Program Transformation for Proving Database Transaction Safety

Thesis by
Michael John Lawley, BSc(Hons)

School of Computing and Information Technology
Faculty of Information and Communication Technology
Griffith University
Nathan, Queensland, Australia 4111

Submitted in fulfillment of the requirements
of the Degree of
Doctor of Philosophy

July 1998

Abstract

In this thesis we propose the use of Dijkstra's concept of a predicate transformer [Dij75] for the determination of database transaction safety [SS89] and the generation of simple conditions to check that a transaction will not violate the integrity constraints in the case that it is not safe. The generation of this simple condition is something that can be done statically, thus providing a mechanism for generating safe transactions.

Our approach treats a database as *state*, a database transaction as a *program*, and the database's integrity constraints as a *postcondition* in order to use a predicate transformer [Dij75] to generate a weakest precondition.

We begin by introducing a set-oriented update language for relational databases for which a predicate transformer is then defined. Subsequently, we introduce a more powerful update language for deductive databases and define a new predicate transformer to deal with this language and the more powerful integrity constraints that can be expressed using recursive rules.

Next we introduce a data model with object-oriented features including methods, inheritance and dynamic overriding. We then extend the predicate transformer to handle these new features. For each of the predicate transformers, we prove that they do indeed generate a weakest precondition for a transaction and the database integrity constraints.

However, the weakest precondition generated by a predicate transformer still involves much redundant checking. For several general classes of integrity constraint, including referential integrity and functional dependencies, we prove that the weakest precondition can be substantially further simplified to avoid checking things we already know to be true under the assumption that the database currently satisfies its integrity con-

straints.

In addition, we propose the use of the predicate transformer in combination with meta-rules that capture the exact incremental change to the database of a particular transaction. This provides a more general approach to generating simple checks for enforcing transaction safety.

We show that this approach is superior to known existing previous approaches to the problem of efficient integrity constraint checking and transaction safety for relational, deductive, and deductive object-oriented databases.

Finally, we demonstrate several further applications of the predicate transformer to the problems of schema constraints, dynamic integrity constraints, and determining the correctness of methods for view updates. We also show how to support transactions embedded in procedural languages such as C.

Acknowledgements

There are many people to whom I owe gratitude for their friendship, encouragement and support in completing this thesis. The following people deserve special mention.

My supervisor, Rodney Topor, has been a mentor in the true sense. He sets standards I strive towards and has been very tolerant of me over the years. I especially enjoyed the time we spent bush-walking in South-East Queensland.

I am also grateful to my close colleagues and friends Gill Dobbie, Alex Lefebvre, and Chris Higgins. They helped round out my background in databases and logic programming, provided feedback on my work and thesis over the years and have been very supportive during difficult times.

I'd like to thank the members of the Department of Computer Science at the University of Melbourne where I spent my early years, and the members of the School of Computing and Information Technology at Griffith University.

My friends and colleagues at the Distributed Systems Technology Centre (DSTC) provided a much too interesting and distracting research environment away from my thesis.

Other people deserving of special thanks are: Sara Holmes, Shawn Parr, Janna Seto, Dean Kuo, Sonya Finnigan, Claire Trickett, and Richard Hagen.

Finally, I owe a special debt of thanks and love to my parents, John and Robin Lawley, and my sisters, Jane and Catherine.

This work was partially supported by an Australian Postgradute Research Award.

Contents

Abstract i

Acknowledgements iii

1 Introduction 1

1.1 Databases and Constraints 1

1.2 Transaction Safety 3

1.3 Thesis Structure 5

2 Relational Databases 7

2.1 Data Model 7

2.2 Update Language 9

2.3 Constraint Transformation 15

2.4 Related Work 22

2.5 Summary 23

3 Deductive Databases 25

3.1 Data Model 25

3.2 Update Language 27

3.3 Constraint Transformation 29

3.4 Related Work 34

3.5 Summary 36

4 Deductive Object-Oriented Databases 39

4.1 Overview 39

4.2 Data Model 43

4.3 Updates 47

4.4 Condition Transformers 49

4.5 Related Work 56
4.6 Summary 58

5 Simplifying the Safety Condition **59**
5.1 Introduction 59
5.2 Direct Simplification 60
5.3 Simplification by Update Propagation 67
5.3.1 Deductive Databases 68
5.3.2 Deductive Object-Oriented Databases 73
5.4 Weakest Precondition Simplification Problem 76
5.5 Summary 82

6 Extensions **83**
6.1 Schema Constraints 83
6.2 Dynamic Constraints 86
6.3 Deferred Updates 87
6.4 Global Updates 90

7 Conclusion **97**
7.1 Summary 97
7.2 Future Work 98

A Notations and Symbols **103**

B Prolog Code **107**

Bibliography **113**

This work has not previously been submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

Chapter 1

Introduction

1.1 Databases and Constraints

Computer database systems are central to the day to day operation of most modern organisations. Essential in the organisation and storage of data in these systems is the integrity and consistency of this data. It is expected that a database management system (DBMS) automatically provide isolation and concurrent access to the data it manages, in addition to just providing fault tolerant storage and efficient retrieval of data. In doing so, application programmers are relieved from having to implement and re-implement locking schemes and other transaction management features and are free to concentrate on the application semantics themselves. However, besides physical data integrity a DBMS should also help manage semantic data integrity.

Integrity constraints such as those in example 1.1 allow the declarative specification of rules which describe consistency criteria for the data in a database to be specified independently of application code and stored by the database management systems itself. Since all access to the data in a database must go through the DBMS, the DBMS is then in a position to enforce the consistency of any update to the database by refusing to commit transactions that would cause the integrity constraints to be violated.

Example 1.1 Given the following set of database relations:

```
emp(E) = E is an employee.
wrk(W) = W is a worker.
```

```
        mngr(M) = M is a manager.
  supply(C,D,I) = company C supplies department D with item I.
 inventory(I,N) = there are N of item I in stock.
    subord(E,M) = employee E is subordinate to manager M.
```

We can define some integrity constraints as follows:

(1.1) A company that supplies Barbie dolls must also supply Ken dolls.
$\forall x, y \exists z (supply(x, y, barbie) \rightarrow supply(x, z, ken))$

(1.2) There can never be less than one of any item in stock.
$\forall x, y (inventory(x, y) \rightarrow y > 0)$

(1.3) The subord relation must be transitively closed.
$\forall x, y, z (subord(x, z) \wedge subord(z, y) \rightarrow subord(x, y))$

(1.4) Every employee must be a worker or a manager, but cannot be both.
$\forall x (emp(x) \rightarrow (wrk(x) \wedge \neg mngr(x)) \vee (mngr(x) \wedge \neg wrk(x)))$

□

Unfortunately, present commercial database implementations fall short in providing direct support for the specification and enforcement of integrity constraints. This means that any constraints that cannot be directly specified to the DBMS must be checked and enforced in all the application code that updates the database. This immediately leads to several problems:

- constraints are represented implicitly and are distributed throughout the application code rather than being explicitly stated in one place;
- it is easy to make a mistake in translating constraints to the appropriate checking code;
- it is difficult and expensive to keep all the application code consistent and up to date as, over time, applications and constraints evolve.

The recent trend towards client/server and open systems, which leads to a proliferation of applications, only compounds these problems.

The limitations of commercial database implementations are not without reason. A naive approach to automatically checking integrity constraints by simply re-checking every constraint against the entire database

after each transaction can result in performing a great deal of unnecessary work. The key insight into avoiding this unnecessary work is the observation that if a databases satisfies its integrity constraints, then only those parts of the database that have been changed by a transaction can cause the integrity constraints to subsequently be violated. For example, an update to an employee's salary cannot violate constraints relating to inventory. Similarly, an update to the inventory count for a single item cannot violate the minimum count constraint (1.2) for other items. One kind of constraint that is supported by current DBMSs is the not-NULL constraint. However, the reason it is supported is because it meshes tightly with (many of) the indexing algorithms that databases use to search for matching records or when performing joins for more complicated queries. Thus it is inexpensive and simple to implement. Primary key and foreign key constraints are also often supported for similar reasons.

While recent work [JJ91, Mar90] has made significant improvements in the efficiency of integrity constraint checking, much of it suffers from two main limitations. The first is the treatment of updates as simply the (disjoint) sets of inserted and deleted facts while ignoring the programs that generate these updates. This means certain semantic information which may allow for more efficient constraint checking is unavailable. The second limitation is performing the checks after performing the updates. This has several implications: write locks must be held while the checks are performed; and constraint violation is detected after it has occurred, potentially quite some time after, making fixing things up difficult and possibly requiring the transaction to be rolled back.

1.2 Transaction Safety

An alternative approach to efficient integrity constraint checking is to enforce transaction safety [SS89]. A database transaction is considered to be *safe* if it cannot violate any of the database's integrity constraints. If so, then no checks need be made when that transaction is executed. The problem is now shifted to one of determining whether a given transaction is safe with respect to a set of integrity constraints.

Example 1.2 Consider the referential integrity constraint 1.1 from exam-

ple 1.1 and a simple transaction which inserts the tuple

$$\texttt{supply}(\texttt{mattel}, \texttt{marketing}, \texttt{barbie}).$$

Traditional integrity constraint checking would require the condition $\exists z\ supply(mattel, z, ken)$ be checked after the tuple was inserted followed by a rollback if the check fails. In contrast, if we perform the check first, making the update conditional on the success of the check, then we avoid the need to possibly rollback. □

Approaching integrity constraint checking from this point of view offers several less well recognised benefits. The traditional approach of checking the constraints at the end of a transaction, immediately before the transaction is committed means that write locks on the affected parts of the database must be held until the constraint checking is finished. On the other hand, when enforcing transaction safety, all necessary checks are performed before executing the update. This means only read locks are required (and acquired) at this stage, possibly allowing for greater concurrency. But, perhaps more importantly, once these checks have been satisfied, the transaction will not have to be rolled back due to integrity constraint violation. The ability to roll-back a transaction requires the maintenance of both old and new database states in some form and is thus quite expensive. Being able to avoid roll-back is a major benefit.

With these advantages, the natural question is why has so much effort been expended on the traditional approach. One reason is that, due to the limited expressive power of SQL, transactions have had to be implemented by embedding the SQL component in a host language such as C. It is this embedding that causes the problem. The nature of languages like C make it very hard to reason about their semantics.

Recent trends in the database field promise to make transaction safety a more viable option for efficient integrity constraint checking. One such trend is the evolution of the SQL standard and current DBMS implementations (e.g., Oracle 8) to include a programming language and *stored programs*. This greatly reduces the need for a host language in which to embed SQL query and update statements. Additionally, the increasing usage of object-oriented and object-relational databases, which provide *methods* which are stored and managed by the DBMS again liberates the database query and update language from reliance on a host language.

In fact, the inclusion of object-oriented features in the proposed SQL-3 standard and the convergence of the SQL-3 [ISO] and ODMG [Cat94] standards indicates that stored programs or methods may become an integral part of future database implementations. However, the extra expressive power of the object-oriented data model, with concepts such as inheritance and overriding, has the potential to nullify or obstruct current techniques for efficient integrity constraint checking using either approach [BD92, JJ91, JK90, Mar90].

Our approach to transaction safety can be broken into two stages. The first stage involves applying the ideas behind Dijkstra's predicate transformer *wp* [Dij75] in the database context where the database represents the *state*, S is a database transaction, H is the conjunction of the integrity constraints, and the weakest precondition as generated by $wp(S, H)$ is the safety condition, guaranteeing that the integrity constraints will hold after execution of the transaction. The second step attempts to simplify the safety condition G using the knowledge that the database initially satisfies all the constraints H.

By combining these two steps we produce a method for ensuring transaction safety that is, in most cases, more powerful than the methods currently described in the literature. By more powerful we mean, either our method produces conditions which are simpler to check than the conditions produced by the other methods or, where the other methods produce optimal solutions, our method also produces optimal solutions but also deals with a larger class of update programs and/or a more complex data model.

1.3 Thesis Structure

In this thesis we first introduce a new technique for efficient checking of integrity constraints in relational databases. This gives us a simple and solid foundation to work from. We then extend this work to handle deductive databases and, finally, deductive object-oriented databases. Not only is it useful to see the effectiveness of this technique when applied to relational databases, which are still the most common databases in use today, but several of the seminal papers on efficient integrity constraint checking apply to the relational model only, so this affords a means for

effective comparison of these techniques with ours.

Deductive databases differ from relational databases essentially by allowing recursive views. This increase in expressive power complicates matters by allowing more complex integrity constraints to be specified and therefore makes their checking and simplification more difficult. Also, the update language we consider is more powerful since it can use this more powerful query language to specify the tuples to be inserted or deleted.

Finally, deductive object-oriented databases require a fundamentally new update operation, namely object creation. We must also deal with inheritance and overriding (i.e., dynamic or late binding).

The remainder of this thesis is structured as follows. Chapter 2 presents the base case of this work. It introduces the relational data model and a set-oriented update language comparable to SQL updates. It then shows how to construct, from a transaction and an integrity constraint, a condition that can be used to guarantee transaction safety. Chapter 3 extends this work to the deductive data model to handle constraints involving recursion, and to deal with a more powerful update language. Chapter 4 takes this work a step further to deal with a deductive object-oriented data model which has inheritance and overriding and a more powerful update language that supports object creation. Chapter 5 gives a formal definition and investigates the problem of simplifying the transaction safety condition given that the database already satisfies its integrity constraints. Chapter 6 demonstrates several applications of and extensions to the previous results. These include maintaining schema constraints for deductive object-oriented databases, dynamic integrity constraints, an approach for handling incrementally specified ad-hoc updates, and an application to the specification of update methods in multidatabases. Finally, Chapter 7 summarises our contributions and indicates directions for further research.

Chapter 2

Relational Databases

This chapter examines the problem of enforcing transaction safety for relational databases. We begin by formally introducing the relational data model, then summarising the strengths and weaknesses of the related integrity constraint checking literature.

After introducing the set-oriented update language which is used to specify the transactions, we describe a method for producing a condition to ensure transaction safety and show how it can be applied to several common constraints.

The material presented in this chapter is based on joint research with Rodney Topor and Mark Wallace that first appeared in [LTW93]. It was developed independently of the work described by Qian [Qia90] where an axiomatisation of database transactions in the manner of Hoare [Hoa69] is described which gives an alternative formulation for the generation of weakest preconditions. While their update language is of equivalent expressive power to the one presented here, it has a difficult semantics and we believe the predicate transformer based approach provides a more direct formulation of the concepts. This makes it simpler to extend the approach to deal with both deductive and deductive object-oriented databases.

2.1 Data Model

Let $\mathcal{U}$ be a *universal domain* consisting of a countably infinite set of constants. A *database* is a tuple $\langle R_1, \ldots, R_n \rangle$, where each R_i is a finite named

relation over $\mathcal{U}^{k_i}$ for some $k_i \geq 0$.[1] A *database update* U is a mapping from one database, $B = \langle R_1, \ldots, R_n \rangle$, to another, $U(B) = \langle R'_1, \ldots, R'_n \rangle$, where each R_i and R'_i have the same arity. We require the mapping to be partially recursive and C-generic for some finite set of constants C. (Essentially C-generic means that constants are not interpreted specially. See [AV90] for a more detailed explanation of the C-genericity condition.)

A *formula* (respectively, *sentence*) is a first-order, function-free formula (resp., sentence) in some fixed language. If R, R_1, ... are relations, then r, r_1, ... are the corresponding predicate symbols in the language. A formula is also known as a *query*. An *integrity constraint* is a sentence.

A *valuation* for a formula F is a mapping of the free variables of F to elements of $\mathcal{U}$. For a database B, formula F (resp., sentence G) and valuation ν for F, $B \models_\nu F$ (resp., $B \models G$) is defined as usual in model theory to mean F (resp., G) is true in B with respect to the valuation ν. We say a valuation ν' extends another valuation ν ($\nu' \geq \nu$) if the domain of ν' is a superset of the domain of ν and the restriction of ν' to the domain of ν is identical to ν. That is, ν' extends ν if it maps at least the same set of variables to the same set of values.

A sentence F is a *precondition* for an update U and a sentence H if, for every database B, $B \models F$ implies $U(B) \models H$.

A precondition F for an update U and a sentence H is a *weakest precondition* for U and H if, for every precondition G for U and H, and for every database B, $B \models G$ implies $B \models F$.

For later use in describing the effects of statements in our update language, we introduce the following notation and two simple lemmas. Let R be a relation in a database B, and $\Phi(\overline{x})$ a first-order formula with free variables $\overline{x} = x_1, \ldots, x_n$. Then $B' = B[R \mapsto R']$ denotes the result of replacing the relation R in B by R'. In practice, R' is either $R \cup \{\overline{x} \mid B \models \Phi(\overline{x})\}$ or $R - \{\overline{x} \mid B \models \Phi(\overline{x})\}$, i.e. $\Phi(\overline{x})$ is evaluated in the current database B and the resulting tuples are added to (resp., deleted from) R.

Lemma 2.1 Corresponding to extending the relation R by the results of the query $\Phi(\overline{x})$ we have $B[R \mapsto R \cup \{\overline{x} \mid B \models \Phi(\overline{x})\}] \models_\nu r(\overline{x})$ if and only if $B \models_\nu r(\overline{x}) \vee \Phi(\overline{x})$.

---

[1] $\mathcal{U}^{k_i} = \overbrace{\mathcal{U} \times \cdots \times \mathcal{U}}^{k_i}$

Lemma 2.2 Corresponding to reducing the relation R by the results of the query $\Phi(\overline{x})$ we have $B[R \mapsto R - \{\overline{x} \mid B \models \Phi(\overline{x})\}] \models_\nu r(\overline{x})$ if and only if $B \models_\nu r(\overline{x}) \wedge \neg\Phi(\overline{x})$.

The proofs of these lemmas are trivial.

2.2 Update Language

We now turn to our update language. What we are looking for is a language sufficiently expressive to be useful, yet not so powerful that we cannot reason about it effectively. In [AV87a, AV87b, AV88b, AV88a, AV90], Abiteboul and Vianu investigate the relative expressive power and properties of a number of database update languages. The languages varied depending on whether they supported bounded or unbounded iteration, were deterministic or non-deterministic, and allowed tuple invention or not.

To illustrate, the number of iterations in a bounded loop is fixed, and known when the loop is entered, whereas the number of iterations an unbounded loop performs is dependent on the operations performed by the loop's body during its execution. An unbounded loop does not necessarily loop indefinitely, although that is certainly a possibility. More importantly, unbounded loops introduce the possibility of non-determinism since the order of iteration over a set may be indeterminate and the loop's body may affect the loop condition, as with the following code:

```
while (p(X) ∧ ¬ q(a)) do
    insert q(X)
```

It might be expected that this code would simply add all elements except a of the relation P to the relation Q, but this is only the case if neither P nor Q contain the element a. If P does contain a (and Q does not), then the set of elements actually added to Q will depend on the order in which the updates are performed.

Our goal is to perform as much optimisation of the integrity constraint check at compile time as possible. This means we cannot assume anything about the contents of the database other than that the integrity constraints are satisfied before the update is performed. Hence we choose

an update language that is deterministic and has bounded iteration. Using bounded iteration means that any program in the language can only make a finite number of changes to the database state and it is this finiteness that simplifies our task of reasoning about the effect of the update programs and their interaction with the integrity constraints.

Our update language is adapted from Wallace [Wal91]. The language is syntactically simpler but can be shown to have the same expressive power when restricted to relational databases [Law92]. The notation we use is intended to capture the semantics of our language succinctly and minimally. It is not intended to provide a usable and expressive means for users to write updates in practice.

$$
\begin{array}{lll}
\textit{program} & ::= & \textit{defns}\,;\ \textit{stmts} \\
\textit{defns} & ::= & \textit{defn} \\
 & \| & \textit{defn}\,;\ \textit{defns} \\
\textit{defn} & ::= & \texttt{define}\ U(\overline{x})\ \texttt{as}\ (\ \textit{stmts}\ ) \\
\textit{stmts} & ::= & \textit{stmt} \\
 & \| & \textit{stmt}\,;\ \textit{stmts} \\
 & \| & \texttt{if}\ \Phi\ \texttt{then}\ (\ \textit{stmts}\ ) \\
 & \| & \texttt{foreach}\ \overline{x} : \Phi(\overline{x})\ \texttt{do}\ (\ \textit{stmts}\ ) \\
\textit{stmt} & ::= & \texttt{insert}_R(\overline{x}) \\
 & \| & \texttt{delete}_R(\overline{x}) \\
 & \| & \texttt{replace}_R(\overline{x},\overline{y})(\overline{x},\overline{z}) \\
 & \| & U(\overline{x})
\end{array}
$$

Figure 2.1: BNF of the update language

Figure 2.1 gives a BNF description of an SQL-like concrete syntax for our update language[2]. Note that at the top level, all variables in the *stmts* section of a *program* must be quantified, and that no recursion is allowed.

Example 2.1 To give a salary increase of 10% to all employees who earn over $10,000 we could write the following statement sequence.

$$
\begin{array}{l}
\texttt{foreach}\ e, s, \textit{snew} : \textit{emp}(e,s) \wedge s > 10000 \wedge \textit{snew} = s \times 1.1\ \texttt{do}\ (\\
\quad \texttt{replace}_{\textit{emp}}(e,s)(e,\textit{snew}) \\
)
\end{array}
$$

[2]This is a variation of the syntax first used in [LTW93], but is semantically equivalent to that language.

Note that the fragment $snew = s \times 1.1$ should be viewed as a constraint and not an assignment statement. □

Example 2.2 Alternatively, we could define a giveRaise update which gives the employee '*e*' a raise of '*p*' percent with the following:

$$
\begin{array}{l}
\texttt{define}\ giveRaise(e,p)\ \texttt{as}\ (\\
\quad \texttt{foreach}\ e, s, snew : emp(e,s) \wedge snew = s \times (1+p)\ \texttt{do}\ (\\
\quad\quad \texttt{replace}_{emp}(e,s)(e, snew)\\
\quad)\\
)
\end{array}
$$

We could then invoke it with:

$$
\begin{array}{l}
\texttt{foreach}\ e, s : emp(e,s) \wedge s > 10000\ \texttt{do}\ (\\
\quad \texttt{giveRaise}(e, 0.1)\\
)
\end{array}
$$

□

While it is not strictly a syntactic constraint, the use of Φ rather than $\Phi(\overline{x})$ in the above BNF indicates that the condition of an *if* statement cannot contain free variables. Also, the variables $\overline{x}$ in a replace statement are *not* required to correspond to the key of the relation R being updated.

Special attention must be given to the semantics of the bounded iteration construct `foreach`. In particular, if *stmt* is of the form

$$\texttt{foreach}\ \overline{x} : \Phi(\overline{x})\ \texttt{do}\ (\ stmt_1\ ; \cdots ;\ stmt_k\ )$$

then it is evaluated as follows: $\Phi(\overline{x})$ is evaluated to generate a set of bindings for $\overline{x}$. Then $stmt_1$ is executed for each $\overline{x}$ binding, then $stmt_2$ is executed for each $\overline{x}$ binding, and so on.

Hence, if the relation P is empty, and the query $q(x,y)$ produces the set of bindings $\{\langle a, 1\rangle,\ \langle b, 2\rangle\}$, then the update

$$\texttt{foreach}\ x, y :\ q(x,y)\ \texttt{do}\ (\ \texttt{insert}_P(x);\ \texttt{delete}_P(y)\ )$$

is equivalent to

$$\texttt{insert}_P(a);\ \texttt{insert}_P(b);\ \texttt{delete}_P(1);\ \texttt{delete}_P(2)$$

or

$$\texttt{insert}_P(b);\ \texttt{insert}_P(a);\ \texttt{delete}_P(2);\ \texttt{delete}_P(1)$$

and *not*

$$\texttt{insert}_P(a);\ \texttt{delete}_P(1);\ \texttt{insert}_P(b);\ \texttt{delete}_P(2)$$

or

$$\texttt{insert}_P(b);\ \texttt{delete}_P(2);\ \texttt{insert}_P(a);\ \texttt{delete}_P(1)$$

This choice of semantics ensures that our language is deterministic and independent of the ordering of the set of bindings. If we were to interleave the execution of the statements making up the body of the `foreach` then the ordering of the set of bindings could influence the semantics of the `foreach` statement.

For example, if the bindings above were instead $\{\langle a, b\rangle,\ \langle b, a\rangle\}$, then the update would result in no change to P. On the other hand, an interleaving semantics would result in non-determinism which would result in either the insertion of $P(a)$ or $P(b)$ depending on the order of the tuples in the result set.

We now introduce a semantically equivalent abstract syntax into which any (non-recursive) update written in the above concrete syntax can be translated. This abstract syntax has the advantage that it is much simpler and does not contain any redundancy; the `if` statement in the concrete syntax is a degenerate case of a `foreach` statement with a condition that contains no free variables. Subsequently, we will use this abstract syntax as our update language since its simplicity makes it easier and more concise to manipulate formally. The concrete syntax is used in examples where its syntactic richness leads to shorter specifications and greater readability.

Let R be a named relation and $\Phi(\overline{x})$ a first-order formula with free variables $\overline{x}$. Then $\forall\overline{x}\ (\Phi(\overline{x}) \rightarrow +r(\overline{x}))$, and $\forall\overline{x}\ (\Phi(\overline{x}) \rightarrow -r(\overline{x}))$ are statements in our language which respectively insert into and delete from the relation R. If $S_1, \ldots, S_n$ are each statements in our language then $(S_1\ ;\ \ldots\ ;\ S_n)$ is a statement in our language. For example, adding the tuple $\overline{a}$ to the relation R would be written $\forall\overline{x}\ (\overline{x} = \overline{a} \rightarrow +r(\overline{x}))$. Sometimes we will abbreviate this statement as +r($\overline{a}$).

The translation from the concrete syntax to the abstract syntax is given

as follows. Where primitive statements of the form

$$\texttt{insert}_R(\overline{a}),$$
$$\texttt{delete}_R(\overline{a}), \text{ and}$$
$$\texttt{replace}_R(\overline{a}, \overline{b})(\overline{a}, \overline{c})$$

do not occur inside an enclosing `foreach`, we replace them with

$$\forall \overline{x}\ (\overline{x} = \overline{a} \rightarrow +r(\overline{x})),$$
$$\forall \overline{x}\ (\overline{x} = \overline{a} \rightarrow -r(\overline{x})), \text{ and}$$
$$\forall \overline{x}\ (\overline{x} = \overline{a} \wedge \overline{y} = \overline{b} \wedge \overline{z} = \overline{c} \rightarrow -r(\overline{x}, \overline{y})\ ;\ +r(\overline{x}, \overline{z}))$$

respectively.

Where primitive statements of the form

$$\texttt{insert}_R(\overline{x}),$$
$$\texttt{delete}_R(\overline{x}), \text{ and}$$
$$\texttt{replace}_R(\overline{x}, \overline{y})(\overline{x}, \overline{z})$$

do occur inside an enclosing `foreach`, we replace them with

$$+r(\overline{x}),$$
$$-r(\overline{x}), \text{ and}$$
$$-r(\overline{x}, \overline{y})\ ;\ +r(\overline{x}, \overline{z})$$

respectively.

Next we translate all occurrences of statements of the form

$$\texttt{foreach}\ \overline{x} : \Phi(\overline{x})\ \texttt{do}\ (\ \textit{stmts}\ )$$

to

$$\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow \textit{stmts})$$

and all occurrences of statements of the form

$$\text{if } \Phi \text{ then } (\ \textit{stmts}\ )$$

to

$$\forall\ (\Phi \rightarrow \textit{stmts})$$

This includes any occurrences of these statement forms inside parameterised update definitions.

Then, we replace all statements of the form

$$\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow U(\overline{x}))$$

where $U(\overline{x})$ is a parameterised update whose translated definition is of the form[3]

$$\texttt{define}\ U(\overline{x})\ \texttt{as}$$
$$(\ \forall \overline{x}_1\ (\Phi_1(\overline{x}_1) \rightarrow \text{stmt}_1)\ ; \cdots ;\ \forall \overline{x}_k\ (\Phi_k(\overline{x}_k) \rightarrow \text{stmt}_k)\ )$$

by the sequence of statements

$$\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow +t(\overline{x}));$$
$$\forall \overline{x}, \overline{x}_1\ (t(\overline{x}) \wedge \Phi_1(\overline{x}_1) \rightarrow \text{stmt}_1);$$
$$\cdots;$$
$$\forall \overline{x}, \overline{x}_k\ (t(\overline{x}) \wedge \Phi_k(\overline{x}_k) \rightarrow \text{stmt}_k)$$

where T is an unused, empty temporary relation. We use the temporary relation to cache the result of the evaluation of $\Phi(\overline{x})$which avoids re-evaluation and isolates the effects any changes made by $\text{stmt}_1, \ldots, \text{stmt}_k$ may have on $\Phi(\overline{x})$. Use of a temporary relation is possible because recursion is not allowed.

