VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Department of Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5328
Fax: +64 4 495 5232
Internet: Tech.Reports@comp.vuw.ac.nz

Foundations of deductive object-oriented database systems

Gillian Dobbie

Technical Report CS-TR-96/13

Abstract

Abstract

The mathematical foundation of the relational data model has contributed to the widespread use of relational database systems. This mathematical foundation gives a clear meaning to the data, provides a basis for database design, allows the use of simple declarative query languages, and enables queries to be optimized automatically. However, the relational data model has limitations. These include the difficulty of representing complex, structured and procedural information that is required in many new database applications, and the restriction to query languages that are not computationally complete.

Two approaches that address some of these problems are deductive databases and object-oriented databases. Deductive databases extend the class of queries that can be expressed, and preserve the mathematical foundation of relational systems and object-oriented databases provide a rich data modeling capability.

In this thesis, after reviewing the areas of deductive databases and object-oriented databases, and related work in which different approaches to combining deductive and object-oriented databases are described, we propose a mathematical foundation for object-oriented systems with deduction. We do this by defining a new language, Gulog, which has the features of object-oriented databases and of deductive databases. From deductive databases it has deduction, predicates, and negation. From object-oriented databases it has inheritance (and multiple inheritance), overriding, functional methods, multi-valued methods, objects, and classes. As in the deductive database literature, a database is a Gulog program and a query is a Gulog goal.

Then, we describe a declarative semantics for Gulog databases and queries. In this setting, a model contains only the ground atoms derived from the rules that are not overridden. We define a class of meaningful programs called simple programs and prove that each simple Gulog program has a unique intuitive model.

Such a semantic description is of little use without procedures that compute answers to queries with respect to a database. There are two ways to approach query evaluation: the first is to provide direct procedures; the second is to provide a translation to a language that has direct query evaluation procedures.

First, we propose direct procedures. We describe a bottom-up query evaluation procedure. Because bottom-up procedures are not goal-directed, top-down procedures are more efficient in some circumstances. So we also describe top-down procedures with and without tabling. We prove that all these procedures are sound, and the bottom-up and the top-down procedure with tabling are complete with respect to the declarative semantics. To find the computed answer for a query using the top-down procedures, all answers for a selected atom are derived, and because of the possibility of overriding, the "minimal" ones are chosen as the computed answer for that atom. These answers are then applied to the rest of the query.

Second, we provide a translation from Gulog to a language that has direct query evaluation procedures, Datalog with negation. Negation replaces overriding in the translation. Such a translation provides another query evaluation procedure, and also allows us to compare results with others from the area of deductive databases.

We next show it is possible to give a mathematical description of a wider range of object-oriented and deductive features. We demonstrate this by including these features into Gulog and extending the analysis we have already performed.

Finally, we identify areas in which Gulog could be extended in the future and discuss its applicability for providing a better understanding of how to effectively apply and implement deductive object-oriented databases.

Publishing Information

A thesis submitted to the University of Melbourne in fulfillment of the requirements for the degree of Doctor of Philosophy.

Author Information

Gillian Dobbie
Department of Computer Science
Victoria University of Wellington
PO Box 600
New Zealand
gill@comp.vuw.ac.nz

Preface

This thesis contains seven chapters, of which the first is an introduction and the seventh is a conclusion. The second chapter introduces concepts and surveys the work of others in the relevant areas; the sources of the material presented in this chapter are cited in the text. The third and fourth chapters are based on joint research with Rodney Topor that first appeared in [44] and later in [47]. The fifth chapter is again based on joint research with Rodney Topor that appeared in [43]. The sixth chapter extends the work presented in the other chapters.

This thesis is less than 100,000 words in length.

Acknowledgements

There are many people who have provided encouragement, support and friendship, without them I would not have completed this thesis. My thanks goes to them all. I would especially like to thank the following people.

I thank both my supervisors, Rodney Topor and Kotagiri Ramamohanarao (Rao), who provided guidance throughout. Rodney was a tolerant supervisor, a meticulous co-author, a fellow bush-walker and a great inspiration. Without him I would not have achieved what I have. Rao provided encouragement, helpful advice, a stimulating work environment, and many anecdotes.

I am also extremely grateful to the members of Rodney's database research group: Alex Lefebvre who read much of my work in its early stages and provided useful feedback and encouragement, Chris Higgins who has read my work in the later stages and again provided valuable feedback, and Mike Lawley who has patiently listened to many of my ideas and spent time teaching me about complexity theory and other areas of Computer Science which were lacking. I also thank the other people in the School of Computing and Information Technology (CIT) at Griffith University, especially Paul Pritchard who was Head of School while I was there.

Rao heads the deductive database research group at the University of Melbourne. Members of this group were supportive and helpful. My special thanks go to James Harland who made logic seem obvious, Peter Stuckey who tolerated my questioning and David Kemp who not only tolerated my questioning, but also provided feedback on my final draft. While visiting Melbourne, Ken Ross also provided ideas and inspiration. I would also like to thank the other people in the Department of Computer Science at the University of Melbourne for their support. In this regard, I particularly thank Peter Poole and Peter Thorne who at different times managed this fine group of people in the seven years I was involved with them. I also thank CITRI (the Collaborative Information Technology Research Institute) for a comfortable environment in which to work. Justin Zobel, who is currently involved in CITRI, has provided inspiration and encouragement throughout.

The latest encouragement has come from my colleagues in the Department of Computer Science at Victoria University of Wellington. In particular, I thank Lindsay Groves who spent time both discussing and reading parts of this thesis.

I have also received valuable feedback from Michael Kifer and anonymous referees, who reviewed papers that have become part of this thesis.

I cannot name everyone here but on top of the people mentioned above I owe thanks to John Shepherd, Philip Dart, Prue Flower, Jane Glatz, Tomoko Akami, Francois Cosquer, David Mills, Michael Cleland, Sarah-Jane Beavitt, Kay Mills, Richard Hagen, Matthew Barry, Lex Scharaschkin, Tim Nickells, Lawrence Reeves, Bruce Baker, Robert Argent and Miles Darby. Finally, I thank my family for their support and understanding.

This work was partially supported by a Melbourne University Postgraduate Scholarship and a CITRI (Collaborative Information Technology Research Institute) Top-up Scholarship. I am also grateful to Griffith University where I was a Visiting Researcher for a year.

Contents

Chapter 1

Introduction

The use of database systems capable of storing and retrieving data is central to the operation of any organization. Different database systems use different data models or abstractions to represent an organization's data. A database system based on a simple model is usually easier to understand and use than a system based on a more complicated data model.

The early seventies saw the development of the relational model [37]. Because of its simplicity, the relational model became the base for a new generation of database systems. Relational databases are adequate for applications requiring only the long term storage of data, the retrieval of data, and the sharing of data in a controlled way. But applications for computer-aided design, software engineering, office automation, image and graphics processing, artificial intelligence and many other areas require an environment for programming and manipulating data, as well as the storage, retrieval and sharing of data. The limitations of relational databases are well documented [40, 63]. They include the difficulty of translating data from a "real world" model to an "implementable" physical model; the use of query languages that are not computationally complete, often embedded in a programming language; the discrepancy between data structures in the query language, and the application language, which is commonly called "the impedance mismatch problem"; the inconvenience of not being able to store operations in the database; the difficulty of ensuring that only semantically correct modifications are made to the database; and the inability to define a new entity type as a specialization of an existing entity type.

From the mid-eighties, various extensions to relational databases and alternative types of systems have been proposed and implemented [13, 55, 83, 95]. Each of these alternatives had different goals, but most attempted to provide a richer data model and address the impedance mismatch problem by providing a uniform language for representing data, views, constraints, queries, and application programs. Two of these extensions are deductive databases and object-oriented databases. Deductive databases extend the class of queries that can be expressed, and preserve the mathematical foundation of relational systems and object-oriented databases have a rich data modeling capability.

In the remainder of this chapter, we introduce the main concepts of deductive databases and object-oriented databases, discuss possible approaches to combining the two models, and describe the structure of the thesis.

1.1 Deductive databases

A deductive database stores facts (or relations) and rules, which derive information from the facts. The same language can be used to express facts, rules and queries. Answers to queries are in turn relations. The following example illustrates some of the features of deductive databases.

Example 1.1.1 Consider a database that stores facts about direct flights.

$$\begin{array}{l} flight(melbourne, sydney). \\ flight(sydney, brisbane). \\ flight(melbourne, alice\_springs). \\ flight(alice\_springs, sydney). \end{array}$$

The first fact states that there is a flight from *melbourne* to *sydney*. Using a deductive database it is easy to derive the different routes between a source and destination using the following rules.

$$\begin{array}{l} stops(Origin, Destination, []) \leftarrow flight(Origin, Destination). \\ stops(Origin, Destination, [Stop \mid Stoplist]) \leftarrow flight(Origin, Stop), \\ \qquad stops(Stop, Destination, Stoplist), \textit{not } in\_list(Stop, Stoplist). \\ in\_list(Head, [Head \mid Tail]). \\ in\_list(Item, [Head \mid Tail]) \leftarrow in\_list(Item, Tail). \end{array}$$

The program, which consists of the rules, is also stored in the database. The program states that there are no stops if there is a direct flight. It includes *Stop* in a list if there is a direct flight from *Origin* to *Stop*, and there are flights from *Stop* to *Destination*, and *Stop* has not already been included in the list.

Consider a query that asks for all the routes from *melbourne* to *brisbane*:

$$?stops(melbourne, brisbane, Stops).$$

The answers to this query with respect to this database are:

$$\begin{array}{l} [sydney] \\ [alice\_springs, sydney]. \quad \square \end{array}$$

Deductive databases extend relational databases by providing capabilities for defining and efficiently evaluating recursive views, thus significantly extending the class of queries that can be expressed [49]. Relational databases coupled with imperative programming language suffer from the impedance mismatch problem where there is a discrepancy between data structures in the query language and the application language. In deductive databases, as the same language is used to express the facts and rules that make up the database, and queries on the database, there is no impedance mismatch problem. Also, because deductive databases store rules in the database, and constraints ensuring that only semantically correct modifications are made to the data can be expressed as rules, constraints can be stored in the database rather than embedded in application programs. Deductive databases preserve the firm mathematical foundation of relational databases and, using this well-understood foundation, much work has been done on their application to problems involving recursive or cyclic data [16, 52, 102], on the semantics of databases with increasingly complex combinations of recursion, negation and aggregates [41, 61, 75, 98, 104, 106], on query optimization [17, 15, 59, 103, 106], and on integrity constraint maintenance [26, 78].

However deductive databases fail to significantly improve the data modeling capabilities of the relational model. For example, it is not possible to encapsulate rules with the data to which they apply, define a new entity type as a specialization of an existing entity type, or describe the relationship between two tuples where one is the value of an attribute of the other.

1.2 Object-oriented databases

An object-oriented database models the real world as a set of related objects. Each object has a set of attributes, and methods for accessing and updating its attributes. Attributes may have complex structured values composed from simple values and objects. Objects are assigned to classes. Based on the set of attributes and methods available to objects in a class, classes are arranged in a lattice. In this lattice subclasses represent specializations of their superclasses. That is, subclasses inherit the attributes and methods of their superclasses and may have additional attributes and methods of their own. The attributes and methods inherited from a superclass may be overridden in the subclass. One of the distinguishing features of the object-oriented model is that the methods are associated with the data. Since the model has rich modeling capabilities, the disparity between real world entities and the way they are modeled in an object-oriented database is less severe than in the relational data model.

The following example illustrates some of the features of object-oriented databases.

Example 1.2.1 This example is part of a running example of Bertino and Martino in [21]. The syntax is similar to the syntax of a language of O_2 [39]. Consider a database that stores details about documents, their title and classification number, and if the document is a technical report, the date on which it appeared. A possible schema is:

```
class TRDate
     type (  month: integer,
             year: integer )

class Document
     type (  title: string,
             classification: string )
     method init(string, string): Document is public

class Tech_Report inherit Document
     type (  date: TRDate)
```

The class *TRDate* has attributes *month* and *year*. The class *Document* has attributes *title* and *classification*, and method *init*. The class *Tech_Report* is a subclass of *Document*, so *Tech_Report* inherits attributes *title* and *classification* and method *init* from *Document*. The class *Tech_Report* also has an attribute *date* whose value is an object of class *TRDate*. Consider the following program with respect to the schema:

```
method body init(t:string, c:string): Document in class Document
{
        self → title = t;
        self → classification = c;
        return(self)
}

method body init(t:string, c:string): Tech_Report in class Tech_Report
{
        self Document@init(t, c);
        self → date → month = current_month;
        self → date → year = current_year;
        return(self)
}
```

```
name gills_report : Tech_Report;
run body
{
    gills_report init("Foundations of OO Systems", "QA76.9")
}
```

The method *init* in class *Document* assigns a *title* and *classification* to the appropriate attributes. The overriding method *init* in class *Tech_Report* specializes the method *init* in *Document*. It applies the method *init* from class *Document* and assigns appropriate values to the attributes of *date*. The object *gills_report* of class *Tech_Report* is created and the method *init* is applied to object *gills_report*.

Consider a query that asks for the *title* and *classification* of each object in *Document* where *Document* is a collection of *Document* objects, including *Tech_Report* objects.

```
select tuple (title: d.title, classification: d.classification)
      from d in Document.
```

The answer to this query with respect to the program and schema is ("Foundations of OO Systems", "QA76.9"). In this query, navigation is carried out through the object identifier, assigned to *d*. □

In relational database design there is a compromise between redundancy of data and efficiency of access to data. When accessing data, it is better to have large relations and repeated information since this avoids expensive join operations. When updating a database, it is better to store the data only once, thus updating it only in one place. Because object-oriented databases support complex objects, and navigation can be carried out through object identifiers, join operations and redundancy of data are less of an issue in object-oriented database design. As attributes and methods are encapsulated, methods can be defined to check semantics when an update occurs. In relational databases, semantic checking is carried out in application programs. Such checks are sometimes forgotten and there can be redundant code where the same checks are carried out in many application programs. Object-oriented databases allow more information to be included in the database.

Object-oriented databases typically extend an object-oriented programming language, such as C++ or Smalltalk, to provide a uniform query and evaluation language [25], although they often also provide an extension of SQL as a separate query language [40, 39, 110]. The ability to define a class hierarchy and redefine methods provides significant opportunity for code reuse, extension and modification [57]. Such databases usually provide environments where there is no impedance mismatch. Several commercial object-oriented databases have appeared on the market. These commercial databases lack a mathematical foundation [25, 39, 67, 88, 110]. The lack of a mathematical foundation restricts the construction of effective design and optimization tools, inhibits usability, and adversely affects reliability.

1.3 Combining deductive and object-oriented databases

It seems natural to try to combine the features of deductive and object-oriented databases as deductive databases only slightly improve the data modeling capabilities of the relational model but preserve the mathematical foundation of relational databases and object-oriented databases provide a rich data modeling capability but lack a mathematical foundation. Such a combination would include objects, classes, functional methods, multi-valued methods, (multiple) inheritance, overriding, negation and deduction.

There have been several attempts to do this [32, 38, 68]. The languages defined in these attempts can be categorized into two groups: a direct and an indirect approach. A direct approach defines the semantics of a language directly whereas an indirect approach defines the semantics of a language by describing a translation from that language to a language that has a well-defined semantics. Kifer, Lausen and Wu [65] give reasons why it is best to provide the semantics of a deductive object-oriented language directly rather than indirectly. They are:

- a translation is merely an algorithm and provides little insight into the nature of object-oriented concepts the language is designed to model,
- even for simple programs, their image is not easily understandable, and
- a direct semantics for a language shows a way of defining a proof theory that is tailored to that language.

There are approaches that do provide the semantics directly. In general these approaches do not present the features in a uniform, simple, and elegant manner. The description of F-logic in [65] provides a direct semantics of a large set of concepts. However, it was demonstrated in [74] that F-logic is not easy to implement.

In this thesis, we intend to provide an alternative description to F-logic with the aim of providing additional insight into how to apply and effectively implement these concepts. We capture the main features of deductive and object-oriented databases in a logic-based language and analyze different aspects of this language in a uniform and simple setting. In order to maintain a simple setting, in the body of this thesis, we consider data and queries only, and postpone the study of updates to future work.

This thesis presents a complete data and query language and studies its properties. Earlier, more restrictive versions of the language were presented in [44] where the language was restricted to objects, predicates, types, functional methods, deduction, inheritance and overriding, and in [45], where multi-valued methods and multiple inheritance were introduced. The language described in the body of this thesis is an extension of that in [45] and includes negation.

1.4 Thesis outline

In this thesis, we describe a new way to combine deductive and object-oriented databases. We describe a language, Gulog, that has objects, classes, functional methods, multi-valued methods, (multiple) inheritance, overriding, predicates, negation and deduction.

Chapter 2 reviews definitions that are used later in the thesis. First, concepts from the areas of logic programming and deductive databases are described. Then, concepts from the area of object-oriented databases are discussed. Finally, related work in which different approaches to combining deductive and object-oriented databases are described in this chapter.

Chapter 3 introduces the new approach to combining deductive and object-oriented databases. It starts with an example that portrays how the concepts introduced in Chapter 2 are expressed in the new language. Then there is a formal description of the syntax of the logic language, Gulog. A deductive object-oriented database is a Gulog program. A query is the body of a Gulog clause. Later when we discuss the semantics of a program we require a class of programs with the following restrictions: no conflicting definitions due to multiple inheritance; all functional methods are single-valued; no cycles in the definition of an atom with respect to negation and deduction; and no cycles in the definition of a method with respect to inheritance and deduction. In Chapter 3, we introduce this class of programs and call them "simple" programs. The model-theoretic or declarative semantics of the language is described. It captures the intended meaning of a deductive object-oriented database. We prove that every simple program has a unique intuitive model. Finally, we show how any Gulog program can be modeled in a subset of Gulog, without negation. This hints at the relationship between inheritance and overriding, and negation.

Chapter 4 describes query evaluation procedures. First, a bottom-up query evaluation procedure is described for deductive object-oriented databases. Next, we prove that this procedure is sound and complete for simple programs with respect to the declarative semantics described in the previous chapter. A query procedure is sound if it returns only expected answers, and complete if it returns all the expected answers with respect to the declarative semantics. As a goal-directed procedure is more efficient under some circumstances, a top-down query evaluation procedure is also described. We prove that the top-down query evaluation procedure is sound for simple programs with respect to the declarative semantics. However, this procedure does not always terminate so is not complete. So, we describe a top-down procedure with a tabling mechanism and prove that this procedure is sound and complete for simple programs with respect to the declarative semantics.

Chapter 5 describes a translation from Gulog to Datalog with negation, a theoretical deductive database language, and proves that the translation is correct for simple Gulog programs. This translation facilitates an indirect approach to query evaluation that was referred to in Section 1.3, thus providing an alternative query evaluation procedure to those presented in Chapter 4. The translation also provides the relationship between a language for deductive object-oriented databases and a well-understood language for deductive databases.

Chapter 6 describes extensions to Gulog, demonstrating how the semantic approach taken in this thesis can be used to describe a wider range of features. The first extension is a generalization from the class of simple programs to a larger class of m-simple programs. There are programs that belong to the larger class, but not the smaller class, that have a natural semantics. The natural semantics of m-simple programs is defined in this chapter. We give a translation to modularly stratified Datalog, and prove that it is correct for m-simple programs. This translation provides an evaluation procedure for m-simple programs in the same way that the previous translation provided an evaluation procedure for simple programs. Part of the restriction to simple and m-simple programs involves eliminating programs that have conflicting definitions due to multiple inheritance. An alternative way to deal with such conflicts is to let the programmer specify from which class a method is inherited. The second extension adds "roled atoms" to the language. Roled atoms provide an alternative way to handle multiple inheritance and also allow the modeling of monotonic inheritance. The third extension adds aggregate and arithmetic operators to the language. The extensions make the language more expressive.

Chapter 7 reviews the main results, describes how they relate to other work, and considers future work.

Chapter 2

Basic Definitions

This chapter provides the background in deductive databases and object-oriented databases, which is assumed in the following chapters. The definitions in Section 2.1 are based on those of Lloyd [77]. The definitions in Section 2.2 are based on those of Atkinson et al. [13], Bertino and Martino [21], and Cattell [31].

2.1 Deductive databases

This section provides definitions and reviews research in the areas of deductive databases and logic programming. Deductive databases are closely related to logic programming. We first restate well-known definitions, and then discuss some of the current research in the area.

Much of the field of logic programming is based on *first-order logic*. First-order logic has two aspects: syntax and semantics. The syntactic aspect is concerned with what formulas are admitted by the grammar of a formal logic. The semantics is concerned with the meaning attached to the symbols in the formulas. A *first-order language* contains variables, propositional connectives, quantifiers, punctuation symbols, constant symbols, function symbols and predicate symbols.

A *term* is defined as follows:

- A variable is a term.
- A constant is a term.
- If f is an n-ary function and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom*. An atom is *ground* if it contains no variables.

A *literal* is an atom or the negation of an atom. A *positive literal* is an atom. A *negative literal* is the negation of an atom. A *program* is a set of clauses where each *clause* has the form $A \leftarrow L_1 \wedge \cdots \wedge L_n$. The *head* of the clause, A, is an atom. The *body* of the clause, $L_1 \wedge \cdots \wedge L_n$, is a conjunction of literals. A *goal* is a clause with an empty head. A clause is *ground* if it contains no variables.

The *semantics* of a logic programming language may be defined in three ways: proof-theoretic, model-theoretic, and fixpoint. The *proof-theoretic* semantics views a program as a conjunction of formulas and the meaning of the program is the set of formulas that can be proved from the program. The *model-theoretic* or declarative semantics considers the meaning of the program to be a particular minimal model for the program. The *fixpoint* semantics defines the meaning of a program as the least fixpoint of an immediate consequence operator associated with the program.

