}m)$ *is the label of the successor of* m.

Let us classify the high-level HQL variables into two distinct sets: $\texttt{Var}_d$ and $\texttt{Var}_a$. The variables which have a correspondence with database attributes belong to the set $\texttt{Var}_d$. Otherwise, the variables are treated as usual variables and belong to $\texttt{Var}_a$. We denote variables in $\texttt{Var}_d$ by the notation $\overline{v}$, in order to differentiate them from the variables in $\texttt{Var}_a$. This leads to four types of dependency which may arise in HQL programs: $x \to y$, $\overline{x} \to y$, $x \to \overline{y}$ and $\overline{x} \to \overline{y}$, where $x, y \in \texttt{Var}_a$ and $\overline{x}, \overline{y} \in \texttt{Var}_d$.

7.4.1.1 Definition of Abstract Transition Function $\overline{\mathscr{T}}$ for `Session` Methods

The abstract labeled transition semantics of various `Session` methods are defined below, where by $\texttt{Var}(exp)$ and $\texttt{Field}(c)$ we denote the set of variables in *exp* and the set of class-fields in the class *c* respectively. The function $\texttt{Map}(v)$ is defined as:

$$\texttt{Map}(v) = \begin{cases} \overline{v} & \text{if } v \text{ has correspondence with a database attribute,} \\ \\ v & \text{otherwise.} \end{cases}$$

Notice that in the definition of $\overline{\mathscr{T}}$ the notation $\widetilde{v}$ stands for either v or $\overline{v}$. Let $\texttt{SF}(\psi)$ denotes the set of subformulas in ψ, and the operator $\ominus$ is defined by $\psi_1 \ominus \psi_2 = \bigwedge\Big(\texttt{SF}(\psi_1) \backslash \texttt{SF}(\psi_2)\Big)$.

The transition semantics for `Session` method m_{save}**:**

$$\begin{aligned} & \overline{\mathscr{T}}[\![{}^{\ell}m_{save}]\!] \\ \overset{def}{=}\ & \overline{\mathscr{T}}[\![{}^{\ell}(\texttt{C}, \phi, \texttt{SAVE(obj)})]\!] \\ \overset{def}{=}\ & \overline{\mathscr{T}}[\![{}^{\ell}(\{c\}, \texttt{FALSE}, \texttt{SAVE(obj)})]\!] \\ \overset{def}{=}\ & \{\langle \ell, \psi \rangle \xrightarrow{\texttt{SAVE}} \langle \texttt{Succ}({}^{\ell}m_{save}), \psi \rangle\} \end{aligned}$$

The transition semantics for `Session` method m_{upd}**:**

$$\overline{\mathscr{T}}[\![{}^{\ell}m_{upd}]\!]$$

$\overset{def}{=}$ $\overline{\mathscr{T}}[\![^{\ell}(\mathtt{C}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp}))]\!]$
$\overset{def}{=}$ $\overline{\mathscr{T}}[\![^{\ell}(\{c\}, \phi, \mathtt{UPD}(\vec{v}, \vec{exp}))]\!]$
$\overset{def}{=}$ $\{\langle \ell, \psi \rangle \xrightarrow{\mathtt{UPD}} \langle \mathtt{Succ}(^{\ell}m_{upd}), \psi' \rangle\}$
where $\psi' = \bigwedge \left\{ \widetilde{y} \to \bar{z}_i \mid y \in \mathtt{Var}[\![\phi]\!], \widetilde{y} = \mathtt{Map}(y),\ \bar{z}_i \in \vec{v} \right\} \bigcup$
$\bigwedge \left\{ \widetilde{y_i} \to \overline{z_i} \mid y_i \in \mathtt{Var}[\![exp_i]\!], exp_i \in \vec{exp}, \widetilde{y_i} = \mathtt{Map}(y_i), \bar{z}_i \in \vec{v} \right\} \bigcup \psi''$

$$\text{and } \psi'' = \begin{cases} \psi \ominus \left(\widetilde{a} \to \bar{z}_i \mid \bar{z}_i \in \vec{v} \wedge a \in \mathtt{Var} \wedge \widetilde{a} = \mathtt{Map}(a) \right) \text{ if } \phi \text{ is } \mathtt{TRUE} \text{ by default.} \\ \psi \text{ otherwise} \end{cases}$$

The transition semantics for `Session` method m_{del}:

$\overline{\mathscr{T}}[\![^{\ell}m_{del}]\!]$
$\overset{def}{=}$ $\overline{\mathscr{T}}[\![^{\ell}(\mathtt{C}, \phi, \mathtt{DEL}())]\!]$
$\overset{def}{=}$ $\overline{\mathscr{T}}[\![^{\ell}(\{c\}, \phi, \mathtt{DEL}())]\!]$
$\overset{def}{=}$ $\{\langle \ell, \psi \rangle \xrightarrow{\mathtt{DEL}} \langle \mathtt{Succ}(^{\ell}m_{del}), \psi' \rangle\}$
where $\psi' = \bigwedge \left\{ \widetilde{y} \to \bar{z} \mid y \in \mathtt{Var}[\![\phi]\!], \widetilde{y} = \mathtt{Map}(y), \bar{z} \in \mathtt{Field}(c) \right\} \bigcup \psi''$

$$\text{and } \psi'' = \begin{cases} \psi \ominus \left(\widetilde{a} \to \bar{z}_i \mid \bar{z}_i \in \vec{v} \wedge a \in \mathtt{Var} \wedge \widetilde{a} = \mathtt{Map}(a) \right) \text{ if } \phi \text{ is } \mathtt{TRUE} \text{ by default.} \\ \psi \text{ otherwise} \end{cases}$$

The transition semantics for `Session` method m_{sel}:

$\overline{\mathscr{T}}[\![^{\ell}m_{sel}]\!]$
$\overset{def}{=}$ $\overline{\mathscr{T}}[\![^{\ell}(\mathtt{C}, \phi, \mathtt{SEL}(f(\vec{exp}'),\ r(\vec{h}(\vec{x})),\ \phi,\ g(\vec{exp}))]\!]$
$\overset{def}{=}$ $\{\langle \ell, \psi \rangle \xrightarrow{\mathtt{SEL}} \langle \mathtt{Succ}(^{\ell}m_{sel}), \psi' \rangle\}$
where $\psi' = \bigwedge \left\{ \widetilde{y} \to \widetilde{z} \mid y \in (\mathtt{Var}[\![\phi]\!] \cup \mathtt{Var}[\![\vec{e}]\!] \cup \mathtt{Var}[\![\phi']\!] \cup \mathtt{Var}[\![\vec{e}']\!]), z \in \mathtt{Var}[\![\vec{x}]\!], \right.$
$\left. \widetilde{y} = \mathtt{Map}(y), \widetilde{z} = \mathtt{Map}(z) \right\} \bigcup \psi$

7.4.2 Assigning Truth Values

We are now in position to use the abstract semantics defined in the previous section to identify possible sensitive database information leakage through high-level HQL variables. After obtaining over-approximation of variable dependency at each program points, we assign truth values to each variable based on their sensitivity level, and we check the satisfiability of propositional formulae representing variable dependency [122].

Since our main objective is to identify the leakage of sensitive database information possibly due to the interaction of high-level variables, we assume, according to the policy, that different security classes are assigned to database attributes. Accordingly, we assign security levels to the variables in $\mathtt{Var}_d$ based on the correspondences. Similarly, we assign the security levels of the variables in $\mathtt{Var}_a$ based on their use in the program. For instance, the variables which are used on output channels, are considered as public variables. Observe that for the variables with unknown security class, it may be computed based on the dependency of it on the other application variables or database attributes of known security classes.

Let $\Gamma : \mathtt{Var} \rightarrow \{L, H, N\}$ be a function that assigns to each of the variables a security class, either public (L) or private (H) or unknown (N).

After computing abstract semantics of HQL program $\mathcal{P}$, the security class of variables with unknown level (N) in $\mathcal{P}$ are upgraded to either H or L, according to the following function:

$$\mathtt{Upgrade}(v) = Z \text{ if } \exists\, (u \rightarrow v) \in \overline{\mathscr{T}}[\![\mathcal{P}]\!].\ \Gamma(u) = Z \wedge \Gamma(u) \neq N \wedge \Gamma(v) = N \tag{7.1}$$

We say that program $\mathcal{P}$ respects the confidentiality property of database information, if and only if there is no information flow from private to public attributes. To verify this property, a corresponding truth assignment function $\hat{\Gamma}$ is used:

$$\hat{\Gamma}(x) = \begin{cases} T & \text{if } \Gamma(x) = H \\ F & \text{if } \Gamma(x) = L \end{cases}$$