Finally, we repeatedly replace all statements of the form

$$\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow \text{stmt}_1\ ; \cdots ;\ \text{stmt}_k)$$

where stmt_i is of the form

$$\forall \overline{x}_i\ (\Phi_i(\overline{x}_i) \rightarrow \textit{stmts})$$

by the sequence of statements

$$\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow +t(\overline{x}));$$
$$\forall \overline{x}, \overline{x}_1\ (t(\overline{x}) \wedge \Phi_1(\overline{x}_1) \rightarrow \textit{stmts});$$
$$\cdots;$$
$$\forall \overline{x}, \overline{x}_k\ (t(\overline{x}) \wedge \Phi_k(\overline{x}_k) \rightarrow \textit{stmts})$$

again T is an unused, empty temporary relation.

This translation process will result in an update corresponding to our abstract syntax.

---

[3]Note that none of the variables in $\overline{x} = x_1, \ldots, x_n$ occur in $\overline{x}_i = x_{i,1}, \ldots, x_{i,m}$ nor do any of the variables in $\overline{x}_i$ occur in $\overline{x}_j = x_{j,1}, \ldots, x_{j,l}$ where $1 \leq i < j \leq k$.

Example 2.3 Translating Example 2.1 showing how to give a salary increase of 10% to all employees who earn over $10,000 we get the following abstract syntax version.

$$\forall e, s\ (emp(e, s) \wedge s > 10000 \rightarrow +t(e, s));$$
$$\forall e, s\ (t(e, s) \rightarrow -emp(e, s));$$
$$\forall e, s, snew\ (t(e, s) \wedge snew = s \times 1.1 \rightarrow +emp(e, snew))$$

where T is a suitable, initially empty, temporary relation. □

Having introduced the syntax of our language, we now turn to its semantics. Recall that an update is a mapping from one database state to another and that there are many different programs that correspond to any particular mapping. In the following we formally define the *effect* of a statement in our language and show that it does indeed define a mapping.

The effect of a statement in our language is given informally as follows. The statement $\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow +r(\overline{x}))$ (resp., $\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow -r(\overline{x}))$) is executed by evaluating $\Phi(\overline{x})$ to produce a set of valuations for $\overline{x}$ and then adding to (resp., deleting from) R each $\overline{x}$ tuple. The statement $(S_1 ; \ldots ; S_n)$ is executed by first executing S_1, then executing S_2, and so on.

The *effect* of S, that is, the update defined by a statement S in our update language, is denoted $[\![S]\!]$ and is given as follows.

1. $[\![\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow +r(\overline{x}))]\!](B) = B[R \mapsto R \cup \{\overline{x} \mid B \models \Phi(\overline{x})\}]$

2. $[\![\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow -r(\overline{x}))]\!](B) = B[R \mapsto R - \{\overline{x} \mid B \models \Phi(\overline{x})\}]$

3. $[\![S_1 ; \ldots ; S_n]\!]$(B) = $[\![S_n]\!](\ldots[\![S_1]\!](B)\ldots)$

where B is a database containing relation R.

Clearly, for every statement S, the mapping $[\![S]\!]$ is a database update.

2.3 Constraint Transformation

In the context of program derivation, Dijkstra [Dij75] defined a predicate transformer wp as follows. For any command S and predicate H, $wp(S, H)$ denotes another predicate F that is the weakest precondition for $[\![S]\!]$ and H. That is, for any predicate G such that execution of S in a

state satisfying G guarantees that S will terminate in a state satisfying H, $G \Rightarrow wp(S, H)$.

Similarly, in our database context, we define a constraint transformer wp for a statement S and a formula H that is a weakest precondition for $[\![S]\!]$ and H. Informally, execution of S in a state satisfying $wp(S, H)$ guarantees that S will terminate in a state satisfying H. The following lemma (2.3) gives a necessary and sufficient condition for a formula to be a weakest precondition with respect to an update and an integrity constraint.

Lemma 2.3 *Let the update S be a statement and the integrity constraint H be a sentence. Then a sentence F is a weakest precondition for $[\![S]\!]$ and H if and only if, for every database B, $B \models F$ if and only if $[\![S]\!](B) \models H$.*

Proof Suppose that, for every database B, we have $B \models F$ if and only if $[\![S]\!](B) \models H$. Clearly, F is a precondition for $[\![S]\!]$ and H. Further, for all databases B and preconditions G for $[\![S]\!]$ and H, $B \models G$ implies $[\![S]\!](B) \models H$, as G is a precondition for $[\![S]\!]$ and H, and hence, by assumption, $B \models F$. That is, F is a weakest precondition for $[\![S]\!]$ and H.

Conversely, suppose that F is a weakest precondition for $[\![S]\!]$ and H and that there exists a database B for which $B \models F$ but $[\![S]\!](B) \not\models H$. This would imply that F is not a precondition for $[\![S]\!]$ and H and therefore neither is it a weakest precondition which violates our supposition.

Now, suppose instead there exists a database B for which $[\![S]\!](B) \models H$ but $B \not\models F$. Since F is a weakest precondition, there must exist a precondition G for $[\![S]\!]$ and H such that $B \models G$. But, if F is a weakest precondition for $[\![S]\!]$ and H, then $B \models G$ implies $B \models F$ which violates our alternative supposition.

Hence, if F is a weakest precondition for $[\![S]\!]$ and H, then $B \models F$ if and only if $[\![S]\!](B) \models H$. □

We introduce the following notation to enhance the readability of the proofs below. Let H be a (possibly open) formula, Φ a formula with k free variables and r a k-ary predicate symbol, for some $k \geq 0$. We will write $H[r \mapsto r \cup \Phi]$ (resp., $H[r \mapsto r - \Phi]$) for the formula resulting from the replacement of every occurrence of $r(\overline{s})$ in H by $(r(\overline{s}) \vee \Phi(\overline{s}))$ (resp., $(r(\overline{s}) \wedge \neg\Phi(\overline{s}))$).

Let S be a statement and H a (possibly open) formula. Then the *condition transformer* $wp(S, H)$ is defined as follows:

1. $wp(\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow +r(\overline{x})), H) = H[r \mapsto r \cup \Phi]$

2. $wp(\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow -r(\overline{x})), H) = H[r \mapsto r - \Phi]$

3. $wp((S_1 \ ; \ \ldots \ ; \ S_n), H) = wp((S_1 \ ; \ \ldots \ ; \ S_{n-1}), wp(S_n, H)), n > 1$

Note that these equations *define* wp, and not the semantics of statements as in [Gri81]. In our case, the semantics of statements were given by the definition of $[\![S]\!]$ on page 15.

This definition gives us the following algorithm for generating $wp(S, H)$ from a concrete syntax representation of an update S and an integrity constraint H:

1. Construct the abstract syntax representation of the concrete syntax update. This will be of the form $S_1; \cdots; S_n$.

2. Let $C = H$ and $i = n$.

3. Construct $wp(S_i, C)$ using the definition 1 or 2, above, as appropriate.

4. Let $i = i - 1$ and $C =$ the result of the previous step.

5. If $i > 0$, then go back to step 3.

The final value of C is $wp(S, H)$.

The following example illustrates how to construct $wp(S, H)$ according to the definition above.

Example 2.4 Consider the example constraint 1.2 from Example 1.1 in Chapter 1 which states that there can never be less than one of any item in stock, and an update that sets the quantity of barbie dolls in stock to 10. Let H be the constraint $\forall x, y\ (inventory(x, y) \rightarrow y > 0)$, and S be the statement $S_1 \ ; \ S_2$, where S_1 is the statement

$$\forall x, y\ (x = \text{'barbie'} \wedge inventory(x, y) \rightarrow -inventory(x, y))$$

and S_2 is the statement

$$\forall x, y\ (x = \text{'barbie'} \wedge y = 10 \rightarrow +inventory(x, y))$$

Then $wp(S, H)$ is constructed as follows:

$$\begin{aligned}
wp(S,H) &= wp(S_1; S_2, H) \\
&= wp(S_1, wp(S_2, H)) \\
&= wp(S_1, \forall x, y\ ((inventory(x,y) \vee \\
&\qquad\qquad (x = \text{'barbie'} \wedge y = 10)) \rightarrow y > 0)) \\
&= \forall x, y\ (((inventory(x,y) \wedge \\
&\qquad \neg(x = \text{'barbie'} \wedge inventory(x,y))) \vee \\
&\qquad (x = \text{'barbie'} \wedge y = 10)) \rightarrow y > 0)
\end{aligned}$$

□

We now present a series of lemmas leading to a proof that the predicate transformer $wp(S, H)$ is the weakest precondition for $[\![S]\!]$ and H (Theorem 2.1).

Lemma 2.4 *Let S be the statement $\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow +r(\overline{x}))$, H a formula with free variables $\overline{z}$, and ν a valuation for $\overline{z}$. Then, for every database B, $B \models_\nu wp(S, H)$ if and only if $[\![S]\!](B) \models_\nu H$.*

Proof The proof is by induction on the structure of H. Lemma 2.1 is used in the base case, $H = r(\overline{x})$. Simple induction is used when $H = H_1 \wedge H_2$, $H = H_1 \vee H_2$, or $H = \neg H_1$. The only nontrivial case is $H = \forall \overline{y}\ H'$, which is given in full below. (By renaming, we can assume that $\overline{x}$ and $\overline{y}$ are disjoint.)

$$\begin{aligned}
& B \models_\nu wp(S, \forall \overline{y}\ H') \\
\iff\ & B \models_\nu (\forall \overline{y}\ H')[r \mapsto r \cup \Phi] && \text{(definition of } wp\text{)} \\
\iff\ & \text{for all } \nu' \geq \nu,\ B \models_{\nu'} H'[r \mapsto r \cup \Phi] && \text{(semantics of } \forall\text{)} \\
\iff\ & \text{for all } \nu' \geq \nu,\ B \models_{\nu'} wp(S, H') && \text{(definition of } wp\text{)} \\
\iff\ & \text{for all } \nu' \geq \nu,\ [\![S]\!](B) \models_{\nu'} H' && \text{(induction hyp.)} \\
\iff\ & [\![S]\!](B) \models_\nu \forall \overline{y}\ H' && \text{(semantics of } \forall\text{)}
\end{aligned}$$

□

Lemma 2.5 *Let S be the statement $\forall \overline{x}\ (\Phi(\overline{x}) \rightarrow -r(\overline{x}))$, H a formula with free variables $\overline{z}$, and ν a valuation for $\overline{z}$. Then, for every database B, $B \models_\nu wp(S, H)$ if and only if $[\![S]\!](B) \models_\nu H$.*

Proof The same as for Lemma 2.4, but using Lemma 2.2 in place of Lemma 2.1. □

Lemma 2.6 *Let S be the sequential statement $(S_1 \; ; \; \dots \; ; \; S_n)$ and H a sentence. Then, for every database B, $B \models wp(S, H)$ if and only if $[\![S]\!](B) \models H$.*

Proof Assume inductively that, for every database B, sentence H', and statement sequence of the form $S_1 \; ; \; \dots \; ; \; S_i$, where $1 \leq i < n$, we have $B \models wp((S_1 \; ; \; \dots \; ; \; S_i), H')$ if and only if $[\![S_1 \; ; \; \dots \; ; \; S_i]\!](B) \models H'$. Then

$$
\begin{aligned}
& B \models wp((S_1 \; ; \; \dots \; ; \; S_n), H) \\
& \qquad \Longleftrightarrow \quad B \models wp((S_1 \; ; \; \dots \; ; \; S_{n-1}), wp(S_n, H)) && \text{(definition of } wp\text{)} \\
& \qquad \Longleftrightarrow \quad [\![S_1 \; ; \; \dots \; ; \; S_{n-1}]\!](B) \models wp(S_n, H)) && \text{(induction hyp.)} \\
& \qquad \Longleftrightarrow \quad [\![S_n]\!]([\![S_1 \; ; \; \dots \; ; \; S_{n-1}]\!](B)) \models H && \text{(Lemma 2.4)} \\
& \qquad \Longleftrightarrow \quad [\![S_1 \; ; \; \dots \; ; \; S_n]\!](B) \models H && \text{(definition of update)}
\end{aligned}
$$

Lemmas 2.4 and 2.5 (with ν empty) provide the base cases. □

We are now in a position to prove that the condition transformer wp actually produces a condition which is the weakest precondition with respect to a particular update statement and condition. That is, $wp(S, H)$ produces a condition such that, if it is true before executing the statement S, then the condition H is guaranteed to be true afterwards.

Theorem 2.1 *Let S be a statement, and H a sentence. Then $wp(S, H)$ is the weakest precondition for $[\![S]\!]$ and H.*

Proof The result follows immediately from Lemmas 2.3, 2.4, 2.5, and 2.6. □

To apply Theorem 2.1 we construct $wp(S, H)$ and evaluate it before performing the statement S. More explicitly, if S is an update statement and H is an integrity constraint, Theorem 2.1 says constructing $wp(S, H)$ gives us the weakest precondition of the update statement and the integrity constraint. By making the performance of the update S conditional on $wp(S, H)$ holding in the current database state, we ensure that the integrity constraint H will not be violated.

We now present several examples of common constraints and show how to construct the weakest precondition for a variety of updates.

We begin with *functional dependencies*. Let P be a relation of arity three where attribute a_1 functionally determines the attribute a_3 We would write this constraint, C_1, as

$$\forall\, x, y_0, y_1, z_0, z_1\ ((p(x, y_0, z_0) \wedge p(x, y_1, z_1)) \rightarrow z_0 = z_1)$$

Example 2.5 Let U be the update $+p(a, b, c)$ or, more formally,

$$\forall x, y, z\ ((x, y, z = a, b, c) \rightarrow +p(x, y, z))$$

where $(x, y, z = a, b, c)$ is shorthand for $(x = a \wedge y = b \wedge z = c)$. Then,

$$
\begin{aligned}
& wp(U, C_1) \\
& \quad = \quad C_1[P \mapsto P \cup (x, y, z = a, b, c)] \\
& \quad \equiv \quad \forall\, x, y_0, y_1, z_0, z_1\ (((p(x, y_0, z_0) \vee (x, y_0, z_0 = a, b, c)) \wedge \\
& \qquad\qquad\qquad (p(x, y_1, z_1) \vee (x, y_1, z_1 = a, b, c))) \rightarrow z_0 = z_1) \\
& \quad \equiv \quad \forall\, x, y_0, y_1, z_0, z_1\ (((p(x, y_0, z_0) \wedge \\
& \qquad\qquad\qquad (p(x, y_1, z_1) \vee (x, y_1, z_1 = a, b, c))) \vee \\
& \qquad\qquad\qquad ((x, y_0, z_0 = a, b, c) \wedge \\
& \qquad\qquad\qquad (p(x, y_1, z_1) \vee (x, y_1, z_1 = a, b, c)))) \rightarrow z_0 = z_1) \\
& \quad \equiv \quad \forall\, x, y_0, y_1, z_0, z_1\ (((p(x, y_0, z_0) \wedge p(x, y_1, z_1)) \vee \\
& \qquad\qquad\qquad ((x, y_0, z_0 = a, b, c) \wedge p(x, y_1, z_1)) \vee \\
& \qquad\qquad\qquad (p(x, y_0, z_0) \wedge (x, y_1, z_1 = a, b, c)) \vee \\
& \qquad\qquad\qquad ((x, y_0, z_0 = a, b, c) \wedge (x, y_1, z_1 = a, b, c))) \rightarrow z_0 = z_1) \\
& \quad \equiv \quad \forall\, x, y_0, y_1, z_0, z_1\ (((p(x, y_0, z_0) \wedge p(x, y_1, z_1)) \rightarrow z_0 = z_1) \wedge \\
& \qquad\qquad\qquad (((x, y_0, z_0 = a, b, c) \wedge p(x, y_1, z_1)) \rightarrow z_0 = z_1) \wedge \\
& \qquad\qquad\qquad ((p(x, y_0, z_0) \wedge (x, y_1, z_1 = a, b, c)) \rightarrow z_0 = z_1) \wedge \\
& \qquad\qquad\qquad (((x, y_0, z_0 = a, b, c) \wedge (x, y_1, z_1 = a, b, c)) \rightarrow z_0 = z_1)) \\
& \quad \equiv \quad C_1 \wedge \forall\, y_0, z_0\ (p(a, y_0, z_0) \rightarrow z_0 = c)
\end{aligned}
$$

Hence, if $C_1 \wedge \forall\, y_0, z_0\ (p(a, y_0, z_0) \rightarrow z_0 = c)$ holds in the current database state, then the update U is safe. That is, it cannot violate the integrity constraint C_1. □

Example 2.6 Let U be the update $-p(a, b, c)$ or, more formally,

$$\forall x, y, z\ ((x, y, z = a, b, c) \rightarrow -p(x, y, z))$$

where $(x, y, z = a, b, c)$ is shorthand for $(x = a \wedge y = b \wedge z = c)$. Then,

$$
\begin{aligned}
& wp(U, C_1) \\
& \quad = \quad C_1[P \mapsto P - (x, y, z = a, b, c)] \\
& \quad \equiv \quad \forall\, x, y_0, y_1, z_0, z_1 \\
& \qquad\qquad (((p(x, y_0, z_0) \wedge \neg(x, y_0, z_0 = a, b, c)) \wedge \\
& \qquad\qquad\quad (p(x, y_1, z_1) \wedge \neg(x, y_1, z_1 = a, b, c))) \rightarrow z_0 = z_1) \\
& \quad \equiv \quad \forall\, x, y_0, y_1, z_0, z_1 \\
& \qquad\qquad ((p(x, y_0, z_0) \wedge p(x, y_1, z_1) \wedge \\
& \qquad\qquad\quad \neg((x, y_0, z_0, y_1, z_1 = a, b, c, b, c))) \rightarrow z_0 = z_1) \\
& \quad \Leftarrow \quad C_1
\end{aligned}
$$

Hence, if C_1 holds in the current database state, then the update U is safe. □

Of more interest, since they involve more than one relation, are the *inclusion dependencies* or *referential constraints*. Let P and Q be relations of arity two where the projection onto a_1 of P is required to be a subset of the projection onto a_1 of Q. We would write this constraint, C_2, as

$$\forall\, x, y\ (p(x, y) \rightarrow \exists z\ q(x, z))$$

This may arise if, for example, P stored actors staring in a film and Q the year in which a film was made. Clearly an actor cannot have starred in a film if it was never made.[4]

Example 2.7 Let U be the update $\forall x, y\ (r(x, y) \rightarrow +p(x, y))$ Then,

$$
\begin{aligned}
& wp(U, C_2) \\
& \quad = \quad C_2[P \mapsto P \cup r(x, y)] \\
& \quad \equiv \quad \forall\, x, y\ ((p(x, y) \vee r(x, y)) \rightarrow \exists z\ q(x, z)) \\
& \quad \equiv \quad \forall\, x, y\ (p(x, y) \rightarrow \exists z\ q(x, z)) \wedge \forall\, x, y\ (r(x, y) \rightarrow \exists z\ q(x, z)) \\
& \quad \equiv \quad C_2 \wedge \forall\, x, y\ (r(x, y) \rightarrow \exists z\ q(x, z))
\end{aligned}
$$

Hence, if $C_2 \wedge \forall\, x, y\ (r(x, y) \rightarrow \exists z\ q(x, z))$ holds in the current database state, then the update U is safe. □

[4] On the other hand, there are many films (such as documentaries) that have no actors.

A discussion of how to simplify $wp(U, C)$ once it is generated is deferred until Chapter 5.

2.4 Related Work

We now summarise the contributions, strengths and weaknesses of related work on relational integrity constraint checking.

Nicolas [Nic82] presents a method which considers an update as two disjoint sets of tuples (*deltas*), one set being the tuples to insert into the database, the other set being those to be deleted. The approach requires that the updates be performed to generate the new database state. By appropriate instantiation of the integrity constraints with the tuples from the deltas, specialised constraints can be formed which are cheaper to check in the updated database than the original integrity constraints because the specialised constraints avoid checking what is already known to be true.

Hsu and Imielinski [HI85] examine multiple-relation updates as captured by deltas and look at the constraint quantifier patterns in an attempt to transform the constraint into a combination of "simpler" constraints which are easier to check.

In contrast, McCune and Henschen [HMN84] and Stemple et al. [SMS87] and Sheard et al. [SS88, SS89] all consider update *programs* rather than the resulting sets of tuples to be inserted and deleted.

McCune and Henschen also generate tests to perform before the update and describe a method related to ours, as described below. Their work is limited in that the update language they consider is less expressive than ours, and the tests they generate are preconditions rather than weakest preconditions. This means that if the test succeeds then the update is safe, but if it fails, they still need to check the original constraint after performing the update.

Stemple, Mazumdar and Sheard [SMS87] show how to generate necessary and sufficient conditions (i.e., a weakest precondition) for checking transaction safety using a heuristic, lemma-driven theorem prover. Their constraint language is more expressive than ours since it allows aggregates and their update language appears to be equivalent in power to ours but it is very difficult to determine this since it is neither formally

described nor very regular. Our predicate transformer provides a more direct mechanism for generating a weakest precondition than their theorem prover.

In subsequent work, Sheard and Stemple [SS88, SS89] attempt to prove that a particular update program *cannot* violate the database's integrity constraints. That is, they attempt to prove that if the database's integrity constraints hold, then the weakest precondition must also hold.

Their use of a theorem prover makes it difficult to quantify the effectiveness and extensibility of their approach. By their own admission, the axiom set used by the theorem prover is very brittle and must be chosen very carefully with a detailed understanding of how the update statements interact with the constraints.

A more detailed review of earlier relational integrity constraint checking work can be found in [SS88].

Benedikt et al. [BGL96] investigate the question of which transaction languages admit first-order weakest preconditions with respect to first-order integrity constraints. They show that the class of transactions that exactly admits first-order weakest preconditions cannot be captured. This contrasts with our approach in that, when restricted to first-order constraints, we also restrict our transactions to those that admit first-order weakest preconditions. For the more powerful transaction languages considered in subsequent chapters, we also allow more powerful integrity constraints, and consequentially more powerful weakest preconditions.

2.5 Summary

In this chapter we introduced a deterministic, set-oriented update language for relational databases. We then applied concepts from the field of program derivation to define a predicate transformer wp for relational database updates specified in this language. We proved that this predicate transformer did indeed generate the weakest precondition with respect to an update and a constraint. From an update U and this weakest precondition $wp(U, C)$, we can then generate a new update $wp(U, C) \rightarrow U$ which we know cannot violate the database's integrity constraints C. This is advantageous since transactions which are safe in this sense do not need to be rolled back due to integrity constraint violation, nor do they require

write locks be held at the end of the transaction while potentially expensive constraint checks are performed.

Furthermore, the predicate transformer can be used for feedback to the programmer writing the original update U. If we can determine that $wp(U, C)$ holds for all databases, then we know the original update U is safe. If not, then the weakest precondition itself indicates to the programmer the conditions under which the update can fail to respect the integrity constraints.

Chapter 3

Deductive Databases

In order to handle a larger class of queries, deductive databases [GMN84, Min88] have evolved as a natural extension to relational databases. They combine the power and expressiveness of logic programming based query languages with the efficiency of relational database engines.

By defining an integrity constraint to be a pair consisting of an atom (the query goal) and a set of deductive rules (the program) defined with respect to the relations in the database, instead of a simple first-order sentence, we can extend our method to deal with deductive databases.

Section 3.1 introduces this notion formally and recasts several of the definitions and terms introduced in section 2.1 in the setting of deductive databases. Section 3.2 introduces the update language, modified from section 2.2 appropriately for deductive databases. Then section 3.3 defines the predicate transformer wp with respect to this data model and update language. Finally, section 3.4 discusses related work and is followed by a summary of the chapter in section 3.5.

The material presented in this chapter is based on joint work with Rodney Topor that first appeared in [LT95].

3.1 Data Model

Let $\mathcal{U}$ be a *universal domain* consisting of a countably infinite set of constants. A *database* is a tuple $\langle R_1, \ldots, R_n \rangle$, where each R_i is a finite named relation over $\mathcal{U}^{k_i}$ for some $k_i \geq 0$. A *database update* U is a mapping from one database, $B = \langle R_1, \ldots, R_n \rangle$, to another, $U(B) = \langle R'_1, \ldots, R'_n \rangle$, where

each R_i and R'_i have the same arity. We require the mapping to be partially recursive and C-generic for some finite set of constants C.

A *formula* (resp., *sentence*) is a first-order, function-free formula (resp., sentence) in some fixed language. If R, $R_1, \ldots$ are relations in databases, then r, $r_1, \ldots$ are the corresponding predicate symbols in the language. A *rule* is a function-free formula of the form $A \leftarrow W$ where A is an atom and W is a first-order formula. All variables in A and free variables in W are assumed to be universally quantified at the front of the formula. If r is a predicate symbol corresponding to the relation R in the database, then there cannot also be a rule defining r (with the same arity). A *program* P is a set of rules. We require a total (two-valued) well-founded model to exist for all databases and programs so we restrict programs to being locally stratified [Prz88].

A *query* is a mapping from a database to a relation. We represent a query as $(q(\overline{x}), P)$ where $q(\overline{x})$ is an atom and P is a program defining q[1]. A *condition* is a query in which the atom has no variables. An *integrity constraint* is a condition.

A *valuation* for a query $(q(\overline{x}), P)$ is a mapping of the free variables $\overline{x}$ of q to elements of $\mathcal{U}$. For a database B, query $(q(\overline{x}), P)$ and valuation ν for $\overline{x}$, $B \models_\nu (q(\overline{x}), P)$ if and only if $M_{B \cup P} \models_\nu q(\overline{x})$ where $M_{B \cup P}$ is the normal model for B and P with domain $\mathcal{U}$. We say a valuation ν' extends another valuation ν ($\nu' \geq \nu$) if the domain of ν' is a superset of the domain of ν and the restriction of ν' to the domain of ν is identical to ν.

A query (q', P') is a *precondition* for an update U and a query (q, P) if, for every database B, $B \models (q', P')$ implies $U(B) \models (q, P)$. A precondition (q', P') for U and (q, P) is a *weakest precondition* for U and (q, P) if, for every precondition (q'', P'') for U and (q, P), and for every database B, $B \models (q'', P'')$ implies $B \models (q', P')$.

A predicate symbol q *directly depends* on a predicate symbol r if r appears in the body of any rule defining q. A predicate symbol q *depends* on a predicate symbol r if q directly depends on r or on another predicate symbol that depends on r.

[1] This does not restrict the power of our query language since we consider P to be the the union of the deductive database rules and any others that may be needed to express the query. It is a notational shorthand to restrict a query to being an atom rather than allowing a formula and then having to introduce more complex definitions and rules to deal with this formula.

Let P be a program and R a relation in the database, with corresponding predicate symbol r. Let P'_r be another program consisting of the union of the set of rules in P with the following additional rules. For every rule Q in P defining a predicate symbol q that depends on r, add an identical rule defining q' (a new predicate which does not appear in P) in which every predicate symbol s in the body of Q that depends on r (as well as every occurrence of r itself) is replaced by s'. Notice that there is no definition of r' in P'_r since r corresponds to a database relation and therefore there can be no rules in P defining r.

Example 3.1 Let E be a relation with corresponding predicate symbol e, and P the following program.

$$q \leftarrow \neg p.$$
$$p \leftarrow e.$$
$$p \leftarrow e \wedge g \wedge p.$$

Then the program P'_e is

$$q \leftarrow \neg p. \qquad q' \leftarrow \neg p'.$$
$$p \leftarrow e. \qquad p' \leftarrow e'.$$
$$p \leftarrow e \wedge g \wedge p. \qquad p' \leftarrow e' \wedge g \wedge p'.$$

Intuitively, we could now add to P'_e a rule defining e' in terms of e such that e', p' and q' correspond to updated versions of e, p and q respectively without actually changing the database B. □

Note that we associate the deductive rules with individual queries. A deductive database in the traditional sense, is a special case of this where all queries share a common set of deductive rules.