The *model-theoretic semantics* of first-order logic is defined by an interpretation that consists of a non-empty domain and which assigns a meaning to each non-logical symbol in the language. Because in logic programming we are interested in the set of ground facts derivable from a set of clauses, we can restrict the interpretations we consider to Herbrand interpretations of a language. To define Herbrand interpretation, we need the notions of Herbrand universe and Herbrand base.

The *Herbrand universe*, U_L, of a language L is the set of all variable free terms that may be constructed from the constants and function symbols of L. If language L has no constants, we add one so the ground terms can be formed.

The *Herbrand base*, B_L, of a language L is the set of all ground atoms that can be formed by using the predicates from L, with ground terms from the Herbrand universe as arguments.

We regard the language L associated with a program P as being fixed, and hence we use both U_L and U_P interchangeably to denote the Herbrand universe of L, and we use B_L and B_P to denote the Herbrand base of L.

A *Herbrand interpretation*, I, for P consists of a set of ground atoms of P which is a subset of the Herbrand base of P. A Herbrand interpretation I of P is a *Herbrand model* of P if each clause of P is true with respect to I.

Generally, an interpretation I is a *model* for a program P if every clause of P is true in I. We write $P \models w$ if a clause w is true in all models of P.

There are classes of programs based on sublanguages that are interesting for different reasons. In *definite programs*, the body of each clause is a conjunction of atoms. They are an interesting class of programs because they are easy to understand and have a simple declarative meaning. A generalization of definite programs is *normal programs*, where the body of each clause is a conjunction of literals (positive and negative literals). There is a subclass of normal programs called *stratified programs* that restrict the use of negation and has a well-defined semantics [90].

Example 2.1.1 Consider the following stratified program P.

$p(x) \leftarrow \neg q(x).$
$q(a).$

Assume a is the only constant of the language associated with P. Then in P $q(a)$ is true, and for any value of x, $p(x)$ is true if $q(x)$ is false. The only value we can assign to x is a. Thus, the only fact we can deduce from P is that $q(a)$ is true. A model or intuitive meaning of P is $\{q(a)\}$. $\square$

There are normal programs that are not stratified and yet have a model that clearly seems to be appropriate.

Example 2.1.2 Consider the following non-stratified program.

$p(a) \leftarrow \neg p(b) \wedge \neg p(c).$
$p(b) \leftarrow \neg p(a).$
$p(c).$

An intuitive model for this program is $\{p(b), p(c)\}$. $\square$

The *well-founded* semantics [105] provides a meaning to all normal programs. The well-founded semantics generalize the stratified approach and provides intuitive, reasonable semantics for non-stratified programs.

An evaluation procedure is used to determine whether a goal is true or false with respect to a program. There are two kinds of evaluation procedures. The *top-down evaluation* *procedure* is related to the proof-theoretic semantics, and the *bottom-up evaluation procedure* to the fixpoint semantics. A bottom-up procedure starts with the program facts and applies the clauses to obtain new facts until all derivable facts have been generated [12]. A top-down procedure starts with the goal and verifies the body of the appropriate clauses in order to make the goal true. SLD-resolution is a common top-down evaluation procedure for definite programs [77]. Optimization techniques make these procedures viable in practice. In particular magic sets [17] focus the procedure on the data relevant to the query for bottom-up procedures and tabling [41] eliminates infinite loops when there is recursion in programs for the top-down procedures. The correctness of a procedure can be measured with respect to the model-theoretic semantics. Suppose S is the well-founded model of a program [105]. A procedure is *sound* if whenever the answer to the goal w is true in the procedure then $S \models w$, and *complete* if whenever $S \models w$ then the answer to the goal w is true in the procedure.

Now, consider deductive databases. A *database* is a program. A *query* is a goal. There is an interesting theoretical language called *Datalog*. A *Datalog program* is a definite program without function symbols. Thus, Datalog programs avoid infinite domains that can lead to infinite sets of answers. A Datalog program is a specialization of a definite program. *Datalog with negation* is Datalog with literals allowed in the body of clauses.

Although allowing negated atoms makes a language more expressive, it also complicates the semantics. *Negation* is defined implicitly; that is, anything that is not true in the program is assumed false. Przymusinski described the model-theoretic semantics for normal programs in [90]. With top-down evaluation, one treats negation as the failure to find a positive solution to an atom [77]. With bottom-up query evaluation, a fact is false if it is not an element of the model generated [12].

Extensions to logic programming include aggregate and arithmetic operators, typing, single-valued data functions, and complex values such as sets. *Arithmetic operators* [33] and *aggregate operators* [104] have been added to logic programming and deductive databases in order to make the language more expressive. In [33], Chandra and Harel represent arithmetic operators as infinite relations over the set of numbers. The standard declarative semantics, bottom-up and top-down query evaluation procedures have been extended to incorporate aggregate operators [60, 75, 85, 92, 94, 104].

Example 2.1.3 Consider a database that stores information about employees in the form $employee(Name, Dept, Salary)$. Using aggregates, it is possible to define a clause to determine the *average salary* for each *department*.

$$\begin{aligned}&avg\_salary\_per\_dept(Dept, AvgSal) \leftarrow\\&\quad group\_by(employee(Name, Dept, Salary), [Dept], AvgSal = avg(Salary)).\end{aligned}$$

This clause groups facts by *department*, and calculates the *average salary* for each *department*. □

Types provide a natural way of capturing the domain concept of relational databases in deductive databases. The fundamental typing scheme is to use typed first-order logic or many-sorted logic [77], where there are several sorts of variables each ranging over a different domain. Many other schemes have been proposed [86, 89]. One scheme that has some of the features required for object-oriented databases was proposed by Hill and Topor [54], who describe a polymorphic type system for logic programming that includes parametric and inclusion polymorphism. Inheritance can be modeled using inclusion polymorphism.

Example 2.1.4 Consider the type declarations:

$spot{:}terrier$
$terrier \leq dog$

which state that constant *spot* is of type *terrier*, and *terrier* is a subtype of *dog*, and a clause

$$(\forall x{:}dog)legs(x, 4).$$

The clause states that all *dog*s have 4 legs, so *spot* has 4 legs. □

Cardelli and Wegner [28] have investigated a more general typing system for functional programming. The authors provide a characterization of a type system that includes abstract data types, parametric polymorphism and multiple inheritance in a consistent framework. The authors introduce a language called Fun that is based on λ-calculus. Although this is a functional programming language rather than a logic programming language, this work does provide insight into strongly typed object-oriented languages.

Single-valued data functions have also been included in logic programming languages [5]. Allowing single-valued data functions provides a way to state functional dependencies explicitly. For example, it provides a way of showing that a *person* has only one *age*. Thus more information can be expressed in the program, which in turn can be used in query optimization. Abiteboul and Hull [5] extend Datalog to incorporate single-valued data functions. These data functions can be facts or derived using clauses. Obviously, a program may not be consistent because a derived function may evaluate to something that is not single-valued. The authors show that detecting consistency is undecidable. However, they do present several cases where consistency is decidable.

It is also recognized that forcing terms to be atomic values is restrictive. Indeed logic programming languages already allow functions, lists and tuples. There are various attempts to add sets and other complex objects to logic programming languages. *Complex objects* are values obtained using set and tuple constructors. In [70], Kuper introduces sets by allowing restricted universal quantifiers in the body of a clause, and considering a two-sorted logic. Naqvi and Tsur describe how sets are dealt with in LDL in [87], using rule rewriting techniques. One of the attractive properties of logic programming is its equivalence to relational algebra. Obviously complex structures upset this relationship. Abiteboul and Beeri [2] present an extended algebra and a logic language with complex objects and prove their equivalence.

To summarize, like relational databases, deductive databases have a firm mathematical foundation. Deductive databases extend relational databases by providing capabilities for defining and efficiently evaluating recursive views, significantly extending the class of queries that can be expressed. As the same language is used to express the facts and clauses that make up the database, and queries on the database, there is no impedance mismatch problem. As deductive databases store clauses in the database, and constraints ensuring that only semantically correct modifications are made to the data can be expressed as clauses, constraints can be stored in the database. Deductive databases slightly improve the data modeling capabilities of the relational model and as we have illustrated above, they can be extended to include negation, aggregates and arithmetic operators, typing, and complex structures.

The main disadvantage of deductive databases is that they fail to significantly improve the data modeling capabilities of the relational model and hence do not address the problem of translating data from a "real world" model to an "implementable" physical model.

2.2 Object-oriented databases

Atkinson et al. [13] proposed an early definition of object-oriented databases. They described the features of object-oriented databases and divided them into three categories: the features a system must have to qualify as an object-oriented database; features that can be added to make a system better; and features where the designer has a choice. In their description of object-oriented databases, Bertino and Martino [21] did not use the categories proposed by Atkinson et al. but adopted descriptions from the definition of Atkinson et al. Although there is still no common model and no standard for object-oriented models, the concepts in the area of object-oriented databases are becoming clearer and there are moves towards standardization, such as ODMG-93 [31]. Because many of the concepts are closely related to similar concepts in the area of object-oriented programming languages, we include references to object-oriented programming languages. There is an example in Section 3.1 that helps clarify the concepts described in this section.

When designing a database, it is necessary to consider the universe of discourse, and define a schema for this universe. The *schema* in an object-oriented model consists of object templates and relationships between these templates. The object templates describe the structure of objects, this includes the structure of both the data and the behavioral aspects of the objects. The object templates are called *classes* or *types*.

The terms "class" and "type" are often used interchangeably. Usually there is no distinction between the two concepts in programming languages. In languages that do differentiate between type and class, such as that of America [11], type is a static concept whereas class is a dynamic concept. Types are used for checking the correctness of a program and cannot be modified at run-time. Classes are not used for checking the correctness of a program. They are used for generating and grouping objects. A class can be modified at run-time. Having different terms for the two concepts is usually unnecessary and complicates the description, so we will continue to use the term type to cover both concepts. Thus a type defines the state and behavior of its instances.

In object-oriented database systems, each real world entity is represented by an *object*. Each object is an instance of a type, so each object belongs to a type. Each object is identified by a unique *object identifier* (oid). Complex objects are objects that are built from other data structures, including other objects. Object identifiers facilitate object containment. Each object has a state and a behavior. In fact, an object is said to encapsulate both state and behavior. That is, no operations outside those specified in the interface can be performed on the object. Encapsulation provides a form of logical data independence where application programs are protected from implementation changes in the lower layers of the system. State is expressed by a set of *attributes* and behavior by a set of *methods*. Methods specify operations on attributes and are used to access the attributes. Thus, methods form the interface between the data and the user. Some systems do not differentiate between methods and attributes.

The definition of a method has two components: signature and body. The signature specifies the name of the method, the types of the arguments, and the type of the result (if there is one). Some languages, like Smalltalk [50] do not require a signature. In other languages, like CLOS [84], the signature is optional. The body represents the implementation of the method and consists of a set of instructions expressed in a programming language. Methods can be functional or multi-valued. A *functional method* returns only one value when the method is applied to an object with a given set of arguments. Such methods are also called single-valued methods. For example, a person has only one birthdate. So *birthdate* could be implemented as a functional method. A *multi-valued method* returns one or more values when the method is applied to an object with a given set of arguments. Multi-valued methods are also called relational methods. For example, a person may have many children. So *children* could be implemented as a multi-valued method. An attribute definition is like a method definition without a body.

Recall that a *type* defines the structure and behavior of objects of a particular kind. It defines the attributes and methods of objects of that type and the relationships in which objects having this type can participate. For example, objects of type *person* can have attributes *name*, *address* and *phone number*, and methods for accessing these attributes. *Address* may be an object so there is a relationship between objects of type *person* and *address*.

Types are organized in an hierarchical manner. *Specialization inheritance* [69] defines an inheritance hierarchy where a subtype is a specialization of its supertype. That is, a subtype inherits the attributes and methods of its supertype, and may also have attributes and methods of its own. For example, *student* may be thought of as a specialization of *person* as *student* has the same characteristics as *person* plus a *student number*. The intention is that an instance of a subtype may be treated like an instance of each of its supertypes.

An obvious generalization of this is *multiple inheritance*, where a type inherits directly from more than one parent. In this case, the types are organized in a directed acyclic graph. For example, *teaching assistant* inherits the characteristics of *student* and *employee*, and may in turn have extra characteristics of its own. Multiple inheritance can lead to conflicting definitions where a type can inherit an attribute or method with the same name and the same number of arguments from different supertypes. The conflicts can be in the method and attribute signatures or in the method and attribute definitions. There are two approaches to resolving conflicting signatures, the conflicts can be prohibited, or the signature in the subtype can be required to conform with all the conflicting signatures, as in CLOS [84]. There are four main approaches to resolving conflicts in method and attribute definitions:

- Prohibit conflicts. This approach is taken in C++ [76] and Eiffel [82]. Eiffel also provides a way to rename attributes and methods in subtypes to help avoid conflicts.

- Provide a static ordering on supertypes that determines which supertype should be inherited from first, next and so on. This approach is taken in CLOS [84], where the ordering on supertypes in which conflicts could occur depends on the order in which they are specified in the list of supertypes for a type.

- Allow the user to write programs that determine which supertype to inherit from. This is the approach taken in O_2 [39].

- Allow any of the conflicting definitions to be correct. This approach is taken in *L&O* [81].

With inheritance, a type inherits attributes and methods from its supertypes. It is sometimes useful to be able to redefine these attribute and method definitions at a lower level in the graph. This redefinition is known as *overriding*.

In [30], Cattell describes four kinds of inheritance. They are: *specification*, where the subtyping depends on something like the value of an attribute; *classification*, where subtypes have the same attributes and methods as supertypes and subtyping is used simply to classify objects; *specialization*, where subtypes may have additional attributes or methods to a supertype; and *implementation*, where subtypes provide different implementations of the methods defined in a supertype. Using specialization inheritance with overriding, it is possible to model these four types of inheritance.

Queries are used to extract information from database systems. In the object-oriented setting, a query is generally formulated on a set of instances of a type and consists of a boolean combination of predicates expressing conditions on the attributes of the objects. Queries can also invoke methods.

Persistence is an important issue in object-oriented database systems. There are three main approaches:

- Persistence is an implicit characteristic of all instances of types. That is the creation of an instance has the effect of inserting the instance in the database. This approach is used in ORION [67].
- Persistence is explicit. An instance created during the execution of a program is deleted at the end of the program, unless it is made persistent. This approach is used in O_2 [39].
- An intermediate approach where types are categorized into persistent types and temporary types. All instances of persistent types are automatically created as persistent instances, whereas this doesn't happen with instances of temporary types. This approach is used in the E language [29].

The advantages of object-oriented databases over relational databases include their data modeling capabilities, such as being able to model complex objects directly, and being able to define an entity type as a specialization of an existing entity type. Other advantages include mechanisms for associating operations with data, sharing method definitions through inheritance, removing the behavioral semantics of objects from application programs and mechanisms for providing an identity that is separate from the state of the object.

A mathematical foundation can give a meaning to the data, provide a basis for database design, allow the use of simple declarative query languages, and enable queries to be optimized automatically. One of the common criticisms [21] is that object-oriented databases are not based on a coherent model or mathematical foundation. This is the issue that we are addressing in this thesis.

2.3 Combining deductive and object-oriented databases

It seems natural to try to combine the features of deductive and object-oriented databases. The result, deductive object-oriented databases, would have a rich modeling capability and a firm mathematical foundation. It is necessary to deal with the following declarative aspects of object-oriented databases: objects with associated state, inheritance including multiple inheritance, attribute and method overriding, and attributes with complex values. It is necessary to consider clauses that can define objects, attributes and methods. In this section, we discuss concepts which arise due to the combination of object-orientation and logic, and review approaches that have influenced the area of deductive object-oriented databases. We divide the approaches into the following groups:

- combining object-orientation with logic to build a programming language,
- studying object-oriented concepts in a mathematical framework, with the aim of formalizing the notions underlying object-oriented systems,
- studying object-oriented concepts in a mathematical framework, with the aim of building object-oriented databases or query languages.

We first describe some properties used in this section, including the non-monotonic behavior of inheritance with overriding, monotonic inheritance, static versus dynamic overriding, different approaches to the interaction of inheritance and recursion, and the credulous versus the skeptical approach to resolving ambiguities due to multiple inheritance.

In the previous section, we described overriding as the redefinition of methods and attributes in subtypes. Overriding causes *non-monotonic* behavior. That is, adding a new clause to a database can make false something that was previously true. For example, suppose *sub* is a subtype of *super* and there is an object *o* of type *sub*, as shown in Figure 2.1.

Suppose there is only one fact in the database and it assigns the value 3 to method *m* for any object of type *super*. Then, due to inheritance and because there are no clauses that define method *m* for objects of type *sub*, we can deduce that object *o* has value 3 for method *m*. Suppose now that we add another fact to the database that assigns the value 4 to method *m* for any object of type *sub*. Because this new clause overrides the old clause for objects of type *sub*, object *o* has value 4 for method *m*, and it is no longer true that object *o* has value 3 for method *m*.

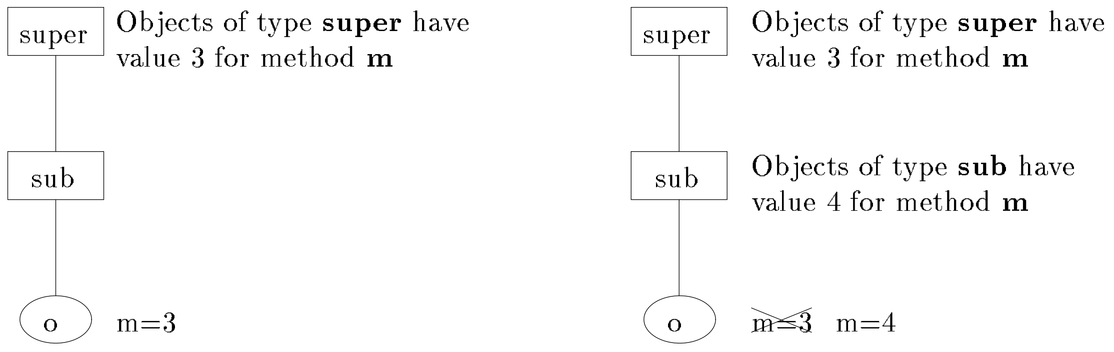
super
Objects of type **super** have value 3 for method **m**
sub
o
m=3
super
Objects of type **super** have value 3 for method **m**
sub
Objects of type **sub** have value 4 for method **m**
o
~~m=3~~
m=4

Figure 2.1: Non-monotonic behavior of overriding

Monotonic inheritance is inheritance without overriding. For example, in Figure 2.1 after the new clause is added with monotonic inheritance object *o* has value 4 and value 3 for method *m*.

Abiteboul et al. [7] give a translation from a deductive object-oriented language to Datalog with negation. In this paper they defined two types of overriding: *static* and *dynamic*. With static overriding, if there is an overriding definition, it overrides the definition in the parent regardless of the value of the body of the definition. With dynamic overriding, an overriding definition overrides only if the body of the overriding clause is true. Suppose again, *sub* is a subtype of *super* and there is an object *o* of type *sub*, as shown in Figure 2.2. Consider a database with two clauses. The first assigns the value 1 to method *m* for any object of type *super*. The second assigns the value 2 to method *m* for any object of type *sub* if the value of method *n* applied to the same object is 2. Suppose that the value of method *n* applied to object *o* is not 2. Then using static overriding, the value of method *m* when applied to object *o* is not 2 and inheritance does not take place, so we can say nothing about the value of method *m* when applied to object *o*. Using dynamic overriding, the value of method *m* when applied to object *o* is not 2, but inheritance does take place so the value of method *m* when applied to object *o* is 1.

The semantics of the interaction of overriding and recursion is not well defined. There are two possibilities in the case of dynamic overriding. Suppose *sub* is a subtype of *super*, and a method *m* is defined on *super* and redefined on *sub*. If the definition of method *m* in *sub* does not override the definition of method *m* in *super* for object *o*, then the definition in *super* is applied. For a recursive call in the body of the clause for method *m* in *super* there are two possibilities. Either the definition of method *m* in *super* is reapplied or the definition in type *sub* is tried again.

Inheritance with overriding has also been studied in the area of Artificial Intelligence [101]. However, in this field dynamic overriding is not considered and types, instances and attributes are not distinguished. Two approaches to conflicting definitions due to multiple inheritance are described in the literature: the *skeptical* and *credulous* approaches. With the skeptical approach, neither of the conflicting definitions is inherited. With the credulous approach, one

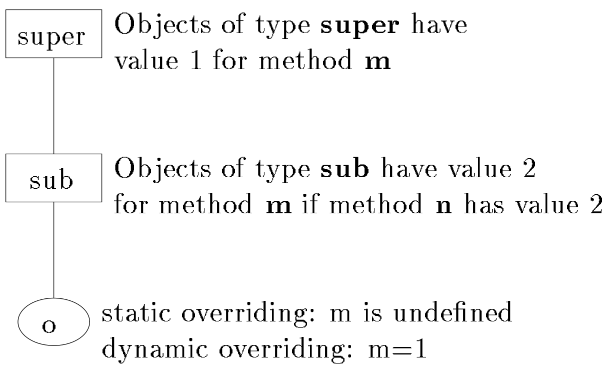
super
Objects of type **super** have value 1 for method **m**
sub
Objects of type **sub** have value 2 for method **m** if method **n** has value 2
o
static overriding: m is undefined
dynamic overriding: m=1

Figure 2.2: Static vs dynamic overriding

of the conflicting definitions is inherited. The standard example used in the literature [101] is the Nixon diamond shown in Figure 2.3. In this figure, *Nixon* is a *Quaker* and a *Republican*; *Quakers* are *Pacifists*; *Republicans* are not *Pacifists*. The conflict is then whether *Nixon* is a *Pacifist* or not. Taking the credulous approach we deduce that either *Nixon* is a *Pacifist*, or *Nixon* is not a *Pacifist*. Taking the skeptical approach we can deduce nothing about *Nixon* and *Pacifism*.