If $\hat{\Gamma}$ does not satisfy any propositional formula in ψ of an abstract state, the analysis will report a possible information leakage.

```
class c1 {
        private int id1, h1, l1;
        c1 { }
        public int getId() { return id1;}
        public void setId(int id) { this.id1 = id;}
        public int getHigh() {return h1;}
        public void setHigh(int x) { this.h1 = x;}
        public int getLow() { return l1;}
        public void setLow(int y) { this.l1 = y;}
}
```

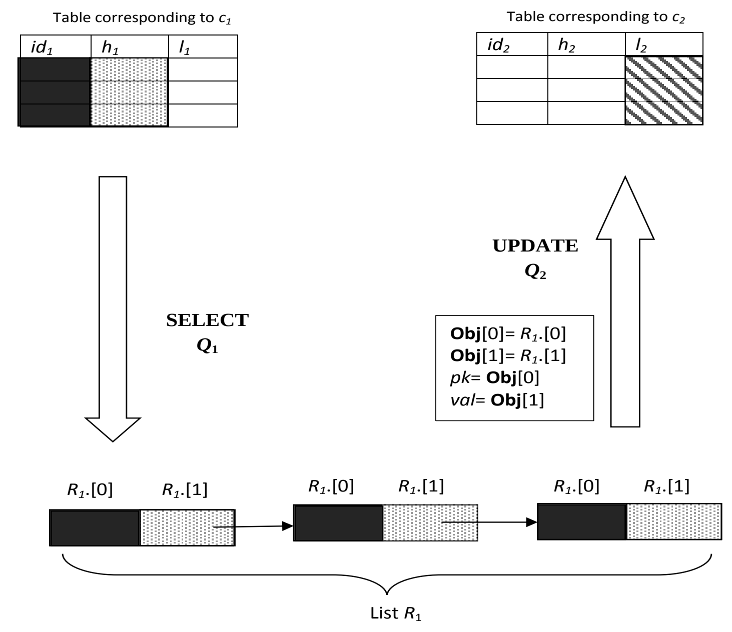
(a) POJO Class c_1

```
class c2 {
        private int id2, h2, l2;
        c2 { }
        public int getId() { return id2;}
        public void setId(int id) { this.id2 = id;}
        public int getHigh() {return h2;}
        public void setHigh(int x) { this.h2 = x;}
        public int getLow() { return l2;}
        public void setLow(int y) { this.l2 = y;}
}
```

(b) POJO Class c_2

```
public class ExClass{
    public static void main(String[] args) {

        Configuration cfg=new Configuration();
        cfg.configure("hibernate.cfg.xml");
        SessionFactory sf=cfg.buildSessionFactory();
        Session ses=sf.openSession();
        Transaction tr=ses.beginTransaction();

                                ......
                                ......

        Query Q1 = session.createQuery("SELECT id1, h1 FROM c1");
        List R1 = Q1.list();
        for(Object[] obj:R1){
            pk=(Int) obj[0];
            val=(Int) obj[1];
            Query Q2 = session.createQuery("UPDATE c2 SET l2 = l2 +1 WHERE id2 = pk AND val=1000");
            int result = Q2.executeUpdate();}

                                ......
                                ......

        tx.commit();
        session.close();}}
```

(c) Class ExClass



(d) Execution view

Figure 7.1: A HQL program and its execution view

7.4.3 Illustration with Example

Let us consider an example in Figure 7.1(c). Here, values of the table corresponding to the class c_1 are used to make a list, and for each element of the list an update is performed on the table corresponding to the class c_2. Observe that there is an information-flow from confidential (denoted by h) to public variables (denoted by l). In fact, the confidential database information h_1 which is extracted at statement 15, affects the public view of the database information l_2 at statement 20. This fact is depicted in Figure 7.1(d).