3.2 Update Language

We modify our update language from Chapter 2 along the same lines as our definition of queries and integrity constraints. Whereas the set-oriented insert and delete commands of Chapter 2 ranged over the results of first-order formulae, they now range over a deductive query, possibly involving recursion. This fundamentally increases the expressive power of the update language since the expressive power of the corresponding underlying

query language has increased [Law92]. For example, we can now write an update which deletes all edges involved in paths from node A to node B in some graph.

Let R be a named relation, and $(q(\overline{x}), P)$ a query with free variables $\overline{x}$. Then $\forall\overline{x}\ ((q(\overline{x}), P) \rightarrow +r(\overline{x}))$, and $\forall\overline{x}\ ((q(\overline{x}), P) \rightarrow -r(\overline{x}))$ are statements in our language. If $S_1, \ldots, S_n$ are individual statements in our language then (S_1 ; ... ; S_n) is a statement in our language. For example, adding the tuple $\overline{a}$ to the relation R could be written $\forall\overline{x}\ ((\overline{x} = \overline{a}, \phi) \ \rightarrow\ +r(\overline{x}))$. Sometimes we will abbreviate this statement as +r($\overline{a}$).

The effect of a statement in our language is given informally as follows. The statement $\forall\overline{x}\ ((q(\overline{x}), P) \ \rightarrow\ +r(\overline{x}))$ (resp., $\forall\overline{x}\ ((q(\overline{x}), P) \ \rightarrow\ -r(\overline{x}))$) is executed by evaluating $(q(\overline{x}), P)$ to produce a set of bindings for $\overline{x}$ and then adding to (resp., deleting from) R each $\overline{x}$ tuple. The statement sequence (S_1 ; ... ; S_n) is executed by first executing S_1, then executing S_2, and so on.

Example 3.2 Consider a database with relations S and E where the tuple $s(x, y)$ indicates that employee x directly supervises employee y and the tuple $e(x, y)$ indicates that employee x earns a salary of y dollars. Let P be the following program.

$$m(x, y) \leftarrow s(x, y).$$
$$m(x, y) \leftarrow s(x, z) \wedge m(z, y).$$
$$q(x, y) \leftarrow m('\mathtt{Jane}', x) \wedge e(x, y).$$
$$r(x, z) \leftarrow t(x, y) \wedge \mathit{times}(y, 1.1, z).$$

Then, to give a salary increase of 10% to all employees supervised (directly or indirectly) by Jane, we could write the following statement sequence.

$$\forall e, s\ ((q(e, s), P) \rightarrow +t(e, s));$$
$$\forall e, s\ ((t(e, s), \phi) \rightarrow -e(e, s));$$
$$\forall e, s\ ((r(e, s), P) \rightarrow +e(e, s))$$

where T is a suitable, initially empty, temporary relation. (We do not bother removing everything from T since nothing else refers to it.) □

The update defined by a statement S in our update language is de-

noted $[\![S]\!]$ and is given inductively as follows.

1. $[\![\forall \overline{x}\ ((q(\overline{x}), P) \rightarrow +r(\overline{x}))]\!](B) = B \cup \{r(\overline{x}) \mid B \models (q(\overline{x}), P)\}$

2. $[\![\forall \overline{x}\ ((q(\overline{x}), P) \rightarrow -r(\overline{x}))]\!](B) = B - \{r(\overline{x}) \mid B \models (q(\overline{x}), P)\}$

3. $[\![S_1\ ;\ \ldots\ ;\ S_n]\!]$(B) = $[\![S_n]\!]([\![S_1\ ;\ \ldots\ ;\ S_{n-1}]\!]$(B)), $n > 1$

Clearly, for every statement S, the mapping $[\![S]\!]$ is a deductive database update.

3.3 Constraint Transformation

Lemma 3.1 is the deductive database equivalent of lemma 2.3 giving the conditions required for a query to be a weakest precondition.

Lemma 3.1 *Let the update S be a statement and the integrity constraint (q, P) a condition. Then a condition (q', P') is a weakest precondition for $[\![S]\!]$ and (q, P) if and only if, for every database B, $B \models (q', P')$ if and only if $[\![S]\!](B) \models (q, P)$.*

Proof Suppose that, for every database B, $B \models (q', P')$ if and only if $[\![S]\!](B) \models (q, P)$. Clearly, (q', P') is a precondition for $[\![S]\!]$ and (q, P). Further, for all databases B and preconditions (q'', P'') for $[\![S]\!]$ and (q, P), $B \models (q'', P'')$ implies $[\![S]\!](B) \models (q, P)$, as (q'', P'') is a precondition for $[\![S]\!]$ and (q, P), and hence, by assumption, $B \models (q', P')$. That is, (q', P') is a weakest precondition for $[\![S]\!]$ and (q, P).

Conversely, suppose that (q', P') is a weakest precondition for $[\![S]\!]$ and (q, P) and that there exists a database B for which $B \models (q', P')$ but $[\![S]\!](B) \not\models (q, P)$. This would imply that (q', P') is not a precondition for $[\![S]\!]$ and (q, P) and is therefore neither a weakest precondition which violates our supposition.

Now, suppose that there exists a database B for which $[\![S]\!](B) \models (q,\ P)$ but $B \not\models (q',\ P')$. Since (q', P') is a weakest precondition, there must exist a precondition (q'', P'') for $[\![S]\!]$ and (q, P) such that $B \models (q'', P'')$. But, if (q', P') is a weakest precondition for $[\![S]\!]$ and (q, P), then $B \models (q'', P'')$ implies $B \models (q', P')$ which violates our alternative supposition.

Hence, if (q', P') is a weakest precondition for $[\![S]\!]$ and (q, P), then $B \models (q',\ P')$ if and only if $[\![S]\!](B) \models (q, P)$. $\square$

For the new update language given in section 3.2 we now define the predicate transformer wp in the context of our deductive data model.

Let S be a statement, (q, P) a condition, and Q a program. Then the *condition transformer* $wp(S, (q, P))$ is defined inductively as follows:

1. $wp(\forall\overline{x}\ ((p(\overline{x}), Q) \rightarrow +r(\overline{x})), (q, P)) =$
$$(q', Q \cup P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \vee p(\overline{x})\})$$
2. $wp(\forall\overline{x}\ ((p(\overline{x}), Q) \rightarrow -r(\overline{x})), (q, P)) =$
$$(q', Q \cup P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \wedge \neg p(\overline{x})\})$$
3. $wp((S_1\ ;\ \ldots\ ;\ S_n), (q, P)) =$
$$wp((S_1\ ;\ \ldots\ ;\ S_{n-1}), wp(S_n, (q, P))), n > 1$$

This definition immediately gives us the following algorithm for constructing $wp(S, (q, P))$, where S is of the form $S_1; \cdots; S_n$ for $n \geq 1$:

1. Let $C = (q, P)$ and $i = n$.
2. Construct $wp(S_i, C)$ using the definition 1 or 2, above, as appropriate.
3. Let $i = i - 1$ and $C =$ the result of the previous step.
4. If $i > 0$, then go back to step 2.

The final value of C is $wp(S, (q, P))$.

Example 3.3 Let S be the statement $\forall x\ ((s(x), \phi) \rightarrow +r(x))$ and H the condition (q, P), where P is as follows.

$$q \leftarrow \neg t.$$
$$t \leftarrow r(x) \wedge \neg m(x).$$

Then,

$$\begin{aligned} wp(S, H) &= wp(S, (q, P)) \\ &= (q', P'_r) \end{aligned}$$

where P'_r is

$$\begin{array}{ll} q \leftarrow \neg t. & q' \leftarrow \neg t'. \\ t \leftarrow r(x) \wedge \neg m(x). & t' \leftarrow r'(x) \wedge \neg m(x). \\ & r'(x) \leftarrow r(x) \vee s(x). \end{array}$$

□

We now present a series of lemmas leading to a proof that $wp(S,(q,P))$ is the weakest precondition for $[\![S]\!]$ and (q,P) (Theorem 3.1).

Lemma 3.2 *Let B be a database, $(q(\overline{x}),P)$ a query where q depends on r, and ν a variable assignment. Then, $B \models_\nu (q(\overline{x}),P)$ if and only if $B \models_\nu (q'(\overline{x}), P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x})$*

Proof The proof is immediate. □

Lemma 3.3 *Let B be a database, $r(\overline{c})$ a ground atom (where r is the predicate symbol corresponding to the database relation R), and $(p(\overline{x}),P)$ a query. Then, $B \cup \{r(\overline{x}) \mid B \models (p(\overline{x}),P)\} \models r(\overline{c})$ if and only if $B \models (r'(\overline{c}), P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \vee p(\overline{x})\})$*

Proof Since there are no clauses defining r' in P'_r, we have

$$\begin{array}{lll} \multicolumn{3}{l}{B \models (r'(\overline{c}), P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \vee p(\overline{x})\})} \\ \iff & B \models (r(\overline{c}), P'_r) \text{ or } B \models (p(\overline{c}), P'_r) & \text{(only def. of } r'\text{)} \\ \iff & B \models r(\overline{c}) \text{ or } B \models (p(\overline{c}), P) & \\ \iff & B \models r(\overline{c}) \text{ or } r(\overline{c}) \in \{r(\overline{x}) \mid B \models (p(\overline{x}),P)\} & \\ \iff & B \cup \{r(\overline{x}) \mid B \models (p(\overline{x}),P)\} \models r(\overline{c}) & \end{array}$$

□

Lemma 3.4 *Let B be a database, $(r(\overline{x}),P)$ a query, and $(p(\overline{x}),Q)$ a query. Then, for all valuations ν, $B - \{r(\overline{x}) \mid B \models (p(\overline{x}),Q)\} \models_\nu (r(\overline{x}),P)$ if and only if $B \models_\nu (r'(x), Q \cup P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \wedge \neg p(\overline{x})\})$*

Proof Similar to the proof of Lemma 3.3. □

Lemma 3.5 *Let B be a database, (q,P) a condition, $(p(\overline{x}),Q)$ a query, and S the statement $\forall\overline{x}\,((p(\overline{x}),Q) \rightarrow +r(\overline{x}))$. Then, $[\![S]\!](B) \models (q,P)$ if and only if $B \models (q', Q \cup P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \vee p(\overline{x})\})$.*

Proof

$$\begin{array}{lll} \multicolumn{3}{l}{B \models (q', Q \cup P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \vee p(\overline{x})\})} \\ \iff & B \cup \{r'(\overline{x}) \mid B \models (r(\overline{x}) \vee p(\overline{x}), Q)\} \models (q', P'_r) & \\ \iff & B \cup \{r(\overline{x}) \mid B \models (r(\overline{x}) \vee p(\overline{x}), Q)\} \models & \\ & \qquad (q', P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x})\}) & \\ \iff & B \cup \{r(\overline{x}) \mid B \models (p(\overline{x}), Q)\} \models (q', P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x})\}) & \\ \iff & B \cup \{r(\overline{x}) \mid B \models (p(\overline{x}), Q)\} \models (q, P) & \text{(Lemma 3.2)} \\ \iff & [\![S]\!](B) \models (q, P) & \text{(def. of } [\![S]\!](B)\text{)} \end{array}$$

□

Lemma 3.6 *Let B be a database, (q, P) a condition, $(p(\overline{x}), Q)$ a query, and S the statement $\forall \overline{x}\ ((p(\overline{x}), Q) \rightarrow -r(\overline{x}))$. Then, $[\![S]\!](B) \models (q, P)$ if and only if $B \models (q', Q \cup P'_r \cup \{r'(\overline{x}) \leftarrow r(\overline{x}) \wedge \neg p(\overline{x})\})$.*

Proof Similar to the proof of Lemma 3.5. □

Lemma 3.7 *Let S be the sequential statement $(S_1 ; \ldots ; S_n)$, and (q, P) a query. Then, for all databases B, $B \models wp(S, (q, P))$ if and only if $[\![S]\!](B) \models (q, P)$.*

Proof Assume inductively that, for every database B, query (q', P), and statement sequence of the form $(S_1 ; \ldots ; S_i)$, where $1 \leq i < n$, then $B \models wp((S_1 ; \ldots ; S_i), (q', P))$ if and only if $[\![S_1 ; \ldots ; S_i]\!](B) \models (q', P)$. Then,

$$
\begin{array}{lll}
B \models wp((S_1 ; \ldots ; S_n), (q, P)) & & \\
\iff & B \models wp((S_1 ; \ldots ; S_{n-1}), wp(S_n, (q, P))) & \text{(defn of } wp\text{)} \\
\iff & [\![S_1 ; \ldots ; S_{n-1}]\!](B) \models wp(S_n, (q, P)) & \text{(induction hyp.)} \\
\iff & [\![S_n]\!]([\![S_1 ; \ldots ; S_{n-1}]\!](B)) \models (q, P) & \text{(induction hyp.)} \\
\iff & [\![S_1 ; \ldots ; S_n]\!](B) \models (q, P) & \text{(defn of update)} \\
\iff & [\![S]\!](B) \models (q, P) &
\end{array}
$$

Lemmas 3.5 and 3.6 provide the base cases. □

We are now in a position to prove that the condition transformer wp actually produces a condition which is the weakest precondition with respect to a particular update statement and condition. That is, $wp(S, (q, P))$ produces a condition such that, if it is true before executing the statement S, then the condition (q, P) is guaranteed to be true afterwards.

Theorem 3.1 *Let S be a statement, and (q, P) a query. Then $wp(S, (q, P))$ is the weakest precondition for $[\![S]\!]$ and (q, P).*

Proof The result follows immediately from Lemmas 3.1, 3.5, 3.6, and 3.7. □

We now present examples of common constraints and show how to construct the weakest precondition with respect to an update. The first example is of a referential integrity constraint and a sequence of two update statements.

Example 3.4 Let S be the statement $(S_1\ ;\ S_2)$ defined as

$$\forall x\ ((s(x), \phi) \rightarrow +r(x))\ ;$$
$$\forall x\ ((s(x), \phi) \rightarrow +m(x))$$

and let H be the condition (q, P), where P is defined as

$$q \leftarrow \neg t.$$
$$t \leftarrow r(x) \wedge \neg m(x).$$

Then,

$$\begin{aligned} wp(S, H) &= wp((S1; S2), H) \\ &= wp(S1, wp(S2, H)) \\ &= wp(S1, (q', P'_m)) \\ &= (q'', P'_{mr}{}') \end{aligned}$$

where P'_m is

$$q \leftarrow \neg t. \qquad q' \leftarrow \neg t'.$$
$$t \leftarrow r(x) \wedge \neg m(x). \qquad t' \leftarrow r(x) \wedge \neg m'(x).$$
$$m'(x) \leftarrow m(x) \vee s(x).$$

and $P'_{mr}{}'$ is P'_m plus the following

$$q'' \leftarrow \neg t''.$$
$$t'' \leftarrow r'(x) \wedge \neg m'(x).$$
$$r'(x) \leftarrow r(x) \vee s(x).$$

□

The next example, an inclusion dependency, also involves a compound update but, more interestingly, includes a *recursive* integrity constraint.

Example 3.5 Let S be the statement $(S1\ ;\ S2)$ defined as

$$\forall x, y\ ((s(x, y), \phi) \rightarrow +e(x, y))\ ;$$
$$\forall x, y\ ((s(x, y), \phi) \rightarrow +r(x, y))$$

and let H be the condition (q, P), where P is defined as

$$q \leftarrow \neg t.$$
$$t \leftarrow tc(x, y) \wedge \neg r(x, y).$$
$$tc(x, y) \leftarrow e(x, y).$$
$$tc(x, y) \leftarrow e(x, z) \wedge tc(z, y).$$

That is, H is the constraint requiring that the relation R be a superset of the transitive closure of the relation E. Then,

$$\begin{aligned} wp(S,H) &= wp((S1;S2),H) \\ &= wp(S1, wp(S2,H)) \\ &= wp(S1,(q',P'_r)) \\ &= (q'',P''_{re}) \end{aligned}$$

where P'_r is

$$\begin{array}{ll} q \leftarrow \neg t. & q' \leftarrow \neg t'. \\ t \leftarrow tc(x,y) \wedge \neg r(x,y). & t' \leftarrow tc(x,y) \wedge \neg r'(x,y). \\ tc(x,y) \leftarrow e(x,y). & r'(x,y) \leftarrow r(x,y) \vee s(x,y). \\ tc(x,y) \leftarrow e(x,z) \wedge tc(z,y). & \end{array}$$

and P''_{re} is

$$\begin{array}{ll} q \leftarrow \neg t. & q'' \leftarrow \neg t''. \\ t \leftarrow tc(x,y) \wedge \neg r(x,y). & t'' \leftarrow tc'(x,y) \wedge \neg r'(x,y). \\ tc(x,y) \leftarrow e(x,y). & tc'(x,y) \leftarrow e'(x,y). \\ tc(x,y) \leftarrow e(x,z) \wedge tc(z,y). & tc'(x,y) \leftarrow e'(x,z) \wedge tc'(z,y). \\ & e'(x,y) \leftarrow e(x,y) \vee s(x,y). \\ q' \leftarrow \neg t'. & \\ t' \leftarrow tc(x,y) \wedge \neg r'(x,y). & \\ r'(x,y) \leftarrow r(x,y) \vee s(x,y). & \end{array}$$

So, (q'',P''_{re}) with P''_{re} as given above is the weakest precondition for the update S and the integrity constraint (q,P). Hence, if (q'',P''_{re}) holds with respect to a particular database, then updating that database using S will not violate the integrity constraint (q,P). □

3.4 Related Work

The subject of efficient integrity constraint checking with respect to deductive databases has been described and studied by many people [SK88, BDM88, GL90, NDCC92, Man90, Deß90, Wal90, Wal91, Wal92, Bay92, LST87,

Oli91]. In general, the various approaches can be classified into several groups: simplified instances, propagated deltas, and transaction safety.

Lloyd et al. [LST87] describe a method which considers updates as two disjoint sets of tuples, Δ^+ being the inserted tuples and Δ^- being the deleted tuples. It is an extension of Nicolas [Nic82] method to stratified deductive databases. The first phase generates an approximation of the sets of atoms whose addition/deletion has been induced by an update. The second phase involves checking (simplified) instances of the integrity constraints relevant to the sets of potentially added/deleted atoms with respect to the updated database state. It has the advantage that it does not require access to the database state to generate the potential updates, but this is offset by extra work that must be done in the second phase.

In [CCD93] and [CD94], Celma et al. propose an extension to [LST87] which involves recording the derivation paths leading to the potential additions/deletions. The second phase then checks only the derivation paths associated with the potential addition of $inconsistent$. The advantage of this method is that, in the presence of disjunctions, only the disjuncts that potentially lead to the addition of $inconsistent$ need be checked while the other disjuncts are ignored. The disadvantage is that sub-paths shared by these recorded derivations will be re-evaluated unnecessarily.

Bry et al. [BDM88] describe a method applicable to stratified databases, but only for single tuple updates. In contrast to Lloyd et al., they compute the consequential additions and deletions exactly in their first phase, then check simplified instances of relevant constraints in a second phase. While a more precise calculation of the induced updates in the first phase may reduce the work required in the second phase, this is offset by the extra effort required by the first phase.

Sadri and Kowalski [SK88] describe an alternative method based on a modified SLDNF proof procedure. Essentially they use the clauses corresponding to the inserted/deleted tuples as the tops of proof trees. This approach follows from the observation that, if the database satisfied the constraints before the update, then any violation of the constraints must involve one of the inserted/deleted tuples.

Although not described as such, Bayer [Bay92] describes a variation on [SK88] involving a rule transformation rather than a modified SLDNF proof procedure “such that the top-down activation of these derivation

rules simulates the bottom-up activation of the initial meta-rules". In contrast to [SK88], this method effectively applies the modified SLDNF proof procedure to a transformed set of rules which capture, exactly, the consequential database additions (similarly to [BDM88]), rather than to the original integrity constraints. Thus, if the goal $add(inconsistent)$ is ever generated during the proof procedure, this indicates that the integrity constraints have been violated by the update.

Olivé [Oli91] also describes a method related to [SK88] and based on the propagation of "internal events" and the use of SLDNF resolution to determine whether events corresponding to the violation of an integrity constraint are caused by a particular update. It is a very similar approach to that of [Bay92] and shares strong links with our approach. However, because it is based on tuple insertion rather than update rules/programs and there is no simplification step, it does not avoid performing checks in, for example, the case where the insertion of one set of tuples makes up for the insertion of another set within the one transaction (see example 5.4).

Wallace [Wal90, Wal91, Wal92] addresses the problem of transaction safety, describing a method for compiling integrity constraints into update procedures. His method extends the methods based on "recursive update consequences" [BDM88] by considering a more powerful update language and a transformation similar to ours. It produces results similar to our method, but these results are not characterised and rely upon an unspecified abstract interpreter or partial evaluator.

3.5 Summary

This chapter extended the concepts introduced in the previous chapter. We introduced an update language for deductive databases and defined the predicate transformer wp which can be used to check the safety of an update with respect to the database's integrity constraints as follows.

If (q, P) is a constraint on a database B then, before executing a statement S, we can check the condition produced by $wp(S, (q, P))$ to determine whether the new database state $[\![S]\!](B)$ will satisfy the constraint (q, P). Again, this weakest precondition can be used to generate safe transactions or as feedback to the programmer.

The extension to deductive databases allows more complex constraints

using recursion to be specified, such as those involving transitive closure. It also allows more complex updates to be specified since the set-oriented bounded iteration construct ranges over the results of a query which may now involved recursion. A natural consequence of having a more powerful update language is that it makes simplifying the resulting weakest precondition that much more difficult.

This provides a foundation for performing integrity constraint checking more efficiently than is possible using previously described methods [BD93, JQ92, JJ91, JK90]. Chapter 5 deals with simplification of the weakest precondition in detail.

Chapter 4

Deductive Object-Oriented Databases

This chapter introduces a deductive object-oriented data model, Gulog, in sections 4.1 and 4.2. The update language is extended to support object creation and deletion in section 4.3. Section 4.4 defines the predicate transformer with respect to the new data model and update language. Finally, section 4.5 examines related work and is followed by a summary of the chapter's contributions in section 4.6.

The material presented in this chapter is based on work that first appeared in [Law95].

4.1 Overview

Gulog, a simple, yet powerful, logical framework for reasoning about deductive object-oriented systems is described by Dobbie and Topor [Dob95, DT93, DT94, DT95]. We use this model as a foundation for a simple database language, which also incorporates updates, in order to describe our results. However, our work is not limited to this model and should generalise easily to other models such as those described by Abiteboul et al. [ALUW93], Jeusfeld et al. [JJ91, JK90], and (practical) subsets of F-logic [KLW90, KLW95].

A Gulog database consists of a schema and an extension. The schema provides the domains of (typed) object identifiers, the types (classes), the type (class) hierarchy, and the types of the relations, rules and methods.

The extension corresponds to a relational database or the extensional part of a deductive database. It stores base relations and the attribute values of objects.

In Gulog, the clause $\{x : \tau\} \vdash x[m \rightarrow a]$ defines a method called m on a type τ. This method maps objects of type τ to the value a.

Not only are methods inherited by subtypes, but they can also be overridden. The clause $\{x : \sigma\} \vdash x[m \rightarrow b] \leftarrow W$ defined on a type σ that is a subtype of τ overrides the inherited method above and maps instances of σ and its subtypes to the value b, but only if W holds. If, for a particular instance of σ or a subtype, W does not hold, then it is mapped to the value a as before.

Usually, overriding is statically defined as a syntactic condition. That is, the existence of a clause defining a method on a subtype causes all clauses defining that method on supertypes to be overridden. For Gulog, it is the, F-logic inspired, *dynamic overriding* that sets Gulog apart from most other languages involving inheritance. The use of dynamic overriding allows Gulog to effectively avoid the problem of multiple inheritance conflict resolution.

Example 4.1 Here is a simple database consisting of schema declarations giving the class hierarchy and typing information for methods and predicates, and then the data itself. The infinite domains of object identifiers $\mathcal{U}_{person}$ and $\mathcal{U}_{student}$ (which would identify the types of $p1$ and $p2$) are omitted. The class $student$ inheriting from $person$ is indicated by writing $student < person$. That the predicate $named$ is of arity one and applies to objects of type $person$ is indicated by $named(person)$. The schema statement $person[name \Rightarrow string]$ indicates that the class $person$ has a method called $name$ with a result of type $string$. Similarly, the schema statement $student[school \Rightarrow string]$ indicates that the class $student$ has a method called $school$ with a result of type $string$. Note that there is no extension for `named` since this is an intensionally defined predicate (see example 4.2 below).

Schema:	Extension:
$student < person.$	$p1[name \rightarrow$ “Damiel”$].$
$named(person).$	$p2[name \rightarrow$ “Raphaela”$].$
$person[name \Rightarrow string].$	
$student[school \Rightarrow string].$	

□

We now observe that Gulog makes no distinction between object identifiers (oids) that refer to objects that "really exist" in the database, rather than oids that are merely elements of the domain of all oids. Since we wish to include the notion of object "creation" and "deletion" in our update language, this is an important concept. As such, we introduce the following convention (which could be considered a restriction on which databases are "valid", or a reification of the domain of a class as opposed to the domain of its corresponding type).

> For every user-defined[1] type τ, we require a base relation $inst_\tau/1$ and a predicate $isa_\tau/1$. The base relation $inst_\tau/1$ is used to denote those objects that "really exist" in the database, and the predicate $isa_\tau/1$ is defined in terms of $inst_\tau/1$ and $isa_\tau_i/1$, where the τ_i's are the immediate subclasses of τ, to reflect the inheritance hierarchy. (A more formal definition is given below.)

The schema from example 4.1 would give rise to the following implicit schema declarations and intensional rules:

Implicit Schema:
$inst_person(person).$
$inst_student(student).$
$isa_person(person).$
$isa_student(student).$

Implicit Rules:
$\{x : student\} \vdash isa_student(x) \leftarrow inst_student(x).$
$\{x : person\} \vdash isa_person(x) \leftarrow inst_person(x).$
$\{x : person\} \vdash isa_person(x) \leftarrow isa_student(x).$

The intuitive meaning of the above predicates is as follows. For all objects for which $inst_person$ holds, that object will be of type $person$ and

[1]The distinction between built-in types and user-defined types has two motivations. Firstly, instances of built-in types are never "created" or "deleted". Secondly, the domains of (the classes corresponding to) built-in types are infinite and reification would thus require infinite relations.

is deemed to "really exist". It does not hold for objects which are subclasses of $person$. For all objects for which $inst_student$ holds, that object will be of type $student$ and is deemed to "really exist". For all objects for which isa_person holds, that object will be of type $person$ or any subclass of $person$ and is deemed to "really exist". For all objects for which $isa_student$ holds, that object will be of type $student$ or any subclass of $student$ and is deemed to "really exist".

Note again that we distinguish between user-defined types (classes) and built-in types, such as `string`, which do not have an associated $inst_\tau$ relation.

Thus the complete database for example 4.1 would be:

Implicit Rules:

$\{x : student\} \vdash isa_student(x) \leftarrow inst_student(x).$
$\{x : person\} \vdash isa_person(x) \leftarrow inst_person(x).$
$\{x : person\} \vdash isa_person(x) \leftarrow isa_student(x).$

Schema:

$student < person.$
$named(person).$
$inst_person(person).$
$inst_student(student).$
$isa_person(person).$
$isa_student(student).$
$person[name \Rightarrow string].$
$student[school \Rightarrow string].$

Extension:

$inst_person(p1).$
$inst_person(p2).$
$p1[name \rightarrow \text{"Damiel"}].$
$p2[name \rightarrow \text{"Raphaela"}].$

Example 4.2 To specify the integrity constraint that requires every person (that really exists in the database) to have a name (an example of a not-NULL constraint), we would include the following rule.

$\{y : string, x : person\} \vdash named(x) \leftarrow x[name \rightarrow y].$

The constraint is then

$\{x : person\} \vdash \forall x \; isa_person(x) \rightarrow named(x)$

which also applies to students because the definition of the implicit rule defining isa_person captures the semantics of inheritance. □

It would not be possible to specify such a constraint without this (or a similar) convention about objects that "really exist" without defining the object creation and deletion as mappings that alter the database's domain. This is something we wish to avoid since it would entail reasoning about systems with changing domains when we deal with the weakest precondition transformation.