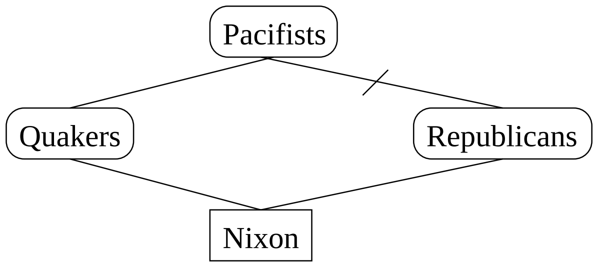

Figure 2.3: Nixon Diamond

We now investigate the various approaches that have influenced deductive object-oriented databases. We investigate two examples of combining object-orientation with logic to provide a programming language. McCabe described *L&O* [81] which is a deductive object-oriented language with overriding. Although the author's motivation was to describe a logic language that supported programming-in-the-large, much of his work can also be applied in the area of databases. McCabe does not distinguish between classes and objects. In [81], McCabe describes two kinds of inheritance. The first is where relations defined in the type template are inherited, not the definitions. This approach is taken in *L*&*O*. In the second, the definitions themselves are inherited. This approach can be modeled in *L*&*O* using the keyword *self*. The two approaches are illustrated in the following example. Suppose *sub* is a subtype of *super* and there is an object *o* of type *sub*, as shown in Figure 2.4. Suppose the value 1 is assigned to method *n* for any object of type *super*, the value 2 is assigned to method *n* for any object of type *sub*, and the method *m* is assigned the value of *n* for any object of type *super*. Using the first approach, method *m* has value 1 for object *o*. Using the second approach, method *m* has value 2 for object *o*. If the clause on *super* is changed such that the method *m* is assigned the value of *n* when applied to the object o, then using the first approach method *m* has value 2 for object *o*. All definitions are inherited unless specifically overridden. Thus, with a conflict due to multiple inheritance, the credulous approach is adopted. The semantics of *L&O* programs is given by translating *L&O* programs to logic programs. This provides little insight into how inheritance, overriding and deduction interact. Query processing too is described using a translation to Prolog. McCabe does mention that Gurr has provided an independent semantics for *L&O*, based on the fact that an *L&O* program comprises independent programs that interact. This approach is similar to that described by Brass and Lipeck [23] where the global semantics of a program is defined in terms of the local semantics of each object in the program.

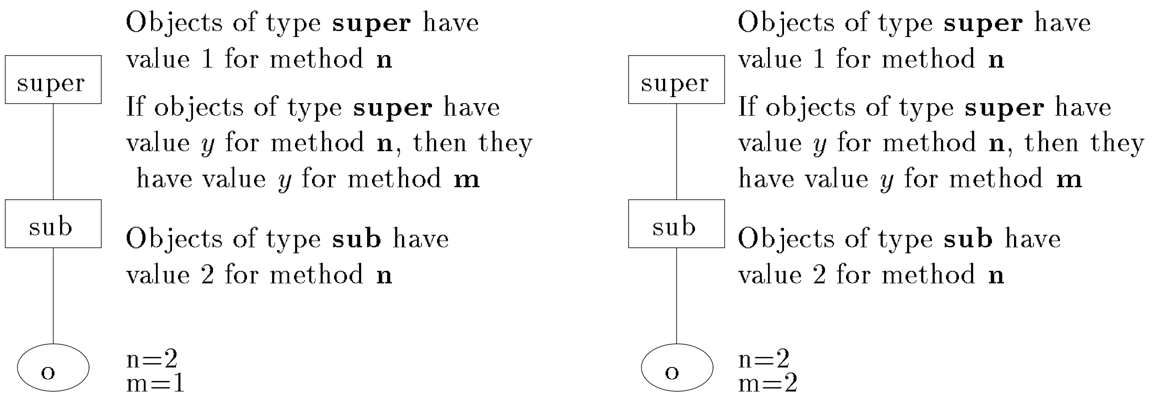
Objects of type **super** have value 1 for method **n**
super
If objects of type **super** have value y for method **n**, then they have value y for method **m**
sub
Objects of type **sub** have value 2 for method **n**
o
n=2
m=1
Objects of type **super** have value 1 for method **n**
super
If objects of type **super** have value y for method **n**, then they have value y for method **m**
sub
Objects of type **sub** have value 2 for method **n**
o
n=2
m=2

Figure 2.4: Inheritance in *L&O*

In [58], Jouannaud, Kirchner, Kirchner and Megrelis describe OBJ, which is a declarative language whose semantics is given by order-sorted equational logic, and whose operational semantics is based on order-sorted term rewriting. This approach stems from the area of artificial intelligence. OBJ supports objects, classes, subclasses, overriding, and multiple inheritance. Because term-rewriting is used, this setting is not conducive to understanding the procedural aspects of object-oriented databases.

There are a number of groups of approaches that aim to formalize the notions underlying object-oriented systems. In [19], Beeri discusses efforts to extend the relational model and its languages so as to capture some object-oriented features. The author describes an object model in which structural inheritance, object identity, and classes are represented. In this model the schema is separate from the data, sets are represented as predicates, and each object is its own identity. The answer to a query is a relation.

One of the initial attempts to formalise the notions underlying object-oriented systems was O-logic [80]. Maier describes how objects, a class hierarchy and methods can be represented in a logic using object identity, labels and types.

Two papers that followed from Maier's work were [36] and [66]. Chen and Warren [36] introduce a logic called C-logic. Each formula in C-logic can be transformed into an equivalent first-order formula. Static type constraints and single-valued labels are not built into the logic but can be added on top of it if needed.

In [66], Kifer and Wu extended O-logic to incorporate sets and a way to deal with inconsistent information, and devised a sound and complete proof procedure.

This work was extended in [65] by Kifer, Lausen, and Wu who define a logic called Frame Logic (or F-logic) that accounts for most of the structural aspects of object-oriented and frame-based languages. Frame-based languages in Artificial Intelligence are also built around the concepts of complex objects, inheritance and deduction. The authors define a model-theoretic semantics and a sound and complete resolution-based proof procedure for F-logic. However, the proof procedure addresses only the monotonic part of F-logic. The authors do not distinguish between classes and objects, but they do differentiate between the relationships between them. That is, the subclass relationship has different semantics from the instance relationship. Attributes and methods are defined as inheritable or non-inheritable. They adopt the credulous approach to conflicting definitions due to multiple inheritance. In F-logic, ground clauses are inherited and overriding is dynamic.

In [8], Ait-Kaci and Nasr define a language called LOGIN and describe a unification procedure that deals with inheritance. They do not differentiate between classes and objects, nor functional and multi-valued methods. They adopt the skeptical approach with regards to conflicts due to multiple inheritance. Ait-Kaci and Podelski went on to define a language called LIFE [9], which has more features than LOGIN. However, when the authors introduce the formal semantics of LIFE they describe only the semantics of LOGIN, on which they claim the semantics of LIFE can be defined. LOGIN is an example of a wider class of languages called feature-description languages [97].

Of the approaches that aim to formalize the notions, [19] does not address behavioral inheritance, and [19, 36, 66] and [80] do not address conflicts due to multiple inheritance or overriding.

There are also a number of groups whose aim was to build object-oriented databases or query languages. In [4], Abiteboul and Grumbach describe COL, a typed logic based language for manipulating complex objects. Besides base and derived relations, base and derived "data functions" are considered. Data functions can be functional or multi-valued.

In [6], Abiteboul and Kanellakis describe IQL, which is an extension of COL. The authors describe an object-based data model and introduce IQL, which they describe as a mathematical model of computation with types and a useful high-level query language. Most significantly, IQL treats inheritance as a specialization of union types.

In [1], Abiteboul has described another language based on the model that is used in IQL. The language is built around the COL and IQL languages and separates the schema and instance. The language has a deductive core with additional elaborate control structures. A fixpoint semantics is provided for the core language with dynamic overriding. Abiteboul then explains how the language and fixpoint procedure can be extended to include static overriding, and discusses the consistency of functional methods, proves that a compile-time check is undecidable, and concludes that a consistency check at runtime is suitable.

Laenens and Vermeir study ordered logic programming in [73], introducing a model and proof theory for their language. This approach stems from the area of artificial intelligence. In their work, the authors do not distinguish between objects, classes and attributes, or the subclass and instance relationships. The skeptical approach is adopted with regard to conflicts due to multiple inheritance.

Brass and Lipeck described a bottom-up query evaluation procedure [24]. In this paper they allow disjunctive information, which provides an alternative way of dealing with conflicts due to multiple inheritance. It is possible to say that one definition or the other is true. However the semantics of disjunctive languages are a lot more complicated.

In [22], Bertino and Montesi define a logical object-oriented language for databases based on LDL [87]. As in *L&O* [81], an object is a theory. The authors distinguish between classes and objects. Classes are used for object grouping and defining methods. There is little type information, and no separation between schema and data. The authors describe two kinds of inheritance: monotonic inheritance and inheritance with overriding. The semantics of overriding is static. Unlike most of the work in this area, the authors investigate updates.

Lou and Ozsoyoglu extend Datalog to include object-oriented features [79]. The resulting language is LLO. In LLO, the schema and data are separate. Methods are defined by clauses, and inheritance is achieved through type and unification mechanisms. The unification mechanism searches the class inheritance hierarchy bottom-up to find the method with "best" matching type. This method is the appropriate method with respect to overriding.

Finally, Abiteboul et al. [7] give a translation from a deductive object-oriented language to Datalog with negation. Although this provides an evaluation procedure for deductive object-oriented languages, it provides little insight into the semantics of such languages.

Of the approaches where the aim is to build a system, [4] does not address inheritance, [4] and [6] do not address overriding, and [1, 4, 6] and [22] do not address multiple inheritance.

2.4 Summary

In this chapter, we first describe concepts and current research in the area of deductive databases, and then we describe the key concepts of object-oriented databases. Deductive databases do not significantly improve the data modeling capabilities of relational databases but they do retain a firm mathematical foundation. Object-oriented databases have powerful data modeling capabilities but lack a mathematical foundation.

In order to reap the benefits from deductive and object-oriented databases, it seems natural to combine them. In Section 2.3, we reviewed related work which describes attempts to combine deduction or logic and object-orientation. These attempts fall naturally into the following three categories:

- combining object-orientation with logic to build a programming language,
- studying object-oriented concepts in a mathematical framework, with the aim of formalizing the notion underlying object-oriented systems,
- studying object-oriented concepts in a mathematical framework, with the aim of building object-oriented databases or query languages.

In this thesis, we follow the third approach and propose a new language, Gulog, that combines the features of deductive object-oriented databases in a uniform and simple manner. We provide both direct and indirect semantics for Gulog, we describe query evaluation procedures and prove they are sound and complete with respect to the semantics for a useful class of programs. The main features of Gulog include inheritance (and multiple inheritance), overriding, functional methods, multi-valued methods, objects, predicates, types, deduction and negation. We later extend Gulog and provide semantics and sound and complete query evaluation procedures with respect to the semantics for these languages that are extensions of Gulog.

Chapter 3

A Simple Deductive Object-Oriented Language

In this chapter, we describe the syntax and semantics of Gulog. Gulog can be considered as either a deductive object-oriented logic or as a deductive object-oriented database programming language. In the latter interpretation, each program in the language, which includes both facts and clauses, corresponds to a database. Similarly, goals in the language correspond to queries.

The syntax of Gulog is based on that of F-logic [65], which originated from Maier's O-logic [80], which in turn was strongly influenced by LOGIN [8]. In order to concentrate on the semantics of inheritance and overriding in a well understood context, schema declarations are separated from data definitions, as in the relational approach. Predicates (or relations) are part of the model. There are four reasons for having predicates in the language:

- the answer to a query can be represented as a predicate,
- the model is closed under query evaluation,
- it is not necessary to create a new object as an answer to a query, and
- predicates eliminate unnatural use of classes.

As in C++, we distinguish between objects and types (or classes). To maintain a general approach, method overriding can be defined on specific instances of a subtype. Our approach to defining the semantics of multi-valued methods is different from that of F-logic in [65], but both involve dynamic rather than static overriding.

3.1 Overview

In this section, we introduce Gulog using an example that models part of a system used to diagnose failures in a system of reactors [34]. In this system, sensors measure flow, temperature and pressure. If a sensor reading is "abnormal", a failure has occurred. The part of the system that we describe records failures in a group of reactors. Each reactor comprises a set of parts, and we want to record what kind of failure has occurred on a particular day for each reactor. In Gulog, we use the following declaration to specify the signature of type *reactor*:

$$reactor[parts \Rightarrow\!\!\!\!\Rightarrow part;\ failure@date \Rightarrow\!\!\!\!\Rightarrow sf]. \tag{3.1}$$

In this declaration, symbols *reactor*, *part*, *date*, and *sf* (system failure) denote types, while *parts* and *failure* denote multi-valued methods. The signature of method *parts* is *reactor*⇛*part*. The signature of method *failure*, which has one argument, is *reactor* × *date*⇛*sf*. A method can be applied to any object in the type on which it is declared, or in any subtype of that type. For example, the method *failure* can be applied to any object of type *reactor*, or in a subtype of *reactor*. Its argument must be of type *date*, or a subtype of *date*. The values of this method are of type sf, or a subtype of sf. The classification hierarchy for failures is given in Figure 3.1.

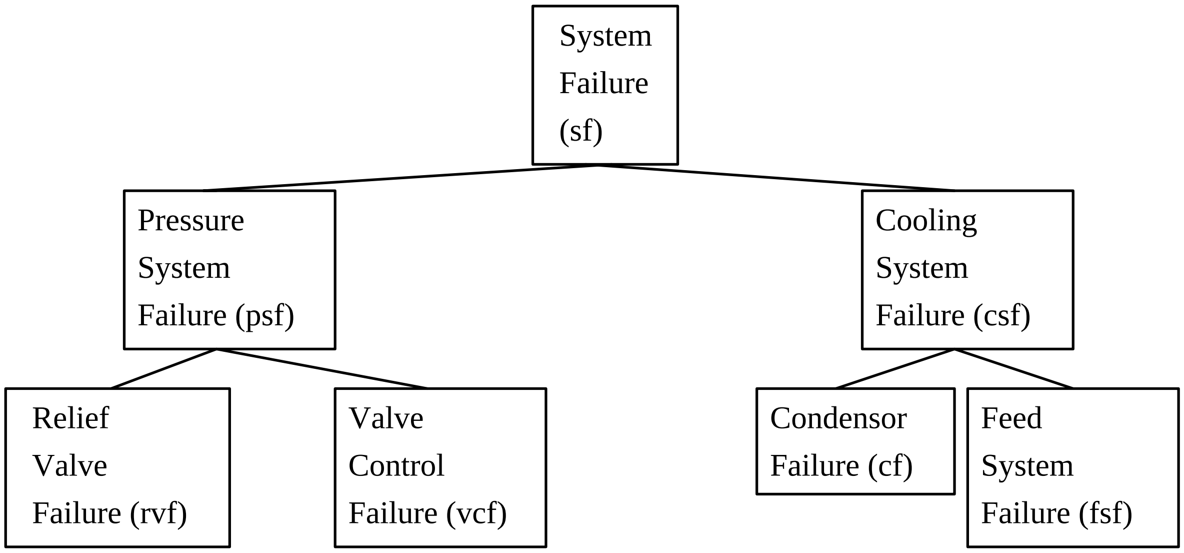
System Failure (sf)
Pressure System Failure (psf)
Cooling System Failure (csf)
Relief Valve Failure (rvf)
Valve Control Failure (vcf)
Condensor Failure (cf)
Feed System Failure (fsf)

Figure 3.1: Failure Classification Hierarchy

In Gulog, this hierarchy is written:

$$psf < sf \tag{3.2}$$
$$csf < sf \tag{3.3}$$
$$rvf < psf \tag{3.4}$$
$$vcf < psf \tag{3.5}$$
$$cf < csf \tag{3.6}$$
$$fsf < csf. \tag{3.7}$$

Declaration 3.2 states that type psf is a subtype of type sf (and sf is a supertype of psf). The type psf thus inherits methods from type sf.

In Gulog we can also represent multiple inheritance, where a type can have more than one parent. The type hierarchy then becomes a type lattice. The restriction that types must form a lattice is sometimes undesirable. However, the restriction simplifies the descriptions of procedures later in the thesis without limiting the expressive power of the language. To illustrate a type lattice, we include a type *total_failure* (tf), to which an object belongs if all measurements are abnormal. In Gulog, this type lattice is represented by declarations 3.2 to 3.7 along with the following declarations:

$$tf < rvf \tag{3.8}$$
$$tf < vcf \tag{3.9}$$
$$tf < cf \tag{3.10}$$
$$tf < fsf. \tag{3.11}$$

The relation $<$ is a non-reflexive partial order on the type symbols in the declarations.

An example of a declaration that indicates that an object is an instance of a type in Gulog is:

$$r1{:}\mathit{reactor}. \tag{3.12}$$

In this declaration, symbol *reactor* denotes a type and symbol $r1$ denotes an object. In this thesis we assume that object identifiers are system generated and we do not investigate the creation of object identifiers. We introduce a new type for each declaration – we explain why we do this below. In this case the new type is $r1$. We say object $r1$ is of type $r1$, or belongs to type $r1$ and is of type *reactor*. We say object o is of *type* τ if there is a declaration $o{:}\tau$, and object o is of *inherited type* τ if o is of type τ or o is of type σ and σ is a subtype of τ.

The measurements from the sensors for a reactor are presented to the system in a predicate *measurements*. The arguments of *measurements* are of type (or a subtype of) *reactor, pressure_sensor, temperature_sensor, relief_valve_sensor, valve_control_sensor, condenser_sensor, feed_system_sensor, sf.* The Gulog declaration is:

$$\begin{aligned}&measurements(reactor, pressure\_sensor, temperature\_sensor, relief\_valve\_sensor,\\&\quad valve\_control\_sensor, condenser\_sensor, feed\_system\_sensor, sf).\end{aligned} \tag{3.13}$$

The signature of predicate $measurements$ is $reactor \times pressure\_sensor \times temperature\_sensor \times relief\_valve\_sensor \times valve\_control\_sensor \times condenser\_sensor \times feed\_system\_sensor \times sf$.

In a set of declarations, there can be only one declaration for each object symbol, and if there are two declarations of a method then their types must be consistent with respect to the standard contravariance principle. Without this restriction type ambiguities would arise. Schema declarations are ground.

We have described the schema declarations. The data is defined by a set of clauses. A program or database (with respect to a set of declarations) is a finite set of clauses. Each object in a program has a single type throughout the program, whereas a variable may be declared to have a different type in each clause in which it occurs. In the following clause:

$$\{x{:}reactor, y{:}sf, d{:}date\} \vdash failure(x, d, y) \leftarrow x[failure@d \twoheadrightarrow y] \tag{3.14}$$

$\{x{:}reactor, y{:}sf, d{:}date\}$ is a variable typing. The symbol $\vdash$ separates the variable typing from the rest of the clause. The atom $\{x{:}reactor, y{:}sf, d{:}date\} \vdash x[failure@d \twoheadrightarrow y]$ is a method atom, while $\{x{:}reactor, y{:}sf, d{:}date\} \vdash failure(x, d, y)$ is a predicate atom. Clause 3.14 defines predicate $failure$ of arity 3. The syntax of predicate atoms and predicate type declarations appears ambiguous, but they can always be distinguished by the context in which they appear.

Every functional method atom has the form $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ where Γ is the variable typing of the variables in $t[m@t_1, \ldots, t_n \rightarrow t']$, $\Gamma \vdash t{:}c$, $\Gamma \vdash t_i{:}c_i$ (for $1 \leq i \leq n$), and $\Gamma \vdash t'{:}c'$ are terms (variables or object symbols) where t has type c, t_i has type c_i, and t' has type c', m is a method symbol, and the terms are consistent with the signature of functional method m of arity n. Similarly, every multi-valued method atom has the form $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$, where Γ is the variable typing of the variables in $t[m@t_1, \ldots, t_n \twoheadrightarrow t']$, $\Gamma \vdash t{:}c$, $\Gamma \vdash t_i{:}c_i$ (for $1 \leq i \leq n$), and $\Gamma \vdash t'{:}c'$ are terms (variables or object symbols) where t has type c, t_i has type c_i, and t' has type c', m is a method symbol, and the terms are consistent with the signature of multi-valued method m of arity n. Every predicate atom has the form $\Gamma \vdash p(t_1, \ldots, t_n)$, where p is a predicate symbol and each $\Gamma \vdash t_i{:}c_i$ is a term consistent with the signature of predicate p of arity n. Predicate atoms allow us to represent both base and derived relations in our language, and provide a convenient way of representing the answers to queries.

Recall declarations 3.2 to 3.7, and consider clause 3.15 that defines functional method *problem* when applied to any object of type sf:

$$\{y{:}sf\} \vdash y[problem \rightarrow \text{“system failure”}]. \tag{3.15}$$

By inheritance, the method *problem* also returns the value “system failure” when applied to any object belonging to a subtype of sf unless the method *problem* has been overridden for that object. Clause 3.16 defines functional method *problem* to return a value “temperature high” for all objects of type csf provided the method *temp* when applied to that object has value *high*:

$$\{y{:}csf\} \vdash y[problem \rightarrow \text{“temperature high”}] \leftarrow y[temp \rightarrow high]. \tag{3.16}$$

Clause 3.16 thus overrides clause 3.15 for objects of type csf whose method *temp* has value *high*. However, if method *temp* of some object of type csf were *low* (not *high*), then the object would inherit the value for the method *problem*, with value “system failure” from type sf.

As described earlier, if there was an object declaration $failure{:}csf$, a new type *failure* is introduced. This is used to maintain a simple and uniform definition of overriding when methods are defined directly on instances. Clause 3.17 defines functional method *problem* to return a value “thermometer broken” for $failure$:

$$failure[problem \rightarrow \text{“thermometer broken”}]. \tag{3.17}$$

Clause 3.17 thus overrides clause 3.16 for object $failure$.