According to the policy, let the database attribute corresponding to variable h_1 is private, whereas the attributes corresponding to id_1, id_2 and l_2 are public. Therefore,

$$\Gamma(\overline{h_1}) = H \quad \text{and} \quad \Gamma(\overline{id_1}) = \Gamma(\overline{id_2}) = \Gamma(\bar{l_2}) = L$$

For other variables in the program, the security levels are unknown. That is,

$$\Gamma(R_1.[0]) = \Gamma(R_1.[1]) = \Gamma(\texttt{obj[0]}) = \Gamma(\texttt{obj[1]}) = \Gamma(pk) = \Gamma(h_2) = N$$

Considering the domain of positive propositional formulae, the abstract semantics yields the following formulae at program point 20 in $\mathcal{P}$:

$$\overline{id_1} \to R_1.[0]; \quad \overline{h_1} \to R_1.[1]; \quad R_1.[0] \to \texttt{obj[0]}; \quad R_1.[1] \to \texttt{obj[1]};$$
$$\texttt{obj[0]} \to pk; \quad \texttt{obj[1]} \to h_2; \quad pk \to \bar{l_2}; \quad \overline{id_2} \to \bar{l_2}; \quad h_2 \to \bar{l_2};$$

According to equation 7.1, the security levels of the variables with unknown security level N are upgraded as below:

$$\Gamma(R_1.[0]) = L, \quad \Gamma(R_1.[1]) = H, \quad \Gamma(\texttt{obj[0]}) = L, \quad \Gamma(\texttt{obj[1]}) = H$$
$$\Gamma(pk) = L, \qquad \Gamma(h_2) = H$$

Finally, we apply the truth assignment function $\overline{\Gamma}$ which does not satisfy the formula $h_2 \to \bar{l_2}$, as $\overline{\Gamma}(h_2) = T$ and $\overline{\Gamma}(\bar{l_2}) = F$ and $T \to F$ is false.

Therefore, the analysis reports that the example program $\mathcal{P}$ is vulnerable to leakage of confidential database data.

7.4.4 Improving the Analysis

The abstract semantics at various levels of abstraction of numerical domain which we defined in chapter 5 can improve the precision of the above analysis results significantly.

This can be done by using reduced Product the analysis results obtained at different levels of abstractions [122]. Let us formally define this among the logical and numerical domains analysis-results. Let $\mathscr{N}$ be the a numerical abstract domain of interest. Suppose $\mathscr{T}^*$ and $\overline{\mathscr{T}}$ represent the set of concrete traces and the set of abstract traces in the propositional formulae domain respectively. Let $(\wp(\mathscr{T}^*), \alpha_0, \gamma_0, \wp(\overline{\mathscr{T}}))$ and $(\wp(\mathscr{T}^*), \alpha_1, \gamma_1, \mathscr{N})$ be two Galois Connections, and let $\Upsilon : \wp(\overline{\mathscr{T}}) \times \mathscr{N} \rightarrow \wp(\overline{\mathscr{T}})$ be a reduced product operator defined as $\Upsilon(\mathcal{X}, \mathfrak{R}) = \mathcal{X}'$, where $\mathcal{X} \in \wp(\overline{\mathscr{T}})$ is a set of partial traces, $\mathfrak{R} \in \mathscr{N}$, and

$$\mathcal{X}' = \Big\{ \langle \ell_i, \psi_k \rangle \mid \langle \ell_i, \psi_j \rangle \in \mathcal{X} \wedge \psi_k = (\psi_j \ominus \{x \rightarrow y \mid y \in \mathfrak{R}\}) \Big\}$$

7.5 Conclusions

In this chapter, we propose a static analysis framework to perform information flow analysis of database program based on the Abstract Interpretation theory. Our approach captures information leakage on "permanent" data stored in a database when a database application manipulates them. We explain how the reduced product of the analysis-results form various abstract domains may improve the precision. This may also lead to a refinement of the non-interference definition that focusses on confidentiality of the data instead of variables.

CHAPTER 8

Conclusions and Future Directions

Preface

In this chapter, we conclude our research works and highlight the possible future research scope.

Dependency analysis of database programs plays 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 thesis proposes a novel approach to compute semantics-based independency among database statements, based on the Abstract Interpretation theory. To this aim, we define an abstract semantics of database programs embedding SQL and HQL in various levels of abstractions, ranging from the domain of intervals to the domain of polyhedra. This computable abstract semantics serves as a powerful basis to design a static semantics-based dependency analyzer for database applications, resulting into a more precise dependency information by removing false alarms. This is also true for undecidable scenarios when the input database instance is unknown. 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 dependency 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. Finally, we have demonstrated two case studies - database code slicing and language-based information leakage analysis - of our proposed dependency refinement approach.

In this research line, we identified the following interesting future scopes:

1. To broaden the application of this proposed analysis to other related software en-

gineering problems, e.g. materialization view creation, data provenance, integrity constraints verification, etc.