4.2 Data Model

We now present a series of definitions relating to Gulog. A full description of Gulog's semantics is beyond the scope of this thesis but can be found in [Dob95, DT93, DT94, DT95]. If the reader is familiar with C-logic, O-logic, F-logic or other similar logics, then Gulog will present few surprises.

Let T be a set of finitely many types. Let $\prec$ be a relation over $T \times T$ defining a partial order on T. Let $\mathcal{U}_{\tau_i}$ be a set of infinitely many, typed, object identifiers for each $\tau_i \in T$.

A tuple of the form $\langle T, \prec, R_1, \ldots, R_l, M_{\Rightarrow 0}, \ldots, M_{\Rightarrow m}, M_{\Rrightarrow 0}, \ldots, M_{\Rrightarrow n} \rangle$ defines a database *schema* S. Each R_i is a tuple $\langle \tau_{j_1}, \ldots, \tau_{j_{k_i}} \rangle$ for some $k_i \geq 0$, specifying the types of the attributes of relation R_i. Similarly, each $M_{\rightarrow i}$ (resp., $M_{\twoheadrightarrow i}$) is a tuple $\langle \tau_{j_0}, \ldots, \tau_{j_{k_i+1}} \rangle$ for some $k_i \geq 0$ specifying the types $\tau_{j_1}, \ldots, \tau_{j_{k_i}}$ of the arguments of the functional (resp., relational) method m_i and the result type $\tau_{j_{k_i+1}}$ when applied to objects of type τ_{j_0}. In this case we say that the method m_i is ***defined on*** class τ_{j_0}.

A tuple of the form $\langle R_1, \ldots, R_l, M_{\rightarrow 0}, \ldots, M_{\rightarrow m}, M_{\twoheadrightarrow 0}, \ldots, M_{\twoheadrightarrow n} \rangle$ defines a database *extension* D. Each R_i is a finite named relation over $\mathcal{U}_{\tau_1} \times \cdots \times \mathcal{U}_{\tau_{k_i}}$ for some $k_i \geq 0$. Each $M_{\rightarrow i}$ is a finite named mapping $\mathcal{U}_{\tau_0} \times \cdots \times \mathcal{U}_{\tau_{k_i}} \mapsto \mathcal{U}_{\tau_{k_i+1}}$. Each $M_{\twoheadrightarrow i}$ is a finite named relation over $\mathcal{U}_{\tau_0} \times \cdots \times \mathcal{U}_{\tau_{k_i+1}}$. Note that the R_is include the implicit base relations $inst_\tau/1$ mentioned above.

A *database* is a tuple $\langle S, D \rangle$. Normally the database would also include an intensional part containing the rules and method definitions. We choose to separate this part and consider it as part of a query (the other part being the goal) for notational convenience when dealing with the weakest precondition transformation.

A *formula* is a first-order, function-free formula in some fixed language $\mathcal{L}$. An atom can be of the following forms: $r_i(x_1, \ldots, x_n)$, $y[m_i@x_1, \ldots, x_k \rightarrow z]$, or $y[m_i@x_1, \ldots, x_k \twoheadrightarrow z]$, corresponding to R_i, $M_{\rightarrow i}$, and $M_{\twoheadrightarrow i}$ respec-

tively and where the x_i, y, z are variables or oids. We write $m/k@c$ to denote a (functional or relational[2]) method (attribute) m of arity k that is applicable to objects in class c (or any subclass of c). We write r_i/n to indicate the predicate corresponding to the atom $r_i(x_1, \ldots, x_n)$ Thus, the r_i/n and $m_i/k@c$ are the predicate symbols of the language $\mathcal{L}$.

A *variable typing* Γ is a set of the form $\{x_1 : \tau_1,\ \ldots,\ x_n : \tau_n\}$.

A *rule* is a function free formula of the form $\Gamma \vdash A \leftarrow W$ where A is an atom, W is a first-order formula, and Γ is a variable typing giving every variable in A and W a type. All variables in A and free variables in W are assumed to be universally quantified at the front of the formula.

If r_i/k (resp., $m_i/k@c$) is a predicate symbol corresponding to the relation R_i (resp., $M_{\rightarrow k}$ or $M_{\twoheadrightarrow k}$) in the database, then there cannot also be a rule defining r_i/k (resp., $m_i/k@c$).

A *program* P is a set of rules. We require a unique preferred model to exist for all databases and programs so we restrict programs to belonging to the class of simple programs [DT94] which means they must be *inheritance-stratified*. Essentially this means the Datalog translation of a simple program is locally stratified.

A *query* is a mapping from a database to a relation. We represent a query as $(\Gamma \vdash A, P)$ where Γ is a variable typing, A is an atomic formula, and P is a program defining A. A *condition* is a query in which the atom has no variables. In this case we omit the (empty) variable typing. An *integrity constraint* is a condition.

A *valuation* for a query $(\Gamma \vdash q(\overline{x}), P)$ is a mapping of each of the free variables x_i of q to elements of $\mathcal{U}_{\tau_i}$. For a database B, query $(\Gamma \vdash q(\overline{x}), P)$ and valuation ν for $\overline{x}$, $B \models_\nu (\Gamma \vdash q(\overline{x}), P)$ if and only if $M_{B \cup P} \models_\nu \Gamma \vdash q(\overline{x})$ where $M_{B \cup P}$ is the unique preferred model for B and P with domain $\mathcal{U}$. We say a valuation ν' extends another valuation ν ($\nu' \geq \nu$) if the domain of ν' is a superset of the domain of ν and the restriction of ν' to the domain of ν is identical to ν.

A query (q', P') is a *precondition* for an update U and a query (q, P) if, for every database B, $B \models (q', P')$ implies $U(B) \models (q, P)$.

A precondition (q', P') for an update U and a query (q, P) is a *weakest precondition* for U and (q, P) if, for every precondition (q'', P'') for U

[2]For our purposes, it is simpler for us to not distinguish notationally between functional and relational methods.

and (q, P), and for every database B, $B \models (q'', P'')$ implies $B \models (q', P')$.

Since one of our goals is to be able to evaluate the effect of an update on a query (such as an integrity constraint) without actually performing the update, we need to determine exactly which predicates and methods are affected by an update. Thus we give a series of definitions which, for a particular base predicate or method t, capture exactly those predicates and methods which may be affected by an update to t.

A predicate s/k *directly depends* on an atom t if t is unifiable with any atom t' in the body of any rule defining s/k.

A method $m/k@c$ *directly depends* on an atom t if t is unifiable with any atom t' in the body of any rule defining $m/k@c$ or $m/k@c'$ where c' is a superclass of c.

Let s' be an instance of a method or predicate s. Then s *depends* on an atom t if s' directly depends on t or, s' directly depends on another atom t' that depends on t.

Note that we do not need to consider overriding in this definition, since method overriding in Gulog is dynamic (being determined by a preference relation over minimal models) and therefore can not be determined independently of the extensional database. That is, when there are definitions for a method m in the superclass $c0$ and the subclass $c1$, evaluation of m for an instance of the subclass $c1$ may still need to involve the definition of m on the superclass $c0$.

Example 4.3 Let $c1$ be a subclass of $c0$, and P be the following program.

$\{x : c0\} \vdash x[m \rightarrow 0].$
$\{x : c1\} \vdash x[m \rightarrow 1] \leftarrow p(x).$
$p(a).$
$inst_c1(a).$
$inst_c1(b).$

Then the query $(\{x : c1, y : int\} \vdash x[m \rightarrow y], P)$ has the two answers $\{x/a, y/1\}$ and $\{x/b, y/0\}$ since the definition of $m/0@c1$ overrides the definition of $m/0@c0$ dynamically. That is, overriding is based on ground instances of clauses rather than the existence of a method definition in a subclass as is the case in languages such as C++, Java, and C-logic. □

However, in adapting our approach to data models like C-logic where overriding is statically determinable, we would incorporate the semantics

of overriding directly into the definition of *directly depends* for methods, above.

Let P be a program and t be an atom defined in the database. Let P'_t be a copy of P with the following additional rules. For every rule Q in P defining a predicate symbol q (or method $m/k@c$) that depends on t, add an identical rule defining q' (resp., $m'/k@c$) in which every predicate symbol s (and method $a/j@d$) in the body of Q that depends on t (including any occurrence of t itself) is replaced by s' (resp., $a'/j@d$). Note, there is no definition of t' in P'_t since t corresponds to data in the database and hence there can be no rules in P defining t.

This definition follows that given in Chapter 3. However, the different definition of *directly depends* given above is sufficient to capture the intricate semantics of inheritance and overriding that Gulog uses.

Example 4.4 Let P be the following program which says there is a path from x to y if there is an edge from x to y or there is an edge to some intermediate point z that is guarded.

$$\{x : p, y : p\} \vdash path(x, y) \leftarrow$$
$$edge(x, y).$$
$$\{x : p, y : p, z : p\} \vdash path(x, y) \leftarrow$$
$$edge(x, z), guard(z), path(z, y).$$

Then, P'_{edge} is as follows.

$$\{x : p, y : p\} \vdash path(x, y) \leftarrow$$
$$edge(x, y).$$
$$\{x : p, y : p, z : p\} \vdash path(x, y) \leftarrow$$
$$edge(x, z), guard(z), path(z, y).$$
$$\{x : p, y : p\} \vdash path'(x, y) \leftarrow$$
$$edge'(x, y).$$
$$\{x : p, y : p, z : p\} \vdash path'(x, y) \leftarrow$$
$$edge'(x, z), guard(z), path'(z, y).$$

□

The above example is essentially identical, but for the typing information, to that which would be obtained using the purely deductive datamodel of the previous chapter. The following example illustrates what happens in the presence of inheritance.

Example 4.5 Let P be the following program where $c0$, $c1$, and $c2$ may represent a hierarchy of product items, m represents the highlighted features to be used for selling the product $m0$ represents a standard feature of all products (of type $c0$) while $m1$ and $m2$ represent optional features specific to products types $c1$ and $c2$. We use short abstract class and method names for clarity.

$c2 < c1 < c0.$
$\{x : c0, y : d\} \vdash x[m \twoheadrightarrow y] \leftarrow x[m0 \twoheadrightarrow y].$
$\{x : c1, y : d\} \vdash x[m \twoheadrightarrow y] \leftarrow x[m1 \twoheadrightarrow y].$
$\{x : c2, y : d\} \vdash x[m \twoheadrightarrow y] \leftarrow x[m2 \twoheadrightarrow y].$

Then, $P'_{m1/0@c1}$ is as follows.

$c2 < c1 < c0.$
$\{x : c0, y : d\} \vdash x[m \twoheadrightarrow y] \leftarrow x[m0 \twoheadrightarrow y].$
$\{x : c1, y : d\} \vdash x[m \twoheadrightarrow y] \leftarrow x[m1 \twoheadrightarrow y].$
$\{x : c2, y : d\} \vdash x[m \twoheadrightarrow y] \leftarrow x[m2 \twoheadrightarrow y].$

$\{x : c1, y : d\} \vdash x[m' \twoheadrightarrow y] \leftarrow x[m1' \twoheadrightarrow y].$
$\{x : c2, y : d\} \vdash x[m' \twoheadrightarrow y] \leftarrow x[m2 \twoheadrightarrow y].$

$\square$

In a similar manner to the previous chapter, the program P'_t will be used in capturing the effects of an update statement. By basing P'_t on the definition of the *directly depends* relationship, we capture the precise semantics of overriding as defined by Gulog.

4.3 Updates

Our update language provides for set-oriented updates of functional attributes, relational attributes, object creation, and object deletion. These operations are primitives, so maintenance of referential integrity is not implicit in the semantics of the deletion operation, although it would be a system enforced integrity constraint. Since we do not allow the database schema to be altered, there are no issues with object migration (i.e., what to do about the existing instances of a class to which has been added a new attribute).

Let $(\Gamma \vdash q(y, \overline{x}, z), P)$ be a query with free variables $y, \overline{x}, z$, where $\overline{x}$ is the vector of variables $x_1, \ldots, x_k$. The following are statements in our update language[3]:

$$\forall \overline{x}\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow +r(\overline{x}))$$
$$\forall \overline{x}\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow -r(\overline{x}))$$

$$\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow +y[m@\overline{x} \twoheadrightarrow z])$$
$$\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow -y[m@\overline{x} \twoheadrightarrow z])$$
$$\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow !y[m@\overline{x} \rightarrow z])$$
$$\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow -y[m@\overline{x} \rightarrow z])$$

$$\forall \overline{x} \mathcal{I} y\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow +y : c)$$
$$\forall x\ ((\Gamma \vdash q(x), P) \rightarrow -x : c)$$

If $S_1, \ldots, S_n$ are statements in our language then $(S_1\ ;\ \ldots\ ;\ S_n)$ is a statement in our language.

The effect of a statement in our language is given informally as follows. A statement of the form $\forall \overline{x}\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow\ op(\overline{x}))$ is executed by evaluating the query $(\Gamma \vdash q(\overline{x}), P)$ to generate a set of bindings for the variables $\overline{x}$, then executing $op(\overline{x})$ for each tuple of bindings for $\overline{x}$. The operation $+r(\overline{x})$ (resp., $-r(\overline{x})$) inserts into (resp., deletes from) the relation R the tuple $\overline{x}$. The operation $+y[m@\overline{x} \twoheadrightarrow z]$ (resp., $-y[m@\overline{x} \twoheadrightarrow z]$) adds to (resp., deletes from) the set-valued attribute m of object y with arguments $\overline{x}$ the value z. The operation $!y[m@\overline{x} \rightarrow z]$ sets the functional attribute m of object y with arguments $\overline{x}$ to the value z. The operation $-y[m@\overline{x} \rightarrow z]$ deletes the value z from the functional attribute m of object y with arguments $\overline{x}$. That is, if m applied to y with arguments $\overline{x}$ had the value z, then it no longer has any value.

The statement $\forall \overline{x} \mathcal{I} y\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow +y : c)$ creates k new objects in class c. It is executed by evaluating the query $(\Gamma \vdash q(\overline{x}), P)$ to generate a set of k bindings for $\overline{x}$, then "inventing" k different bindings $o_i, 1 < i < k$ for y such that $B \not\models inst_c(o_i)$. These new objects are then added to $inst_c$ to indicate that they now "really exist".

The operation $-x : c$ deletes the object x from class c. What this really means is that the update $-inst_c(x)$ is actually performed, to indicate that the object x no longer "really exists".

[3]The quantifier $\mathcal{I}$ is used here to indicate object identifier invention. It can only occur in this context.

The sequence of statements $(S_1 \,;\, \ldots \,;\, S_n)$ is executed by evaluating S_1 then S_2 up to S_n in order.

The *update* represented by a statement S in our update language is a mapping from one database to another. It is denoted $[\![S]\!]$ and is defined inductively as follows.

1. $[\![\forall \overline{x}\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow\ +r(\overline{x}))]\!](B) =$
   $B \cup \{r(\overline{x}) \mid B \models (\Gamma \vdash q(\overline{x}), P)\}$
2. $[\![\forall \overline{x}\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow\ -r(\overline{x}))]\!](B) =$
   $B - \{r(\overline{x}) \mid B \models (\Gamma \vdash q(\overline{x}), P)\}$
3. $[\![\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow\ +y[m@\overline{x} \twoheadrightarrow z])]\!](B) =$
   $B \cup \{y[m@\overline{x} \twoheadrightarrow z] \mid B \models (\Gamma \vdash q(y, \overline{x}, z), P)\}$
4. $[\![\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow\ -y[m@\overline{x} \twoheadrightarrow z])]\!](B) =$
   $B - \{y[m@\overline{x} \twoheadrightarrow z] \mid B \models (\Gamma \vdash q(y, \overline{x}, z), P)\}$
5. $[\![\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow !y[m@\overline{x} \rightarrow z])]\!](B) =$
   $B - \{y[m@\overline{x} \rightarrow z] \mid B \models (\Gamma \vdash q(y, \overline{x}, x') \wedge y[m@\overline{x} \rightarrow z], P)\}$
   $\cup \{y[m@\overline{x} \rightarrow z] \mid B \models (\Gamma \vdash q(y, \overline{x}, z), P)\}$
6. $[\![\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow\ -y[m@\overline{x} \rightarrow z])]\!](B) =$
   $B - \{y[m@\overline{x} \rightarrow z] \mid B \models (\Gamma \vdash q(y, \overline{x}, z), P)\}$
7. $[\![\forall \overline{x} \mathcal{I} y\ ((\Gamma \vdash q(\overline{x}), P) \rightarrow +y : c)]\!](B) =$
   $B \cup \{inst_c(o) \mid o \in O\}$,
   where $O \subset \mathcal{U}_c$
   and $|O| = |\{\overline{x} \mid B \models (\Gamma \vdash q(\overline{x}), P)\}|$
   and $O \cap \{x \mid B \models (inst_c(x), \phi)\} = \phi$
8. $[\![\forall x\ ((\Gamma \vdash q(x), P) \rightarrow -x : c)]\!](B) =$
   $B - \{inst_c(x) \mid B \models ((\Gamma \vdash q(x), P)\}$
9. $[\![S_1 \,;\, \ldots \,;\, S_n]\!](B) = [\![S_n]\!]([\![S_1 \,;\, \ldots \,;\, S_{n-1}]\!](B)), n > 1$

4.4 Condition Transformers

We now define a condition transformer wp for a statement S and a condition (q, P) in the context of our deductive object-oriented data model.

Let S be a statement, (q, P) a condition, and Q a program. Then the *condition transformer* $wp(S, (q, P))$ is defined inductively as follows:

1. $wp(\forall\overline{x}\ ((\Gamma \vdash p(\overline{x}), Q) \rightarrow\ +r(\overline{x})), (q, P)) =$
   $(q', Q \cup P'_r \cup \{\Gamma \vdash r'(\overline{x}) \leftarrow r(\overline{x}) \vee p(\overline{x})\})$

2. $wp(\forall\overline{x}\ ((\Gamma \vdash p(\overline{x}), Q) \rightarrow\ -r(\overline{x})), (q, P)) =$
   $(q', Q \cup P'_r \cup \{\Gamma \vdash r'(\overline{x}) \leftarrow r(\overline{x}) \wedge \neg p(\overline{x})\})$

3. $wp(\forall y, \overline{x}, z\ ((\Gamma \vdash p(y, \overline{x}, z), Q) \rightarrow +y[m@\overline{x} \twoheadrightarrow z]), (q, P)) =$
   $(q', Q \cup P'_{m/k@c} \cup$
   $\{\Gamma \vdash y[m'@\overline{x} \twoheadrightarrow z] \leftarrow y[m@\overline{x} \twoheadrightarrow z] \vee p(y, \overline{x}, z)\})$

4. $wp(\forall y, \overline{x}, z\ ((\Gamma \vdash p(y, \overline{x}, z), Q) \rightarrow -y[m@\overline{x} \twoheadrightarrow z]), (q, P)) =$
   $(q', Q \cup P'_{m/k@c} \cup$
   $\{\Gamma \vdash y[m'@\overline{x} \twoheadrightarrow z] \leftarrow y[m@\overline{x} \twoheadrightarrow z] \wedge \neg p(y, \overline{x}, z)\})$

5. $wp(\forall y, \overline{x}, z\ ((\Gamma \vdash p(y, \overline{x}, z), Q) \rightarrow !y[m@\overline{x} \rightarrow z]), (q, P)) =$
   $(q', Q \cup P'_{m/k@c} \cup$
   $\{\Gamma \vdash y[m'@\overline{x} \rightarrow z] \leftarrow$
   $(y[m@\overline{x} \rightarrow z] \wedge \neg p(y, \overline{x}, x')) \vee p(y, \overline{x}, z)\})$

6. $wp(\forall y, \overline{x}, z\ ((\Gamma \vdash p(y, \overline{x}, z), Q) \rightarrow -y[m@\overline{x} \rightarrow z]), (q, P)) =$
   $(q', Q \cup P'_{m/k@c} \cup$
   $\{\Gamma \vdash y[m'@\overline{x} \rightarrow z] \leftarrow y[m@\overline{x} \rightarrow z] \wedge \neg p(y, \overline{x}, z)\})$

7. $wp(\forall\overline{x}\mathcal{I}y\ ((\Gamma \vdash p(\overline{x}), Q) \rightarrow +y : c), (q, P)) =$
   $(q', Q \cup P'_c \cup$
   $\{\{y : c\} \vdash \mathit{inst_c}'(y) \leftarrow \mathit{inst_c}(y) \vee (p(\overline{x}) \wedge y = f(\overline{x})\})$

8. $wp(\forall x\ ((\Gamma \vdash p(x), Q) \rightarrow -x : c), (q, P)) =$
   $(q', Q \cup P'_c \cup$
   $\{\{x : c\} \vdash \mathit{inst_c}'(x) \leftarrow \mathit{inst_c}(x) \wedge \neg p(x)\})$

9. $wp((S_1\ ;\ \ldots\ ;\ S_n), (q, P)) =$
   $wp((S_1\ ;\ \ldots\ ;\ S_{n-1}), wp(S_n, (q, P))), \quad n > 1$

Here, $f(\overline{x})$ acts like a Skolem function in that it maps each valuation for $\overline{x}$ to a member of the set O as given in the formal definition of the effect of an object creation statement. Thus, $O \equiv \{f((\overline{x}) \mid B \models (q(\overline{x}), P)\}$.

Rules 3 and 4 for multi-valued methods are essentially the same as rules 1 and 2 for predicates. Rule 5 encodes a "replace" which is effectively a delete followed by an insert while rule 6 is similar to rules 2 and 4. Rule 9 describes repeated application of the transformation to deal with update sequences in the same manner as for the previous chapters. Rule 8 encodes deletion from the implicit relation $inst_c$ used to encode those objects that "really exist". Rule 7 is perhaps the most tricky. It encodes the arbitrary selection of n object identifiers from the appropriate domain $\mathcal{U}_c$ where n is the number of solutions to the query $(\Gamma \vdash q(\overline{x}), P)$ and marking them as objects that "really exist" by adding them to the implicit relation $inst_c$. It also ensures that none of the selected object identifiers already "really exists" by making sure that they are not already members of $inst_c$.

We now present a series of lemmas leading to a proof that $wp(S, (q, P))$ is the weakest precondition for $[\![S]\!]$ and (q, P). You will notice that the proofs make no mention of the inheritance hierarchy or method overriding. This is because, as mentioned in Section 4.1, the mapping from a program P to P'_c caters for inheritance and the semantics of overriding is implicit when we write $B \models (q, P)$.

Lemma 4.1 *Let B be a database, $(\Gamma \vdash q(\overline{x}), P)$ a query where q depends on r, and ν a variable assignment. Then, $B \models_\nu (\Gamma \vdash q(\overline{x}), P)$ if and only if $B \models_\nu (q'(\overline{x}), P'_r \cup \{\Gamma \vdash r'(\overline{x}) \leftarrow r(\overline{x})\})$.*

Proof The proof is immediate. □

Lemma 4.2 *Let B be a database, $r(\overline{c})$ an atom, and $(\Gamma \vdash q(\overline{x}), P)$ a query. Then, $B \cup \{r(\overline{x}) \mid B \models (\Gamma \vdash q(\overline{x}), P)\} \models r(\overline{c})$ if and only if $B \models (r'(\overline{c}), P'_r \cup \{\Gamma \vdash r'(\overline{x}) \leftarrow r(\overline{x}) \vee q(\overline{x}$*

Proof

$$
\begin{aligned}
& B \models (r'(\overline{c}), P'_r \cup \{\Gamma \vdash r'(\overline{x}) \leftarrow r(\overline{x}) \vee q(\overline{x})\}) \\
\iff\quad & B \models (r(\overline{c}), P'_r) \quad or \quad B \models (q(\overline{c}), P'_r) \qquad \text{(only def. of } r'\text{)} \\
\iff\quad & B \models r(\overline{c}) \quad or \quad B \models (q(\overline{c}), P) \\
\iff\quad & B \models r(\overline{c}) \quad or \quad r(\overline{c}) \in \{r(\overline{x}) \mid B \models (q(\overline{x}), P)\} \\
\iff\quad & B \cup \{r(\overline{x}) \mid B \models (q(\overline{x}), P)\} \models r(\overline{c})
\end{aligned}
$$

□

Lemma 4.3 *Let B be a database, (q, P) a condition, $(\Gamma \vdash p(y, \overline{x}, z), Q)$ a query, and S the statement $\forall y, \overline{x}, z((\Gamma \vdash p(y, \overline{x}, z), Q) \rightarrow +y[m@\overline{x} \twoheadrightarrow z])$. Then, $[\![S]\!](B) \models (q, P)$ if and only if $B \models wp(S, (q, P))$.*

Proof

$$
\begin{aligned}
& B \models wp(S,(q,P)) \\
\iff\quad & B \models (q', Q \cup P'_{m/k@c} \cup \\
& \qquad \{\Gamma \vdash y[m'@\overline{x} \twoheadrightarrow z] \leftarrow y[m@\overline{x} \twoheadrightarrow z] \vee p(y,\overline{x},z)\}) \\
\iff\quad & B \cup \{y[m'@\overline{x} \twoheadrightarrow z] \mid B \models (y[m@\overline{x} \twoheadrightarrow z] \vee p(y,\overline{x},z), Q)\} \models \\
& \qquad (q', P'_{m/k@c}) \\
\iff\quad & B \cup \{y[m@\overline{x} \twoheadrightarrow z] \mid B \models (y[m@\overline{x} \twoheadrightarrow z] \vee p(y,\overline{x},z), Q)\} \models \\
& \qquad (q', P'_{m/k@c} \cup \{y[m'@\overline{x} \twoheadrightarrow z] \leftarrow y[m@\overline{x} \twoheadrightarrow z]\}) \\
\iff\quad & B \cup \{y[m@\overline{x} \twoheadrightarrow z] \mid B \models (\Gamma \vdash p(y,\overline{x},z), Q)\} \models \\
& \qquad (q', P'_{m/k@c} \cup \{y[m'@\overline{x} \twoheadrightarrow z] \leftarrow y[m@\overline{x} \twoheadrightarrow z]\}) \\
\iff\quad & B \cup \{y[m@\overline{x} \twoheadrightarrow z] \mid B \models (\Gamma \vdash p(y,\overline{x},z), Q)\} \models (q,P) \\
\iff\quad & [\![S]\!](B) \models (q,P)
\end{aligned}
$$

□

Lemma 4.4 *Let B be a database, (q,P) a condition, $(\Gamma \vdash p(y,\overline{x},z), Q)$ a query, and S the statement $\forall y,\overline{x},z((\Gamma \vdash p(y,\overline{x},z), Q) \rightarrow -y[m@\overline{x} \twoheadrightarrow z])$. Then, $[\![S]\!](B) \models (q,P)$ if and only if $B \models wp(S,(q,P))$.*

Proof Similar to the proof of lemma 4.3. □

Lemma 4.5 *Let B be a database, (q,P) a condition, $(\Gamma \vdash p(y,\overline{x},z), Q)$ a query, and S the statement $\forall y,\overline{x},z((\Gamma \vdash p(y,\overline{x},z), Q) \rightarrow !y[m@\overline{x} \rightarrow z])$. Then, $[\![S]\!](B) \models (q,P)$ if and only if $B \models wp(S,(q,P))$.*