Negated atoms are allowed in the body of a clause. Clause 3.18 is used to validate the data produced. A temperature sensor is recognized as malfunctioning if there is a condenser failure and the temperature is normal.

$$\{y{:}cf\} \vdash y[malfunction \rightarrow temp\_sensor] \leftarrow \neg y[temp \rightarrow high] \wedge \neg y[temp \rightarrow low] \tag{3.18}$$

The method *malfunction* has the value *temp_sensor* when applied to an object of type cf, if the method *temp* when applied to the same object does not return the value *high* and does not return the value *low*. A literal is an atom or a negated atom. We call a negated atom a negative literal.

With multiple inheritance, the signature of a method cannot be inherited from more than one parent because of the constraint that there can be only one type definition for each n-ary method symbol in a set of declarations. However, the value of a method may be inherited from more than one parent. This can cause ambiguities. An ambiguity arises if a method is defined in more than one path to a common type in the type lattice, and neither of the types on which the method is defined is more specific than the other. For example, consider the type lattice described in declarations 3.2 to 3.11. Class tf is more specific than types rvf, vcf, cf, and fsf, which in turn are more specific than types csf and psf, which are more specific than sf. If method *problem* is defined in types cf and fsf only, then method *problem* is defined in more than one path and neither cf nor fsf is more specific than the other so there is an ambiguity. However, if method *problem* is defined in types sf and cf only, then method *problem* is defined in more than one path and redefined in a more specific type so there is no ambiguity. Currently, to simplify our presentation, we prohibit programs with ambiguities due to multiple inheritance. We discuss alternative ways of dealing with multiple inheritance in Section 6.2.

A goal or query is a clause with an empty head:

$$\{x{:}reactor, y{:}sf\} \vdash \; \leftarrow x[failure@17May \twoheadrightarrow y]. \tag{3.19}$$

This query asks what *system failures* occured on $17May$, and on what *reactor* they occurred.

Gulog contains the essential features of objects, types, functional methods, multi-valued methods, inheritance, overriding, multiple inheritance, negation and deduction. When atoms are restricted to predicate atoms, the language becomes a form of typed Datalog.

3.2 Syntax

In this section we describe the syntax of declarations, and of programs with respect to a set of declarations.

The letters τ and σ denote type symbols, a denotes an object symbol, p denotes a predicate symbol, x, y denote variable symbols, and s, t, u terms. Subscripts and superscripts are also used.

The subtype relationship and types of objects, predicates and methods are expressed using type declarations.

Definition A *declaration* is defined as follows:

- If τ and τ' are type symbols then $\tau < \tau'$ is a *type lattice* declaration. We say τ is a subtype of τ', τ' is a supertype of τ, and $\tau <_t \tau'$. For any type τ'' such that $\tau' <_t \tau''$, τ is also a subtype of τ'', τ'' is a supertype of τ, and $\tau <_t \tau''$. If $\tau <_t \tau''$ or $\tau = \tau''$ we write $\tau \leq_t \tau''$.

- If a is an object symbol and τ is a type symbol then $a : \tau$ is an *object declaration*. We say a is an object of inherited type τ. For any type τ' such that τ' is a supertype of τ, a is also of inherited type τ'.

- If p is an n-ary predicate symbol and $\tau_1, \ldots, \tau_n$ are type symbols then $p(\tau_1, \ldots, \tau_n)$ is a *predicate declaration*. We say the signature of predicate p of arity n is $\tau_1 \times \cdots \times \tau_n$.

- If m is an n-ary functional method symbol and $\tau, \tau_1, \ldots, \tau_n, \tau'$ are type symbols then $\tau[m@\tau_1, \ldots, \tau_n \Rightarrow \tau']$ is a *functional method declaration*. We say the signature of functional method m of arity n is $\tau \times \tau_1 \times \cdots \times \tau_n \Rightarrow \tau'$. If m is a 0-ary functional method symbol, we write $\tau[m \Rightarrow \tau']$.

- If m is an n-ary multi-valued method symbol and $\tau, \tau_1, \ldots, \tau_n, \tau'$ are type symbols then $\tau[m@\tau_1, \ldots, \tau_n \Rrightarrow \tau']$ is a *multi-valued method declaration*. We say the signature of multi-valued method m of arity n is $\tau \times \tau_1 \times \cdots \times \tau_n \Rrightarrow \tau'$. If m is a 0-ary multi-valued method symbol, we write $\tau[m \Rrightarrow \tau']$.

For each object declaration, we introduce another type to the type lattice. As in [8], the maximal element in the type lattice is $\top$ and the minimal element is $\bot$. Let $a : \tau$ be an object declaration, we introduce type a, such that $a <_t \tau$ and object a is of type a, and inherited type τ. In a set of declarations, the following conditions must hold:

- there can be only one declaration for each object symbol,

- for any two functional method declarations on method m with signatures $\sigma \times \sigma_1 \times \cdots \times \sigma_{n-1} \Rightarrow \sigma_n$ and $\tau \times \tau_1 \times \cdots \times \tau_{n-1} \Rightarrow \tau_n$ where $\sigma < \tau$, $\sigma_i \geq \tau_i$ for all $1 \leq i < n$, and $\sigma_n \leq \tau_n$,

- for any two multi-valued method declarations on method m with signatures $\sigma \times \sigma_1 \times \cdots \times \sigma_{n-1} \Rrightarrow \sigma_n$ and $\tau \times \tau_1 \times \cdots \times \tau_{n-1} \Rrightarrow \tau_n$ where $\sigma < \tau$, $\sigma_i \geq \tau_i$ for all $1 \leq i < n$, and $\sigma_n \leq \tau_n$.

The relation $<_t$ is a non-reflexive partial order on the type symbols.

Example 3.2.1 Let D be a set of declarations that includes the following:

- $r1$:*reactor* declares that there is an object $r1$ that is of inherited type *reactor*.
- $psf < sf$ declares that psf (pressure system failure) is a subtype of sf (system failure).
- *reactor*[*parts*$\Rrightarrow$*part*] declares a 0-ary multi-valued method *parts*, which when applied to objects of type *reactor* returns values of type *part*. The signature of 0-ary multi-valued method *parts* is *reactor*$\Rrightarrow$*part*.
- *measurements(reactor, pressure_sensor, temperature_sensor, relief_valve_sensor, valve_control_sensor, condenser_sensor, feed_system_sensor, sf)* declares an 8-ary predicate *measurements*, with signature $reactor \times pressure\_sensor \times temperature\_sensor \times relief\_valve\_sensor \times valve\_control\_sensor \times condenser\_sensor \times feed\_system\_sensor \times sf$. $\square$

Example 3.2.2 Consider the set of declarations $D = \{tf < rvf, tf < vcf, rvf[problem \Rightarrow string], vcf[problem \Rightarrow string]\}$. These declarations obey the constraints described above. Thus, the type definitions do not conflict. $\square$

We now define data definitions. Data is defined by a program with respect to a set of declarations. As described above, the declarations define a set of types arranged in a lattice, a set of typed object symbols, a set of functional method symbols each with a given signature, a set of multi-valued method symbols each with a given signature, and a set of predicate symbols each with a given signature. It is possible for the same symbol to be used for both functional and multi-valued methods and predicates, or for a method or predicate of arity n and a method or predicate of arity m where $n \neq m$.

So that we can construct non-ground terms and clauses, we need to assign types to the variables.

Definition If x is a variable and τ is a type then $x{:}\tau$ is a *typed variable* defining variable x to be of type τ. A *variable typing* is a set of typed variables where each variable is distinct, and will be denoted by Γ.

We say an object or variable s is of type τ if τ is its type, and s is of inherited type τ if s is of type σ and $\sigma <_t \tau$.

The definitions of terms, atoms, and clauses are similar to those in logic programming except that in Gulog we include type information. The following definitions are understood with respect to a set of declarations.

Definition A *term* is defined as follows:

- $\Gamma \vdash x : \tau$ is a term if x is a variable and $x : \tau \in \Gamma$. We usually write $\Gamma \vdash x$ when the type of x is obvious. The variable x can be assigned an object a if and only if a is of inherited type τ.
- $\Gamma \vdash a : \tau$ is a term if a is an object of type τ. We write the term simply as a when the type of a is obvious.

The term $\Gamma \vdash t : \tau$ is called a *ground term* if t is not a variable.

Example 3.2.3 Let x be a variable then $\{x{:}reactor\} \vdash x{:}reactor$ is a term. The variable typing $\{x{:}reactor\}$ is separated from $x{:}reactor$ by the $\vdash$ symbol.

Assume the declaration $failure{:}psf$. Then $failure$ is a term. $\Box$

Definition An *atom* is defined as follows:

- If p is an n-ary predicate symbol with signature $\tau_1 \times \cdots \times \tau_n$ and there exists a variable typing Γ that contains type assignments for all the variables in $t_1, \ldots, t_n$, and $\Gamma \vdash t_1 : \sigma_1, \ldots, \Gamma \vdash t_n : \sigma_n$ are terms with $\sigma_1 \leq_t \tau_1, \ldots, \sigma_n \leq_t \tau_n$, then $\Gamma \vdash p(t_1, \ldots, t_n)$ is a *predicate atom.*

- If m is an n-ary functional method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \Rightarrow \tau'$ and there exists a variable typing Γ that contains type assignments for all the variables in $t, t_1, \ldots, t_n$ and t', and $\Gamma \vdash t : \sigma, \Gamma \vdash t_1 : \sigma_1, \ldots, \Gamma \vdash t_n : \sigma_n$, and $\Gamma \vdash t' : \sigma'$ are terms with $\sigma \leq_t \tau$, $\sigma_1 \leq_t \tau_1, \ldots, \sigma_n \leq_t \tau_n$ and $\sigma' \leq_t \tau'$, then $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ is a *functional method atom.*

- If m is an n-ary multi-valued method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \Rrightarrow \tau'$ and there exists a variable typing Γ that contains type assignments for all the variables in $t, t_1, \ldots, t_n$ and t', and $\Gamma \vdash t : \sigma, \Gamma \vdash t_1 : \sigma_1, \ldots, \Gamma \vdash t_n : \sigma_n$, and $\Gamma \vdash t' : \sigma'$ are terms with $\sigma \leq_t \tau$, $\sigma_1 \leq_t \tau_1, \ldots, \sigma_n \leq_t \tau_n$ and $\sigma' \leq_t \tau'$, then $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is a *multi-valued method atom.*

An atom is a *method atom* if it is a functional method atom or a multi-valued method atom. The atom $\Gamma \vdash A$ is a *ground atom* if A contains no variables. Let $\rightsquigarrow$ denote $\rightarrow$ or $\twoheadrightarrow$, and M be a 0-ary method atom $\Gamma \vdash t[m@ \rightsquigarrow t']$. The atom M can be written $\Gamma \vdash t[m \rightsquigarrow t']$.

Example 3.2.4 Assume the appropriate declarations including $failure{:}psf$ and $psf < sf$. Then $\{x{:}sf\} \vdash x[problem \rightarrow$ "system failure"], $failure[problem \rightarrow$ "system failure"], $\{x{:}reactor, y{:}sf, d{:}date\} \vdash x[failure@d \twoheadrightarrow y]$ are atoms. $\Box$

Definition Let $\Gamma \vdash A$ be an atom. Then $\Gamma \vdash A$ and $\Gamma \vdash \neg A$ are *literals.* The atom $\Gamma \vdash A$ is a *positive literal*, and $\Gamma \vdash \neg A$ is a *negative literal.*

Clauses are used to define methods and predicates. Each clause also has type information.

Definition If $\Gamma \vdash A, \Gamma_1 \vdash B_1, \ldots, \Gamma_n \vdash B_n$ are atoms where the variables common to $\Gamma, \Gamma_1, \ldots, \Gamma_n$ have the same types then $\Gamma \cup \Gamma_1 \cup \cdots \cup \Gamma_n \vdash (A \leftarrow B_1 \wedge \cdots \wedge B_n)$ is a *clause.* If $\Gamma \cup \Gamma_1 \cup \cdots \cup \Gamma_n$ is empty we write $A \leftarrow B_1 \wedge \cdots \wedge B_n$. A is the *head* of the clause, and $B_1 \wedge \cdots \wedge B_n$ is the *body*.

Let $\Gamma \vdash A$ be $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$), where $\Gamma \vdash t{:}\sigma$ is a term. We say method m is defined on type σ.

We can write the following compound atoms:

- $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t_1; \ldots; m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]$, as a shorthand for $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t_1] \wedge \cdots \wedge t[m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]$.

- $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t'[m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]]$, as a shorthand for $\Gamma \vdash t[m_1@t_{1,1}, \ldots, t_{1,n} \rightarrow t'] \wedge t'[m_j@t_{j,1}, \ldots, t_{j,k} \rightarrow t_j]$.

- $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow \{t'_1, \ldots, t'_p\}]$, as a shorthand for $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t'_1] \wedge \cdots \wedge t[m@t_1, \ldots, t_n \twoheadrightarrow t'_p]$.

If $a_1, \ldots, a_p$ are object symbols we can write $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow \{a_1, \ldots, a_p\}] \leftarrow B_1 \wedge \cdots \wedge B_m$ as a shorthand for the p clauses:

$$\begin{array}{ll} \Gamma & \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow a_1] \leftarrow B_1 \wedge \cdots \wedge B_m \\ & \vdots \\ \Gamma & \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow a_p] \leftarrow B_1 \wedge \cdots \wedge B_m. \end{array}$$

Let $\Gamma \vdash A$ be $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$) where $\Gamma \vdash t{:}\sigma$ is a term. We say $\Gamma \vdash A$ is of type σ. The type of a predicate atom is undefined. If the head of a clause is an atom of type σ, then the clause is of type σ.

Definition A *goal* is a clause of the form $\Gamma \vdash \;\leftarrow B_1 \wedge \cdots \wedge B_n$, that is, a clause with an empty head. If Γ is empty we write $\leftarrow B_1 \wedge \cdots \wedge B_n$.

The type of a goal is undefined.

Definition A *program* with respect to a set of declarations is a finite set of clauses. Let P be a program with respect to a set of declarations D. We write P when the set of declarations under consideration is clear.

Example 3.2.5 Assume appropriate declarations D including $psf < sf$ and $failure{:}psf$. Consider the program P' with respect to the set of declarations D:

$\{x{:}sf\} \vdash x[action \twoheadrightarrow$ "call supervisor"$]$
$\{x{:}psf\} \vdash x[action \twoheadrightarrow \{$"lower pressure", "call supervisor"$\}] \leftarrow x[pressure \rightarrow high]$
$\{x{:}psf\} \vdash x[action \twoheadrightarrow \{$"increase pressure", "call supervisor"$\}] \leftarrow$
$\quad x[pressure \rightarrow low]$.

Program P' is a shorthand for P:

$\{x{:}sf\} \vdash x[action \twoheadrightarrow$ "call supervisor"$]$
$\{x{:}psf\} \vdash x[action \twoheadrightarrow$ "lower pressure"$] \leftarrow x[pressure \rightarrow high]$
$\{x{:}psf\} \vdash x[action \twoheadrightarrow$ "call supervisor"$] \leftarrow x[pressure \rightarrow high]$
$\{x{:}psf\} \vdash x[action \twoheadrightarrow$ "increase pressure"$] \leftarrow x[pressure \rightarrow low]$
$\{x{:}psf\} \vdash x[action \twoheadrightarrow$ "call supervisor"$] \leftarrow x[pressure \rightarrow low]$. □

3.3 Substitutions

Bindings or substitutions are computed in the unification procedure. We now define substitution, an instance of an expression, and how to combine two substitutions. These concepts are similar to the corresponding concepts in logic programming, except that they are typed and hence must satisfy additional restrictions.

We first define greatest lowest bound. Recall from Section 3.2 that types in a set of declarations form a type lattice.

Definition Let S be the set of types in a set of declarations D. Type $\sigma \in S$ is a *lower bound* of $X \subseteq S$ with respect to D if $\sigma \leq_t \tau$ for all $x \in X$, and is the *greatest lower bound* of X with respect to D, denoted $glb(X)$, if for all lower bounds σ' of X, $\sigma' \leq_t \sigma$.

Definition Let D be a set of declarations. Let $\Gamma \vdash \{x_1/t_1, \ldots, x_n/t_n\}$ be a pair where $x_1, \ldots, x_n$ are distinct variables, $\Gamma \vdash t_1, \ldots, \Gamma \vdash t_n$ are terms, and no t_i equals x_i.

$\Gamma \vdash \{x_1/t_1, \ldots, x_n/t_n\}$ is a *substitution* if for each i, for some τ_i and σ_i, $\Gamma \vdash x_i : \tau_i$ and $\Gamma \vdash t_i : \sigma_i$ are terms, and $\sigma_i \leq_t \tau_i$ with respect to D.

The empty substitution is denoted by ε.

Definition An *expression* $\Gamma \vdash T$ is a term, a literal, a conjunction of literals or a clause. A *simple expression* is a term, a predicate atom, or a method atom.

We now describe how substitutions are applied to expressions.

Definition Let D be a set of declarations. Let $\theta = (\Gamma_\theta \vdash \{x_1/t_1, \ldots, x_k/t_k\})$ be a substitution and $E = (\Gamma_E \vdash T)$ an expression. For any i, such that $1 \leq i \leq k$, let $\Gamma_\theta \vdash x_i : \tau_i$ and $\Gamma_\theta \vdash t_i : \sigma_i$ be terms and $x_i : \tau'_i$ be a typed variable in E (with respect to Γ_E). We say $dom(\theta) = \{x_1, \ldots, x_k\}$, $range(\theta) = \{t_1, \ldots, t_k\}$ and $vars(T)$ is the set of all variables in T.

We construct the instance $E\theta$ of E as follows:

- Construct T' from T by simultaneously replacing each occurrence of a variable x_i in T by t_i.
- Construct Γ', a variable typing for the variables in T', as follows. For each variable x in T', if x is identical to a t_i, then the typing of x in Γ' is $x{:}glb(\sigma_i, \tau'_i)$ with respect to D; otherwise the typing of x in Γ' is just that in Γ_E.
- If for each variable x_i in $vars(T) \cap dom(\theta)$, t_i is a variable and $glb(\sigma_i, \tau'_i)$ with respect to D is not $\perp$, or t_i is an object and $\sigma_i \leq_t \tau'_i$ with respect to D then the *instance* $E\theta$ of E is $\Gamma' \vdash T'$; otherwise, $E\theta$ is undefined.

If $E\theta$ is ground, then $E\theta$ is a *ground instance* of E.

Example 3.3.1 Assume D is the set of declarations $failure{:}sf$, $tf < fsf$, $fsf < csf$, $cf < csf$, $tf < cf$, $csf < sf$. Let E be $\{x{:}fsf, y{:}string\} \vdash x[problem \rightarrow y]$ and substitution θ be $\{x{:}sf, x'{:}cf\} \vdash \{x/x'\}$ with respect to declarations D, then the instance $E\theta$ is $\{x'{:}tf, y{:}string\} \vdash x'[problem \rightarrow y]$.

Let θ_1 be $\{x{:}sf\} \vdash \{x/failure\}$. Instance $E\theta_1$ is undefined, because $sf \not\leq fsf$. □

Let $S = \{E_1, \ldots, E_n\}$ be a finite set of expressions and θ a substitution. Then $S\theta$ denotes the set $\{E_1\theta, \ldots, E_n\theta\}$. We now describe how substitutions are composed.

Definition Let D be a set of declarations and θ and σ be the substitutions $\Gamma_\theta \vdash \theta'$ and $\Gamma_\sigma \vdash \sigma'$ where θ' is $\{x_1/t_1, \ldots, x_n/t_n\}$, σ' is $\{y_1/s_1, \ldots, y_m/s_m\}$.

We construct the composition $\theta\sigma$ of θ and σ as follows:

- Construct γ from $\{x_1/t_1\sigma, \ldots, x_n/t_n\sigma, y_1/s_1, \ldots, y_m/s_m\}$ (with respect to Γ_θ and Γ_σ) by deleting any binding $x_i/t_i\sigma$ for which $x_i = t_i\sigma$, and deleting any binding y_j/s_j for which $y_j \in \{x_1, \ldots, x_n\}$.
- Construct Γ, the variable typing for the variables in γ as follows. For each variable x in γ, if $x{:}\tau \in \Gamma_\theta$ and $x{:}\tau' \in \Gamma_\sigma$ then include $x{:}glb(\tau, \tau')$ with respect to D in Γ, otherwise if $x{:}\tau \in \Gamma_\theta$ and there is no type specified for x in Γ_σ, or $x{:}\tau \in \Gamma_\sigma$ and there is no type specified for x in Γ_θ then include $x{:}\tau$ in Γ.
- If for any variable x, where $x{:}\tau \in \Gamma_\theta$ and $x{:}\tau' \in \Gamma_\sigma$, and $glb(\tau, \tau')$ with respect to D is $\perp$, then $\theta\sigma = \emptyset$, else $\theta\sigma = \Gamma \vdash \gamma$.

Example 3.3.2 Assume appropriate declarations including $rvf < psf$, $psf < sf$, *failure : rvf*, and $f2 : rvf$. Let substitutions $\theta = \{x_1{:}sf, x_2{:}psf, x_3{:}rvf, y_2{:}sf, y_3{:}rvf\} \vdash \{x_1/y_2, x_2/y_3, x_3/failure\}$ and $\sigma = \{x_1{:}psf, y_2{:}sf, y_3{:}rvf\} \vdash \{x_1/failure, y_2/x_1, y_3/f2\}$. Then the composition of θ and σ is $\theta\sigma = \{x_1{:}psf, x_2{:}psf, x_3{:}rvf, y_2{:}sf, y_3{:}rvf\} \vdash \{x_2/f2, x_3/failure, y_2/x_1, y_3/f2\}$.