2. Designing of new ad-hoc abstract domains suitable for analysis and verification of database programs in new applicative scenarios.
3. Extending the analysis to a distributed scenario with multiple transactions and heterogeneous database systems.
4. Enhancement of `SemDDA` to support other popular mainstream languages C, C++, Java, HQL, etc. Besides, we shall also consider sub-queries, dynamically generated queries, JOIN queries and string data-type along with numerical attributes.

References

[1] Gotocode. `http://www.gotocode.com`. [Online; accessed 20-Dec-2010], (we now archived at: `https://github.com/angshumanjana/GotoCode`).

[2] Md. Imran Alam and Raju Halder. Refining Dependencies for Information Flow Analysis of Database Applications. In *International Journal of Trust Management in Computing and Communications*. Inderscience, 2016.

[3] Frances E. Allen. Control Flow Analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[4] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Compututer Programming*, 64:3–28, 2007.

[5] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. *ACM Trans. Program. Lang. Syst.*, 2:56–76, 1980.

[6] Morton M Astrahan and Donald D Chamberlin. Implementation of a structured english query language. *Communications of the ACM*, 18(10):580–588, 1975.

[7] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The PPL: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. Technical report, Dipartimento di Matematica, Università di Parma, Italy, 2006. http://www.cs.unipr.it/ppl/.

[8] Roberto Bagnara, Patricia M Hill, and Enea Zaffanella. Widening operators for powerset domains. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3):413–414, 2007.

[9] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proc. of the 19th int. symposium on Software testing and analysis*, pages 13–24, Trento, Italy, 2010. ACM Press.

[10] Elena Baralis and Jennifer Widom. An Algebraic Approach to Rule Analysis in Expert Database Systems. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 475–486. Morgan Kaufmann Publishers Inc., 1994.

[11] Christian Bauer and Gavin King. *Hibernate in Action*. Manning Publications Co., 2004.

[12] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., 2006.

[13] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[14] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.

[15] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, 2006.

[16] Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):14:1–14:39, 2010.

[17] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. PLUTO: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*, 2008.

[18] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca. Software salvaging based on conditions. In *Proceedings of the 10th International Conference on Software Maintenance (ICSM '94)*, pages 424–433, Victoria, British Columbia, Canada, September 1994. IEEE Computer Society.

[19] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11–12):595–607, 1998.

[20] Salvador Cavadini. Secure slices of insecure programs. In *Proc. of the ACM symposium on Information, computer and communications security*, pages 112–122, Tokyo, Japan, 2008. ACM Press.

[21] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. A history and evaluation of system r. *Communications of the ACM*, 24(10):632–646, 1981.

[22] L. Chen, A. Minè, and Patrick Cousot. A Sound Floating-Point Polyhedra Abstract Domain. In *Proc. of the 6th Asian Symposium on PLS*, pages 3–18, 2008.

[23] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance As Dependency Analysis. In *Proceedings of the 11th ICDPL*, DBPL'07, pages 138–152, 2007.

[24] Jingde Cheng. Dependence analysis of parallel and distributed programs and its applications. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 370–377. IEEE, 1997.

[25] NV Chernikoba. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[26] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, Berlin, Heidelberg, 2003. Springer-Verlag.

[27] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 1990.

[28] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. A Suite of Abstract Domains for Static Analysis of String Values. *Softw. Pract. Exper.*, 45(2):245–287, 2015.

[29] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[30] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the POPL'77*, pages 238–252, 1977.

[31] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press.

[32] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3):103–179, 1992.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[34] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

[35] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *ACM SIGPLAN Notices*, volume 37, pages 178–190. ACM, 2002.

[36] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the POPL '78*, pages 84–96, 1978.

[37] Arjun Dasgupta, Vivek Narasayya, and Manoj Syamala. A static analysis framework for database applications. In *IEEE International Conference on Data Engineering*, pages 1403–1414. IEEE, 2009.

[38] Christopher John Date. *An introduction to database systems*. Pearson Education India, 2006.

[39] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[40] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19:236–243, 1976.

[41] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 169–185, Philadelphia, PA, 2012. Springer-Verlag LNCS.

[42] James Elliott, Tim O'Brien, and Ryan Fowler. *Harnessing Hibernate*. O'Reilly, first edition, 2008.

[43] Ramez Elmasri and Sham Navathe. *Fundamentals of database systems*. Pearson London, 2016.

[44] Ramez Elmasri and Shamkant Navathe. *Fundamentals of database systems*. Addison-Wesley Publishing Company, 2010.