Proof

$$B \models wp(S,(q,P))$$

$$
\begin{aligned}
\iff\quad & B \models (q', Q \cup P'_{m/k@c} \cup \\
& \quad \{\Gamma \vdash y[m'@\overline{x} \to z] \leftarrow \\
& \qquad (y[m@\overline{x} \to z] \wedge \neg p(y, \overline{x}, x')) \vee p(y, \overline{x}, z)\}) \\
\iff\quad & B \cup \{y[m'@\overline{x} \to z] \mid \\
& \quad B \models ((y[m@\overline{x} \to z] \wedge \neg p(y, \overline{x}, x')) \vee p(y, \overline{x}, z), Q)\} \\
& \qquad \models (q', P'_{m/k@c}) \\
\iff\quad & B \cup \{y[m'@\overline{x} \to z] \mid \\
& \quad B \models ((y[m@\overline{x} \to z] \wedge \neg p(y, \overline{x}, x')) \vee p(y, \overline{x}, z), Q)\} \\
& \qquad \models (q', P'_{m/k@c}) \\
\iff\quad & B \cup \{y[m'@\overline{x} \to z] \mid B \models (y[m@\overline{x} \to z], Q)\} \\
& \quad - \{y[m'@\overline{x} \to z] \mid B \models (y[m@\overline{x} \to z] \wedge p(y, \overline{x}, x'), Q)\} \\
& \quad \cup \{y[m'@\overline{x} \to z] \mid B \models (\Gamma \vdash p(y, \overline{x}, z), Q)\} \\
& \qquad \models (q', P'_{m/k@c}) \\
\iff\quad & B - \{y[m@\overline{x} \to z] \mid B \models (y[m@\overline{x} \to z] \wedge p(y, \overline{x}, x'), Q)\} \\
& \quad \cup \{y[m@\overline{x} \to z] \mid B \models (\Gamma \vdash p(y, \overline{x}, z), Q)\} \\
& \qquad \models (q', P'_{m/k@c} \cup \{y[m'@\overline{x} \to z] \leftarrow y[m@\overline{x} \to z]\}) \\
\iff\quad & B - \{y[m@\overline{x} \to z] \mid B \models (y[m@\overline{x} \to z] \wedge p(y, \overline{x}, x'), Q)\} \\
& \quad \cup \{y[m@\overline{x} \to z] \mid B \models (\Gamma \vdash p(y, \overline{x}, z), Q)\} \models (q, P) \\
\iff\quad & [\![S]\!](B) \models (q, P)
\end{aligned}
$$

□

Lemma 4.6 *Let B be a database, (q, P) a condition, $(\Gamma \vdash p(y, \overline{x}, z), Q)$ a query, and S the statement $\forall y, \overline{x}, z((\Gamma \vdash p(y, \overline{x}, z), Q) \to -y[m@\overline{x} \to z])$. Then, $[\![S]\!](B) \models (q, P)$ if and only if $B \models wp(S, (q, P))$.*

Proof Similar to the proof of lemma 4.4. □

Lemma 4.7 *Let B be a database, (q, P) a condition, and S the statement $\forall \overline{x} \mathcal{I} y\ ((\Gamma \vdash p(\overline{x}), Q) \to +y : c)$. Then, $[\![S]\!](B) \models (q, P)$ if and only if $B \models wp(S, (q, P))$.*

Proof

$B \models wp(S, (q, P))$

$$\begin{aligned}
&\Longleftrightarrow \quad B \models (q', Q \cup P'c \cup \{\{y : c\} \vdash inst_c'(y) \leftarrow \\
&\qquad\qquad inst_c(y) \vee (p(\overline{x}) \wedge y = f(\overline{x}))\}) \\
&\Longleftrightarrow \quad B \cup \{inst_c'(y) \mid B \models (inst_c(y) \vee (p(\overline{x}) \wedge y = f(\overline{x})), Q)\} \models \\
&\qquad\qquad (q', P'c) \\
&\Longleftrightarrow \quad B \cup \{inst_c(y) \mid B \models (inst_c(y) \vee (p(\overline{x}) \wedge y = f(\overline{x})), Q)\} \models \\
&\qquad\qquad (q', P'c \cup \{inst_c'(x) \leftarrow inst_c(x)\}) \\
&\Longleftrightarrow \quad B \cup \{inst_c(o) | o \in O\} \models (q, P) \\
&\Longleftrightarrow \quad [\![S]\!](B) \models (q, P)
\end{aligned}$$

□

Lemma 4.8 *Let B be a database, (q, P) a condition, $(\Gamma \vdash p(x), Q)$ a query, and S the statement $\forall \overline{x}\ ((\Gamma \vdash p(x), Q) \to -x : c), (q, P)$ Then, $[\![S]\!](B) \models (q, P)$ if and only if $B \models wp(S, (q, P))$.*

Proof

$$\begin{aligned}
&B \models wp(S, (q, P)) \\
&\quad \Longleftrightarrow \quad B \models (q', Q \cup P'c \\
&\qquad\qquad \cup \{\{x : c\} \vdash inst_c'(x) \leftarrow inst_c(x) \wedge \neg p(x)\}) \\
&\quad \Longleftrightarrow \quad B \cup \{inst_c'(x) \mid B \models (inst_c(x) \wedge \neg p(x), Q)\} \models (q', P'c) \\
&\quad \Longleftrightarrow \quad B - \{inst_c(x) \mid B \models (\Gamma \vdash p(x), Q)\} \models \\
&\qquad\qquad (q', P'c \cup \{inst_c'(x) \leftarrow inst_c(x)\}) \\
&\quad \Longleftrightarrow \quad B - \{inst_c(x) \mid B \models (\Gamma \vdash p(x), Q)\} \models (q, P) \\
&\quad \Longleftrightarrow \quad [\![S]\!](B) \models (q, P)
\end{aligned}$$

□

Lemma 4.9 *Let B be a database, (q, P) a condition, query, and S the sequential statement $(S_1 ; \ldots ; S_n)$. Then, $[\![S]\!](B) \models (q, P)$ if and only if $B \models wp(S, (q, P))$.*

Proof Assume inductively that, for every statement $((S_1 ; \ldots ; S_i)$, where $1 \leq i < n$, and query (q', P), we have $B \models wp((S_1 ; \ldots ; S_i), (q', P))$ if and only if $[\![S_1 ; \ldots ; S_i]\!](B) \models (q', P)$. Then,

$$\begin{aligned}
&B \models wp((S_1 ; \ldots ; S_n), (q, P)) \\
&\quad \Longleftrightarrow \quad B \models wp((S_1 ; \ldots ; S_{n-1}), wp(S_n, (q, P))) && \text{(definition of } wp\text{)} \\
&\quad \Longleftrightarrow \quad [\![S_1 ; \ldots ; S_{n-1}]\!](B) \models wp(S_n, (q, P)) && \text{(induction hyp.)} \\
&\quad \Longleftrightarrow \quad [\![S_n]\!]([\![S_1 ; \ldots ; S_{n-1}]\!](B)) \models (q, P) && \text{(induction hyp.)} \\
&\quad \Longleftrightarrow \quad [\![S_1 ; \ldots ; S_n]\!](B) \models (q, P) && \text{(definition of update)} \\
&\quad \Longleftrightarrow \quad [\![S]\!](B) \models (q, P)
\end{aligned}$$

Lemmas 4.3, 4.4, 4.5, 4.6, 4.7, and 4.8 provide the base cases. □

We are now in a position to prove that the condition transformer wp actually produces a condition which is the weakest precondition with respect to a particular update statement and condition. That is, $wp(S,(q,P))$ produces a condition such that, if it is true before executing the statement S, then the condition (q,P) is guaranteed to be true afterwards.

Theorem 4.1 *Let S be a statement, and (q,P) a query. Then $wp(S,(q,P))$ is the weakest precondition for $[\![S]\!]$ and (q,P).*

Proof The result follows immediately from Lemmas 3.1, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9. □

For the sake of clarity of presentation in the following two examples we will deal with formulas rather than programs since, in this case, they are equivalent.

Example 4.6 Consider a constraint C which says every member of every department must be an existing person - a kind of referential integrity constraint to avoid dangling references. C is

$$\{x:dept, y:emp\} \vdash \forall x,y(x[members \twoheadrightarrow y] \rightarrow inst_emp(y))$$

Now let $S1$ be the update that deletes all employees called john.

$$\forall\, x(\{x:emp\} \vdash x[name \rightarrow john] \rightarrow -x:emp)$$

Thus we have:

$$\begin{aligned}
&wp(S1,C)\\
&\quad = \{x:dept, y:emp\} \vdash \forall x,y(x[members \twoheadrightarrow y] \rightarrow\\
&\qquad\qquad (inst_emp(y) \wedge \neg y[name \rightarrow john]))\\
&\quad \equiv \{x:dept, y:emp\} \vdash \forall x,y((x[members \twoheadrightarrow y] \rightarrow\\
&\qquad\qquad inst_emp(y)) \wedge\\
&\qquad\qquad (x[members \twoheadrightarrow y] \rightarrow\\
&\qquad\qquad \neg y[name \rightarrow john]))\\
&\quad \equiv C \wedge \{x:dept, y:emp\} \vdash \forall x,y(x[members \twoheadrightarrow y] \rightarrow\\
&\qquad\qquad \neg y[name \rightarrow john])
\end{aligned}$$

□

It should be clear from the previous example (and from examples in previous chapters) that in many cases we can simplify the weakest precondition because we can make the assumption that the integrity constraint C holds for the current database state.

Example 4.7 To illustrate the potential of these simplifications, consider the same constraint C as for example 4.6. Now let $S2$ be the complex update that deletes all employee objects with the name john and removes them as members from any departments they may belong to.

$$\forall\, x(\{x : emp\} \vdash x[name \rightarrow john] \rightarrow -x : emp);$$
$$\forall\, x, y(\{x : dept, y : emp\} \vdash y[name \rightarrow john] \rightarrow -x[members \twoheadrightarrow y])$$

Thus we have:

$$\begin{aligned}
& wp(S2, C) \\
&= \{x : dept, y : emp\} \vdash \\
&\qquad \forall x, y((x[members \twoheadrightarrow y] \wedge \neg y[name \rightarrow john]) \rightarrow \\
&\qquad\qquad (inst_emp(y) \wedge \neg y[name \rightarrow john])) \\
&= C \wedge \{x : dept, y : emp\} \vdash \\
&\qquad \forall x, y((x[members \twoheadrightarrow y] \wedge \neg y[name \rightarrow john]) \rightarrow \\
&\qquad\qquad \neg y[name \rightarrow john]) \\
&= C
\end{aligned}$$

Since we know that C hold in the current database state, we can simplify this check to true, i.e., we know the update $S2$ is *safe* with respect to the constraint C and thus we do not need to check anything before or after performing $S2$. □

4.5 Related Work

Most attempts to deal with integrity constraints in a deductive object-oriented setting involve simply applying existing techniques for relational or deductive databases to restricted aspects of the deductive object-oriented data model. For example, Benzaken and Doucet's early work [BD92, BD93], which applies existing integrity constraint simplification methods for relational databases to an object-oriented data model, suffers because the

restrictions they apply to their data model effectively removes its object-oriented and deductive features. More recently, in [BD95, BS97] they adopt an approach similar to Sheard and Stemple in [SS89]. They use a predicate transformer based technique in combination with a tableaux theorem prover in order to prove transaction safety. However, their predicate transformer is not exact, so it does not produce *strongest* postconditions. This means that they are really performing a kind of abstract interpretation which may conservatively determine a transaction to be unsafe when it is, in fact, safe.

Bertino et al. [BCB97] extend the method by Lloyd et al. [LST87] from Datalog to what they call "Chimera Extended Datalog". Their approach involves providing a translation from the Chimera deductive object-oriented model into their extended datalog model. In this aspect, their approach is similar to ours. However, their integrity constraint checking method deals only with simple tuple insertion and deletion rather than the more expressively powerful update programs we deal with.

Jagadish and Qian [JQ92] describe a method that associates a simplified constraint with the classes over whose objects the constraint ranges. This means when an object is updated, only the constraints associated with that class need to be checked, and only for that object. However, it is likely that a complex update will update objects from several classes at a time and that these classes will be related by integrity constraints. Thus there is the danger that the same constraint will be evaluated several times.

Jeusfeld et al. [JJ91, JK90] reduce the problem to the deductive database case by providing a mapping from their deductive object-oriented language level to the deductive database level. They then use well known integrity checking techniques for deductive databases [BDM88] which results in a loss of information and therefore less efficient checking routines. Their work attempts to compensate for this lost information to regain the efficiency of the original deductive technique. However, since they do not deal with an update language, just a set of inserted and deleted facts, they are unable to recognise simplifications that are possible because of the form of the original update program. For example, when the same set of facts are inserted into two relations for which referential integrity must be maintained. On the other hand, their data model is more expressive than

ours and they allow (limited) constraints and updates on the database schema.

We believe our approach is promising since it deals directly with update programs and can therefore make use of the semantics implicit in them. In an object-oriented environment, these programs are likely to be in the form of methods and thus stored, managed by and available to the DBMS.

4.6 Summary

The object model framework provides an expressive mechanism for modelling and representing information about the world. Gulog is an attempt to provide a sound mathematical foundation for reasoning about object-based databases. It addresses concepts such as object identity, methods, and overriding as well as retaining concepts and strengths from relational and deductive databases.

In this chapter we introduced an update language, including object creation, for a deductive object-oriented data model based on Gulog. Furthermore, we defined the predicate transformer wp for updates in this language and proved that it generates the weakest precondition for a given update and constraint. The condition generated by the predicate transformer can be used to check for transaction safety *before* the update is performed.

In the following chapter we investigate the problem of simplifying the weakest precondition given that the current database state already satisfies the integrity constraints.

Chapter 5

Simplifying the Safety Condition

5.1 Introduction

Until now the focus has been on transaction safety. In the previous chapters we have shown how to check transaction safety, ensuring that rollbacks need not be performed due to integrity constraint violation. We now address the larger goal of reducing or eliminating the complexity and expense of the safety check. To do this we follow the well established principle of efficient integrity constraint checking that optimisations can be made by observing that we know the constraints hold in the initial database state. That is we examine how to simplify $wp(U, C)$ given that C already holds in the initial database state.

Hence, when we talk of a *simpler* condition, we have the goal of reducing to a minimum the amount of work required to enforce transaction safety. So, when we have two conditions that are syntactically related, as is the case in the simplifications we consider below, we can clearly say one condition is simpler than another if the former has fewer distinct conjuncts than the latter. That is, if P and Q are sentences, then P is simpler than $P \wedge Q$ since $cost(P \wedge Q) = cost(P) + cost(Q)$.

The structure of this chapter is as follows. In section 5.2 we examine some examples of typical integrity constraints, update statements and their associated safety conditions. We show how to find simpler safety conditions, then present results for more general forms of integrity con-

straints and update statements. In section 5.3 we examine an alternative approach to generating simple safety conditions based on the generation of meta-rules that capture the changes to the intensional database with respect to an update statement. In section 5.4 we describe formally the general problem of integrity constraint simplification and discuss some open problems, tradeoffs and limits on what can be achieved. Finally, section 5.5 summarises the results presented in this chapter.

5.2 Direct Simplification

Beginning with some simple examples to get a feel for the method, we then move on to more complex examples illustrating the limitations of others' work and the generality and power of our approach. Note that the simple examples are included to illustrate the completeness of our approach and are not meant to indicate that other methods cannot handle them nor that they might not be a more straightforward approach in that particular instance.

We begin this chapter using the deductive data model given in Chapter 3. However, for first-order constraints, we omit the empty programs and use the simpler syntax of Chapter 2. We adopt the convention that constants not enclosed in quotes, such as a in example 5.1, represent parameters that will be known at runtime. Also to simplify presentation, we omit extraneous attributes.

Example 5.1 Consider a simple not-NULL constraint and the insertion of a single tuple.

Let C_p be a constraint and $U(a)$ a parameterised update defined as follows:

$$\begin{array}{rcl} C_p & = & \forall x(p(x) \rightarrow x \neq NULL) \\ U(a) & = & \forall x\ (x = a \rightarrow +p(x)) \end{array}$$

Then we apply the wp transformation and simplify as follows:

$$\begin{array}{rcl} wp(U(a), C_p) & = & \forall x((p(x) \vee x = a) \rightarrow x \neq NULL) \\ & \equiv & \forall x(x = a \rightarrow x \neq NULL) \\ & = & a \neq NULL \end{array}$$

since C_p holds in the initial database state.

This result is exactly as expected; assuming that the integrity constraints hold before performing the update, then only the tuple being inserted needs to be checked. □

Example 5.2 Consider two attributes in different relations, P and Q, governed by a not-NULL constraint. The update copies some subset of Q, determined by the predicate $r(x)$, into P.

Let C_p and C_q be constraints and U an update defined as follows:

$$\begin{aligned} C_p &= \forall x(p(x) \rightarrow x \neq NULL) \\ C_q &= \forall x(q(x) \rightarrow x \neq NULL) \\ U &= \forall x\ (q(x) \wedge r(x) \rightarrow +p(x)) \end{aligned}$$

Then we apply the wp transformation and simplify as follows:

$$\begin{aligned} wp(U, C_p) &= \forall x((p(x) \vee (q(x) \wedge r(x))) \rightarrow x \neq NULL) \\ &\equiv \forall x((q(x) \wedge r(x)) \rightarrow x \neq NULL) \end{aligned}$$

which is subsumed by C_q so can be further simplified to $true$. □

Here we see an example where knowledge of the source of the tuples to be inserted into P can be utilised to avoid unnecessary checking. Integrity constraint checking methods, such as those described by Nicolas [Nic82] and Hsu and Imielinski [HI85], that operate simply on the deltas (the materialised sets of tuples to be added to and deleted from the database), would still need to check that each inserted tuple was not-NULL.

Example 5.3 Consider again the previous example, but for some condition $\Phi(x)$ that is much more complex and expensive to check than not-NULL.

Let C'_p and C'_q be constraints and U an update defined as follows:

$$\begin{aligned} C'_p &= \forall x(p(x) \rightarrow \Phi(x)) \\ C'_q &= \forall x(q(x) \rightarrow \Phi(x)) \\ U &= \forall x\ (q(x) \wedge r(x) \rightarrow +p(x)) \end{aligned}$$

Then we apply the wp transformation and simplify as follows:

$$\begin{aligned} wp(U, C'_p) &= \forall x((p(x) \vee (q(x) \wedge r(x))) \rightarrow \Phi(x)) \\ &\equiv \forall x((q(x) \wedge r(x)) \rightarrow \Phi(x)) \end{aligned}$$

which is subsumed by C'_q so can be further simplified to $true$. □

Clearly there is a potentially large efficiency gain here in avoiding the check of $\Phi(x)$ entirely when compared again to Nicolas's and Hsu and Imielinski's methods These would need to check $\Phi(a)$ for every tuple a inserted into P (i.e., every tuple satisfying $q(x) \wedge r(x)$).

Example 5.4 Consider a simple referential integrity constraint and the insertion of a set of tuples.

Let C_{pq} be a constraint and U an update defined as follows:

$$\begin{aligned} C_{pq} &= \forall x(p(x) \rightarrow q(x)) \\ U &= \forall x\ (r(x) \rightarrow +p(x)) \end{aligned}$$

Then we apply the wp transformation and simplify as follows:

$$\begin{aligned} wp(U, C_{pq}) &= \forall x((p(x) \vee r(x)) \rightarrow q(x)) \\ &= C_{pq} \wedge \forall x(r(x) \rightarrow q(x)) \\ &\equiv \forall x(r(x) \rightarrow q(x)) \end{aligned}$$

since C_{pq} holds in the initial database state. □

Similarly to example 5.1, we see that it is only necessary to check referential integrity for the newly inserted tuples.

While the previous examples were straightforward, we now examine an example where the preservation of the knowledge of the source of the tuples to be inserted allows much greater simplification.

Example 5.5 Let C_{pq} be the constraint from the previous example, and C_{rq} a constraint and U an update defined as follows:

$$\begin{aligned} C_{rq} &= \forall x(r(x) \rightarrow q(x)) \\ U &= \forall x\ (r(x) \rightarrow +p(x)) \end{aligned}$$

Then we apply the wp transformation and rewrite as follows:

$$\begin{aligned} wp(U, C_{pq}) &= \forall x((p(x) \vee r(x)) \rightarrow q(x)) \\ &= C_{pq} \wedge C_{rq} \end{aligned}$$

which can be simplified to $true$. □

Again, methods based on deltas cannot perform this kind of simplification since they do not retain the knowledge that the tuples being inserted into P come from R which we already know satisfies the referential integrity constraint.

We now formalise these examples to obtain some general results about the possibility of simplifying the weakest precondition $wp(U, C)$ given that C holds in the current database state.

Lemma 5.1 *Let B be a database, D_i a disjunction of literals $l_{i,j}$ possibly containing an existential quantifier whose scope is the disjunction, H a first-order constraint of the form $\forall \overline{x}\ (D_1 \wedge \cdots \wedge D_n)$, and S a statement. Suppose that the constraint H is true in B and that the predicate corresponding to some relation that S inserts into (resp., deletes from) occurs negatively (resp., positively) in H. Then there exists a formula wp', simpler than $wp(S, H)$, such that wp' is true in B if and only if $wp(S, H)$ is true in B.*

Proof For simplicity of presentation and without loss of generality, we assume S is a single insert or delete statement and that the updated predicate appears only once in the constraint H. We can assume a single update statement since, if G is simpler than $wp(S, H)$, then $wp(S', G)$ is simpler than $wp(S', wp(S, H))$. If the updated predicate occurs more than once in the constraint, then that only offers further opportunity for simplification, since we can simplify one occurrence at a time.

Let S be the statement $\forall \overline{x}'\ (r(\overline{x}') \rightarrow +p(\overline{x}'))$, and l_{ij} be $\neg p(\overline{u})$. Then, assuming $B \models H$

$$
\begin{aligned}
& wp(S, H) \\
& \quad = \ \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_{i-1} \wedge D_{i+1} \wedge \cdots \wedge D_n \wedge \\
& \qquad\qquad\qquad \exists \overline{y}\ (l_{i1} \vee \cdots \vee \neg(p(\overline{u}) \vee r(\overline{u})) \vee \cdots \vee l_{i,k_i})) \\
& \quad \equiv \ \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_{i-1} \wedge D_{i+1} \wedge \cdots \wedge D_n \wedge \\
& \qquad\qquad\qquad \exists \overline{y}\ (l_{i1} \vee \cdots \vee (\neg p(\overline{u}) \wedge \neg r(\overline{u})) \vee \cdots \vee l_{i,k_i})) \\
& \quad \equiv \ \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_n \wedge \\
& \qquad\qquad\qquad \exists \overline{y}\ (l_{i1} \vee \cdots \vee l_{ij-1} \vee \neg r(\overline{u}) \vee l_{ij+1} \vee \cdots \vee l_{i,k_i})) \\
& \quad \equiv \ H \wedge \forall \overline{x} \exists \overline{y}\ (l_{i1} \vee \cdots \vee l_{ij-1} \vee \neg r(\overline{u}) \vee l_{ij+1} \vee \cdots \vee l_{i,k_i}) \\
& \quad \equiv \ \forall \overline{x} \exists \overline{y}\ (r(\overline{u}) \rightarrow l_{i1} \vee \cdots \vee l_{ij-1} \vee l_{ij+1} \vee \cdots \vee l_{i,k_i}) \\
& \qquad\qquad\qquad \text{(since, by assumption, } B \models H\text{)}
\end{aligned}
$$

This last formula is simpler to check than $wp(S, H)$ because it consists of only one of the n conjuncts in $wp(S, H)$. The case for deletion is similar. □

Both referential integrity constraints and functional dependencies can

be expressed as constraints in the form required by Lemma 5.1. Hence they can be simplified in this manner.

Also, depending on the form of S and H, greater simplifications than Lemma 5.1 suggests may be possible, as the following example illustrates.

Example 5.6 Let H be a constraint and S a statement defined as follows:

$$\begin{aligned} H &= \forall \overline{x}\,(p(\overline{x}) \rightarrow q(\overline{x})) \\ S &= \forall \overline{x}\,(r(\overline{x}) \rightarrow +p(\overline{x}))\ ;\ \forall \overline{x}\,(r(\overline{x}) \rightarrow +q(\overline{x})) \end{aligned}$$

Then,

$$\begin{aligned} wp(S,H) &= \forall \overline{x}\,(p(\overline{x}) \vee r(\overline{x}) \rightarrow q(\overline{x}) \vee r(\overline{x})) \\ &\equiv \forall \overline{x}\,(p(\overline{x}) \rightarrow (q(\overline{x}) \vee r(\overline{x}))) \wedge \\ &\quad\ \forall \overline{x}\,(r(\overline{x}) \rightarrow (q(\overline{x}) \vee r(\overline{x}))) \\ &\equiv \forall \overline{x}\,(p(\overline{x}) \rightarrow (q(\overline{x}) \vee r(\overline{x}))) \\ &\equiv true \end{aligned}$$

given that H is true in B. Thus S may be performed safely without checking $wp(S,H)$ at all. □

It is worth noting that for this example, the methods described in [Nic82], [LST87], and [BDM88] would all require checks equivalent to

$$\forall \overline{x}\,(r(\overline{x}) \rightarrow q(\overline{x}))$$

whereas our approach requires no checking at all.

Under certain circumstances, notably when inserting and deleting facts, we may also simplify the weakest precondition when the constraint contains an existential quantifier following the outermost universal quantifier.

Example 5.7 (See [MH89, p. 58].) Let H be a constraint and S a statement defined as follows:

$$\begin{aligned} H &= \forall x \exists y\,(p(x) \rightarrow q(x,y)) \\ S &= -q(a,b) \end{aligned}$$

Then,

$$\begin{aligned}
wp(S,H) &= \forall x \exists y\ (p(x) \rightarrow q(x,y) \wedge x,y \neq a,b) \\
&\equiv \forall x \exists y\ (x \neq a \rightarrow (p(x) \rightarrow q(x,y) \wedge x,y \neq a,b)) \wedge \\
&\quad\ \forall x \exists y\ (x = a \rightarrow (p(x) \rightarrow q(x,y) \wedge x,y \neq a,b)) \\
&\equiv \forall x \exists y\ (x \neq a \rightarrow (p(x) \rightarrow q(x,y))) \wedge \\
&\quad\ \exists y\ (p(a) \rightarrow q(a,y) \wedge y \neq b) \\
&\equiv \forall x \exists y\ (p(x) \rightarrow q(x,y)) \wedge \\
&\quad\ \exists y\ (p(a) \rightarrow q(a,y) \wedge y \neq b) \\
&\equiv H \wedge \exists y\ (p(a) \rightarrow q(a,y) \wedge y \neq b) \\
&\equiv \exists y\ (p(a) \rightarrow q(a,y) \wedge y \neq b)
\end{aligned}$$

given that H is true in B. □

Furthermore, because we treat updates as a whole, we can also simplify some constraints whose outermost quantifier is existential.

Example 5.8 Let H be a constraint and S a statement defined as follows:

$$\begin{aligned}
H &= \exists \overline{x}\ p(\overline{x}) \\
S &= -p(\overline{a})\ ;\ +p(\overline{c})
\end{aligned}$$

Then,

$$\begin{aligned}
wp(S,H) &= \exists \overline{x}\ (p(\overline{x}) \wedge \neg(\overline{x} = \overline{a})) \vee \overline{x} = \overline{c} \\
&\equiv \exists \overline{x}\ (p(\overline{x}) \wedge \neg(\overline{x} = \overline{a})) \vee \exists \overline{x}\ \overline{x} = \overline{c} \\
&\equiv \exists \overline{x}\ (p(\overline{x}) \wedge \neg(\overline{x} = \overline{a})) \vee \mathit{true} \\
&\equiv \mathit{true}
\end{aligned}$$

given that H is true in B. □

This example illustrates one instance of how our approach does not suffer from treating an *update* as a *delete* followed by an *insert*. Again, delta based methods need to treat *update* as a primitive operation and thus introduce extra complexity into the checking and simplification process [JJ91], or they treat updates as deletes followed by inserts and suffer because they cannot associate the subsequently inserted tuples with those that were deleted.