Assume appropriate declarations including $psf < sf$ and $csf < sf$. Let $\theta = \{x_1{:}sf, x_2{:}psf\} \vdash \{x_1/x_2\}$ and $\sigma = \{x_2{:}csf, x_3{:}csf\} \vdash \{x_2/x_3\}$. Then $\theta\sigma = \emptyset$, because $\Gamma_\theta \vdash x_2{:}psf$ and $\Gamma_\sigma \vdash x_2{:}csf$, and $glb(psf, csf)$ is $\perp$. □

The substitution $\Gamma \vdash \theta$ is a *ground substitution*, with respect to a set of declarations D, if $range(\theta)$ contains no variables. The substitution $\Gamma' \vdash \theta'$ is a ground instance of a substitution $\Gamma \vdash \theta$ with respect to a set of declarations D if there is a ground substitution $\Gamma_\sigma \vdash \sigma$ with respect to D such that $\Gamma' \vdash \theta' = (\Gamma \vdash \theta)(\Gamma_\sigma \vdash \sigma)$ and $\Gamma' \vdash \theta'$ is a ground substitution with respect to D. The set of all ground instances of $\Gamma \vdash \theta$ with respect to D is written $[\Gamma \vdash \theta]_D$.

Example 3.3.3 Assume a set of declarations D including $psf < sf$ and $failure{:}psf$. Let $\theta = \{x_1{:}sf, x_2{:}psf\} \vdash \{x_1/x_2\}$. Then $[\theta]_D$, the set of all ground instances of θ with respect to D, contains only one element, and it is $\{x_1{:}sf, x_2{:}psf\} \vdash \{x_1/failure, x_2/failure\}$. □

Definition Let P be a program with respect to a set of declarations D. Then $[P]_D$ denotes the set of all ground instances of clauses of P with respect to D. We write $[P]$ when the set of declarations under consideration is clear.

3.4 Simple programs

To ensure that programs have well-defined semantics, we impose restrictions that ensure they are "stratified" and that functional methods really are single-valued. Programs that satisfy these restrictions are called "simple" programs. Before defining the classes of stratified and simple programs, we must define a relationship "possibly overrides" between ground instances of clauses.

Informally a ground clause C defining a method m possibly overrides a ground clause C' defining the same method if C is defined on a more specific type, and produces different values when applied to the same object and arguments.

Definition Let D be a set of declarations, P be a program with respect to D, C be $\Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t'] \leftarrow B$, (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_k \twoheadrightarrow t'] \leftarrow B$) and C' be $\Gamma' \vdash s[m@s_1, \ldots, s_k \rightarrow s'] \leftarrow B'$ (respectively, $\Gamma' \vdash s[m@s_1, \ldots, s_k \twoheadrightarrow s'] \leftarrow B'$), where $C, C' \in P$, $\Gamma \vdash t : \sigma$ is a term, $\Gamma' \vdash s : \tau$ is a term and $\sigma <_t \tau$. If there are ground instances $C\theta$ of C and $C'\theta'$ of C' such that $t\theta = s\theta'$, $t_i\theta = s_i\theta'$ for all i, $1 \le i \le k$, and $t'\theta \neq s'\theta'$ then clause $C\theta$ *possibly overrides* clause $C'\theta'$ with respect to P and D.

Example 3.4.1 Assume the set of declarations D including $failure{:}csf$ and $csf < sf$. Consider the following program P with respect to declarations D.

$\{x{:}sf\} \vdash x[problem \rightarrow$ "system failure"$]$
$\{x{:}csf\} \vdash x[problem \rightarrow$ "cooling system failure"$] \leftarrow x[problem \rightarrow$ "system failure"$]$

Then $[P]_D$ is:

$$failure[problem \rightarrow \text{"system failure"}] \tag{3.20}$$

$$failure[problem \rightarrow \text{"cooling system failure"}] \leftarrow failure[problem \rightarrow \text{"system failure"}] \tag{3.21}$$

Based on the above declaration, we say clause 3.21 possibly overrides clause 3.20. □

A program is inheritance-stratified if there are no cycles in the definition of a method. That is, the definition of a method does not depend on an inherited definition of the same method.

Definition A program P with respect to a set of declarations D is *inheritance-stratified* (or *i-stratified*) if there exists a mapping μ from the set of ground atoms to the set of non-negative integers such that, for every literal in every ground instance $C\theta$ of every clause C in P,

1. $\mu(A') \leq \mu(A)$, where A' is a positive literal in the body of the clause instance $C\theta$, and A is the head of the clause instance $C\theta$,

2. $\mu(A') < \mu(A)$, where $\neg A'$ is a negative literal in the body of the clause instance $C\theta$, and A is the head of the clause instance $C\theta$,

3. $\mu(B') < \mu(A)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where B' is a positive literal or $\neg B'$ is a negative literal in the body of $C'\theta'$, and A is the head of the clause instance $C\theta$, and

4. $\mu(A') \leq \mu(A)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where A' is the head of $C'\theta'$, and A is the head of $C\theta$.

The integer mapped to a ground atom is the strata or level of that atom.

Example 3.4.2 Consider the declarations D and the program P with respect to declarations D in Example 3.4.1. Method *problem* is a functional method. It has either the value "system failure" or "cooling system failure". If it has value "system failure" then it should also have value "cooling system failure" (due to deduction). If it doesn't have value "cooling system failure" then it should have value "system failure" (due to inheritance). This is contradictory. Thus we conclude that program P has no natural meaning. The i-stratification of $[P]_D$ requires:

$\mu(failure[problem \rightarrow \text{“system failure”}]) \leq \mu(failure[problem \rightarrow \text{“cooling system failure”}])$
from 1 in the definition of inheritance-stratified,
$\mu(failure[problem \rightarrow \text{“cooling system failure”}]) \leq \mu(failure[problem \rightarrow \text{“system failure”}])$
from 4, and
$\mu(failure[problem \rightarrow \text{“system failure”}]) < \mu(failure[problem \rightarrow \text{“system failure”}])$
from 3,
which is unsatisfiable. Hence P is not i-stratified. □

Note that i-stratification is defined in terms of ground atoms and clauses and is thus similar to local stratification [90]. However, in the definition of i-stratification, it may not be obvious to the reader why $\mu(A') \leq \mu(A)$ is required rather than $\mu(A') < \mu(A)$, where A' is the head of a ground clause that possibly overrides a ground clause whose head is A. We motivate this requirement in the following example.

Example 3.4.3 Assume the set of declarations D including $cf < csf$, $csf < sf$ and $failure{:}cf$. Consider the program P with respect to D:

$\{x{:}sf\} \vdash x[severity \twoheadrightarrow high]$
$\{x{:}csf\} \vdash x[severity \twoheadrightarrow medium]$
$\{x{:}cf\} \vdash x[severity \twoheadrightarrow high]$.

There are no cycles in the definition of method *severity* in P with respect to D. The i-stratification of P with respect to D requires:

$\mu(failure[severity \twoheadrightarrow medium]) \leq \mu(failure[severity \twoheadrightarrow high])$
$\mu(failure[severity \twoheadrightarrow high]) \leq \mu(failure[severity \twoheadrightarrow medium])$.

As expected, P with respect to D is i-stratified. If the definition of i-stratification required $\mu(A') < \mu(A)$, where A' is the head of a ground clause that possibly overrides a ground clause whose head is A, then this program, which has a natural meaning would not be i-stratified. □

We also talk about the level or stratum of a goal. The *level* of a goal G is the larger of:

- the maximum level of the ground predicates and methods of the positive literals in G, and
- $1 +$ the maximum level of the ground predicates and methods of the negative literals in G.

For simplicity, we disallow programs with ambiguities that arise due to multiple inheritance. In Section 6.2, we discuss alternative ways of dealing with multiple inheritance that are less restrictive but more complicated.

We first describe what an unambiguous program is.

Definition Let P be a program with respect to a set of declarations D. For each type σ in D, let $R(\sigma)$ be $\{< m_{\rightarrow}, n, \tau >|$ functional method $m_{\rightarrow}$ of arity n is defined on type τ, and $\sigma \leq_t \tau\} \cup \{< m_{\twoheadrightarrow}, n, \tau >|$ multi-valued method $m_{\twoheadrightarrow}$ of arity n is defined on type τ, and $\sigma \leq_t \tau\}$. Program P is *unambiguous* with respect to multiple inheritance if and only if, for every type σ in D, if $< m_{\rightarrow}, n, \tau_1 > \in R(\sigma)$ (respectively, $< m_{\twoheadrightarrow}, n, \tau_1 > \in R(\sigma)$) and $< m_{\rightarrow}, n, \tau_2 > \in R(\sigma)$ (respectively, $< m_{\twoheadrightarrow}, n, \tau_2 > \in R(\sigma)$), where $\tau_1 \neq \tau_2$, then either $\tau_1 <_t \tau_2$ or $\tau_2 <_t \tau_1$.

A program is *ambiguous* if it is not unambiguous.

Example 3.4.4 Assume appropriate declarations D including $tf < cf$, $cf < csf$, $tf < fsf$, and $fsf < csf$. Consider the following program P with respect to D:

$\{x{:}cf\} \vdash x[problem \rightarrow$ "condenser failure"]
$\{x{:}fsf\} \vdash x[problem \rightarrow$ "feed system failure"].

An object of type *tf* could inherit the value "condenser failure" for method *problem* from *cf* or inherit the value "feed system failure" for method *problem* from *fsf* or inherit both values.

Based on the above definition, $R(tf) = \{< problem, 0, cf >, < problem, 0, fsf >\}$. In the type lattice $cf \not<_t fsf$ and $fsf \not<_t cf$. As expected this program is ambiguous.

Assuming declarations D from above, consider the program P' with respect to D:

$\{x{:}csf\} \vdash x[problem \rightarrow$ "cooling system failure"]
$\{x{:}fsf\} \vdash x[problem \rightarrow$ "check history"]

Based on the above definition, $R(csf) = \{< problem, 0, csf >\}$, $R(fsf) = \{< problem, 0, csf >, < problem, 0, fsf >\}$, $R(cf) = \{< problem, 0, csf >\}$, and $R(tf) = \{< problem, 0, csf >, < problem, 0, fsf >\}$. It is not obvious why this program is unambiguous. With respect to type tf, the definition of method *problem* on type fsf is more specific than the definition on csf. So, this program has a natural meaning. □

In [5], Abiteboul and Hull showed that compile-time consistency checking to ensure that functional methods are single-valued is undecidable. To ensure that functional methods really are single-valued, we impose the following condition on programs.

Definition A variable z is *restricted* with respect to the terms t, $t_1, \ldots, t_n$ in the conjunction C if C contains a method atom $t'[m'@t'_1, \ldots, t'_k \rightarrow z]$ and every variable in t', $t'_1, \ldots, t'_k$ either occurs in t, $t_1, \ldots, t_n$ or is itself restricted with respect to t, $t_1, \ldots, t_n$ in the remainder of C.

For example, the variable z is restricted with respect to a and x in the conjunction $a[m'@x \rightarrow y] \wedge a[m''@y \rightarrow z]$.

Definition A program P with respect to a set of declarations D is *well-defined* if the following three conditions hold:

- For each clause $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow u] \leftarrow C$ in P with respect to D, u is either an object symbol or is a variable that is restricted with respect to t, $t_1, \ldots, t_n$ in C. This is the usual "safety" condition.

- For each functional method m of signature $\tau \times \tau_1 \times \cdots \times \tau_n \Rightarrow \tau'$, and for each type $\sigma \leq_t \tau$, program P with respect to D does *not* contain two clauses

  $$\begin{array}{l} \Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow u] \leftarrow C \\ \Gamma' \vdash t'[m@t'_1, \ldots, t'_n \rightarrow u'] \leftarrow C' \end{array}$$

  such that $\Gamma \vdash t{:}\sigma$, $\Gamma' \vdash t'{:}\sigma$ are terms, and atoms $t[m@t_1, \ldots, t_n \rightarrow x]$ and $t'[m@t'_1, \ldots, t'_n \rightarrow x']$ (where x and x' are new variables) are unifiable. This condition ensures that, for objects a, $a_1, \ldots, a_n$ and functional method atom $a[m@a_1, \ldots, a_n \rightarrow t]$, there is at most one clause of each type whose head is unifiable with the atom. This, together with the first condition, ensures that functional methods really are single-valued.

- P with respect to D is unambiguous with respect to multiple inheritance.

The following example illustrates how the conditions are applied to exclude unreasonable programs.

Example 3.4.5 Assume appropriate declarations D. Consider program P with respect to D:

$$\begin{array}{l} \{x{:}csf, y{:}string\} \vdash x[problem \rightarrow y] \leftarrow temp\_problem(x, y) \\ \{x{:}csf\} \vdash x[problem \rightarrow \text{"cooling system failure"}] \leftarrow x[temp \rightarrow high]. \end{array}$$

In this program, an object of type *csf* may have more than one value for the functional method *problem*. Thus the 0-ary functional method *problem* is not single-valued. More formally, based on the definition of "well-defined" above, this program is not well-defined because $\{x{:}csf, z{:}string\} \vdash x[problem \rightarrow z]$ and $\{x{:}csf, z'{:}string\} \vdash x[problem \rightarrow z']$ are unifiable. In addition, the program is not well-defined because the variable y is not restricted with respect to x in the first clause in P. $\square$

We are particularly interested in the class of simple programs, because each simple program has a preferred model, and because, as we shall show, there are bottom-up and top-down procedures that compute this model [45].

Definition A program P with respect to a set of declarations D is *simple* if the following two conditions hold:

- P with respect to D is i-stratified, and

- P with respect to D is well-defined.

3.5 Semantics

In this section, we describe the declarative semantics of programs. As in logic programming, the meaning of a program or database can be given by a preferred model. We first define typed Herbrand interpretations and models. A simple program may have more than one minimal model, so we define a priority relation that is used to identify preferred models of a program.

We start by defining an interpretation of a program P with respect to a set of declarations D. An interpretation assigns a truth value to every ground instance of a clause in P.

Definition The *typed Herbrand base* of P with respect to a set of declarations D is the set of all ground atoms that can be formed from the objects, predicates, functional and multi-valued methods in P with respect to D. We denote the Herbrand base of P with respect to D by B_P with respect to D.

A *typed Herbrand interpretation* I for P with respect to a set of declarations D consists of a set of ground atoms of P.

Definition Let Γ be a variable typing and D be a set of declarations. A *variable assignment* $\mathcal{V}$ (with respect to Γ) is an assignment to each variable x of an object a such that, if $x{:}\tau \in \Gamma$ and $a{:}\tau' \in D$, then $\tau' \leq_t \tau$ (with respect to D).

Definition Let $\Gamma \vdash t : \tau$ be a term and $\mathcal{V}$ a variable assignment (with respect to Γ). Then the *term assignment* $\mathcal{T}$ (based on $\mathcal{V}$ with respect to Γ) is the assignment to $\Gamma \vdash t : \tau$ of an object a such that:

- If t is a variable x, then $\mathcal{T}(t) = \mathcal{V}(x)$.
- If t is an object symbol, then $\mathcal{T}(t) = t$.

Definition Let I be an interpretation, and $\mathcal{T}$ a term assignment (with respect to Γ). Then a literal L is given a *truth value* in I (with respect to $\mathcal{T}$ and Γ) as follows:

- If L is $\Gamma \vdash p(t_1, \ldots, t_n)$, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then L is true in I (with respect to $\mathcal{T}$ and Γ) if and only if $p(t'_1, \ldots, t'_n) \in I$. If L is $\Gamma \vdash \neg p(t_1, \ldots, t_n)$, then L is true in I (with respect to $\mathcal{T}$ and Γ) if and only if $p(t'_1, \ldots, t'_n) \notin I$.
- If L is $\Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t_n]$, and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then L is true in I (with respect to $\mathcal{T}$ and Γ) if and only if $t'[m@t'_1, \ldots, t'_k \rightarrow t'_n] \in I$. If L is $\Gamma \vdash \neg t[m@t_1, \ldots, t_k \rightarrow t_n]$, then L is true in I (with respect to $\mathcal{T}$ and Γ) if and only if $t'[m@t'_1, \ldots, t'_k \rightarrow t'_n] \notin I$.
- If L is $\Gamma \vdash t[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$, and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then L is true in I (with respect to $\mathcal{T}$ and Γ) if and only if $t'[m@t'_1, \ldots, t'_k \twoheadrightarrow t'_n] \in I$. If L is $\Gamma \vdash \neg t[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$, then L is true in I (with respect to $\mathcal{T}$ and Γ) if and only if $t'[m@t'_1, \ldots, t'_k \twoheadrightarrow t'_n] \notin I$.

A ground instance of a clause is given a truth value in I (with respect to $\mathcal{T}$ and Γ) in the normal way.

Example 3.5.1 Suppose $I = \{r1[parts \twoheadrightarrow valve], r1[parts \twoheadrightarrow heat\_exchanger]\}$.
Then the following ground instances are true in I:

$r1[parts \twoheadrightarrow valve]$
$\neg r2[parts \twoheadrightarrow valve]$ □

Because of the presence of inheritance and overriding, we require one more concept before we can define a "model" of a set of clauses.

Definition Let P be a program with respect to a set of declarations D and I an interpretation. A ground clause C *overrides* a ground clause C' with respect to P and I if:

- C possibly overrides C' with respect to P, and
- the bodies of C and C' are both true in I.

Example 3.5.2 Assume appropriate declarations D, including $csf < sf$ and $failure{:}csf$. Consider program P with respect to D:

$$\{x{:}sf\} \vdash x[severity \twoheadrightarrow high]$$
$$\{x{:}csf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{"tell supervisor"}],$$

and interpretation $I_1 = \{failure[severity \twoheadrightarrow low], failure[treatment \rightarrow \text{"tell supervisor"}]\}$. The ground clause $C = failure[severity \twoheadrightarrow low] \leftarrow failure[treatment \rightarrow \text{"tell supervisor"}]$ overrides $C' = failure[severity \twoheadrightarrow high]$ with respect to P and I_1.

Now, consider P with respect to D and interpretation $I_2 = \{failure[severity \twoheadrightarrow low]\}$. Although C possibly overrides C' with respect to P, C does not override C' with respect to P and I_2, because the body of C is not true in I_2. □

Definition Let P be a program with respect to a set of declarations D and I an interpretation. We say that I is a *model* of P with respect to D if, for every ground instance C' of each clause in P, either:

- C' is true in I, or
- there exists another ground instance C of a clause in P with respect to D such that C overrides C' with respect to I.

If I is a model of P with respect to D, we write $I \models P$.

Example 3.5.3 Consider the declarations D and program P with respect to D given in Example 3.5.2, and interpretations $I_1 = \{failure[severity \twoheadrightarrow high]\}$, $I_2 = \{failure[severity \twoheadrightarrow low], failure[treatment \rightarrow \text{"tell supervisor"}]\}$, and $I_3 = \{failure[severity \twoheadrightarrow low]\}$. Because every ground instance of each clause in P is true in I_1, $I_1 \models P$. Because the ground clause $failure[severity \twoheadrightarrow low] \leftarrow failure[treatment \rightarrow \text{"tell supervisor"}]$ overrides $failure[severity \twoheadrightarrow high]$ with respect to P and I_2, and there are no other ground instances of clauses in P, $I_2 \models P$. Finally, because $failure[severity \twoheadrightarrow high]$ is not true in I_3, and is not overridden by $failure[severity \twoheadrightarrow low] \leftarrow failure[treatment \rightarrow \text{"tell supervisor"}]$ with respect to P and I_3, $I_3 \not\models P$. □

Note that a program with respect to a set of declarations may have more than one model, and that not every ground instance of a clause need be true in a model. As in [90], we can define a priority relationship between ground atoms (using possible overriding as well as negation), and use this to define a preference relationship between models. As in [90], we prefer models in which there are fewer occurrences of higher priority atoms. We say a model of a program P with respect to a set of declarations D is *preferred* if no models of P are preferable to it.

Definition Let P be a program with respect to a set of declarations D. We define a priority relation $<_p$ based on P between ground atoms with respect to D. We write $a \leq_p b$ to denote $a <_p b$ or $a = b$. For every ground instance $C\theta$ of a clause C in P with respect to D:

- $A \leq_p B$, where A is the head of $C\theta$, and B is a ground atom in the body of $C\theta$,
- $A <_p B$, where A is the head of $C\theta$, and B is a ground negative literal in the body of $C\theta$,
- $A <_p B'$ for every ground instance $C'\theta'$ of C' in P, which possibly overrides $C\theta$, where A is the head of $C\theta$ and B' is an atom in the body of $C'\theta'$,
- $A \leq_p A'$ for every ground instance $C'\theta'$ of C' in P, which possibly overrides $C\theta$, where A is the head of $C\theta$ and A' is the head of $C'\theta'$,

and for all ground atoms A, B, C and F:

- if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$,
- if $A \leq_p B$ and $B <_p C$ (respectively, $F <_p A$) then $A <_p C$ (respectively, $F <_p B$).

The goal of the following preference relation is to minimize higher priority atoms as much as possible.

Definition Suppose that M and N are two distinct models of program P with respect to a set of declarations D. Then N is *preferable* to M ($N \ll M$) if, for every atom A in $N - M$, there is an atom B in $M - N$ such that $A <_p B$. We write $N \leq M$ if $N \ll M$ or $N = M$. We say model N is a *preferred* model of P if there are no models of P preferable to N.

Example 3.5.4 Consider the declarations D and program P with respect to D given in Example 3.5.2, and the interpretations:

$$I_1 = \{failure[severity \twoheadrightarrow high]\}$$
$$I_2 = \{failure[severity \twoheadrightarrow low], failure[treatment \rightarrow \text{“tell supervisor”}]\}.$$

Since all the ground instances of clauses of P with respect to D are true or overridden in each of these interpretations, I_1 and I_2 are both models of P. The priority relationship gives the following relationships between the ground atoms of P: Because $failure[severity \twoheadrightarrow high] <_p failure[treatment \rightarrow \text{“tell supervisor”}]$ and there is no atom $A \in I_2 - I_1$ such that there exists $B \in I_1 - I_2$, where $A <_p B$, $I_1 \ll I_2$. Thus I_1 is preferable to I_2. □

The following results are analogous to Przymusinski's results for stratified logic programs [90]. They motivate our definition of simple programs.