[45] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Lang. and Sys.*, 9(3):319–349, 1987.

[46] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable Verification of Hybrid Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[47] Dina Goldin, Srinath Srinivasa, and Vijaya Srikanti. Active databases as information systems. In *Database Engineering and Applications Symposium, 2004. IDEAS'04. Proceedings. International*, pages 123–130. IEEE, 2004.

[48] Dina Goldin, Srinath Srinivasa, and Bernhard Thalheim. Is= dbs+ interaction: towards principles of information system design. In *International Conference on Conceptual Modeling*, pages 140–153. Springer, 2000.

[49] Diganta Goswami, Rajib Mall, and Prosenjit Chatterjee. Static slicing in Unix process environment. *Softw., Pract. Exper.*, 30(1):17–36, 2000.

[50] Jay Greenspan and Brad Bulger. *MySQL/PHP database applications*. John Wiley & Sons, Inc., 2001.

[51] Raju Halder. Language-based security analysis of database applications. In *3rd International Conference on Computer, Communication, Control and Information Technology (C3IT'15)*, pages 1–4. IEEE Press, 2015.

[52] Raju Halder and Agostino Cortesi. Abstract Interpretation of Database Query Languages. *Computer Languages, Systems & Structures*, 38:123–157, 2012.

[53] Raju Halder and Agostino Cortesi. Abstract Program Slicing of Database Query Languages. In *Proceedings of the the 28th Symposium On Applied Computing - Special Track on Database Theory, Technology, and Applications*, pages 838–845, Coimbra, Portugal, 2013. ACM Press.

[54] Raju Halder, Angshuman Jana, and Agostino Cortesi. Data Leakage Analysis of the Hibernate Query Language on a Propositional Formulae Domain. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 23:23–44, 2016.

[55] Raju Halder, Matteo Zanioli, and Agostino Cortesi. Information leakage analysis of database query languages. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, pages 813–820, Gyeongju, Korea, 24–28 March 2014. ACM Press.

[56] William GJ Halfond and Alessandro Orso. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183. ACM, 2005.

[57] William GJ Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

[58] R. J. Hall. Automatic extraction of executable program subsets by simultaneous program slicing. *The Journal of Automated Software Engineering*, 2(1):33–53, 1995.

[59] Christian Hammer. Experiences with PDG-Based IFC. In *Proc. of the Engineering Secure Software and Systems*, pages 44–60, Pisa, Italy, 2010. Springer-Verlag.

[60] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for java based on path conditions in dependence graphs. In *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pages 87–96, Arlington, VA, 2006. IEEE.

[61] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *IJIS*, 8:399–422, 2009.

[62] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[63] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 25–34. IEEE, 2005.

[64] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on PLS*, 12(1):26–60, 1990.

[65] Susan Horwitz and Thomas Reps. The Use of Program Dependence Graphs in Software Engineering. In *Proc. of the 14th ICSE*, pages 392–411. ACM Press, 1992.

[66] Charles Hymans and Eben Upton. Static Analysis of Gated Data Dependence Graphs. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, pages 197–211, 2004.

[67] Jean-Louis Imbert. Fourier's Elimination: Which to Choose? In *PPCP*, pages 117–129, 1993.

[68] Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pages 342–360. Elsevier, 1989.

[69] AMERICAN NATIONAL STANDARD INSTITUTE. Information technology-database languages-SQL-part 2: Foundation (SQL/foundation), 2003.

[70] Shachar Itzhaky, Tomer Kotek, Noam Rinetzky, Mooly Sagiv, Orr Tamir, Helmut Veith, and Florian Zuleger. On the automated verification of web applications with embedded SQL. *arXiv preprint arXiv:1610.02101*, 2016.

[71] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. *ACM SIGSOFT Software Engineering Notes*, 19(5):2–10, 1994.

[72] Lingxiao Jiang. *Scalable Detection of Similar Code: Techniques and Applications*. PhD thesis, Davis, CA, USA, 2009.

[73] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proc. of the 36th ACM SIGPLAN Conference on PLDI*, PLDI '15, pages 291–302. ACM, 2015.

[74] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37(1-3):113–138, 2000.

[75] Jonas S Karlsson, Amrish Lal, Cliff Leung, and Thanh Pham. Ibm db2 everyplace: A small footprint relational database system. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 230–232. IEEE, 2001.