Lemma 5.2 *Let B be a database, D_i a disjunction of literals $l_{i,j}$, H a first-order constraint of the form $\forall \overline{x}\ (D_1 \wedge \cdots \wedge D_n)$, and S a statement. Suppose that the constraint H is true in B and that S affects negatively a literal $l_{i,j}$ and positively a literal $l_{i,j+1}$. Then there exists a formula wp', simpler than $wp(S,H)$, such that wp' is true in B if and only if $wp(S,H)$ is true in B.*

Proof For simplicity of presentation, we assume S is a sequence of only two simple statements and that the updated predicates appear only once each in the constraint H. Let S be the statement

$$\forall \overline{x}\ (r(\overline{x}) \rightarrow +p(\overline{x}))\ ;\ \forall \overline{x}\ (s(\overline{x}) \rightarrow +q(\overline{x})),$$

where s does not depend on p, $l_{i,j}$ be $\neg p(\overline{u})$, and $l_{i,j+1}$ be $q(\overline{v})$. Then,

$$\begin{aligned}
&wp(S,H) \\
&\quad = \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_{i-1} \wedge D_{i+1} \wedge \cdots \wedge D_n \wedge \\
&\qquad\quad (l_{i,1} \vee \cdots \vee \neg(p(\overline{u}) \vee r(\overline{u})) \vee (q(\overline{v}) \vee s(\overline{v})) \vee \cdots \vee l_{i,k_i})) \\
&\quad \equiv \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_{i-1} \wedge D_{i+1} \wedge \cdots \wedge D_n \wedge \\
&\qquad\quad (l_{i,1} \vee \cdots \vee (\neg p(\overline{u}) \wedge \neg r(\overline{u})) \vee q(\overline{v}) \vee s(\overline{v}) \vee \cdots \vee l_{i,k_i})) \\
&\quad \equiv \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_{i-1} \wedge D_{i+1} \wedge \cdots \wedge D_n \wedge \\
&\qquad\quad ((D_i \vee s(\overline{v})) \wedge (l_{i,1} \vee \cdots \vee \neg r(\overline{u}) \vee q(\overline{v}) \vee s(\overline{v}) \vee \cdots \vee l_{i,k_i}))) \\
&\quad \equiv \forall \overline{x}\ (D_1 \wedge \cdots \wedge D_{i-1} \wedge D_{i+1} \wedge \cdots \wedge D_n \wedge \\
&\qquad\quad D_i \wedge (r(\overline{u}) \rightarrow (l_{i,1} \vee \cdots \vee q(\overline{v}) \vee s(\overline{v}) \vee \cdots \vee l_{i,k_i}))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{since } B \models \forall \overline{x}\ D_i) \\
&\quad \equiv H \wedge \forall \overline{x}\ (l_{i,1} \vee \cdots \vee l_{i,j-1} \vee l_{i,j+1} \vee \cdots \vee l_{i,k_i} \vee \neg r(\overline{u}) \vee s(\overline{v})) \\
&\quad \equiv \forall \overline{x}\ (r(\overline{u}) \rightarrow (s(\overline{v}) \vee l_{i,1} \vee \cdots \vee l_{i,j-1} \vee l_{i,j+1} \vee \cdots \vee l_{i,k_i}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{since } B \models H)
\end{aligned}$$

Again, it is straightforward to extend this to constraints containing the updated predicate more than once and updates consisting of more than a single statement. □

Lemma 5.2 is an important simplification not offered by most other integrity constraint simplification methods since it takes account of the consequences of an update sequence. For example, if we also know that $\forall \overline{x}\ (r(\overline{x}) \rightarrow s(\overline{x}))$, then we could further simplify the above to $true$. This would be the case if $R \equiv S$.

Of course, it is not always possible to simplify $wp(S,H)$ as the following example illustrates.

Example 5.9 Let H be a constraint and S a statement defined as follows:

$$\begin{aligned} H &= \exists \overline{x}\ p(\overline{x}) \\ S &= -p(\overline{a}) \end{aligned}$$

Then,

$$wp(S,H) = \exists \overline{x}\ (p(\overline{x}) \wedge \overline{x} \neq \overline{a})$$

which cannot be simplified and must thus always be checked. □

Note that this inability to simplify is not a limitation of our approach, but rather an inherent property of certain constraints; there is no method which could avoid making this check.

At present, we are unable to completely characterise the conditions under which a constraint is simplifiable with respect to an update using this direct approach. However, we have identified certain important classes of integrity constraints which we can simplify, including those constraints expressing referential integrity and functional dependencies, as demonstrated in lemmas 5.1 and 5.2.

5.3 Simplification by Update Propagation

So far, this chapter has examined a *direct* approach to simplifying the condition $wp(S, H)$ for what are essentially first-order constraints. Work on integrity constraint checking for the deductive database system EKS built at ECRC adopted a different approach [Bay92]. Specifically, it focused on the propagation of incremental changes to the intensional database defined by the database and its associated rules.

This method captured the exact incremental change to the intension database by means of a set of meta-rules generated from the rule base, and the set of tuples corresponding to a particular update's deltas.

For consistency with their approach we adopt the convention that constraints are expressed in the negative, as rules of the form:

$$inconsistent \leftarrow \neg C,$$

Then, if the fact $inconsistent$ is added to the intensional database, we can conclude that the integrity constraints have been violated by the update. The drawbacks of their approach centre around the need to evaluate some terms of the transformed rules with respect to the old database state and other terms with respect to the new database state.

In the following we show how to augment their technique with our weakest precondition transformation, to produce transformed rules which need only be evaluated with respect to the old database state.

5.3.1 Deductive Databases

We begin with a brief overview of the results presented by Bayer in [Bay92] then present our extensions which allow for a more complex update language and for the transformed rules to be evaluated solely with respect to the initial database state.

In the following, we assume a database rule has the form:

$$A \leftarrow L_1 \wedge \cdots \wedge L_n$$

The meta-rules for induced additions of A are:

- for each positive literal L_i:

  $add(A) \leftarrow add(L_i) \wedge new(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n).$

- and for each negative literal $\neg L_i$:

  $add(A) \leftarrow del(L_i) \wedge new(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n).$

where $new(L_i)$ indicates that L_i must be evaluated in the updated database.

For induced deletions it is slightly more complicated. It is not enough that a derivation path to a virtual fact be deleted, we must ensure that no other derivation paths exist.

The meta-rules for induced deletions of A are:

- for each positive literal L_i:

  $del(A) \leftarrow del(L_i) \wedge old(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n) \wedge \neg new(A).$

- and for each negative literal $\neg L_i$:

  $del(A) \leftarrow add(L_i) \wedge old(L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n) \wedge \neg new(A).$

In addition, for every tuple $p(\overline{a})$ added to a base relation P, there is the associated base fact $add(p(\overline{a}))$, and for every tuple $p(\overline{a})$ deleted from a base relation P, there is the associated base fact $del(p(\overline{a}))$. These correspond to the positive and negative deltas usually considered to characterise an update for the purposes of integrity constraint checking. So, naturally, the same tuple cannot be both added to and deleted from a base relation.

This is the essence (ignoring the “bottom-up propagation” evaluation method) of the technique described in [Bay92].

Example 5.10 Given the following program

$$
\begin{aligned}
& inconsistent \leftarrow \\
& \quad p(X) \wedge \neg q(X). \\
& q(X) \leftarrow r(X, Y).
\end{aligned}
$$

we would generate the following meta-rules

$$
\begin{aligned}
& add(inconsistent) \leftarrow \\
& \quad add(p(X)) \wedge \neg new(q(X)). \\
& add(inconsistent) \leftarrow \\
& \quad del(q(X)) \wedge new(p(X)). \\
& add(q(X)) \leftarrow \\
& \quad add(r(X, Y)). \\
& del(inconsistent) \leftarrow \\
& \quad del(p(X)) \wedge \neg old(q(X)) \wedge \neg new(inconsistent). \\
& del(inconsistent) \leftarrow \\
& \quad add(q(X)) \wedge old(p(X)) \wedge \neg new(inconsistent). \\
& del(q(X)) \leftarrow \\
& \quad del(r(X, Y)) \wedge \neg new(q(X)).
\end{aligned}
$$

□

To eliminate the meta-predicate new from the above meta-rules, we make use of the weakest precondition predicate transformer wp described in chapter 3.

In the above transformation, the set of generated meta-rules were essentially independent of the update, since it was implicit in the *old/new* distinction and the set of $add/1$, $del/1$ base facts. In eliminating the meta-predicate new through the use of the predicate transformer, the meta-rules become specific to a particular update. Thus we need to encode this information explicitly. We do this by adding (a representation of) the update as an extra argument to the meta-rules.

Now, for every rule of the form:

$$A \leftarrow L_1 \wedge \cdots \wedge L_n$$

we generate:

- $add(U, A) \leftarrow add(U, L_i) \wedge wp(U, L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n).$

- $del(U, A) \leftarrow del(U, L_i) \wedge L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n \wedge \neg wp(U, A)$.

for positive literals L_i, and

- $add(U, A) \leftarrow del(U, L_i) \wedge wp(U, L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n)$.
- $del(U, A) \leftarrow add(U, L_i) \wedge L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n \wedge \neg wp(U, A)$.

for negative literals $\neg L_i$.

In addition, for every base relation A we generate the rules:

$$add(U, A) \leftarrow wp(U, A) \wedge \neg A.$$

$$del(U, A) \leftarrow A \wedge \neg wp(U, A).$$

Note that these meta-rules are dependent on the update language only indirectly; through the predicate transformer wp. Since we have previously defined wp for an update language which allows a sequence of set-oriented updates, so do these meta-rules now allow for our more expressive update language.

Example 5.11 Given the same program as example 5.10

$$inconsistent \leftarrow p(X) \wedge \neg q(X).$$
$$q(X) \leftarrow r(X, Y).$$

we would generate the following meta-rules

$$add(U, inconsistent) \leftarrow add(p(X)) \wedge \neg wp(U, q(X)).$$
$$add(U, inconsistent) \leftarrow del(q(X)) \wedge wp(U, p(X)).$$
$$add(U, q(X)) \leftarrow add(U, r(X, Y)).$$
$$del(U, inconsistent) \leftarrow del(p(X)) \wedge \neg q(X) \wedge \neg wp(U, inconsistent).$$
$$del(U, inconsistent) \leftarrow add(q(X)) \wedge p(X) \wedge \neg wp(U, inconsistent).$$

$$del(U, q(X)) \leftarrow del(U, r(X, Y)) \wedge \neg wp(U, q(X)).$$
$$add(U, p(X)) \leftarrow wp(U, p(X)) \wedge \neg p(X).$$
$$del(U, p(X)) \leftarrow p(X) \wedge \neg wp(U, p(X)).$$
$$add(U, r(X, Y)) \leftarrow wp(U, r(X, Y)) \wedge \neg r(X, Y).$$
$$del(U, r(X, Y)) \leftarrow r(X, Y) \wedge \neg wp(U, r(X, Y)).$$

These meta-rules are much more useful than those of example 5.10 since they can be evaluated with respect to just the initial state rather than requiring both the initial and updated states. □

We now revisit the two lemmas 5.1 and 5.2 from section 5.2.

Lemma 5.3 *Let B be a database, D_i a disjunction of literals $l_{i,j}$ possibly containing an existential quantifier whose scope is the disjunction, H a first-order constraint of the form $\forall\overline{x}\ (D_1 \wedge \cdots \wedge D_n)$, and S a statement. Suppose that the constraint H is true in B and that the predicate corresponding to some relation that S inserts into (resp., deletes from) occurs negatively (resp., positively) in H. Then there exists a formula wp', simpler than $wp(S,H)$, such that wp' is true in B if and only if $wp(S,H)$ is true in B.*

Proof Let $l_{i,j}$ be the term $p(\overline{s})$ and S be the update $\forall\overline{x}\ (r(\overline{x}) \rightarrow +p(\overline{x}))$ which we write, Prolog-style, as the single element list: $[ins(p(\overline{x}), r(\overline{x}))]$. Then the constraint would be expressed in clausal form as follows:

$$\begin{array}{lcl} inconsistent & \leftarrow & \neg D_1. \\ & \vdots & \\ inconsistent & \leftarrow & \neg l_{i,1} \wedge \cdots \wedge p(\overline{s}) \wedge \cdots \wedge \neg l_{i,k_i}. \\ & \vdots & \\ inconsistent & \leftarrow & \neg D_n. \end{array}$$

and the update S would give us the propagation meta-rule

$$add([ins(p(\overline{x}), r(\overline{x}))], p(\overline{x})) \leftarrow r(\overline{x}).$$

Applying the transformation given above results in the following meta-rules

$$\begin{array}{l} add(U, inconsistent) \leftarrow \\ \qquad\qquad add(U, \neg D_1). \\ \quad\vdots \\ add(U, inconsistent) \leftarrow \\ \qquad\qquad add(U, p(\overline{s})) \wedge \\ \qquad\qquad wp(U, \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j-i} \wedge \neg l_{i,j+1} \wedge \cdots \wedge \neg l_{i,k_i}). \\ \quad\vdots \\ add(U, inconsistent) \leftarrow \\ \qquad\qquad add(U, \neg D_n). \end{array}$$

This can be simplified (via straightforward unfolding) to

$$\begin{array}{l} add([ins(p(\overline{x}), r(\overline{x}))], inconsistent) \leftarrow \\ \qquad\qquad r(\overline{s}) \wedge \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j-i} \wedge \\ \qquad\qquad \neg l_{i,j+1} \wedge \cdots \wedge \neg l_{i,k_i}. \end{array}$$

(assuming that $p(\overline{x})$ does not occur in $D_j, i \neq j$.)

Then, wp' is

$$r(\overline{s}) \wedge \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j-i} \wedge \neg l_{i,j+1} \wedge \cdots \wedge \neg l_{i,k_i}$$

which is simpler than $wp(U, C)$. □

This is an equivalent result to that obtained in lemma 5.1.

Lemma 5.4 *Let B be a database, D_i a disjunction of literals $l_{i,j}$, H a first-order constraint of the form $\forall\overline{x}\ (D_1 \wedge \cdots \wedge D_n)$, and S a statement. Suppose that the constraint H is true in B and that S affects negatively a literal $l_{i,j}$ and positively a literal $l_{i,j+1}$. Then there exists a formula wp', simpler than $wp(S, H)$, such that wp' is true in B if and only if $wp(S, H)$ is true in B.*

Proof Let S be the statement

$$\forall\overline{x}\ (r(\overline{x}) \rightarrow +p(\overline{x}))\ ;\ \forall\overline{x}\ (s(\overline{x}) \rightarrow +q(\overline{x})),$$

where s does not depend on p, $l_{i,j}$ be $\neg p(\overline{u})$, and $l_{i,j+1}$ be $q(\overline{v})$. Again, we write the statement S as the Prolog term: $[ins(p(\overline{x}), r(\overline{x})), ins(q(\overline{x}), s(\overline{x}))]$. Then the constraint would be expressed in clausal form as follows:

$$\begin{array}{lcl} inconsistent & \leftarrow & \neg D_1. \\ & \vdots & \\ inconsistent & \leftarrow & \neg l_{i,1} \wedge \cdots \wedge p(\overline{u}) \wedge \neg q(\overline{v}) \wedge \cdots \wedge \neg l_{i,k_i}. \\ & \vdots & \\ inconsistent & \leftarrow & \neg D_n. \end{array}$$

and the update S would give us the propagation meta-rules

$$\begin{array}{lcl} add([ins(p(\overline{x}), r(\overline{x})), ins(q(\overline{x}), s(\overline{x}))], p(\overline{x})) & \leftarrow & r(\overline{x}). \\ add([ins(p(\overline{x}), r(\overline{x})), ins(q(\overline{x}), s(\overline{x}))], q(\overline{x})) & \leftarrow & s(\overline{x}). \end{array}$$

Applying the propagation transformation results in the following meta-rules

$$\begin{array}{lcl} add(U, inconsistent) & \leftarrow & add(U, \neg D_1). \\ & \vdots & \\ add(U, inconsistent) & \leftarrow & \\ \quad add(U, p(\overline{s})) \wedge wp(U, \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j-i} \wedge \neg l_{i,j+1} \wedge \cdots \wedge \neg l_{i,k_i}). & & \\ add(U, inconsistent) & \leftarrow & \\ \quad del(U, q(\overline{s})) \wedge wp(U, \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j} \wedge \neg l_{i,j+2} \wedge \cdots \wedge \neg l_{i,k_i}). & & \\ & \vdots & \\ add(U, inconsistent) & \leftarrow & add(U, \neg D_n). \end{array}$$

Then,

$$\begin{aligned} wp' = \quad & add([ins(p(\overline{x}), r(\overline{x})), ins(q(\overline{x}), s(\overline{x}))], inconsistent) \\ \equiv \quad & r(\overline{s}) \wedge wp(U, \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j-i} \wedge \neg l_{i,j+1} \wedge \cdots \wedge \neg l_{i,k_i}). \\ \equiv \quad & r(\overline{s}) \wedge \neg l_{i,1} \wedge \cdots \wedge \neg l_{i,j-i} \wedge \neg q(\overline{v}) \wedge \neg s(\overline{v}) \wedge \\ & \qquad \neg l_{i,j+2} \wedge \cdots \wedge \neg l_{i,k_i}. \end{aligned}$$

□

This is an equivalent result to that obtained in lemma 5.2.

Less than 100 lines of simple (uncommented) Prolog code (see Appendix B) is needed to implement the wp and update propagation transformations and runs quickly enough for interactive use; more than sufficient for a compile-time transformation. The resulting tests, after straightforward constant propagation and similar standard optimisations, effectively factor in the assumption that C already holds in B[1].

5.3.2 Deductive Object-Oriented Databases

Turning our attention to the interesting problem of adapting this method to Gulog, we immediately encounter the problem of overriding.

Consider the following program:

Schema:	Extension:
$student < person.$	$john : student.$
$person[p \Rightarrow any].$	$jane : student.$
$person[q \Rightarrow any].$	$john[p \rightarrow a].$
$\{X : person, Y : any\} \vdash X[r \rightarrow Y] \leftarrow X[p \rightarrow Y].$	$jane[p \rightarrow a].$
$\{X : student, Y : any\} \vdash X[r \rightarrow Y] \leftarrow X[q \rightarrow Y].$	$jane[q \rightarrow b].$

Gulog's overriding semantics mean that $john[r \rightarrow a]$ and $jane[r \rightarrow b]$ are the only answers to the query $\{X : student, Y : any\} \vdash X[r \rightarrow Y]$. The atom $jane[r \rightarrow a]$ is not an answer because the rule for $student$s has an answer for $jane$ and thus overrides the rule for $person$s.

Deleting the fact $jane[q \rightarrow b]$ not only induces the deletion of $jane[r \rightarrow b]$, but also induces the addition of $jane[r \rightarrow a]$. This is because the rule for

[1]However, see 5.4 for further information about this. Propagation does not do everything. There is still a place for Semantic Query Optimisation [CGM90].

$student$s applied to $jane$ no longer supplies an answer and thus no longer overrides the rule for $person$s.

When overriding is involved like this, there are three ways an addition can be induced. (We ignore object creation and destruction for the moment.) Firstly, it can be induced directly by an addition in the body of the rule from the most specific subclass. (By *most specific subclass* we mean the subclass lowest in the hierarchy in which code defining the method exists.) Secondly, it can be induced directly by an addition in the body of a rule from some other class which is not overridden by a more specific subclass. Thirdly, as illustrated above, it can be induced by a deletion which causes a rule in a more specific subclass to no longer override a rule in a less specific subclass.

This fact can be most clearly understood by considering the translation of a Gulog program into its Datalog equivalent as described by Dobbie and Topor in [DT94].

Consider the following Gulog program:

$$c_1 < c_2.$$
$$c_2[m \Rightarrow any].$$
$$\{X : c_2, Y : any\} \vdash \quad X[m \rightarrow Y] \leftarrow q(X, Y).$$
$$\{X : c_1, Y : any\} \vdash \quad X[m \rightarrow Y] \leftarrow r(X, Y).$$

The essentials of a translation to Datalog are as follows:

$$m_c_2(X, Y) \leftarrow \quad isa_c_2(X) \wedge q(X, Y) \wedge \neg em_{c_1}(X).$$
$$m_c_2(X, Y) \leftarrow \quad isa_c_1(X) \wedge r(X, Y).$$

$$em_{c_2}(X) \leftarrow \quad isa_c_2(X) \wedge q(X, Y).$$
$$em_{c_1}(X) \leftarrow \quad isa_c_1(X) \wedge r(X, Y).$$

That is, we replace every Gulog method-term $m/k@c'$ with a corresponding predicate m_c of arity $k + 1$ where c is the most general superclass of c defining a method m of arity k, the first argument is the object on which the method is invoked, and the remaining k arguments correspond to the original k arguments to the method. To capture the type information the predicate $isa_c'(X)$ is inserted at the beginning of the clause

body. (This effectively makes all clauses range-restricted.) In addition, we append to the bodies the literals $\neg em_{c_i}(\overline{X})$ for every subclass c_i of c'.

Finally, for every clause that defines a method-term $m/k@c$ we add a clause defining $em_c/1$ where the body is the same as that defining the method $m/k@c$ but with method terms substituted by their $m_c/k+1$ equivalents and again with the term $isa_c(X)$ inserted.

The em clauses are used to capture the overriding semantics. That is, $em_c(o)$ is true if there is a possible answer for the method m applied to the object o using just the definition of m from the class c. Thus, if $em_c(o)$ is true, then there can be no answers for m applied to o derived from superclasses of c.

Applying the translation from section 5.3.1 (again assuming, for clarity, the update U performs no object creation or deletion), we generate the meta-rules:

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_2(X) \wedge add(U, q(X, Y)) \wedge \neg wp(U, em_{c_1}(X)).$$

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_2(X) \wedge del(U, em_{c_1}(X)) \wedge wp(U, q(X, Y)).$$

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_1(X) \wedge add(U, r(X, Y)).$$

$$del(U, em_{c_2}(X)) \leftarrow \\ \quad isa_c_2(X) \wedge del(U, q(X, Y)) \wedge \neg wp(U, em_{c_2}(X)).$$

$$del(U, em_{c_1}(X)) \leftarrow \\ \quad isa_c_1(X) \wedge del(U, r(X, Y)) \wedge \neg wp(U, em_{c_1}(X)).$$

Unfolding and simplifying these rules gives:

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_2(X) \wedge add(U, q(X, Y)) \wedge \neg isa_c_1(X).$$

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_1(X) \wedge add(U, q(X, Y)) \wedge \neg wp(U, r(X, Z)).$$

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_1(X) \wedge del(U, r(X, Z_1)) \wedge \neg wp(U, r(X, Z_2)).$$

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_1(X) \wedge add(U, r(X, Y))$$

Clearly, the three rules beginning with $isa_c_1(X)$ in the body correspond to the three cases detailed on page 74.

Note that for the sake of clarity we have omitted the meta-rules that would normally be generated for each of the base relations.

Extending this to allow for updates which may perform object creation or destruction, the translation from section 5.3.1 gives the following meta-rules (before simplification):

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_2(X) \wedge add(U, q(X, Y)) \wedge \neg wp(U, em_{c_1}(X)).$$
$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_2(X) \wedge del(U, em_{c_1}(X)) \wedge wp(U, q(X, Y)).$$
$$add(U, m(X, Y)) \leftarrow \\ \quad add(U, isa_c_2(X)) \wedge wp(U, q(X, Y) \wedge \neg em_{c_2}(X)).$$

$$add(U, m(X, Y)) \leftarrow \\ \quad isa_c_1(X) \wedge add(U, r(X, Y)).$$
$$add(U, m(X, Y)) \leftarrow \\ \quad add(U, isa_c_1(X)) \wedge wp(U, r(X, Y)).$$

$$del(U, em_{c_2}(X)) \leftarrow \\ \quad isa_c_2(X) \wedge del(U, q(X, Y)) \wedge \neg wp(U, em_{c_2}(X)).$$
$$del(U, em_{c_2}(X)) \leftarrow \\ \quad del(U, isa_c_2(X)) \wedge q(X, Y) \wedge \neg wp(U, em_{c_2}(X)).$$
$$del(U, em_{c_1}(X)) \leftarrow \\ \quad isa_c_1(X) \wedge del(U, r(X, Y)) \wedge \neg wp(U, em_{c_1}(X)).$$
$$del(U, em_{c_1}(X)) \leftarrow \\ \quad del(U, isa_c_1(X)) \wedge r(X, Y) \wedge \neg wp(U, em_{c_1}(X)).$$

Thus, to apply our integrity constraint checking mechanism to Gulog databases we first translate those methods relevant to the constraints into their Datalog equivalent and then apply the update propagation transformation to the results.

5.4 Weakest Precondition Simplification Problem

We call the problem tackled in this chapter the *Weakest Precondition Simplification Problem* (WPSP).

The general form of the WPSP can be stated as follows. Given a constraint (sentence) H, an update statement S, a cost function $cost$, and a bound b such that $b < cost(wp(S, H))$, does there exist a constraint C such that $cost(C) \leq b$ and $wp(S, H) \stackrel{H}{\equiv} C$. That is, does there exist a bounded-cost constraint C such that the truth value of $wp(S, H)$ is the same as the truth value of C in all models of H.

Transaction Safety is a simpler problem than WPSP since, given H and S as above, we only need to determine if $wp(S, H) \stackrel{H}{\equiv} true$.

This statement of the WPSP immediately raises the question, what is an appropriate choice for the cost function. There are several possibilities.

- *Syntactic Complexity*

  This measure usually counts the number of terms in some normal form of the queries. For example, $p(x)$ would be considered simpler than $p(x) \vee q(x)$. Due to the structural similarity of the queries and the nature of the transformers we use, this is arguably a reasonable measure.

- *Semantic Complexity*

  This is a fairly course grained measure. It distinguishes between, for example, a query that requires a fixpoint computation (due to recursion) and a (simpler) query that requires only joins. For example, see Chandra and Harel's work on query complexity [CH82] and Lawley's on update language complexity [Law92].

- *Query Engine Relative*

  This is closely related to the cost functions traditionally used in Semantic Query Optimisation. It may be some combination of the above measures plus estimates based on the relative sizes of the actual database tables and may even take account of statistics from the history of previous query execution times.

Ideally, one would like the cost function to be useful in guiding the simplification process itself, but this is really only feasible for a syntactic complexity cost function.

Theorem 5.1 *Let H be a (first-order) statement and S a sentence as per the the relational data model of chapter 2. Then,* WPSP *is co-recursively enumerable (co-r.e.).*

Proof To see that WPSP is at least co-r.e., imagine enumerating all formulas simpler than $wp(S, H)$ and testing, in parallel, each one for equivalence with $wp(S, H)$. Given that there are no function symbols, this enumeration will be finite. However, the equivalence test for first-order formulas is co-r.e. and not recursive, thus giving a co-r.e. solution for WPSP.

To show that WPSP is not recursive, we show that the subset of WPSP instances corresponding to Transaction Safety are equivalent to determining unsatisfiability.

In order to show that satisfiability is not recursive, it is shown that, given an instance of the *Post correspondence problem* (PCP) a formula P can be constructed that encodes this instance. Given this encoding, we show that an instance of WPSP can be constructed that also encodes the PCP instance.

Let P be a formula encoding an instance of PCP and q be a predicate not occurring in the encoding P. Let H and S be defined as follows:

$$\begin{aligned} H &= q(a) \\ S &= \forall\,(P \rightarrow -q(a)) \end{aligned}$$

Applying the weakest precondition transformation gives:

$$\begin{aligned} wp(S,H) &= q(a) \wedge \neg P \\ &= H \wedge \neg P \end{aligned}$$

If we now let the bound b be $cost(true)$, then this reduces WPSP to the question of whether $wp(S,H)$ is satisfiable in all models of H. Clearly, $wp(S,H)$ is satisfiable in all models of H if and only if P is not satisfiable in any model of H. Since q does not occur in P, this is equivalent to determining unsatisfiability of P, which is co-r.e. and not recursive. □

If we now consider a condition H and statement S as per the deductive data model of chapter 3, we can argue that WPSP is still co-recursively enumerable.

The argument is similar to the proof of Theorem 5.1: $wp(S,H)$ can be essentially an arbitrary condition and programs must be locally stratified (thus having a unique perfect model), unsatisfiability is co-r.e. and WPSP must also be co-r.e..

However, this argument also relies upon the ability to enumerate all conditions G with $cost(G)$ less than some bound $b < wp(S,H)$. While simple and practical syntactic cost functions are easily defined for first-order queries, the same cannot be said for conditions in our deductive data model due to the presence of recursion. It is difficult to say anything more concrete about the decidability of WPSP in this case without a full

treatment of cost functions for conditions involving recursion, which is beyond the scope of this thesis.

On a more positive note, lemmas 5.1 and 5.2 show that for certain useful and reasonably general forms of constraint and update there does exist such a bounded-cost constraint for a syntactic cost function and a bound approximately equal to $cost(wp(S, H))$.

We now discuss the application of Semantic Query Optimisation to the results of the weakest-precondition and propagation techniques.

Example 5.12 Consider the integrity constraint:

$$inconsistent \leftarrow c1 \vee c2.$$

where $c1$ and $c2$ are defined by the following rules:

$$c1 \leftarrow p(x) \wedge q(y) \wedge r(x, y).$$
$$c2 \leftarrow p(x) \wedge q(y) \wedge s(x, y).$$

Let U be the update:

$$\forall x, y\ ((p(x) \wedge q(y) \wedge r(x, y) \wedge t(x, y)) \rightarrow +s(x, y))$$

Applying the weakest precondition and update propagation transformations gives us the following rules:

$$add(inconsistent) \leftarrow add(c1) \vee add(c2).$$
$$add(c2) \leftarrow p(x) \wedge q(y) \wedge add(s(x, y)).$$
$$add(s(x, y)) \leftarrow p(x) \wedge q(y) \wedge r(x, y) \wedge t(x, y).$$

The body of $add(inconsistent)$ can be simplified to $false$ but doing this requires recognising that $add(s(x, y))$ must be false as a consequence of $c1$ being false.