Lemma 3.5.1 *Let P be a simple program with respect to a set of declarations D in which there are no negative literals, and no clause is possibly overridden by any other clause. Then P with respect to D has a unique minimal model, which is a preferred model.*

Proof Let $\{M_i\}$ be the set of models of P with respect to D. There are three parts to the proof. First we want to prove that the intersection of these models $\cap\{M_i\}$ is a model. This part of the proof follows from a similar proof for the classical case in [48]. We include it here for completeness. Suppose that the intersection of these models $\cap\{M_i\}$ is not a model. Then there must be a ground instance of a clause in P with repect to D whose body is true in every M_i but whose head is not true in $M \in \{M_i\}$. This is clearly a contradiction as there is no overriding, it would imply that M does not model $C\theta$. Thus M cannot be a model of P.

Second we want to prove that $\cap\{M_i\}$ is the unique minimal model. This is clearly true since every other model is a superset of $\cap\{M_i\}$.

Finally we want to prove that $\cap\{M_i\}$ is a preferred model. Suppose $\cap\{M_i\}$ is not a preferred model. Then there is a model $M \in \{M_i\}$ such that for every atom $A \in M - \cap\{M_i\}$ there is an atom $B \in \cap\{M_i\} - M$ such that $A <_p B$. Because $\cap\{M_i\}$ is a model and there is no $B \in \cap\{M_i\} - M$, this is a contradiction. Thus, $\cap\{M_i\}$ is a preferred model of P with respect to D. $\Box$

Before proving the following lemma we define "conflicts", "(atom, type) pairs", "(clause, type) pairs", an ordering on (clause, type) pairs, and the "level" of a program.

Definition Let A be the ground method atom $a[m@a_1, \ldots, a_k \rightarrow b]$ and A' the ground method atom $a[m@a_1, \ldots, a_k \rightarrow b']$, or let A be $a[m@a_1, \ldots, a_k \twoheadrightarrow b]$ and A' be $a[m@a_1, \ldots, a_k \twoheadrightarrow b']$. We say A *conflicts* with A' if $b \neq b'$.

Definition Let P be a program with respect to a set of declarations D. The set of ground *(atom, type) pairs* of P with respect to D is $\{(a, \tau) \mid a$ in B_P with respect to D is a ground instance of an atom a' and τ is the type of $a'\}$.

We define ground "(clause, type) pairs", an ordering on them, and the "level" of a program.

Definition Let P be a program with respect to a set of declarations D. The set of ground *(clause, type) pairs* of P with respect to D is $\{(p, \tau) \mid p$ is a ground instance of a clause p' in P with respect to D and τ is the type of the clause $p'\}$.

The ground (clause, type) pairs of an i-stratified program can be *partitioned* into levels based on the following relationship.

Definition Let P be an i-stratified program with respect to a set of declarations D. For all clauses $\Gamma \vdash A \leftarrow B$ of type τ and $\Gamma' \vdash A' \leftarrow B'$ of type τ' in $[P]_D$, if $\mu(A) < \mu(A')$ or $\Gamma \vdash A \leftarrow B$ possibly overrides $\Gamma' \vdash A' \leftarrow B'$ then $(A \leftarrow B, \tau) <_l (A' \leftarrow B', \tau')$.

Let $P_1^t, \ldots, P_k^t$ be a partitioning on the ground (clause, type) pairs of an i-stratified program P with respect to a set of declarations D that satisfies the $<_l$ relationship described above. Then P with respect to D has *level* k.

Let P_i^t be a partition in $P_1^t, \ldots, P_k^t$, where $1 \leq i \leq k$. Then P_i is the *untyped partition* of P_i^t, such that $P_i = \{C \mid (C, \tau) \in P_i^t\}$.

We now show that the above definition does define a partitioning on simple programs. Recall the definition of i-stratification. There can be a relationship between mappings of ground atoms A and A', $\mu(A) < \mu(A')$, if there are ground clauses of the following forms in $[P]_D$:

1. $A' \leftarrow \cdots \wedge A \wedge \cdots$,
2. $A' \leftarrow \cdots \wedge \neg A \wedge \cdots$,
3. $A' \leftarrow \cdots \wedge B \wedge \cdots$, $B' \leftarrow \cdots \wedge A \wedge \cdots$ where $B' \leftarrow \cdots \wedge A \wedge \cdots$ possibly overrides $A' \leftarrow \cdots \wedge B \wedge \cdots$, or
4. $A' \leftarrow \cdots \wedge B \wedge \cdots$, $B' \leftarrow \cdots \wedge \neg A \wedge \cdots$ where $B' \leftarrow \cdots \wedge \neg A \wedge \cdots$ possibly overrides $A' \leftarrow \cdots \wedge B \wedge \cdots$.

We want to show it is impossible for P to contain a ground clause defining A' that possibly overrides a ground clause defining A. In case 1, the clause $A' \leftarrow \cdots \wedge A \wedge \cdots$ would possibly override a clause with head A. Then $\mu(A) < \mu(A)$. Such a program is not i-stratified. Case 2 is similar to case 1. In case 3, the clause $A' \leftarrow \cdots \wedge B \wedge \cdots$ would possibly override a clause with head A. Because $B' \leftarrow \cdots \wedge A \wedge \cdots$ would possibly override the clause with head A, $\mu(A) < \mu(A)$. This program is not i-stratified either. Case 4 is similar to case 3. Thus the definition above does define a partitioning on simple programs.

Lemma 3.5.2 *Let P be a simple program with respect to a set of declarations D and $P_1^t, \cdots, P_k^t$ be a partitioning of P with respect to D. Then $P_1 \cup \cdots \cup P_i$ for $i \leq k$ is a simple program.*

Proof We want to show that $P_1 \cup \cdots \cup P_i$ with respect to D is well-defined and i-stratified. Suppose $P_1 \cup \cdots \cup P_i$ with respect to D is not well-defined. Then P with respect to D is not well-defined. This is a contradiction.

Suppose now that $P_1 \cup \cdots \cup P_i$ with respect to D is not i-stratified. Then there are clauses $\Gamma \vdash A \leftarrow B$ and $\Gamma' \vdash A' \leftarrow B'$ in $P_1 \cup \cdots \cup P_i$ with respect to D that do not satisfy the mapping μ. This is also a contradiction. Thus $P_1 \cup \cdots \cup P_i$, for $i \leq k$, is a simple program. □

Lemma 3.5.3 *Let P be a simple program with respect to a set of declarations D and $P_1^t, \ldots, P_{k+1}^t$ be a partitioning of the ground (clause, type) pairs of P as described above. Suppose $P_1 \cup \cdots \cup P_k$ has a unique preferred model M_k. Then there exists a unique minimal extension of M_k to a unique preferred model M_{k+1} of P.*

Proof This proof is based on the construction of a minimal extension of M_k to M_{k+1}. First, we define some terminology used in the proof. We say a ground (clause, type) pair (C, τ) conflicts with a ground (atom, type) pair (A', τ'), if the head of clause C conflicts with atom A' and $\tau \neq \tau'$. Let $M|_i$ be the restriction of the set M to the atoms in the heads of ground instances of clauses in $P_1 \cup \cdots \cup P_i$. Let M_i^t be a set of (atom, type) pairs (A, τ) where A is in M_i, A is a ground instance of $\Gamma \vdash A'$, and τ is the type of $\Gamma \vdash A'$.

Consider the set P_{k+1}^t of ground (clause, type) pairs. Modify P_{k+1}^t by removing all (clause, type) pairs whose head conflicts with an (atom, type) pair in M_k^t. Remove all (clause, type) pairs where the clause contains a negative literal $\neg A_i$ such that A_i is an atom in an (atom, type) pair in M_k^t. From the remaining clauses, remove all the negative literals (they are all true in M_k^t). Call the resulting set of clauses P_{res}^t. Let $P_{res} = \{C \mid (C, \tau) \in P_{res}^t\}$.

Consider the program $P^* = P_{res} \cup M_k$. There are no negative literals in P^*, and no clause in P^* is possibly overridden by another clause in P^*. By Lemma 3.5.1, P^* has a unique minimal model, M_{k+1}. We now show that M_{k+1} is the unique preferred model of P.

We first want to prove that M_{k+1} is a model of P. We know that M_{k+1} is a model of $P^* = P_{res} \cup M_k$. Let $C = A \leftarrow L_1 \wedge \cdots \wedge L_n \in P_{k+1}$ and suppose $L_1 \wedge \cdots \wedge L_n$ is true in M_{k+1}. If $C \in P_{res}$ then C is true in M_{k+1}. If $C \notin P_{res}$ then C is overridden by some clause in $P_1 \cup \cdots \cup P_k$ with respect to M_{k+1}.

We next want to prove that M_{k+1} is a preferred model of P. Suppose M_{k+1} is not a preferred model of P. Then there exists another model M of P such that $M \ll M_{k+1}$. That is, $\forall A \in M - M_{k+1}$, $\exists B \in M_{k+1} - M$ such that $A <_p B$. First note that $M|_k = M_k$. For if $M|_k \subset M_k$, then $M|_k$ is a model of $P_1 \cup \cdots \cup P_k$ such that $M|_k \ll M_k$, then M_k would not be the preferred model of $P_1 \cup \cdots \cup P_k$. If $M_k \subset M|_k$, then M differs from M_{k+1} at a level less than $k+1$, and it would not be possible for M to be a preferred model of P. Then M is an extension of M_k. But M_{k+1} is the minimal extension of M_k. So M_{k+1} and M only differ on level $k+1$ atoms. As M_{k+1} is the minimal unique extension of M_k it is impossible that $M \ll M_{k+1}$.

Finally we want to prove that M_{k+1} is the only preferred model of P. Suppose there exists some preferred model M of P. Then $M \not\ll M_{k+1}$ and $M_{k+1} \not\ll M$. As above $M|_k = M_k$. Because M_{k+1} is the minimal unique extension of M_k then $M_{k+1} \ll M$, which is a contradiction. □

Theorem 3.5.1 *Let P be a simple program with respect to a set of declarations D. Then P with respect to D has exactly one preferred model, which we denote M_P. For every other model M, we have $M_P \ll M$.*

Proof We prove this theorem by induction on the level of program P with respect to D.

Suppose the level of P with respect to D is 1. Then P with respect to D is a program in which there are no negative literals, and no clause is overridden by any other clause. In Lemma 3.5.1 we proved that P with respect to D has a unique minimal model, which is a preferred model M_P. We now want to prove that M_P is the unique preferred model of P with respect to D. Suppose there is another preferred model of P with respect to D, M, such that $M \not\ll M_P$ and $M_P \not\ll M$. This is clearly impossible because M_P is the unique minimal model of P with respect to D and because P with respect to D is a program in which there are no negative literals, and no clause is overridden by any other clause.

Assume that the result holds for a program P with respect to D of level k. Then $P = P_1 \cup \cdots \cup P_k$ has a unique preferred model M_P.

Suppose now that P with respect to D has maximum level $k+1$. We proved in Lemma 3.5.3 that if M'_P is a preferred model of $P_1 \cup \cdots \cup P_k$, there is an extension of M'_P to M_P, which is a preferred model of $P_1 \cup \cdots \cup P_{k+1}$ and hence P with respect to D. By Lemma 3.5.3 and the induction hypothesis, P with respect to D has a preferred model M_P, and M_P is unique.

By definition, as M_P is the preferred model, for every other M of P, $M_P \ll M$. □

Theorem 3.5.2 *Let P be a simple program with respect to a set of declarations D. Then M_P does not depend on the particular i-stratification chosen for P with respect to D.*

Proof There may be many i-stratifications of a program. The definition of preferred model is independent of a particular i-stratification. □

This preferred model semantics agrees with our intuition about the interaction between inheritance, overriding and deduction. We can now give the definition of a correct answer for a program and a goal.

Definition Let P be a simple program with respect to a set of declarations D and G a goal. An *answer* for $P \cup \{G\}$ is a ground substitution for variables of G. Let G be the goal $\Gamma \vdash \; \leftarrow L_1 \wedge \cdots \wedge L_k$, and θ an answer for $P \cup \{G\}$. Then θ is a *correct answer* for $P \cup \{G\}$ if $(L_1 \wedge \cdots \wedge L_k)\theta$ is true in M_P.

3.6 Expressibility

In this section, we prove that any queries, which can be expressed in Gulog, can be expressed in a subset of Gulog without negation. We call the subset of Gulog without negation, Gulog$^+$. The relationship between Gulog and Gulog$^+$ is interesting because, not only does it show how expressive Gulog$^+$ is, it also hints at the relationship between inheritance and overriding, and negation.

To prove this conjecture, we need a model to work within. We first define a data model for a schema and then a database with respect to a schema. We assume a given universal domain of object identifiers, U. Let there be a set of object types, T. Boolean, integer, real, char and string are special object types. A type lattice, $<$, is a relation over T. Predicate signatures T_{R_i} are a tuple $(T_1, \ldots, T_n)$ where $T_1, \ldots, T_n \in T$. Functional method signatures $M_{i\Rightarrow}$ are a tuple $(T, T_1, \ldots, T_n, T')$ where $T, T_1, \ldots, T_n, T' \in T$. Multi-valued method signatures $M_{i\Rightarrow\!\!\!\Rightarrow}$ are a tuple $(T, T_1, \ldots, T_n, T')$ where $T, T_1, \ldots, T_n, T' \in T$.

The schema is $S =< T, <, T_{R_1}, \ldots, T_{R_k}, M_{0\Rightarrow}, \ldots, M_{j\Rightarrow}, M_{0\Rightarrow\!\!\!\Rightarrow}, \ldots, M_{n\Rightarrow\!\!\!\Rightarrow} >$.

Example 3.6.1 Consider the declarations:

$cf[temp \Rightarrow level]$	$temp\_sensor{:}part$	$failure{:}cf$
$cf[problem \Rightarrow string]$	$low{:}level$	$high{:}level$
$cf[malfunction \Rightarrow part]$	$normal{:}level$	$answer(part)$

These would be modeled as the following schema, S:

$$\begin{aligned}&T = \{cf, level, temp\_sensor, part, failure, low, high, normal\},\\ &< \,= \{(temp\_sensor, part), (failure, cf), (low, level), (high, level), (normal, level)\},\\ &temp_{\Rightarrow} = (cf, level),\\ &problem_{\Rightarrow} = (cf, string),\\ &malfunction_{\Rightarrow} = (cf, part)\\ &T_{answer} = (part).\end{aligned}$$

The only surprise here may be that object identifiers like *temp_sensor* are treated as types. Recall that this was introduced in Section 3.2. □

We now describe a database with respect to a schema S. The domains (without considering inheritance), D_{T_i}, for types $T_i \in T$ are sets of object identifiers. The domains (considering inheritance), D_{τ_j}, for types $T_j \in T$ are $D_{\tau_j} = D_{T_j} \cup \bigcup D_{T_i}$, where $T_i < T_j$ in S. Ground method and predicate atoms are stored in the database. The relation R_i is a relation over $D_{\tau_1} \times \cdots \times D_{\tau_n}$, where $T_{R_i} = (T_1, \ldots, T_n)$ in S. The relation $M_{i\rightarrow}$ is a relation over $D_{\tau} \times D_{\tau_1} \times \cdots \times D_{\tau_n} \times D_{\tau'}$, where $T_{M_{i\Rightarrow}} = (T, T_1, \ldots, T_n, T')$ in S. The relation $M_{i\twoheadrightarrow}$ is a relation over $D_{\tau} \times D_{\tau_1} \times \cdots \times D_{\tau_n} \times D_{\tau'}$, where $M_{i\Rrightarrow} = (T, T_1, \ldots, T_n, T')$ in S. The database is $B_S =< D_{T_1}, \ldots, D_{T_p}, R_1, \ldots, R_k, M_{0\rightarrow}, \ldots, M_{j\rightarrow}, M_{0\twoheadrightarrow}, \ldots, M_{n\twoheadrightarrow} >$.

Example 3.6.2 Consider the schema, S, in Example 3.6.1. Now consider the following Gulog program:

$failure[problem \rightarrow$ "thermometer broken"$]$
$failure[temp \rightarrow normal]$

It would be modeled as the following database, B_S:

$$\begin{aligned}&D_{failure} = \{failure\}\\ &D_{normal} = \{normal\}\\ &problem_{\rightarrow} = \{(failure, \text{“thermometer broken”})\}\\ &temp_{\rightarrow} = \{(failure, normal)\} \ \square\end{aligned}$$

We denote the domain $D_{T_1} \cup \ldots \cup D_{T_p}$ of a database B_S by $D(B_S)$. Facts are part of the database and clauses that define predicates and methods are considered as part of the query, as we describe in the following definition of query. Now, because there are constants in the query that are not in the database, the output of a query on a database is not restricted to the constants in the domain of the database. The elements in a database are usually uninterpreted. There is a set of constants which are in the query but not in the domain of the database which must be interpreted. Let C be the set of constants in the query Q with schema S that are not in the domain $D(B_S)$ of database B_S. A query Q with schema S and constants C is a partial recursive function that, given a database B_S as input, produces as output a relation over $D(B_S) \cup C$. Furthermore, every query must satisfy the following consistency criterion: If two databases B_S and B'_S are isomorphic using an isomorphism from $D(B_S)$ into $D(B'_S)$, then their outputs are also isomorphic under the same isomorphism.

We define two languages based on Gulog, one that allows negative literals in the body of clauses and one that does not.

Definition A *Gulog*$^+$ *program* with respect to a schema S is a finite set of clauses of the form $A \leftarrow A_1 \wedge \cdots \wedge A_n$ where $A, A_1, \ldots, A_n$ are atoms.

Let $a(x_1, \ldots, x_n)$ be a predicate atom with variables $x_1, \ldots, x_n$. A *Gulog*$^+$ *query* with respect to a schema S is a pair (a, P), where a is a predicate symbol and P is a Gulog$^+$ program with respect to S that includes a definition of a.

Let $Q = (a, P)$ be an Gulog$^+$ query with respect to S. The meaning of Q is defined by $Q(B_S) = \{(c_1, \ldots, c_n) \mid a(c_1, \ldots, c_n) \in M_{B,P}\}$, where $M_{B,P}$ is the unique preferred model for P with respect to S that extends the interpretation $B_S =< D_{T_1}, \ldots, D_{T_p}, R_1, \ldots, R_k,$ $M_{0_{\rightarrow}}, \ldots, M_{j_{\rightarrow}}, M_{0_{\twoheadrightarrow}}, \ldots, M_{n_{\twoheadrightarrow}} >$ by assigning relations to the derived predicates and methods in P.

Definition A *Gulog program* with respect to a schema S is a finite set of clauses of the form $A \leftarrow L_1 \wedge \cdots \wedge L_n$ where A is an atom, $L_1, \ldots, L_n$ are literals, and no atom depends negatively on itself.

Let $a(x_1, \ldots, x_n)$ be a predicate atom with variables $x_1, \ldots, x_n$. A *Gulog query* with respect to a schema S is a pair (a, P), where a is a predicate symbol and P is a Gulog program with respect to S that includes a definition of a.

Let $Q = (a, P)$ be a Gulog query with respect to S. The meaning of Q is defined by $Q(B_S) = \{(c_1, \ldots, c_n) \mid a(c_1, \ldots, c_n) \in M_{B,P}\}$, where $M_{B,P}$ is the unique preferred model for P with respect to S that extends the interpretation $B_S =< D_{T_1}, \ldots, D_{T_p}, R_1, \ldots, R_k,$ $M_{0_{\rightarrow}}, \ldots, M_{j_{\rightarrow}}, M_{0_{\twoheadrightarrow}}, \ldots, M_{n_{\twoheadrightarrow}} >$ by assigning relations to the derived predicates and method in P.

Example 3.6.3 Consider the schema S in Example 3.6.1 and the database B_S in Example 3.6.2. Now consider the Gulog query Q where $a = answer$ and P is:

$$\{y{:}cf\} \vdash y[malfunction \rightarrow temp\_sensor] \leftarrow \neg y[temp \rightarrow high] \wedge \neg y[temp \rightarrow low]$$
$$\{y{:}part\} \vdash answer(y) \leftarrow failure[malfunction \rightarrow y]$$

The database B_S is extended by the unique preferred model of P:

$$malfunction_{\rightarrow} = \{(failure, temp\_sensor)\}$$
$$answer = \{temp\_sensor\}$$

and $Q(B_S) = temp\_sensor$. □

In the following theorem we prove that any Gulog query can be expressed as a Gulog$^+$ query with respect to a schema and database as long as the schema and database contain certain information. This information is described in the statement of the theorem. The proof of the theorem describes a translation from Gulog queries to Gulog$^+$ queries. The intuition behind the translation is for any negative literal $\neg A$ in the Gulog query, we introduce a method into the Gulog$^+$ query that returns false if A is true and defaults to true otherwise.