[76] Jonathan A. Kelner and Daniel A. Spielman. A Randomized Polynomial-time Simplex Algorithm for Linear Programming. In *Proc. of the 38th Annual ACM Symposium on Theory of Computing*, pages 51–60. ACM, 2006.

[77] Tim King, Clark Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for smt. In *Proc. of the 14th ICFMCAD*, 2014. Lausanne, Switzerland, to appear.

[78] Bogdan Korel and Jurgen Rilling. Program Slicing in Understanding of Large Programs. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, pages 145–152, Ischia, Italy, June 1998. IEEE Computer Society.

[79] Jens Krinke. Information flow control and taint analysis with dependence graphs. In *3rd International Workshop on Code Based Security Assessments (CoBaSSA 2007)*, pages 6–9, 2007.

[80] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive Program Analysis As Database Queries. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 1–12. ACM, 2005.

[81] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[82] Filippo Lanubile and Giuseppe Visaggio. Extracting Reusable Functions by Flow Graph-Based Program Slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[83] Loren Larsen and Mary Jean Harrold. Slicing object-oriented software. In *Proceedings of the 18th ICSE*, pages 495–505, Berlin, Germany, 1996. IEEE CS.

[84] Alon Y. Levy and Yehoshua Sagiv. Queries Independent of Updates. In *Proceedings of the 19th International Conf. on VLDB*, pages 171–181. Morgan Kaufmann Publishers Inc., 1993.

[85] Bixin Li. Analyzing information-flow in java program based on slicing technique. *SIGSOFT Softw. Eng. Notes*, 27:98–103, 2002.

[86] Francesco Logozzo. Class invariants as abstract interpretation of trace semantics. *Computer Languages, Systems & Structures*, 35:100–142, 2009.

[87] Kevin Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, Inc., 2008.

[88] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of the 4th Workshop on Program Comprehension (WPC '96)*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society.

[89] Isabella Mastroeni and Damiano Zanardini. Data dependencies and program slicing: from syntax to abstract semantics. In *Proc. of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 125–134, 2008.

[90] Phil Mcminn, Chris J. Wright, and Gregory M. Kapfhammer. The Effectiveness of Test Coverage Criteria for Relational Database Schema Integrity Constraints. *ACM Trans. Softw. Eng. Methodol.*, 25(1):8:1–8:49, 2015.

[91] Jim Melton. Database language sql. In *Handbook on Architectures of Information Systems*, pages 105–132. Springer, 1998.

[92] Todd Millstein, Alon Levy, and Marc Friedman. Query Containment for Data Integration Systems. In *Proc. of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on PDS*, pages 67–75. ACM, 2000.

[93] Antoine Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects, Second Symposium, PADO*, pages 155–172, 2001.

[94] Antoine Minè. The Octagon Abstract Domain. *Higher Order Symbol. Comput.*, 19(1):31–100, 2006. http://www.astree.ens.fr/.

[95] Antoine Miné. The Octagon Abstract Domain. *CoRR*, abs/cs/0703084, 2007.

[96] Flemming Nielson and N Jones. Abstract interpretation: a semantics-based tool for program analysis. *Handbook of logic in computer science*, 4:527–636, 1994.

[97] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[98] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press Oxford, 1990.

[99] Elizabeth J. O'Neil. Object/relational mapping 2008: Hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 1351–1356, New York, NY, USA, 2008. ACM.

[100] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, 1984.

[101] Jens Palsberg. Type-based Analysis and Applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, pages 20–27, Snowbird, Utah, USA, 2001. ACM.

[102] A. Podgurski and L. A. Clarke. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Trans. on Software Engineering*, 16(9):965–979, 1990.

[103] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems*, 25:117–158, 2003.

[104] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[105] Xavier Rival. Abstract dependences for alarm diagnosis. In *Asian Symposium on Programming Languages and Systems*, pages 347–363. Springer, 2005.

[106] Michelle Elaine Ruse. *Model checking techniques for vulnerability analysis of Web applications*. PhD thesis, 2013. https://lib.dr.iastate.edu/etd/13211.

[107] Philip Russom. Managing big data. *TDWI Best Practices Report, TDWI Research*, pages 1–40, 2013.

[108] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:5–19, 2003.

[109] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17:517–548, 2009.

[110] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. Selinq: tracking information across application-database boundaries. In *ACM SIGPLAN Notices*, volume 49, pages 25–38. ACM, 2014.