This example illustrates why the propagation method alone is not sufficient to do all possible optimisations. In this case an unrelated constraint ($c1$) supplies knowledge about an update condition that potentially affects another constraint ($c2$).

In general recognising these situations is a non-trivial task requiring some form of theorem proving approach as taken by Sheard and Stemple [SS88, SS89] and Benzaken and Schaefer [BS97]. □

Example 5.13 Consider the constraint requiring that the relation P be non-empty. We write this constraint C as $\exists \overline{x}\ p(\overline{x})$. Let U be the update

$$\forall \overline{x}\ (q(\overline{x}) \rightarrow -p(\overline{x}))$$

Then, $wp(U, C)$ is

$$\exists \overline{x}\ (p(\overline{x}) \wedge \neg q(\overline{x}))$$

Even though we know that C holds before performing U, we cannot use this information to simplify $wp(U, C)$ (unless $q/1$ is defined in terms of $p/1$). □

From these examples we can see that finding a simpler condition is potentially as hard as the normal query optimisation problem.

Semantic Query Optimisation [CGM90] is a technique for using the semantic information captured in integrity constraints to simplify the evaluation of database queries. It allows a simpler query to be evaluated if it produces the same answers as the original query given that the integrity constraints are satisfied by the database.

More precisely, given a constraint H, a cost function $cost$, a query Q, and a bound b such that $b < cost(Q)$, Semantic Query Optimisation techniques attempt to find another query C such that $cost(C) \leq b$ and $Q \overset{H}{\equiv} C$.

Since $wp(S, (q, P))$ is a query, but is also syntactically similar to the integrity constraint (q, P), simplifying $wp(S, (q, P))$ is a special case of Semantic Query Optimisation. That is, by adopting the propagation approach we could take advantage of the syntactic similarity of $wp(S, (q, P))$ to (q, P), and then apply known Semantic Query Optimisation techniques to the result (i.e., $(del(q), P)$).

To illustrate this, we briefly introduce some terminology and definitions adapted from Godfrey et al. [GGM96].

Let (q, P) be a query. Then q' is an *unfolding* of q with respect to P if and only if

- $q' = q$;
- $q' = A_1 \wedge \cdots \wedge A_{i-1} \wedge B\theta \wedge A_{i+1} \wedge \cdots \wedge A_n$, where $q'' = A_1 \wedge \cdots \wedge A_{i-1} \wedge A_i \wedge A_{i+1} \wedge \cdots \wedge A_n$, q'' is an unfolding of q, and there is a rule $A \leftarrow B$ in P for which $A\theta = A_i\theta$ and θ is an mgu.

q' is a *complete unfolding* if q' consists solely of atoms referring to relations in the database, and not to rules in P.

Let q' be an unfolding of q with respect to P, and IC be the set of rules defining the database integrity constraints. q' is a *null-unfolding* of q if and only if $P \cup IC \not\models \exists q'$.

Example 5.14 Now consider example 5.12 above and the application of a simple semantic query optimisation technique involving query unfolding and the identification of null-unfoldings.

In this case, q is the query $add(inconsistent)$ and IC is the set of rules:

$$\leftarrow p(x) \wedge q(y) \wedge r(x, y),$$
$$\leftarrow p(x) \wedge q(y) \wedge s(x, y)$$

The set of complete unfoldings of q contains one element:

$$p(x) \wedge q(y) \wedge r(x, y) \wedge t(x, y)$$

which is clearly a null-unfolding and can be eliminated, leaving an empty set of unfoldings. This indicates that the query $add(inconsistent)$ has no answers and the transaction is safe. □

Results applicable to Semantic Query Optimisation, such as those of Levy and Sagiv [LS95], are applicable to WPSP. Levy and Sagiv restrict their work to first-order integrity constraints, showing that Semantic Query Optimisation can be completely done for queries (in our case, weakest preconditions) with certain restricted forms of recursive rules. However, the presence of negation and order constraints in the recursive rules can make the problem undecidable. In the case of deductive and especially deductive object-oriented databases this is likely to be a common problem. Godfrey et al. [GGM96] allow for recursion not only in the query, but also in the integrity constraints. However, they do not handle negation in the integrity constraints and queries.

Obviously, as for Semantic Query Optimisation, the general case of WPSP for databases and integrity constraints with both recursion and negation is a large and difficult problem with much potential for future work.

5.5 Summary

This chapter has built on the results from previous chapters to show how the weakest precondition can be simplified to produce a more efficient test for transaction safety.

We have shown how to apply a simple syntactic transformation along with the weakest precondition predicate transformer to the datalog transformation of a Gulog program that results in a simple and efficient safety condition. Combined with the meta-rule generation for update propagation, this gives us a method for generating simple safety conditions to enforce transaction safety which accounts for the database initially satisfying its integrity constraints.

Additionally, we have identified and given a formal statement of the WPSP and given some general results relating to it. Notably, we have proven that WPSP is co-r.e. and we have shown that certain common cases of integrity constraints, even in very general forms can be directly simplified. Also, we noted that the problem of simplifying $wp(S, H)$ given that H is true in B, is a special case of Semantic Query Optimisation [CGM90].

Chapter 6

Extensions

In this chapter we describe four extensions and alternative applications of the predicate transformer. These extensions are mostly unrelated, however they further demonstrate the power and usefulness of this technique.

Section 6.1 examines the problem of maintaining the integrity of an object-oriented database's schema by representing the schema and its associated integrity constraints explicitly in the database. Section 6.2 looks at the subclass of dynamic constraints called *transition constraints* which specify restrictions on updates that can be made to the database in terms of the old and new database states. Section 6.3 addresses some limitations of the update language we have considered so far and shows how to provide support for updates embedded in languages such as C. Finally, section 6.4 investigates the application of transaction safety to the problem of writing correct global updates for autonomous distributed databases.

6.1 Schema Constraints

Traditional integrity constraint checking focuses on updates to the data in the database (the *object-level*). We would like to extend this to include updates of the database schema as well (the *meta-level*). This is motivated by two factors. Firstly, there is the natural aesthetic and economy to using a single mechanism for managing updates and constraints at both the object and meta-levels. Secondly, it is common for modern databases, and especially object-oriented databases to make the database schema explicitly available as meta-data in a meta-schema.

The key to achieving the unification of constraint checking at both the object and meta-levels is the meta-schema; we must be able to represent the database schema at the object level, thus giving rise to a *meta-database*. Having done this, the remainder is relatively straightforward. Schema updates can be represented as updates to the meta-database, and schema constraints as integrity constraints on the meta-database.

Once the schema and schema updates have been expressed at the object level, we can apply any of the usual integrity checking methods, albeit with one caveat. Integrity checking methods that are applied *after* performing the relevant updates need to separate the implementation of the actual schema update from the update of its representation in the meta-database, otherwise the check may need to be evaluated with respect to a database with an inconsistent schema. This may break the query evaluation mechanism. However, integrity constraint checking methods such as ours, which ensure transaction safety by performing the check before carrying out the actual update, do not suffer from this restriction since the check will always be evaluated with respect to the original consistent schema.

In the following we present a (partial) meta-schema for Gulog and some typical schema constraints.

Example 6.1 Let $inh/2$ be the base relation defining the class inheritance hierarchy, and $isa/2$ be its transitive closure as defined by the program P:

$\{C, S{:}class\} \vdash isa(C, S) \leftarrow inh(C, S).$
$\{C, S, I{:}class\} \vdash isa(C, S) \leftarrow inh(C, I) \wedge isa(I, S).$

The schema constraint restricting the class hierarchy to be acyclic would then be $(\neg isa(C, C), P)$.

If U is the update $+inh(emp, person)$, then the weakest precondition $wp(U, (\neg isa(C, C), P))$ is $wp(U, (\neg isa'(C, C), P'))$ where P' is:

$\{C, S{:}class\} \vdash isa'(C, S) \leftarrow isa(C, S).$
$\{S{:}class\} \vdash isa'(emp, S) \leftarrow isa(person, S).$
$\{C{:}class\} \vdash isa'(C, person) \leftarrow isa(C, emp).$
$\{C, S{:}class\} \vdash isa'(C, S) \leftarrow isa(C, emp), isa(person, S).$

$\{C, S{:}class\} \vdash isa(C, S) \leftarrow inh(C, S).$
$\{C, S, I{:}class\} \vdash isa(C, S) \leftarrow inh(C, I), isa(I, S).$

Unfolding the constraint $(\{C{:}class\} \vdash \neg isa'(C, C), P')$ one level gives:

$(\{C{:}class\} \vdash$
$\quad \neg isa(C, C) \wedge \neg isa(person, emp) \wedge$
$\quad \neg isa(person, emp) \wedge \neg(isa(C, emp) \wedge isa(person, C)), P')$

This can then be simplified, assuming $\{C{:}class\} \vdash (\neg isa(C, C), P)$ is true, to the simpler condition $\{C{:}class\} \vdash (\neg isa(person, emp), P)$.

Hence, for the generic update $+inh(Class1, Class2)$, which adds the declaration $Class1 < Class2$ to the schema, the safe version is

$$\{C{:}class\} \vdash \neg isa(Class2, Class1) \rightarrow +inh(Class1, Class2)$$

□

Example 6.1 illustrates how one can use the wp transformer to produce *safe* schema updates from simple updates and schema constraints.

Example 6.2 Consider the constraints:

$$IC_1 \equiv \{O1, O2{:}object\} \vdash \forall O1, O2\ (O1[self \rightarrow O2] \Rightarrow O1 = O2)$$
$$IC_2 \equiv \{O{:}object, C{:}class\} \vdash \forall O, C\ (O[self \rightarrow O] \Leftrightarrow inst(O, C))$$

These require that an object's $self$ attribute refer to itself and that every object that exists has such an attribute.

We can now write a method new which can be used to create objects satisfying these constraints.

$\{O{:}object, C{:}class\} \vdash C[new \rightarrow O] \leftarrow (+O{:}C\ ;\ !O[self \rightarrow O]).$

If U is the update $person[new \rightarrow Oid]$, then the weakest preconditions $wp(U, IC_1)$ and $wp(U, IC_2)$ are:

$\{O1, O2{:}object\} \vdash$
$\quad \forall O1, O2\ (\ (O1[self \rightarrow O2] \vee (O1 = Oid \wedge O2 = Oid))$
$\qquad \Rightarrow O1 = O2\ )$

and

$\{O1, O2{:}object\} \vdash$
$\quad \forall O, C\ (\ (O[self \rightarrow O] \vee (O = Oid))$
$\qquad \Leftrightarrow (inst(O, C) \vee (O = Oid \wedge C = person))\ )$

both of which simplify to $true$ which indicates the update U is safe with respect to these integrity constraints. □

The examples in this section illustrate how it is relatively straightforward to express simple schema integrity constraints and updates in terms of a meta-schema and then to apply our integrity constraint technique to generate safe updates if necessary

6.2 Dynamic Constraints

Using the wp transformer gives us a simple means to deal with certain kinds of dynamic integrity constraints, namely those which can be specified in terms of the old and new database states. These are sometimes called *transition constraints.*

To specify these dynamic integrity constraints we need to be able to refer to relations in each of the old and new states. We introduce the following convention for doing so. Let q be a predicate symbol, then $new{:}q$ is new predicate symbol referring to q in the *new* database state. To illustrate, let S be an update statement, and $q(\overline{c})$ a ground atom, then $B \models (new{:}q(\overline{c}), P)$ if and only if $[\![S]\!](B) \models (q(\overline{c}), P)$. Recalling the definition of a weakest precondition and lemma 2.3, we see that $(new{:}q(\overline{x}), P)$ is equivalent to $wp(S, (q(\overline{x}), P))$. Hence, a dynamic integrity constraint can be treated as a static integrity constraint parameterised by an update statement (denoted here by S).

Hence we should really write $new(S){:}q$ instead of $new{:}q$ since it is parameterised by the update statement S. However, since the parameter is the same for every usage of $new{:}$, we omit it.

Example 6.3 We would write the dynamic constraint "salaries may not decrease" as follows:

$$\forall Name, X, Y \ ((sal(Name, X) \wedge new{:}sal(Name, Y)) \rightarrow X \leq Y)$$

where $new{:}sal$ refers to the relation sal in the updated database. This dynamic constraint is equivalent to the parameterised constraint:

$$\forall Name, X, Y \ ((sal(Name, X) \wedge wp(S, sal(Name, Y))) \rightarrow X \leq Y)$$

where S is an update statement.

Given a specific update $S(K)$ such as

$$\forall X, Y \ (\ (sal(\text{'Jane'}, X) \wedge Y \ is \ X + K) \\ \rightarrow (-sal(\text{'Jane'}, X) \ ; \ +sal(\text{'Jane'}, Y)) \ )$$

we can then expand the dynamic constraint as follows.

$$
\begin{aligned}
& G(K) \\
& \quad = \ \forall\, Name, X, Y\ ((sal(Name, X) \wedge wp(S, sal(Name, Y))) \rightarrow \\
& \qquad\qquad X \leq Y) \\
& \quad \equiv \ \forall\, Name, X, Y\ ((sal(Name, X) \wedge \\
& \qquad\qquad ((Name = \text{'Jane'} \wedge Y\ is\ X + K) \vee \\
& \qquad\qquad\ (Name \neq \text{'Jane'} \wedge Y = X))) \rightarrow X \leq Y) \\
& \quad \equiv \ \forall X\ (sal(\text{'Jane'}, X) \rightarrow X \leq X + K) \\
& \quad \equiv \ \forall X\ (sal(\text{'Jane'}, X) \rightarrow 0 \leq K)
\end{aligned}
$$

which refers only to the current database state and can now be used as the safety condition for $S(K)$ to generate the conditional update $G(K) \rightarrow S(K)$ which expands to:

$$
\begin{aligned}
& \forall X, Y\ (0 \leq K \wedge (sal(\text{'Jane'}, X) \wedge Y\ is\ X + K) \rightarrow \\
& \qquad (-sal(\text{'Jane'}, X)\ ;\ +sal(\text{'Jane'}, Y)))
\end{aligned}
$$

This conditional update is guaranteed not to violate the dynamic constraint since the update will only be executed if $K > 0$. □

From the above we see that a simple extension to our syntax to allow reference to both the current and potential new database states enables us to express dynamic integrity constraints. Furthermore, it is straightforward to apply the predicate transformer to generate a safety condition for each update which refers only to the current database state.

6.3 Deferred Updates

One drawback of the approach presented so far is the limited expressive power of the update language. While it is more powerful that SQL due to the ability to evaluate recursive queries, it is less powerful in a practical sense since SQL updates are nearly always embedded in some computationally complete programming language such as C or a proprietary language such as Visual Basic or Oracle's PL/SQL. To extend the applicability of the weakest precondition approach to embedded updates, we introduce the concept of *deferred updates*.

First, let us consider how a DBMS interacts with an application containing some embedded SQL queries and updates. Typically, the DBMS and the application will be separate processes which communicate using some kind of inter-process communication (IPC) protocol such as remote procedure calls, message queues, or sockets/pipes. In all these cases, what the DBMS sees, with respect to an individual transaction, is essentially a stream of individual SQL query and update operations bracketed by begin and end transaction markers. It is these individual SQL query and update operations and their sequencing which corresponds to our update language.

Example 6.4 Consider the following pseudo-code fragment which might exist in a database application program.

```
$$ BEGIN_TRANSACTION $$;
limit := 0;
n := $$ SELECT count(*) FROM R $$;
while( n > MaxRows ) {
    limit++;
    $$ DELETE FROM R WHERE R.attr < :limit $$;
    n := $$ SELECT count(*) FROM R $$;
}
$$ COMMIT_TRANSACTION $$;
```

What the DBMS will see is an SQL query followed by a possibly empty sequence of SQL update and query statements such as:

```
SELECT count(*) FROM R;
DELETE FROM R WHERE R.attr < 1;
SELECT count(*) FROM R;
DELETE FROM R WHERE R.attr < 2;
SELECT count(*) FROM R;
DELETE FROM R WHERE R.attr < 3;
SELECT count(*) FROM R;
```

For any particular execution of this program fragment, the sequence will correspond to a valid update program in our language (if we ignore the query statements[1]). □

[1] This is perfectly valid since they do not change the database state and the technique we will describe performs everything within the same transaction so read dependencies are preserved.

So that we can reap the benefits of performing our optimised integrity checking before actually executing the updates, we can do is defer the execution of the individual update statements until the commit request is received. At this point we can then generate the weakest precondition and apply the propagation technique to check the constraints. If this determines they will be satisfied, then we can actually execute the updates and commit the transaction.

Because there may be queries interspersed in the sequence of update statements, we need to make sure that they are evaluated with respect to the various intermediate states of the database. This is straightforward since we can simply generate and evaluate the weakest precondition of the query with respect to the update sequence seen so far.

With respect to the sequence given in example 6.4, the DBMS would execute the following sequence of queries and updates before checking any integrity constraints and committing the transaction.

```
SELECT count(*) FROM R;
SELECT count(*) FROM R WHERE R.attr >= 1;
SELECT count(*) FROM R WHERE R.attr >= 2;
SELECT count(*) FROM R WHERE R.attr >= 3;
DELETE FROM R WHERE R.attr < 1;
DELETE FROM R WHERE R.attr < 2;
DELETE FROM R WHERE R.attr < 3;
```

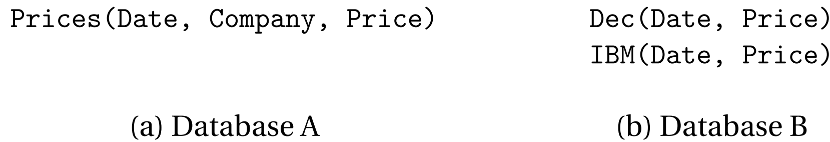
Figure 6.3 illustrates via a simple Prolog program how a DBMS can evaluate such a stream of query and update operations, deferring the actual execution of the update operations until the end of the transaction (after checking the integrity constraints), while still allowing queries to be evaluated and their answers returned during the course of the transaction.

```
process_stream([begin_trans |T], [[] |Output], _Updates) :-
        eval(begin_trans, _),
        process_stream(T, Output, []).

process_stream([query(Q) |T], [Results |Output], Updates) :-
        wp(Updates, Q, Q1),
        eval(Q1, Results),
        process_stream(T, Output, Updates).

process_stream([update(U) |T], [[] |Output], Updates) :-
        process_stream(T, Output, [U |Updates]).

process_stream([commit_trans |T], [[] |Output], Updates) :-
        constraints(IC),
        wp(Updates, IC, IC1),
        ( eval(IC1, _) ->
            eval(Updates, _),
            eval(commit_trans, _)
        ;
            eval(abort_trans, _)
        ),
        process_stream(T, Output, []).

process_stream([abort_trans |T], [[] |Output], Updates) :-
        eval(abort_trans, _),
        process_stream(T, Output, []).
```

Figure 6.1: Prolog code to remember the sequence of updates as a stream is processed instead of executing them. When a query is found in the stream it evaluates the wp of the query with respect to the sequence of updates remembered so far. At the end of the transaction, the integrity constraints are checked, and if satisfied, the update sequence executed.

6.4 Global Updates

A common approach to integrating (relational) databases is to use view definitions. These are mostly satisfactory for providing an integrated query-only database, but are far from satisfactory if one wishes to be able to perform updates. We present a pragmatic approach to providing support for updates to these views using *update methods*. An update method is associated with a view/virtual table and labelled either *INSERT* or *DELETE*.



```
Prices(Date, Company, Price)          Dec(Date, Price)
                                      IBM(Date, Price)
```

(a) Database A (b) Database B

Figure 6.2: Stock Market database schemata

```
CREATE VIEW stocks (Date, Company, Price) AS
    ( SELECT Date, 'DEC', Price FROM B.Dec )
UNION
    ( SELECT Date, 'IBM', Price FROM B.IBM )
UNION
    ( SELECT Date, Company, Price FROM A.Prices );
```

Figure 6.3: Integration of tables of databases “A” and “B”

Whenever an attempt is made to insert or delete tuples from one of these views, the corresponding method is invoked. This method can then arrange for the appropriate underlying tables to be updated.

Furthermore, we introduce a formal definition of *correctness* for these update methods and show how the correctness of a particular update method can be determined statically. Normally, correctness simply ensures that the result of executing the update method is to effectively insert (delete) the appropriate tuples into the view. In some cases this may be too strict a requirement in which case we allow user-defined correctness constraints using user-defined postconditions. These postconditions are essentially dynamic integrity constraints which are associated specifically with an individual update method.

Figure 6.2 gives the relational schemata for two databases storing stock market information. Figure 6.3 shows a view definition that one might use to integrate these two databases.

The update methods for this view can then be defined as in Figures 6.4 and 6.5. We revert to an SQL-like concrete syntax similar to that in Chapter 2. Given this, the translation to our abstract syntax is straightforward and therefore omitted.

```
DEFINE INSERT (theDate, theCompany, thePrice) INTO stocks AS
SWITCH( theCompany )
(
    CASE 'DEC':
        INSERT INTO B.Dec VALUES (theDate, thePrice)
    CASE 'IBM':
        INSERT INTO B.IBM VALUES (theDate, thePrice)
    DEFAULT:
        INSERT INTO A.Prices
        VALUES (theDate, theCompany, thePrice)
);
```

Figure 6.4: Insert method for the Stocks integration view.

```
DEFINE DELETE (theDate, theCompany, thePrice) FROM stocks AS
SWITCH( Company )
(
    CASE 'DEC':
        DELETE FROM B.Dec D
        WHERE D.Date = theDate
            AND D.Price = thePrice
    CASE 'IBM':
        DELETE FROM B.IBM I
        WHERE I.Date = theDate
            AND I.Price = thePrice
    DEFAULT:
        DELETE FROM A.Prices P
        WHERE P.Date = theDate
            AND P.Company = theCompany
            AND P.Price = thePrice
);
```

Figure 6.5: Delete method for the Stocks integration view.

In defining the correctness criteria for view update methods we take a general approach by allowing the specification of an arbitrary SQL 3 style condition as a postcondition for an update method. We say that an update method is *correct* if it will never violate the postcondition, regardless of the database contents and the values of the update method's input parameters.

For example, a possible postcondition for an insert (resp., delete) method may be that the tuple(s) inserted are (resp., are not) returned in a subse-

quent query on the view. Figures 6.6 and 6.7 give an example of how we can specify these postconditions.

```
DEFINE INSERT (theDate, theCompany, thePrice) INTO stocks AS
...
POSTCONDITION
FOR SOME stocks SB
  (
    SB.Date = theDate
    AND SB.Company = theCompany
    AND SB.Price = thePrice
  );
```

Figure 6.6: Postcondition for inserting tuples into the Stocks view.

```
DEFINE DELETE (theDate, theCompany, thePrice) FROM stocks AS
...
POSTCONDITION
FOR ALL stocks SB
  (
    NOT (SB.Date = theDate
        AND SB.Company = theCompany
        AND SB.Price = thePrice)
  );
```

Figure 6.7: Postcondition for deleting tuples from the Stocks view.

Again, for the purposes of simplicity and clarity of presentation, in the following, we will assume the postcondition has been translated to a first-order predicate logic formula and the updates into our abstract syntax.

So, in order to prove that a particular update method is *correct*, we need to show that, for any database B and constants $c_1, \ldots, c_n$, the updates

$$\forall \overline{x}\ ((x_1 = c_1 \wedge \cdots \wedge x_n = c_n) \rightarrow +R(\overline{x}))$$

and

$$\forall \overline{x}\ ((x_1 = c_1 \wedge \cdots \wedge x_n = c_n) \rightarrow -R(\overline{x}))$$

cannot violate their corresponding correctness conditions.

Let U be an update method and P its corresponding correctness condition. Furthermore, let IC be the formula corresponding to any integrity

constraints that have been defined on the database. We say that the update method U is *correct* if and only if $IC \rightarrow wp(U, P) \equiv true$.

To illustrate the use of wp, consider the INSERT method from Figure 6.4 and its corresponding correctness condition given above. Let a_1, a_2, a_3 be the formal parameters of the update method U, then, in abstract syntax form, the update method is:

$$
\begin{aligned}
&\forall Date, Price(a_2 = \text{'DEC'} \wedge Date = a_1 \wedge Price = a_3) \rightarrow \\
&\qquad +B.Dec(a_1, a_3); \\
&\forall Date, Price(a_2 = \text{'IBM'} \wedge Date = a_1 \wedge Price = a_3) \rightarrow \\
&\qquad +B.IBM(a_1, a_3); \\
&\forall Date, Company, Price(Date = a_1 \wedge Company = a_2 \wedge Price = a_3 \wedge \\
&\qquad a_2 \neq \text{'DEC'} \wedge a_2 \neq \text{'IBM'}) \rightarrow +A.Prices(a_1, a_2, a_3))
\end{aligned}
$$

and the correctness condition P is $stocks(a_1, a_2, a_3)$. Which, after expanding the view definition of `stocks` is:

$$
\begin{aligned}
&(a_2 = \text{'DEC'} \wedge B.Dec(a_1, a_3)) \vee \\
&(a_2 = \text{'IBM'} \wedge B.IBM(a_1, a_3)) \vee \\
&(a_2 \neq \text{'DEC'} \wedge a_2 \neq \text{'IBM'} \wedge A.Prices(a_1, a_2, a_3))
\end{aligned}
$$

Applying the definition above, $wp(U, P)$ is:

$$
\begin{aligned}
&(a_2 = \text{'DEC'} \wedge \\
&\qquad (B.Dec(a_1, a_3) \vee (a_2 = \text{'DEC'} \wedge a_1 = a_1 \wedge a_3 = a_3))) \vee \\
&(a_2 = \text{'IBM'} \wedge \\
&\qquad (B.IBM(a_1, a_3) \vee (a_2 = \text{'IBM'} \wedge a_1 = a_1 \wedge a_3 = a_3))) \vee \\
&(a_2 \neq \text{'DEC'} \wedge a_2 \neq \text{'IBM'} \wedge \\
&\qquad (A.Prices(a_1, a_2, a_3) \vee a_1 = a_1 \wedge a_2 = a_2 \wedge a_3 = a_3))
\end{aligned}
$$

which can be simplified to $true$. Hence, we can say that the update method U is *correct*.

In this section we have presented a pragmatic approach to providing support for updates to database views using update methods. We introduced the notion of correctness for these methods based on postconditions and showed how to prove that an update method does not violate its correctness condition.

In doing so we have ignored the SQL related issues of NULLs, and duplicate rows in tables. We feel this is justified on several counts. SQL's

handling of NULLs is complex and irregular. A full and satisfactory treatment of NULLs is therefore beyond the scope of this thesis. Allowing duplicate rows in tables violates Codd's original formulation of the relational model and it has been argued [Dat95] that databases can and should be designed to avoid duplicate rows.

Several interesting avenues for further research include automatic or semi-automatic generation of update methods, based on the criteria for view updates outlined by Date [Dat95] or based on the correctness conditions themselves, and support for simple aggregate functions such as COUNT and SUM.

We have implemented update methods as a component of the DISTOPIA multidatabase prototype [BKL+95].

Chapter 7

Conclusion

7.1 Summary

In this thesis we approach the integrity constraint checking problem for deductive object-oriented databases by attempting to determine transaction safety and producing safe transactions, those that cannot violate the integrity constraints, from unsafe ones.

In Chapter 2 we introduced the use of a predicate transformer for generating weakest preconditions for relational databases and a set-oriented update language. Chapter 3 built on this foundation to deal with a more powerful update language and deductive databases. That is, we allowed recursion in the specification of both queries and the conditions of the update language. Then, in Chapter 4, we extended the predicate transformer to deal with object creation, inheritance and overriding in the deductive object-oriented data model Gulog [Dob95, DT93, DT94, DT95]. At this point we have a powerful tool at our disposal which can be used to generate the weakest precondition for a transaction and the database's integrity constraints. By making the transaction conditional on the generated weakest precondition, we now have a safe update.

In Chapter 5 we investigated two approaches to simplifying the generated weakest precondition under the assumption that the database satisfies its integrity constraints. We gave a formal statement of the problem including a proof that it is co-recursively enumerable.