Theorem 3.6.1 *Let Q be a Gulog query with respect to a database B_S and a schema S with meaning $Q(B_S)$. Let S include new symbols σ_1 and σ_2 in T, with $\sigma_1 < \sigma_2$ in the type lattice. For every negated predicate literal $\Gamma \vdash \neg p(t_1, \ldots t_n)$ in Q, where $\Gamma \vdash t_1{:}\tau_1, \ldots, \Gamma \vdash t_n{:}\tau_n$ are terms, include $(\sigma_2, \tau_1, \ldots, \tau_n, boolean)$ in a new relation $\$p_{\Rrightarrow}$ in S. For every negated functional method literal $\Gamma \vdash \neg t[m@t_1, \ldots t_n \rightarrow t']$ in Q, where $\Gamma \vdash t{:}\tau, \Gamma \vdash t_1{:}\tau_1, \ldots, \Gamma \vdash t_n{:}\tau_n, \Gamma \vdash t'{:}\tau'$ are terms, include $(\sigma_2, \tau, \tau_1, \ldots, \tau_n, \tau', boolean)$ in a new relation $\$m_{\rightarrow\Rrightarrow}$ in S. For every negated multi-valued method literal $\Gamma \vdash \neg t[m@t_1, \ldots t_n \twoheadrightarrow t']$ in Q, where $\Gamma \vdash t{:}\tau, \Gamma \vdash t_1{:}\tau_1, \ldots, \Gamma \vdash t_n{:}\tau_n, \Gamma \vdash t'{:}\tau'$ are terms, include $(\sigma_2, \tau, \tau_1, \ldots, \tau_n, \tau', boolean)$ in a new relation $\$m_{\twoheadrightarrow\Rrightarrow}$ in S. Let the domain of D_{σ_2} include the new object identifier obj in B_S. There is a Gulog$^+$ query Q' with schema S with meaning $Q'(B_S)$ such that $Q'(B_S) = Q(B_S)$.*

Proof We prove this theorem by describing a translation from a Gulog program P with respect to a schema S to a Gulog$^+$ program P' with respect to a schema S. Obviously any clause in P with respect to S with no negation in the body is a clause in P' with respect to S. Let $\Gamma \vdash C_{pos}$ be a conjunction of positive literals and $\Gamma \vdash \neg C_1 \wedge \cdots \wedge \neg C_j$ be a conjunction of negative literals. To simplify the proof, we introduce m_name_i. For each $\Gamma \vdash C_i$, if $\Gamma \vdash C_i$ is a predicate atom $\Gamma \vdash p(t_1, \ldots, t_n)$ then $m\_name_i = \$p_p@t_1, \ldots, t_n$; if $\Gamma \vdash C_i$ is a functional method atom $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ then $m\_name_i = \$m_{\rightarrow}@t, t_1, \ldots, t_n, t'$; if $\Gamma \vdash C_i$ is a multi-valued method atom $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$ then $m\_name_i = \$m_{\twoheadrightarrow}@t,\ t_1, \ldots, t_n, t'$. Any clause $A \leftarrow C_{pos} \wedge \neg C_1 \wedge \cdots \wedge \neg C_j$ in P with respect to S is translated to:

$$\begin{array}{l}
\Gamma \cup \{x':\sigma_2\} \vdash x'[m\_name_1 \twoheadrightarrow true] \leftarrow C_{pos} \\
\vdots \\
\Gamma \cup \{x':\sigma_2\} \vdash x'[m\_name_j \twoheadrightarrow true] \leftarrow C_{pos} \\
\Gamma \cup \{x':\sigma_1\} \vdash x'[m\_name_1 \twoheadrightarrow false] \leftarrow C_1 \\
\vdots \\
\Gamma \cup \{x':\sigma_1\} \vdash x'[m\_name_j \twoheadrightarrow false] \leftarrow C_j \\
\Gamma \vdash A \leftarrow obj[m\_name_1 \twoheadrightarrow true] \wedge \cdots \wedge obj[m\_name_j \twoheadrightarrow true]
\end{array}$$

in P' with respect to S.

We need to show that $Q'(B_S) = Q(B_S)$. Let $M_{B,P'}$ be the unique preferred model of P' with respect to S which extends B_S and $M_{B,P}$ be the unique preferred model of P with respect to S which extends B_S. Consider P', if $C_i \in M_{B,P'}$ where $C_i \in \{C_1, \ldots, C_j\}$ then $obj[m\_name_1 \twoheadrightarrow false], \ldots, obj[m\_name_j \twoheadrightarrow false] \in M_{B,P'}$, $obj[m\_name_1 \twoheadrightarrow true], \ldots,$ $obj[m\_name_j \twoheadrightarrow true] \notin M_{B,P'}$, and $A \notin M_{B,P'}$. Similarly if $C_i \in M_{B,P}$ where $C_i \in \{C_1, \ldots, C_j\}$ then $A \notin M_{B,P}$.

If $C_1, \ldots, C_j \notin M_{B,P'}$ then $obj[m\_name_1 \twoheadrightarrow false], \ldots, obj[m\_name_j \twoheadrightarrow false] \notin M_{B,P'}$, and if $C_{pos} \in M_{B,P'}$ then $obj[m\_name_1 \twoheadrightarrow true], \ldots, obj[m\_name_j \twoheadrightarrow true] \in M_{B,P'}$, and $A \in M_{B,P'}$. If $C_1, \ldots, C_j \notin M_{B,P}$ and $C_{pos} \in M_{B,P}$ then $A \in M_{B,P}$.

It follows that $Q'(B_S) = Q(B_S)$. $\Box$

The following example demonstrates the translation given in the proof of Theorem 3.6.1.

Example 3.6.4 Consider a schema that models the set of declarations D:

$obj:\sigma_1$	$cf[temp \Rightarrow level]$	$temp\_sensor:part$
$\sigma_1 < \sigma_2$	$cf[problem \Rightarrow string]$	$low:level$
$\sigma_2[\$temp_{\rightarrow}@cf, level \Rrightarrow boolean]$	$cf[malfunction \Rightarrow part]$	$high:level$
$answer(part)$	$failure:cf.$	

Consider the database B_S which models the ground atoms:

$$\begin{array}{l}
failure[problem \rightarrow \text{"thermometer broken"}] \\
failure[temp \rightarrow normal].
\end{array}$$

Consider the Gulog program P with respect to D:

$$\begin{array}{l}
\{y:cf\} \vdash y[malfunction \rightarrow temp\_sensor] \leftarrow \neg y[temp \rightarrow high] \wedge \neg y[temp \rightarrow low] \\
\{y:part\} \vdash answer(y) \leftarrow failure[malfunction \rightarrow y].
\end{array}$$

Consider now the program P' with respect to D:

$$
\begin{array}{l}
\{y{:}cf\} \vdash y[malfunction \rightarrow temp\_sensor] \leftarrow \\
\quad obj[\$temp_{\rightarrow}@y, high \twoheadrightarrow true] \wedge obj[\$temp_{\rightarrow}@y, low \twoheadrightarrow true] \\
\{y{:}part\} \vdash answer(y) \leftarrow failure[malfunction \rightarrow y] \\
\{x'{:}\sigma_1, y{:}cf\} \vdash x'[\$temp_{\rightarrow}@y, high \twoheadrightarrow false] \leftarrow y[temp \rightarrow high] \\
\{x'{:}\sigma_1, y{:}cf\} \vdash x'[\$temp_{\rightarrow}@y, low \twoheadrightarrow false] \leftarrow y[temp \rightarrow low] \\
\{x'{:}\sigma_2, y{:}cf\} \vdash x'[\$temp_{\rightarrow}@y, high \twoheadrightarrow true] \\
\{x'{:}\sigma_2, y{:}cf\} \vdash x'[\$temp_{\rightarrow}@y, low \twoheadrightarrow true].
\end{array}
$$

Now consider the queries $Q = (answer, P)$ and $Q' = (answer, P')$. The meaning of Q is $Q(B_S) = \{temp\_sensor\}$ and the meaning of Q' is $Q'(B_S) = \{temp\_sensor\}$. $\Box$

3.7 Summary

The syntax of Gulog is similar to that of LOGIN [8] and F-logic [65]. In order to concentrate on the semantics of inheritance and overriding in a well understood context, schema declarations are separated from data definitions, as in the relational approach. As in C++, we distinguish between objects and types (or classes). Object identifiers are restricted to constants rather than the arbitrary terms of F-logic [65] and IQL [6]. This restriction limits the expressive power of the language, but primarily with regard to dynamic aspects such as object creation, which we are not considering. To maintain a general approach, method overriding can be defined on specific instances of a subtype.

Interestingly, although we include typing, inheritance, overriding, functional and multi-valued methods in Gulog, the results in this chapter are similar to those for Datalog with negation. The semantics of deductive object-oriented logics have been defined in a similar way in [4, 27, 65, 73]. Unlike these approaches, we use a different definition of model, claiming that it describes a natural description of programs and has an interesting and useful theory associated with it. Also in [4], there is no concept of inheritance or overriding. In the definition of the perfect model of an F-logic program [65], the authors include extra axioms in the program to deal with equivalence predicates and inheritance, and take the perfect model of the augmented program. Unlike our approach, inheritance and overriding are not dealt with in the definition of model. Also if there are conflicts due to multiple inheritance, a program may have multiple "perfect" models. In [27], the authors consider interpretations of units of a program and combine them to give an interpretation of the program. The model of a program is regarded as a function over Herbrand sets rather than a simple Herbrand set. Our approach considers the program as a single unit. Laenens and Vermiers' [73] approach is similar to that described above except that the authors do not distinguish between classes and objects, or between functional and multi-valued methods, and models are based on the more complicated 3-valued semantics.

Our approach to sets, using multi-valued methods, is the same as that described in [36], and very similar to the approach in [65]. We express set containment rather than set equality. The advantage of this approach is that, without supporting higher-order logic, we can still support many aspects of set manipulation. In comparison, LDL [87] is more expressive but computationally expensive. In COL [4], the author disallows infinite sets thus avoiding the computational problems of LDL. However, this limits the expressiveness of the language.

In Gulog, an answer to a query is a substitution as in logic programming. However, in Gulog, the substitutions are typed and objects rather than values are substituted. Our approach is most similar to that in IQL [6], ILOG [56] and LLO [79] where the answer is a collection of object identifiers. This is different from F-logic [65] and LIFE [9], where there is basically no distinction between types and instantiations of types. In [14], answers to DTL queries are a set of homogeneous elements. That is every answer in the set has the same type, which is the “minimal” type with respect to subtyping.

We have also proved that any Gulog program can be expressed in a subset of Gulog, without negation. This shows how negation can be modeled using inheritance and overriding.

Chapter 4

Query Evaluation

We have considered the declarative semantics of programs. Procedures that compute answers for queries or goals with respect to a program are obviously necessary. It is desirable that the answers computed by these procedures are equivalent to the answers with respect to the declarative semantics, that is, that the procedures are sound and complete with respect to the declarative semantics.

There are two common approaches to query evaluation. The first approach is to process the query with respect to the program directly without translation [24, 65, 73]. This approach provides insight into the needs of deductive object-oriented systems, and can be efficient. In this chapter, we describe this approach. The second is to translate the program and query to a language for which efficient query evaluation procedures exist [7, 46, 81]. This approach provides efficient evaluation procedures, but does not aid our understanding of deductive object-oriented databases. In Chapter 5, we describe the second approach.

We describe a bottom-up procedure and prove it is sound and complete with respect to the declarative semantics described in Section 3.5. Because this procedure is not goal directed, it is inefficient when computing the answers to some queries. We then describe a goal directed top-down procedure. This procedure is sound with respect to the declarative semantics, but is not complete because it does not always terminate. We extend this procedure with a tabling mechanism, and prove that the new procedure is sound and complete with respect to the declarative semantics.

4.1 Bottom-up evaluation

In [12], Apt et al. defined a bottom-up procedure to compute the standard model of a stratified logic program. In this section, we describe a similar bottom-up procedure to evaluate the preferred model of a simple program. This procedure differs in three ways: it is based on the possibility of overriding, as well as the presence of negation; the immediate consequence operator is a mapping between typed interpretations rather than interpretations; and atoms are not included in the typed interpretation if they conflict with atoms that are part of the interpretation. (Recall the definition of conflicting atoms from Section 3.5.)

In a simple program with only functional methods, there is only one ground instance of a clause in which a value is assigned to functional method m of arity n for object o for type τ, so the immediate consequence operator T may be defined to include an atom A in $T(I)$ only if A does not conflict with an atom in I. But in a simple program with multi-valued methods, there may be many ground instances of clauses in which a value is assigned to multi-valued method m of arity n for object o for type τ, so A may be included in $T(I)$ if A does not conflict with any atom in I *of a different type*. Recall the definition of "conflict" in Section 3.5 does not consider the types of the atoms.

Example 4.1.1 Assume a set of declarations D including $failure{:}sf$. Consider program P with respect to the set of declarations D:

$$\begin{array}{l} \{x : sf\} \vdash x[severity \twoheadrightarrow high] \\ \{x : sf\} \vdash x[severity \twoheadrightarrow low] \leftarrow x[treatment \rightarrow \text{“tell supervisor”}] \\ \{x : sf\} \vdash x[treatment \rightarrow \text{“tell supervisor”}]. \end{array}$$

Let $I = \{failure[treatment \rightarrow \text{“tell supervisor”}], failure[severity \twoheadrightarrow high]\}$. Intuitively, we would expect $failure[severity \twoheadrightarrow low] \in T(I)$ because it is not overridden by another clause in $[P]_D$ with respect to I. Using the immediate consequence operator defined in [44], which doesn't consider typing, $failure[severity \twoheadrightarrow low]$ conflicts with $failure[severity \twoheadrightarrow high]$, so $failure[severity \twoheadrightarrow low] \notin T(I)$. $\Box$

Obviously we require some way of recording the type on which a multi-valued method is defined.

A *typed interpretation* (or t-interpretation) is an extension to an interpretation in which each ground atom A is replaced by a pair (A, τ), where τ is a type associated with A. The set of all typed interpretations forms a complete lattice. We denote the least element by $\emptyset$ and the order relation on the lattice is denoted by $\subseteq$. The immediate consequence operator is a mapping from t-interpretations to t-interpretations.

Definition The *immediate consequence operator* is defined as follows: Let P be a set of ground (clause, type) pairs, I a t-interpretation and τ a type. Then $T_P(I) = I \cup \{ (A, \tau) \mid (C, \tau)$ is a (clause, type) pair of P, $C = A \leftarrow B_1 \wedge \cdots \wedge B_k$, for each B_i and type τ_i, $(B_i, \tau_i) \in I$, and, if A is a method atom and A' is another method atom with $(A', \tau') \in I$, and $\tau \neq \tau'$, then A and A' do not conflict $\}$.

If the immediate consequence operator is a monotonic operator then we can take its fixpoint. We now define monotonic and show that the immediate consequence operator is not monotonic. Recall that the set of all typed interpretations of a program forms a complete lattice. The least element is $\emptyset$ and the elements of the lattice are denoted by I, J, M. We say that an operator T is *monotonic* if $I \subseteq J$ implies $T(I) \subseteq T(J)$. In this example we show that the immediate consequence operator, T_P, is not monotonic.

Example 4.1.2 Assume a set of declarations D including $csf < sf$, and $failure{:}csf$. Consider the program P with respect to the set of declarations D:

$$\begin{array}{l} \{x{:}sf\} \vdash x[problem \rightarrow \text{“system failure”}] \\ \{x{:}csf\} \vdash x[problem \rightarrow \text{“temperature high”}] \leftarrow x[temp \rightarrow high]. \end{array}$$

Let $I = \emptyset$ and $J = \{(failure[temp \rightarrow high], csf)\}$. Then $I \subseteq J$, $T_P(I) = \{(failure[problem \rightarrow \text{“system failure”}], sf)\}$ and $T_P(J) = \{(failure[temp \rightarrow high], csf), (failure[problem \rightarrow \text{“temperature high”}], csf)\}$. Clearly $T_P(I) \not\subseteq T_P(J)$, thus T_P is not monotonic. $\Box$

If we can prove that T_P is monotonic in restricted circumstances, then we can take the fixpoint of T_P in those circumstances.

Proposition 4.1.1 *Let P be a simple program with respect to a set of declarations D, and $P_1^t, \ldots, P_k^t$ be a partitioning of the ground (clause, type) pairs of P with respect to D as described in Section 3.5.*

1. *T_{P_1} is monotonic.*
2. *Let M_{k-1} be the model of $P_1 \cup \cdots \cup P_{k-1}$. T_{P_k} is monotonic if I is initially M_{k-1}.*

Proof We prove part 2 by induction on the maximum level of (clause, type) pairs in P with respect to D. We prove part 1 in the first step of the induction.

Suppose the maximum level of P with respect to D is 1, then there is no negation or possible overriding in P_1. Clearly T_{P_1} is monotonic. This proves part 1.

Assume the result holds for programs whose maximum level is $k-1$. Then M_{k-1} is the fixpoint of $T_{P_{k-1}}$.

Suppose the maximum level is k. Let P_k^- be P_k without the (clause, type) pairs where the clause contains a negative literal which is false in M_{k-1}, and where the negative literals are omitted from clauses in (clause, type) pairs if the negative literal is true in M_{k-1}. Let P_k^* be P_k^- without the (clause, type) pairs where an (atom, type) pair in M_{k-1} conflicts with the (clause, type) pair. Taking the fixpoint of T_{P_k} with $I = M_{k-1}$ initially is the same as taking the fixpoint of $T_{P_k^*}$ with $I = M_{k-1}$ initially. P_k^* is a program without negation or possible overriding so T_{P_k} with $I = M_{k-1}$ initially is monotonic.

Thus T_P restricted in this way is monotonic. □

We now describe a procedure for computing a model for a program P with respect to a set of declarations D using T_P in this way.

Definition The *powers* of T_P for an interpretation I are defined:

$$\begin{aligned}&T_P\!\uparrow\!0(I) = I\\ &T_P\!\uparrow\!(n+1)(I) = T_P(T_P\!\uparrow\!n(I))\\ &T_P\!\uparrow\!\omega(I) = \textstyle\bigcup_{n=0}^{\infty} T_P\!\uparrow\!n(I).\end{aligned}$$

We described a partial ordering on (clause, type) pairs of a program in Section 3.5. The following example illustrates why it is necessary to also take typing into account when partitioning the program.

Example 4.1.3 Assume the set of declarations D including $csf < sf$ and $failure{:}csf$. Consider the following program with respect to D:

$$\begin{aligned}&\{x{:}sf\} \vdash x[severity \twoheadrightarrow high]\\ &\{x{:}csf\} \vdash x[severity \twoheadrightarrow low]\end{aligned}$$

Under the partitioning described in [44], in which typing is not considered, the ground instances of these two clauses could be in the same level. A possible model for this program would then be $\{failure[severity \twoheadrightarrow high]\}$. Using the ordering described in Section 3.5, the model of this program is $\{failure[severity \twoheadrightarrow low]\}$ as expected. □

Let $P_1^t, \ldots, P_n^t$ be a partitioning on the ground (clause, type) pairs of program P with respect to a set of declarations D. Then an interpretation, M_P^*, is computed using the immediate consequence operator to find a fixpoint of T_{P_1}, T_{P_2} and so on. (For simplicity, we abbreviate $T_{P_i^t}$ to T_{P_i}.)

$$\begin{aligned}&M_1' = T_{P_1}\!\uparrow\!\omega(\emptyset),\\ &M_i' = T_{P_i}\!\uparrow\!\omega(M_{i-1}'), \text{ for } 1 < i \leq n\\ &M_n = \{A \mid (A,t) \in M_n'\}\\ &M_P^* = M_n.\end{aligned}$$

Each M_i' is a fixpoint of T_{P_i}, and we show below that M_P^* is a rather special model of P.

Example 4.1.4 Assume the set of declarations D including $psf < sf$ and $failure{:}psf$. Consider program P with respect to D:

$$\{x{:}sf\} \vdash x[pressure \rightarrow high]$$
$$\{x{:}sf\} \vdash x[action \twoheadrightarrow \text{“call supervisor”}]$$
$$\{x{:}psf\} \vdash x[action \twoheadrightarrow \text{“increase pressure”}] \leftarrow \neg x[pressure \rightarrow high]$$

One of the possible i-stratifications of P gives the following ordering on ground (clause, type) pairs:

$$P_1^t = \{(failure[pressure \rightarrow high], sf)\}$$
$$P_2^t = \{(failure[action \twoheadrightarrow \text{“increase pressure”}] \leftarrow \neg failure[pressure \rightarrow high], psf)\}$$
$$P_3^t = \{(failure[action \twoheadrightarrow \text{“call supervisor”}], sf)\}$$

M_P^* is computed as follows:

$$T_{P_1} \uparrow 0(\emptyset) = \emptyset$$
$$M_1' = T_{P_1} \uparrow \omega(\emptyset) = \{(failure[pressure \rightarrow high], sf)\}$$

$$T_{P_2} \uparrow 0(M_1') = \{(failure[pressure \rightarrow high], sf)\}$$
$$M_2' = T_{P_2} \uparrow \omega(M_1') = \{(failure[pressure \rightarrow high], sf)\}$$

$$T_{P_3} \uparrow 0(M_2') = \{(failure[pressure \rightarrow high], sf)\}$$
$$M_3' = T_{P_3} \uparrow \omega(M_2') = \{(failure[pressure \rightarrow high], sf), (failure[action \twoheadrightarrow \text{“call supervisor”}], sf)\}$$

$$M_P^* = \{failure[pressure \rightarrow high], failure[action \twoheadrightarrow \text{“call supervisor”}]\} \quad \Box$$

Note that each M_j is a model for $\bigcup_{i \leq j} P_i$, and that M_P^* is a model of P, indeed the preferred model of P, in this example. In fact, these properties hold for all simple programs, and the following key result holds.

Theorem 4.1.1 *(Soundness and completeness) Let P be a simple program with respect to a set of declarations D. Then M_P^* does not depend on the particular i-stratification chosen for P with respect to D, and M_P^* is the preferred model M_P of P with respect to D.*

Proof The construction of M_P in Theorem 3.5.1 is precisely the construction of the model M_P^*, so M_P^* is equal to M_P. In Theorem 3.5.2 we proved that this construction is independent of the chosen i-stratification. $\Box$

Thus this procedure is sound and complete with respect to the declarative semantics. However, a problem with this procedure is the difficulty of assigning strata to the ground instances of clauses before starting the computation, and we are investigating means of avoiding this task. Another problem is that, like other naive bottom-up evaluation procedures, this one is not goal directed, indicating the need for optimizations perhaps based on program transformation and suggesting the investigation of goal directed top-down procedures such as those considered in the next section.

4.2 Top-down evaluation

In this section we outline an alternative evaluation procedure that might be more efficient than the bottom-up evaluation procedure described in the previous section. This top-down procedure is based on typed unification and a variant of SLDNF-resolution [77]. The resolution procedure differs from SLDNF-resolution as ground computed answers for a selected subgoal are computed, filtered to eliminate overridden answers, and then substituted in the rest of the goal. The answers derived by this procedure are ground.