[111] Soumya Sen, Anjan Dutta, Agostino Cortesi, and Nabendu Chaki. A New Scale for Attribute Dependency in Large Database Systems. In *CISIM*, volume 7564 of *LNCS*, pages 266–277. 2012.

[112] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 203–217, Washington, DC, USA, 2007. IEEE Computer Society.

[113] Geoffrey Smith. Principles of secure information flow analysis. In M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307, Novï£¡ Smokovec, Slovakia, 2007. Springer US.

[114] Richard N. Taylor and Leon J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, (3):265–278, 1980.

[115] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.

[116] Frank Tip. A Survey of Program Slicing Techniques. Technical report, 1994.

[117] G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Notices*, 26(6):107–119, 1991.

[118] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, 1996.

[119] Gary Wassermann, Carl Gould, Zhendong Su, and Premkumar Devanbu. Static Checking of Dynamically Generated Queries in Database Applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.

[120] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[121] David Willmor, Suzanne M. Embury, and Jianhua Shao. Program Slicing in the Presence of a Database State. In *Proceedings of the 20th IEEE ICSM*, ICSM '04, pages 448–452, 2004.

[122] Matteo Zanioli and Agostino Cortesi. Information leakage analysis by abstract interpretation. In *Proc. of the 37th int. conf. on Current trends in theory and practice of computer science*, pages 545–557, Novï£¡ Smokovec, Slovakia, 2011. Springer LNCS 6543.

[123] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. Sails: static analysis of information leakage with sample. In *Proc. of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*, pages 1308–1313, Trento, Italy, 2012. ACM Press.

[124] Jianjun Zhao. Multithreaded dependence graphs for concurrent java program. In *pdse*, page 13. IEEE, 1999.

Publications

Journal Publications

1. Angshuman Jana, Raju Halder, K. V. Abhishekh, S. D. Ganni, and Agostino Cortesi, “Extending Abstract Interpretation to Dependency Analysis of Database Applications", *IEEE Transactions on Software Engineering*, IEEE, 2018. (Accepted for publication, Preprint available at: https://ieeexplore.ieee.org/document/8423692/)

2. Raju Halder, Angshuman Jana, and Agostino Cortesi, “Data Leakage Analysis of the Hibernate Query Language on a Propositional Formulae Domain", *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems*, Volume 23: 23-44, Springer, 2016.

Conference Publications

1. Angshuman Jana and Raju Halder, “Defining Abstract Semantics for Static Dependence Analysis of Relational Database Applications", in Proc. of the *12th International Conference on Information Systems Security (ICISS '16)*, Springer LNCS 10063, pp. 151-171, MNIT Jaipur, India, 2016.

2. Angshuman Jana, Raju Halder, Nabendu Chaki, and Agostino Cortesi, “Policy-based Slicing of Hibernate Query Language", in Proc. of the *14th International Conference on Computer Information Systems and Industrial Management Applications (CISIM '15)*, Springer LNCS 9339, pp. 267-281, Warsaw, Poland, 2015.

3. Angshuman Jana, Raju Halder, and Agostino Cortesi, “ Verification of Hibernate Query Language by Abstract Interpretation", in Proc. of the *5th International Conference on Intelligence Science and Big Data Engineering (IScIDE '15)*, Springer LNCS 9243, pp. 116-128, Suzhou, China, 2015.

Poster Presentations

1. Angshuman Jana and Raju Halder, “ SemDDA: A Semantics-based Database Dependency Analyzer", Appeared in the Poster Session of *14th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '16)*, IEEE, IIT Kanpur, India, 2016.

2. Angshuman Jana, Raju Halder, and Agostino Cortesi, " Abstract Interpretation of Hibernate Query Language", Appeared in Student Poster Session and Student Research Competition of *42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*, ACM, Mumbai, India, 2015.

<u>Other Related Publications</u>

1. Angshuman Jana, Md. Imran Alam, and Raju Halder, "A Symbolic Model Checker for Database Programs", in Proc. of the *13th International Conference on Software Technologies (ICSOFT '18)*, SciTePress, pp. 381-388, Porto, Portugal, 2018.

2. Bharat Kumar Ahuja, Angshuman Jana, Ankit Swarnkar, and Raju Halder, " On Preventing SQL Injection Attacks", *Advanced Computing and Systems for Security (ACSS '16)*, Springer AISC 395, Kolkata, India, 2016.