The first approach to simplification identifies several general classes of integrity constraint, that include referential integrity and functional de-

pendencies, and for which substantial simplifications can be made, often determining that nothing need be checked at all. That is, that the transaction is safe. If something does need to be checked, then that check can be used as the (simpler) condition for a safe transaction. At present, we are unable to completely characterise the conditions under which a constraint is simplifiable with respect to an update.

The second approach is more general. It involves capturing exactly the incremental change that a transaction will make to the database in terms of a query made with respect to just the current database state. By expressing the integrity constraints as rules within the database, we then simply need to determine whether the fact $inconsistent$ will be added to the database state. If not, then the transaction is safe. Again, this test for the incremental addition of $inconsistent$ can be used as a simple condition for a safe transaction.

In Chapter 6 we investigated several extensions and further applications of the predicate transformer. We illustrated briefly the use of our technique for enforcing schema constraints. We showed how to extend the class of integrity constraints our technique is applicable to to include dynamic (or transition) constraints; those that enable constraints with respect to the old and new database states to be specified. We also showed how we can support transactions that are specified via an embedding in some other programming language, such as C. Finally, we developed a technique for checking that methods defined to update views in the context of federated databases do not violate consistency criteria associated with these views.

7.2 Future Work

In Chapter 6 we described a number of further applications of the predicate transformer. Clearly we have just scratched the surface of schema integrity constraints and there is much more that can be investigated. For example, the evaluation procedures of Gulog place a well-formedness restriction on Gulog programs called *i-stratification*. Essentially this is a stratification condition on the inheritance hierarchy and its interaction with overriding in methods. Method overriding in Gulog can be used to implement negation, thus the *i-stratification* condition has a strong simi-

larity to various stratification conditions in deductive databases and logic programming. In particular, *i-stratification* is closely modelled on *local stratification* [Prz88] and any inheritance stratified Gulog program can be translated into an equivalent locally stratified Datalog program. Given a suitable meta-schema for representing Gulog programs, it should be possible to state the *i-stratification* condition as a (quite complex) schema constraint. However, it should still be simple to apply our predicate transformer and update propagation technique to generate guards for the incremental addition (and deletion) of rules to the database. How efficient these checks would be remains to be seen.

Another interesting application of this technique would be to the specification and use of database triggers. The traditional approach to specification of database triggers is the Event-Condition-Action (ECA) rule. With this approach, every trigger consists of a primitive update event such as `INSERT EMPLOYEE(`$x_1, \ldots, x_n$`)` or `DELETE DEPARTMENT(`$x_1, \ldots, x_n$`)`, a condition to evaluate, and an action to perform if the condition is satisfied.

One common use of ECA rules is to support integrity checking and maintenance. Thus the referential integrity constraint requiring all employees to belong to an existing department $\forall E, D\ (belongs(E, D) \rightarrow dept(D))$ could be translated into the set of ECA rules:

```
ON INSERT belongs(E, D)
CHECK NOT dept(D)
DO ABORT;

ON DELETE dept(D)
CHECK EXISTS E belongs(E, D)
DO DELETE belongs(E, D);
```

The first of these simply aborts the transaction if the constraint is violated, the second performs any necessary extra updates to ensure the constraint is maintained.

Generating these ECA rules by hand from the integrity constraints is a time-consuming and potentially error-prone task. Ceri and Widom [CW90] show how to automate this process.

The main drawback with the trigger approach to integrity constraint

checking and enforcement is that the events are too fine-grained since transactions usually consist of a set of related updates. For example, imagine the constraint `ON INSERT p(X) CHECK q(X) DO something`, and the update $\forall$ X (r(X) $\rightarrow$ (+p(X) ; $-$q(X))). Clearly, when we perform this update the trigger will be activated since the event `INSERT p(X)` occurs, but the condition will always be false.

It would be interesting to investigate the use of predicate transformers to deal with entire transactions rather than individual updates in an attempt to avoid triggers from firing unnecessarily. One possible way to do this might involve eliminating the event part of the ECA and allowing dynamic integrity constraints for the condition part.

There is still no widely accepted deductive object-oriented data model in the way the relational and deductive data models are. In this thesis we have chosen to use Gulog for our deductive object-oriented data model, but this choice is somewhat arbitrary. This choice has meant we have had to deal with some interesting semantics such as dynamic overriding. It would be instructive to also apply this technique in the context of other deductive object-oriented data models such as O_2 [Deu90] and their particular semantic quirks.

Some of these other data models also have associated update languages which allow either unbounded iteration (while loops) or, the semantically equivalent, recursive invocation of user-defined update procedures. These languages are more powerful [Law92] than the update languages we have presented. While we have shown how it is possible to apply our technique to languages with these features, the use of deferred updates is probably mostly useful for ad-hoc transactions. Adopting our technique to handle such powerful update languages is an open problem. A possible approach may be to generate preconditions rather than weakest preconditions by approximating the effect of while loops. This would naturally mean that some safe transactions would still have guards generated for them.

Finally, we propose implementing a complete prototype of this technique, along with implementations of several of the other major integrity constraint checking techniques, for each of the relational, deductive, and deductive object-oriented data models as appropriate. These implementations would then allow empirical results to be gathered as to the rela-

tive efficiency of each of the techniques in the context of various mixes of transaction, numbers of users, likelyhood of constraint violation, and kind of constraint. This would be a non-trivial undertaking. There are many variables to account for, and several of the techniques require quite powerful theorem provers or partial evaluation software. Also, the query optimiser available in an underlying DBMS may make a large difference to the results.

Appendix A

Notations and Symbols

$[\![S]\!]$
: the update defined by the statement S 15

P'_r
: a program consisting of the union of the set of rules in P with the following additional rules. For every rule Q in P defining a predicate symbol q that depends on r, add an identical rule defining q' (a new predicate which does not appear in P) in which every predicate symbol s in the body of Q that depends on r (as well as every occurrence of r itself) is replaced by s' 27

relational model

$\mathbf{B}[R \mapsto R']$
: the result of replacing the relation R in B by R' 8

$wp(S, H)$
: the predicate transformer that generates the weakest precondition for $[\![S]\!]$ and H 16

abstract update language

$\forall\overline{x}\ (\Phi(\overline{x}) \rightarrow -r(\overline{x}))$
: delete each of the tuples $\overline{x}$ satisfying $\Phi(\overline{x})$ from the relation R . 12

$\forall\overline{x}\ (\Phi(\overline{x}) \rightarrow +r(\overline{x}))$
: insert each of the tuples $\overline{x}$ satisfying $\Phi(\overline{x})$ into the relation R . . 12

concrete update language

$\texttt{delete}_R(\overline{x})$
: delete the tuple $\overline{x}$ from the relation R 10

`foreach` $\overline{x}$:$\Phi(\overline{x})$ `do` (***stmt***$_1$; $\cdots$; ***stmt***$_k$)
evaluate $\Phi(\overline{x})$ to produce a set of valuations for $\overline{x}$ then, for each $stmt_i$, $1 \leq i \leq k$, execute *stmt*$_i$ for each valuation 10

`insert`$_R(\overline{x})$
insert the tuple $\overline{x}$ into the relation R 10

`replace`$_R(\overline{x}, \overline{y})(\overline{x}, \overline{z})$
replace the tuple $\overline{x}, \overline{y}$ in the relation R with the tuple $\overline{x}, \overline{z}$ 10

deductive model

$\forall \overline{x}\ ((q(\overline{x}), P) \rightarrow -r(\overline{x}))$
delete the tuples satisfying (q($\overline{x}$), P) from the relation R 28

$\forall \overline{x}\ ((q(\overline{x}), P) \rightarrow +r(\overline{x}))$
insert the tuples satisfying (q($\overline{x}$), P) into the relation R 28

$(q(\overline{x}), P)$
a query where $q(\overline{x})$ is an atom and P is a program defining q . . 26

$wp(S, (q, P))$
the condition transformer that generates the weakest precondition for $[\![S]\!]$ and (q, P) 30

deductive object-oriented model

$\Gamma \vdash y[m@\overline{x} \rightarrow z]$
a single-valued method called m applied to the object y with arguments $\overline{x}$ and returning z 40

$\Gamma \vdash y[m@\overline{x} \twoheadrightarrow z]$
a multi-valued method called m applied to the object y with arguments $\overline{x}$ and returning z 40

r_i/n
the predicate corresponding to the atom $r_i(x_1, \ldots, x_n)$ 44

$m/k@c$
a (functional or relational) method m of arity k that is applicable to objects in class c (or any subclass of c) 44

$(\Gamma \vdash A, P)$
a Gulog query where Γ is a variable typing, A is an atomic formula, and P is a program defining A 44

$c_1 < c_2$
c_1 is a subclass of c_2 40

Γ
a variable typing equivalent to $\{x_1{:}\tau_1,\ \ldots,\ x_n{:}\tau_n\}$ 44

$\forall\overline{x}\ ((\Gamma \vdash\ q(\overline{x}), P) \rightarrow +r(\overline{x}))$
insert the tuples satisfying ($\Gamma \vdash$ q($\overline{x}$), P) into the relation R 48

$\forall\overline{x}\ ((\Gamma \vdash\ q(\overline{x}), P) \rightarrow -r(\overline{x}))$
delete the tuples satisfying ($\Gamma \vdash$ q($\overline{x}$), P) from the relation R . . . 48

$\forall y, \overline{x}, z\ ((\Gamma \vdash\ q(y, \overline{x}, z), P) \rightarrow +y[m@\overline{x} \twoheadrightarrow z])$
insert each of the tuples satisfying ($\Gamma \vdash$ q($\overline{x}$), P) into the set-valued method m 48

$\forall y, \overline{x}, z\ ((\Gamma \vdash q(y, \overline{x}, z), P) \rightarrow -y[m@\overline{x} \twoheadrightarrow z])$
delete each of the tuples satisfying ($\Gamma \vdash$ q($\overline{x}$), P) from the set-valued method m 48

$\forall y, \overline{x}, z\ ((\Gamma \vdash\ q(y, \overline{x}, z), P) \rightarrow !y[m@\overline{x} \rightarrow z])$
sets the value of a single-valued method m 48

$\forall y, \overline{x}, z\ ((\Gamma \vdash\ q(y, \overline{x}, z), P) \rightarrow -y[m@\overline{x} \rightarrow z])$
unsets the value of a single-valued method m 48

$\forall\overline{x}\mathcal{I}y\ ((\Gamma \vdash\ q(\overline{x}), P) \rightarrow +y{:}c)$
for each of the tuples satisfying ($\Gamma \vdash$q($\overline{x}$), P), create a new instance of c 48

$\forall x\ ((\Gamma \vdash\ q(x), P) \rightarrow -x{:}c)$
delete each instance x of c that satisfies ($\Gamma \vdash$q($\overline{x}$), P) 48

Appendix B

Prolog Code

The Prolog code in this appendix is a minimal implementation of the predicate transformer wp for generating weakest preconditions for deductive databases, as well as performing update propagation.

In conjunction with an appropriate partial evaluator [Sah, LM95], this can be used to help automate the task of determining transaction safety.

This code has been run on many of the examples in this thesis. There is no perceptible delay in producing the transformed code corresponding to the update propagation rules for the weakest precondition. However, the partial evaluator `Mixtus` [Sah] takes a noticeably long time to process these rules and fails to completely simplify the rules. This seems mainly due to Mixtus' goal of handling full Prolog, including its various non-logical behaviours, which leads to it being more conservative than is otherwise necessary for handling the purely declarative language we use.

types.pl

```
%    michael lawley
%    24/11/96
%
%    These are some Mercury style type definitions for the
%    simplified deductive IC checking generator.
%
%    They are not actually used, but serve as documentation.

:- type lit
    --->        pos(atom)
    |   neg(atom).
```

```
:- type rule
    --->        rule(lit, list(lit))
    |   wrule(wp, list(wp)).

:- type wp
    --->        wp(update, lit).

:- type update
    --->        list(stmt).

:- type stmt
    --->        ins(atom, atom)
    |   del(atom, atom).
```

gen_wp.pl

```
%    michael lawley
%    24/11/96
%
%    Simple wp generation.

%
%    Takes an update (list of at least one stmt), and a list of
%    atoms representing the base relations and produces a list
%    of rules corresponding to the wp's for each base relation.
%
% :- pred gen_wp_relns(update::in, list(atom)::in, list(rule)::out).
%
gen_wp_relns(_, [], []).
gen_wp_relns(U, [H |T], WPs) :-
    gen_wp_reln(U, H, WPs0),
    gen_wp_relns(U, T, WPs1),
    append(WPs0, WPs1, WPs).


%
%    Takes an update and a single atom and produces a list of
%    rules corresponding to the wp for the base relation.
%
%    It doesn this by generating the rules for wp(S, A) where
%    S is the last statement in the update U, then calling
%    gen_wp_rules to generate the rules for wp(U', wp(S, A))
%    where U is (U' ; S).
%
% :- pred gen_wp_reln(update::in, atom::in, list(rule)::out).
%
gen_wp_reln([], _, []).
gen_wp_reln([S1,S2|Ss], A, Rules) :-
```

```
    append(Start, [End], [S|Ss]),
    copy_term([A,End], [A1,End1]),
    gen_rules(End1, A1, Rules0),
    ( Start = [] ->
        Rules = Rules0
    ;
        gen_wp_rules(Start, Rules0, Rules1),
        gen_wp_reln(Start, A, Rules2),
        append(Rules0, Rules1, RulesTmp)
        append(RulesTmp, Rules2, Rules)
    ).


%
%    Generates the rules for wp(S, A) where S is a single statement
%    and A is a positive occurrence of an atom.
%
% :- pred gen_rules(stmt::in, atom::in, list(rule)::out).
%
gen_rules(ins(Targ, Cond), A, Rules) :-
    U = [ins(Targ, Cond)],
    ( A = Targ ->
        Rules = [rule(wp(U,A), [pos(A)]),
                 rule(wp(U,A), [pos(Cond)])]
    ;
        Rules = [rule(wp(U,A), [pos(A)])]
    ).

gen_rules(del(Targ, Cond), A, Rules) :-
    U = [del(Targ, Cond)],
    ( A = Targ ->
        Rules = [rule(wp(U,A), [pos(A), neg(Cond)])]
    ;
        Rules = [rule(wp(U,A), [pos(A)])]
    ).


%
%       Takes an update U, and a list of rules and generates the
%       wp's for each of the rules.
%
% :- pred gen_wp_rules(update::in, list(rule)::in, list(rule)::out).
%
gen_wp_rules(_, [], []).
gen_wp_rules(U, [H |T], WPs) :-
    gen_wp_rule(U, H, WPs0),
    gen_wp_rules(U, T, WPs1),
    append(WPs0, WPs1, WPs).
```

```
%
%       Takes an update and a single rule and generates the rule
%       corresponding to the wp.
%
% :- pred gen_wp_rule(update::in, rule::in, list(rule)::out).
%
gen_wp_rule(U, rule(Head, Body), WPs) :-
    ( Head = wp(U1, C) ->
        append(U, U1, U2),
        WPs = [ wrule(wp(U2, C), WPBody) ]
    ;
        WPs = [ wrule(wp(U, Head), WPBody) ]
    ),
    wrap_lits(U, Body, WPBody).

gen_wp_rule(U, wrule(wp(U1, Head), Body), WPs) :-
    append(U, U1, U2),
    WPs = [ wrule(wp(U2, Head), WPBody) ],
    wrap_wps(U, Body, WPBody).


% :- pred wrap_lits(update::in, list(lit)::in, list(wp)::out).
%
wrap_lits(_, [], []).
wrap_lits(U, [pos(H) |T], [pos(wp(U, H)) |WPT]) :-
    wrap_lits(U, T, WPT).
wrap_lits(U, [neg(H) |T], [neg(wp(U, H)) |WPT]) :-
    wrap_lits(U, T, WPT).


% :- pred wrap_wps(update::in, list(wp)::in, list(wp)::out).
%
wrap_wps(_, [], []).
wrap_wps(U, [pos(wp(U1, H)) |T], [pos(wp(U2, H)) |WPT]) :-
    append(U, U1, U2),
    wrap_wps(U, T, WPT).
wrap_wps(U, [neg(wp(U1, H)) |T], [neg(wp(U2, H)) |WPT]) :-
    append(U, U1, U2),
    wrap_wps(U, T, WPT).
```

gen_petra.pl

```
%    michael lawley
%    24/11/96
%
%    Quick Implementation of Petra's "top-down analysis" method
%    for IC checking.
```

```
% :- pred gen_reln_deltas(update::in, list(atom)::in,
%                         list(rule)::out).
%
gen_reln_deltas(_, [], []).
gen_reln_deltas(U, [H |T],
                [rule(add(H), [pos(wp(U, H)), neg(H)]),
                 rule(del(H), [neg(wp(U, H)), pos(H)]) |Rules]) :-
    gen_reln_deltas(U, T, Rules).


% :- pred gen_rule_deltas(update::in, list(rule)::in,
%                         list(rule)::out).
%
gen_rule_deltas(_, [], []).
gen_rule_deltas(U, [H |T], Rules) :-
    gen_rule_add(U, H, Rules0),
    gen_rule_del(U, H, Rules1),
    gen_rule_deltas(U, T, Rules2),
    append(Rules1, Rules2, Rules3),
    append(Rules0, Rules3, Rules).


% :- pred gen_rule_add(update::in, rule::in, list(rule)::out).
%
gen_rule_add(U, rule(L, Body), Rules) :-
    findall(rule(add(L), [pos(add(Lit)) |WRest]),
            (
                delete(pos(Lit), Body, Rest),
                wrap_lits(U, Rest, WRest)
            ),
            Rules0),
    findall(rule(add(L), [pos(del(Lit)) |WRest]),
            (
                delete(neg(Lit), Body, Rest),
                wrap_lits(U, Rest, WRest)
            ),
            Rules1),
    append(Rules0, Rules1, Rules).


% :- pred gen_rule_del(update::in, rule::in, list(rule)::out).
%
gen_rule_del(U, rule(L, Body), Rules) :-
    findall(rule(del(L), [pos(del(Lit)), neg(wp(U, L)) |Rest]),
            delete(pos(Lit), Body, Rest),
            Rules0),
    findall(rule(del(L), [pos(add(Lit)), neg(wp(U, L)) |Rest]),
            delete(neg(Lit), Body, Rest),
```

```
        Rules1),
    append(Rules0, Rules1, Rules).
```

Bibliography

[ALUW93] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, pages 32–41, Washington, DC, 1993.

[AV87a] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification (extended abstract). In *Proc. Sixth ACM Symposium on Principles of Database Systems*, pages 260–268, San Diego, California, March 1987.

[AV87b] S. Abiteboul and V. Vianu. Transaction languages for database update and specification. Technical Report 715, INRIA, September 1987.

[AV88a] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. Technical Report 900, INRIA, September 1988.

[AV88b] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proc. Seventh ACM Symposium on Principles of Database Systems*, pages 240–250, Austin, Texas, March 1988.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(1):181–229, 1990.

[Bay92] P. Bayer. Update propagation for integrity checking, materialized view maintenance and production rule triggering. Technical Report ECRC-92-10, ECRC, February 1992.

[BCB97] E. Bertino, B. Catania, and S. Bressan. Integrity constraint checking in chimera. In *Proc. of the 2nd International Workshop on Constraint Database Systems (CDB'97)*, Lecture Notes in Computer Science, pages 160–186, Delphi, Greece, January 1997. Springer-Verlag.

[BD92] V. Benzaken and A. Doucet. Enforcement tests generation for integrity constraint checking based on simplification methods in object-oriented database systems. In *Proc. French Database Conference*, 1992.

[BD93] V. Benzaken and A. Doucet. Themis: a database programming language with integrity contraints. In *Proc. Fourth International Workshop on Database Programming Languages*, New York, N.Y., 1993. Springer-Verlag.

[BD95] V. Benzaken and A. Doucet. Themis: a database programming language handling integrity constraints. *VLDB Journal*, 4(3), July-August 1995.

[BDM88] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In *Proc. First International Conference on Extending Database Technology*, pages 488–505, Venice, Italy, February 1988.

[BGL96] M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. In *Proc. Fifteenth ACM Symposium on Principles of Database Systems*, pages 117–127, Montreal, Quebec Canada, 1996.

[BKL+95] L. Bell, D. Kuo, M.J. Lawley, M. Orlowska, and R. Topor. An architectural overview of DISTOPIA. In *Proc. of the First DSTC Symposium*, Brisbane, Australia, July 1995. http://www.dstc.edu.au/DDU/publications/papers/doia.html.

[BS97] V. Benzaken and X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, Lecture Notes in Computer Science, Jyvaškyla;, Finland, June 1997. Springer-Verlag.

[Cat94] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kauffmann, 1994.

[CCD93] M. Celma, J.C. Casamayor, and H. Decker. Improving integrity checking by compiling derivation paths. In M.E. Orlowska and M. Papazoglou, editors, *Proc. 4th Australian Database Conference*, pages 150–160, Brisbane, Australia, February 1993.

[CD94] M. Celma and H. Decker. Integrity checking in deductive databases - the ultimate method? In R. Sacks-Davis, editor,

Proc. 5th Australasian Database Conference, pages 136–146, Christchurch, New Zealand, January 1994.

[CGM90] U.S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.

[CH82] A.K. Chandra and D. Harel. Structure and complexity of relational queries. *Journal of Computer and System Science*, 25:99–128, 1982.

[CW90] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proc. Sixteenth International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Australia, August 1990.

[Dat95] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley Systems Programming Series. Addison-Wesley, sixth edition, 1995.

[Deß90] S. Deßloch. Enforcing integrity in the KBMS KRISYS. In *Proc. Second Workshop on Foundations of Models and Languages for Dataand Objects*, pages 123–138, Aigen Austria, September 1990. email: dessloch@informatik.uni-kl.de.

[Deu90] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Dob95] G. Dobbie. *Foundations of Deductive Object-Oriented Database Systems.* PhD thesis, University of Melbourne, February 1995.

[DT93] G. Dobbie and R.W. Topor. A model for inheritance and overriding in deductive object-oriented systems. In Gopal Gupta, George Mohay, and Rodney W. Topor, editors, *Proc. of the 16th Australian Computer Science Conference*, volume 15, pages 625–634, Brisbane, Queensland, February 1993.

[DT94] G. Dobbie and R.W. Topor. Representing inheritance and overriding in Datalog. *Computers and Artificial Intelligence*, 13(2–3):133–158, 1994.

[DT95] G. Dobbie and R.W. Topor. On the declarative and procedural semantics of deductive object-oriented systems. *Journal of Intelligent Information Systems*, 4(2):193–219, March 1995.

[GGM96] O. Godfrey, J. Gryz, and J. Minker. Semantic query optimization for bottom-up evaluation. In *Foundations of Intelligent Systems, 9th International Symposium*, volume 1079 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1996.

[GL90] U. Griefahn and S. Lüttringhaus. Top-down integrity constraint checking for deductive databases. In *Proc. Seventh International Conference on Logic Programming*, pages 130–144, 1990.

[GMN84] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 12(2):153–185, 1984.

[Gri81] D. Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1981.

[HI85] A. Hsu and T. Imielinski. Integrity checking for multiple updates. In *Proc. 1985 ACM SIGMOD International Conference on Management of Data*, pages 152–167, 1985.

[HMN84] L.J. Henschen, W.W. McCune, and S.A. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances in Data Base Theory*, pages 145–169. Plenum Press, New York, 1984.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[ISO] ISO/IEC. ISO/IEC DIS 9075-2 – Part 2: Data Base Language SQL – Part 2: SQL foundation (for SQL3).

[JJ91] M. Jeusfeld and M. Jarke. From relational to object-oriented integrity simplification. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. Second International Conference on Deductive and Object-Oriented Databases*, Lecture Notes in Computer Science, pages 460–477. Springer-Verlag, December 1991.

[JK90] M. Jeusfeld and E. Krüger. Deductive integrity maintenance in an object-oriented setting. Technical Report MIP-9013, Iniversität Passau, 1990.

[JQ92] H.V. Jagadish and X. Qian. Integrity maintenance in an object-oriented database. In *Proc. Eighteenth International Conference on Very Large Data Bases*, pages 469–480, 1992.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical report 90/14 (revised), Department of Computer Science, State University of New York at Stony Brook, August 1990.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, July 1995.

[Law92] M.J. Lawley. On the power of database update languages. In G.K. Gupta and C.D. Keen, editors, *Proc. of the 15th Australian Computer Science Conference*, pages 517–528, Hobart, Australia, January 1992.

[Law95] M.J. Lawley. Transaction safety in deductive object-oriented databases. In *Proc. Fourth International Conference on Deductive and Object-Oriented Databases*, pages 395–410, Singapore, 1995.

[LM95] M. Leuschel and B. Martens. Partial deduction of the ground representation and its application to integrity checking. In J.W. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, pages 495–509, Portland, Oregon, December 1995.

[LS95] A.Y. Levy and Y. Sagiv. Semantic query optimisation in datalog programs (extended abstract). In *Proc. Fourteenth ACM Symposium on Principles of Database Systems*, pages 163–173, San Jose, California, 1995.

[LST87] J.W. Lloyd, E.A. Sonenberg, and R.W. Topor. Integrity constraint checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, December 1987.

[LT95] M.J. Lawley and R.W. Topor. Transaction safety in deductive databases using weakest preconditions. Technical Report CIT-95-13, School of Computing and Information Technology, Griffith University, 1995.

[LTW93] M.J. Lawley, R.W. Topor, and M. Wallace. Using weakest preconditions to simplify integrity constraint checking. In M.E. Orlowska and M. Papazoglou, editors, *Proc. 4th Australian Database Conference*, pages 161–170, Brisbane, Australia, February 1993.

[Man90] R. Manthey. Integrity and recursion: two key issues for deductive databases. In *Information Systems and AI: Integration*

Aspects, number 474 in Lecture Notes in Computer Science, pages 104–126. Springer-Verlag, 1990.

[Mar90] V.M. Markowitz. Referential integrity revisited: An object-oriented perspective. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proc. Sixteenth International Conference on Very Large Data Bases*, pages 578–589, Brisbane, Australia, August 1990.

[MH89] W.W. McCune and L.J. Henschen. Maintaining state constraints in relational databases: A proof theoretic basis. *Journal of the ACM*, 36(1):46–68, 1989.

[Min88] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kauffmann, 1988.

[NDCC92] G. Nüssel, H. Decker, M. Celma, and J.C. Casamayor. A complete proof procedure for efficient integrity checking in deductive databases. In *Proc. 3rd International Workshop on the Deductive Approach to Information Systems and Databases*, Catalonia, Spain, September 1992.

[Nic82] J.-M. Nicolas. Logic for improving integrity checking in relational database. *Acta Informatica*, 18:227–253, 1982.

[Oli91] A. Olivé. Integrity constraints checking in deductive databases. In Guy M. Lohman, Amilcar Sernadas, and Rafael Camps, editors, *Proc. Seventeenth International Conference on Very Large Data Bases*, pages 513–523, Barcelona, Spain, September 1991.

[Prz88] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Minker [Min88], pages 193–216.

[Qia90] X. Qian. An axiom system for database transactions. *Information Processing Letters*, 36(4):183–189, 1990.

[Sah] D. Sahlin. Mixtus, an automatic partial evaluator for full prolog. http://www.sics.se/isl/mixtus.html.

[SK88] F. Sadri and R.A. Kowalski. A theorem-proving approach to database integrity. In Minker [Min88], pages 313–162.

[SMS87] D. Stemple, S. Mazumdar, and T. Sheard. On the modes and meaning of feedback to transaction designers. In U. Dayal and I. Traiger, editors, *Proc. 1987 ACM SIGMOD International Conference on Management of Data*, pages 374–386, San Francisco, California, 1987. ACM Press.

[SS88] T. Sheard and D. Stemple. Automatic verification of database transaction safety. TR 88-29, University of Masschusetts at Amherst, April 1988.

[SS89] T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, September 1989.

[Wal90] M. Wallace. Compiling declarative constraints into methods. Unpublished paper, February 1990.

[Wal91] M. Wallace. Compiling integrity checking into update procedures. In *Proc. Twelfth International Joint Conference on Artificial Intelligence*, pages 903–908, August 1991.

[Wal92] M. Wallace. Compiling integrity checking into update procedures. Technical Report ECRC-92-19, ECRC, 1992.