Example 4.2.1 Assume a set of declarations D including $psf < sf$ and $failure{:}psf$. Consider simple program P with respect to D that has only functional methods.

$$\{x{:}sf\} \vdash x[action \rightarrow \text{"call supervisor"}]$$
$$\{x{:}psf\} \vdash x[action \rightarrow \text{"lower pressure"}]$$

The answer to goal $\{y{:}string\} \vdash \leftarrow\{failure[action \rightarrow y]\}$ could be substitution $\{y{:}string\} \vdash \{y/\text{"call supervisor"}\}$ or $\{y{:}string\} \vdash \{y/\text{"lower pressure"}\}$. There must be some mechanism for deciding which is the overriding answer and hence computed answer. □

We start by describing the unification procedure.

Definition Let S be a finite set of simple expressions, where a simple expression is a term, a method atom, or a predicate atom. A substitution θ is called a *unifier* for S if $S\theta$ is a singleton. A unifier θ for S is called a *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

Definition Let S be a finite set of simple expressions $\{\Gamma_1 \vdash E_1, \ldots, \Gamma_n \vdash E_n\}$ where the variables in each E_i are disjoint. The *disagreement set*, $D_k = \Gamma \vdash D'_k$, of S is defined as follows. Locate the leftmost symbol position at which not all E_i, $1 \leq i \leq n$, have the same symbol and extract from each expression in S the symbol at that position. Let D'_k be the set of all such symbols, and Γ be the variable typing of the variables in D'_k.

The unification algorithm takes a finite set of simple expressions S with respect to a set of declarations D as input, and outputs the most general unifier if the set is unifiable, otherwise it reports that the set is not unifiable. It is defined as follows:

1. Put $k = 0$, and the substitution $\sigma_0 = \varepsilon$.

2. If $S\sigma_k$ is a singleton then stop and σ_k is the mgu of S; otherwise find the disagreement set $D_k = \Gamma \vdash D'_k$ of $S\sigma_k$.

3. Let v and t be in D'_k. If $\Gamma \vdash v{:}\tau$ and $\Gamma \vdash t{:}\tau'$ are terms, v is a variable, and $\tau' \leq_t \tau$ with respect to D then put $\sigma_{k+1} = \sigma_k(\Gamma \vdash \{v/t\})$; otherwise if v and t are both variables and the greatest lower bound (glb) of (τ, τ') with respect to D is not $\perp$, then $\Gamma' = \Gamma - \{t{:}\tau'\} \cup \{t{:}glb(\tau,\tau')\}$ and $\sigma_{k+1} = \sigma_k(\Gamma' \vdash \{v/t\})$. In either case, increment k and go to step 2. If neither case holds, stop and report that S is not unifiable.

Note that because there are no function symbols in our language, no occur check is necessary.

Example 4.2.2 Assume the set of declarations D including $tf < cf$, $tf < fsf$, $failure{:}fsf$. Let S_1 be the set of expressions $\{\{x{:}cf, y{:}string\} \vdash x[problem \rightarrow y], failure[problem \rightarrow \text{"valve failure"}]\}$ to be unified. Then the disagreement set D_0 is $\{x{:}cf\} \vdash \{x, failure\}$. Because the type of $failure$ is fsf, and fsf is not less than or equal to cf with respect to D, S is not unifiable.

Next, let S_2 be the set of expressions $\{\{x{:}cf, y{:}string\} \vdash x[problem \rightarrow y], \{x'{:}fsf\} \vdash x'[problem \rightarrow \text{"valve failure"}]\}$ to be unified. Now the disagreement set D_0 is $\{x{:}cf, x'{:}fsf\} \vdash \{x, x'\}$, and the unifier σ_1 is $\{x{:}cf, x'{:}tf\} \vdash \{x/x'\}$. The next disagreement set D_1 is $\{y{:}string\} \vdash \{y, \text{"valve failure"}\}$, and the (most general) unifier σ_2 is $\{x{:}cf, x'{:}tf, y{:}string\} \vdash \{x/x', y/\text{"valve failure"}\}$. Thus, $S_2\sigma_2$ is $\{\{x'{:}tf\} \vdash x'[problem \rightarrow \text{"valve failure"}]\}$. □

We can now introduce our top-down evaluation procedure. It is similar to SLDNF-resolution except that our procedure must deal with answers that are only possibly true for method atoms, that is, answers that could be overridden. For each selected literal, a set of possible answers is computed, and a minimization operation is applied to find which of the possible answers have not been overridden.

We define the "standard form" of an atom where the value of a method atom is a variable. If an atom in a goal is $\Gamma \vdash \leftarrow t[m@t_1, \ldots, t_n \rightarrow b]$, where b is an object symbol, this atom may unify with the head of a clause that is overridden by another clause and hence return an incorrect answer. To ensure that only correct answers are computed, the value of each method atom in the goal must be a variable. We illustrate this problem and a solution in the following example.

Example 4.2.3 Assume a set of declarations D including $failure{:}psf$ and $psf < sf$. Consider the program P with respect to D:

$$\{x{:}sf\} \vdash x[treatment \twoheadrightarrow \text{"check readings"}] \tag{4.1}$$
$$\{x{:}sf\} \vdash x[treatment \twoheadrightarrow \text{"call supervisor"}] \tag{4.2}$$
$$\{x{:}psf\} \vdash x[treatment \twoheadrightarrow \text{"lower pressure"}] \leftarrow pressure(x, high) \tag{4.3}$$
$$\{x{:}psf\} \vdash x[treatment \twoheadrightarrow \text{"raise pressure"}] \leftarrow \neg pressure(x, high) \tag{4.4}$$
$$\{x{:}psf\} \vdash x[treatment \twoheadrightarrow \text{"call supervisor"}] \tag{4.5}$$
$$pressure(failure, low) \tag{4.6}$$

Consider program P and the goal $G \leftarrow failure[treatment \twoheadrightarrow \text{"check readings"}]$. In the procedure we are about to describe it is necessary to obtain all the answers to $P \cup \{\{y{:}string\} \vdash failure[treatment \twoheadrightarrow y]\}$, find the minimum answer with respect to typing, and test if $y =$ "check readings". If the atom in G is not rewritten in standard form, the atom unifies only with the head of clause 4.1 in program P. This is the minimum answer with respect to typing, so the goal is true. When overriding is taken into account, this goal should obviously not be true with respect to program P and declarations D. □

A method atom is in standard form if it has the form $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow x]$ (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow x]$), where $\Gamma \vdash t$, $\Gamma \vdash t_1, \ldots, \Gamma \vdash t_n$ are terms and x is a variable. Every predicate atom, every equality atom, and every negative literal is in standard form. If an atom is in standard form we call it a standard atom.

A positive literal not in standard form can easily be rewritten as the conjunction of two standard positive literals. Let A be a non-standard positive literal $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow a]$ (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow a]$), where a is an object. Let x be a variable, the variable typing Γ' be $\Gamma \cup \{x : \tau\}$ where a is of type τ, and $=$ the equality predicate. Then $\Gamma' \vdash t[m@t_1, \ldots, t_n \rightarrow x] \wedge x = a$ (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow x] \wedge x = a$) is the standard form of A.

Example 4.2.4 Consider the declarations, and goal G in Example 4.2.3. The standard form of goal G is $\{y{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow y] \wedge y =$ "check readings". □

We assume that R is a computation rule that selects literals such that equality and negative literals are not selected unless they are ground. The ground equality atoms and ground negative literals act as filters rather than answer generators. The condition that the negative literals are evaluated last is like the safeness condition in SLDNF.

Before we describe the top-down evaluation procedure, we informally describe different kinds of answers. We refer to three kinds of answers in the following description, computed answers, possible answers and atom answers. Answers are substitutions. Let P be a program with respect to a set of declarations D and $\Gamma \vdash L$ be a positive literal. Informally, the possible answers for $P \cup \{\Gamma \vdash \leftarrow L\}$ are all the possible answers for $P \cup \{\Gamma \vdash \leftarrow L\}$ without taking overriding into account. The atom answers for $P \cup \{\Gamma \vdash \leftarrow L\}$ are the minimal possible answers for $P \cup \{\Gamma \vdash \leftarrow L\}$ with respect to typing. The minimization eliminates overridden answers. Let G be a goal. The computed answers for $P \cup \{G\}$ are a combination of the atom answers for the literals in G. The definitions of computed answer and atom answer are mutually recursive. Assume initially that there is a way to find the atom answers for $P \cup \{\Gamma \vdash \leftarrow L\}$ where $\Gamma \vdash L$ is a positive literal. (We will define such a way later.)

We start by defining a failed goal tree and a goal tree.

Definition Let P be a program with respect to a set of declarations D and $G_i = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a goal. Then G_{i+1} is *derived* from G_i using substitution θ_{i+1} if the following conditions hold:

- $\Gamma \vdash L_m$ is a positive literal in G_i called the selected atom.
- θ_{i+1} is an atom answer for $P \cup \{\Gamma \vdash \leftarrow L_m\}$.
- G_{i+1} is the goal $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta_{i+1}$.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *goal refutation of rank 0* of $P \cup \{G\}$ consists of a sequence $G_0 = G, G_1, \ldots, G_n = \Box$ of goals, and a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that for each i, G_{i+1} is derived from G_i with substitution θ_{i+1}.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *failed goal tree of rank 0* for $P \cup \{G\}$ is defined as follows:

- Each node of the tree is a non-empty goal.
- The root node is G.
- Only positive literals are selected.
- Let $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a non-leaf node in the tree and suppose that $\Gamma \vdash L_m$ is selected. For each atom answer θ for $P \cup \{\Gamma \vdash \leftarrow L_m\}$, this node has a child that is $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$.
- Let $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a leaf node in the tree and suppose that $\Gamma \vdash L_m$ is selected. There are no atom answers for $P \cup \{\Gamma \vdash \leftarrow L_m\}$.

Example 4.2.5 Consider program P with respect to the set of declarations D in Example 4.2.3. Let goal G be $\{y{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow y] \wedge y =$ "check readings". Figure 4.1 is a failed goal tree of rank 0 for $P \cup \{G\}$. Goal "call supervisor" = "check readings" is derived from G using substitution $\theta = \{y{:}string\} \vdash \{y/\text{"call supervisor"}\}$, which is an atom answer for $P \cup \{\{y{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow y]\}$. It may seem surprising that these are the only atom answers. They are the answers computed using the clauses of the most specific type. Later we describe a way to find atom answers. $\Box$

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *goal refutation of rank k+1* of $P \cup \{G\}$ consists of a sequence $G_0 = G, G_1, \ldots, G_n = \Box$ of goals, and a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that for each i, either

- G_i is $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$, the selected literal $\Gamma \vdash L_m$ in G_i is a ground negative literal $\Gamma \vdash \neg A_m$, there is a failed goal tree of rank k for $P \cup \{\Gamma \vdash \leftarrow A_m\}$, θ_{i+1} is the identity substitution, and G_{i+1} is $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$; or

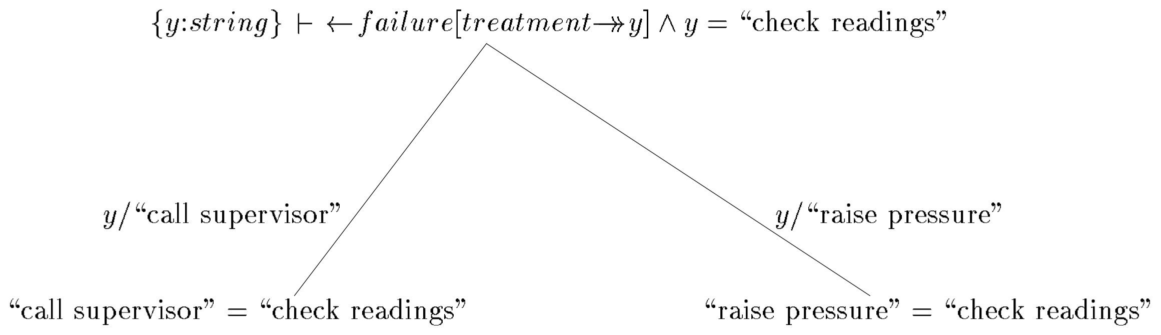
$\{y{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow y] \wedge y =$ "check readings"
$y/$"call supervisor"
$y/$"raise pressure"
"call supervisor" = "check readings"
"raise pressure" = "check readings"

Figure 4.1: Failed goal tree of rank 0

- G_{i+1} is derived from G_i with substitution θ_{i+1}.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *failed goal tree of rank k+1* for $P \cup \{G\}$ is defined as follows:

- Each node of the tree is a non-empty goal.
- The root of the tree is G.
- Let $G' = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a non-leaf node of the tree where $\Gamma \vdash L_m$ is selected, either
  - $\Gamma \vdash L_m$ is a positive literal, and for each atom answer θ for $P \cup \{\Gamma \vdash \leftarrow L_m\}$, G' has a child that is $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$, or
  - $\Gamma \vdash L_m$ is a negative literal $\Gamma \vdash \neg A_m$, there is a failed goal tree of rank k for $P \cup \{\Gamma \vdash \leftarrow A_m\}$, and G' has a child $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$.
- Let $G'' = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a leaf node of the tree where $\Gamma \vdash L_m$ is selected, either
  - $\Gamma \vdash L_m$ is a positive literal and there are no atom answers for $P \cup \{\Gamma \vdash \leftarrow L_m\}$, or
  - $\Gamma \vdash L_m$ is a negative literal $\Gamma \vdash \neg A_m$ and there is a goal refutation of rank k for $P \cup \{\Gamma \vdash \leftarrow A_m\}$.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *goal refutation* of $P \cup \{G\}$ is a goal refutation of rank k of $P \cup \{G\}$, for some positive integer k. A goal refutation of $P \cup \{G\}$ via R is a goal refutation of $P \cup \{G\}$ in which computation rule R is used to select atoms.

A *failed goal tree* for $P \cup \{G\}$ is a failed goal tree of rank k for $P \cup \{G\}$, for some positive integer k.

We make two observations. The first is each failed goal tree is finite. The second is that a failed goal tree (respectively, refutation) of rank k is also a failed goal tree (respectively, refutation) of rank j for all $j > k$. We can now define the computed answer for a program and goal.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *computed answer* for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to variables of G, where $\theta_1, \ldots, \theta_n$ is the sequence of substitutions used in a goal refutation of $P \cup \{G\}$ via R.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *goal derivation* of $P \cup \{G\}$ consists of a finite sequence $G_0 = G, G_1, \ldots, G_n$ of goals, and a sequence $\theta_1, \ldots, \theta_n$ of substitutions such that:

- For each $i < n$, G_i is $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$. Let $\Gamma \vdash L_m$ be the selected literal. Either
  - $\Gamma \vdash L_m$ is a positive literal, with an atom answer θ. In this case G_{i+1} is $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$, and θ_{i+1} is θ. Or
  - $\Gamma \vdash L_m$ is a negative literal $\Gamma \vdash \neg A_m$, and there is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A_m\}$. In this case G_{i+1} is $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$, and θ_{i+1} is the identity substitution.
- And either
  - G_n is empty,
  - $G_n = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$, $\Gamma \vdash L_m$ is the selected positive literal and there is no atom answer for $P \cup \{\Gamma \vdash \leftarrow L_m\}$, or
  - $G_n = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$, $\Gamma \vdash L_m$ is a ground negative atom $\Gamma \vdash \neg A_m$, $\Gamma \vdash L_m$ is selected and there is a goal refutation of $P \cup \{\Gamma \vdash \leftarrow A_m\}$.

A goal derivation of $P \cup \{G\}$ via R is a goal derivation of $P \cup \{G\}$ in which a computation rule R is used to select atoms.

Definition Let P be a program with respect to a set of declarations D, and G a goal. A *goal tree* for $P \cup \{G\}$ is defined as follows:

- The root of the tree is G.
- Each node of the tree is a (possibly empty) goal.
- Let $G' = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a non-leaf node of the tree where $\Gamma \vdash L_m$ is selected, either
  - $\Gamma \vdash L_m$ is a positive literal, and for each atom answer θ for $P \cup \{\Gamma \vdash \leftarrow L_m\}$, G' has a child $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$, or
  - $\Gamma \vdash L_m$ is a negative literal $\Gamma \vdash \neg A_m$, there is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A_m\}$ and G' has a child $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$.
- Let $G'' = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a leaf node of the tree where $\Gamma \vdash L_m$ is selected, either
  - $\Gamma \vdash L_m$ is a positive literal and there is no atom answer for $P \cup \{\Gamma \vdash \leftarrow L_m\}$, or
  - $\Gamma \vdash L_m$ is a negative literal $\Gamma \vdash \neg A_m$, and there is a goal refutation for $P \cup \{\Gamma \vdash \leftarrow A_m\}$.

A goal tree of $P \cup \{G\}$ via R is a goal tree of $P \cup \{G\}$ in which a computation rule R is used to select atoms.

Example 4.2.6 Consider program P with respect to the set of declarations D in Example 4.2.3, the goal

$$G = \{y{:}string, z{:}level\} \vdash \leftarrow failure[treatment \twoheadrightarrow y] \wedge pressure(failure, z)$$

and the goal tree in Figure 4.2. The root of the goal tree is $\{y{:}string, z{:}level\} \vdash \leftarrow failure[treatment \twoheadrightarrow y] \wedge pressure(failure, z)$. Initially, the atom $\{y{:}string, z{:}level\} \vdash failure[treatment \twoheadrightarrow y]$ is selected. Assume $\{\{y{:}string\} \vdash \{y/\text{“raise pressure”}\}, \{y{:}string\} \vdash \{y/\text{“call supervisor”}\}\}$ are the atom answers for $P \cup \{\{y{:}string, z{:}level\} \vdash \leftarrow failure[treatment \twoheadrightarrow y]\}$.

Now consider the goal $\{z{:}level\} \vdash \leftarrow pressure(failure, z)$. Assume the atom answer for $P \cup \{\{z{:}level\} \vdash \leftarrow pressure(failure, z)\}$ is $\{\{z{:}level\} \vdash \{z/low\}\}$.

One goal refutation of $P \cup \{G\}$ consists of a sequence of goals:

$$G_0 = \{y{:}string, z{:}level\} \vdash \leftarrow failure[treatment \twoheadrightarrow y] \wedge pressure(failure, z),$$
$$G_1 = \{z{:}level\} \vdash \leftarrow pressure(failure, z),$$
$$G_2 = \Box,$$

and a sequence of substitutions $\theta_1 = \{y{:}string\} \vdash \{y/\text{“raise pressure”}\}$, $\theta_2 = \{z{:}level\} \vdash \{z/low\}$. Then $\{y{:}string, z{:}level\} \vdash \{y/\text{“raise pressure”}, z/low\}$ and $\{y{:}string, z{:}level\} \vdash \{y/\text{“call supervisor”}, z/low\}$ are the computed answers for $P \cup \{G\}$. $\Box$

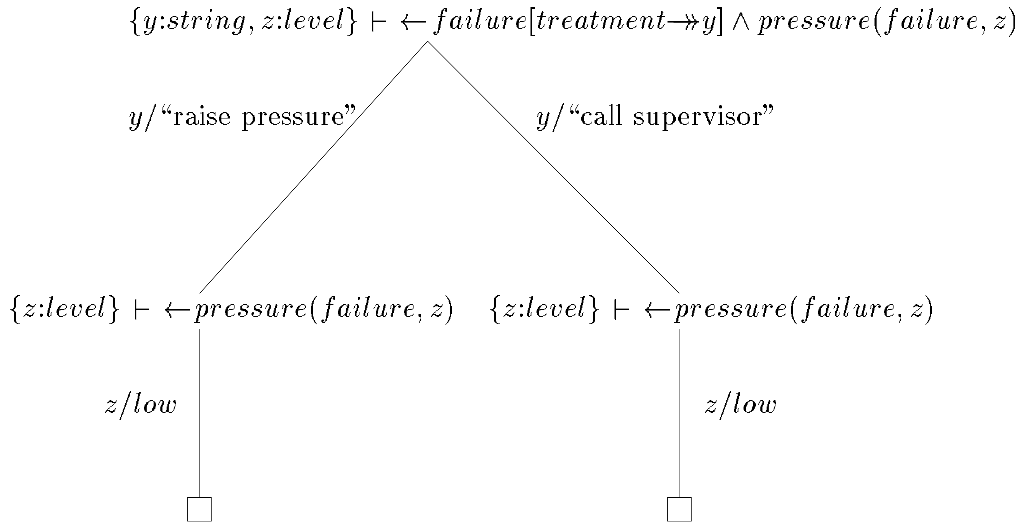
{y:string, z:level} ⊢ ←failure[treatment↠y] ∧ pressure(failure, z)
y/“raise pressure”
y/“call supervisor”
{z:level} ⊢ ←pressure(failure, z)
{z:level} ⊢ ←pressure(failure, z)
z/low
z/low

Figure 4.2: Goal tree for $P \cup \{G\}$

We have defined goal tree and computed answer assuming there is a definition for “atom answer”. We now define “atom answer”. Informally, atom answers are the minimal possible answers. Possible answers are all answers regardless of overriding. We now define possible answers and minimal possible answers.

Definition Let P be a program with respect to a set of declarations D, and $G = \Gamma_m \vdash \leftarrow A_m$ be a standard positive literal. Suppose $C = \Gamma \vdash A \leftarrow L_1 \wedge \cdots \wedge L_k$ is a clause of P where the variables of G and C are standardized apart. An *atom derivation* of $P \cup \{G\}$ consists of goals G, G', type τ, substitution θ and clause C where

- θ is the mgu of $\Gamma \vdash A_m$ and $\Gamma \vdash A$,

- G' is the goal $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_k)\theta$, and
- τ is the type of $\Gamma \vdash A$.

To determine the atom answers, it is necessary to compare possible answers. We can only compare ground instances of answers. We now define ground instances of (substitution, type) pairs, possible answers and atom answers.

Definition Let P be a program with respect to a set of declarations D. Suppose the substitution θ is $\Gamma \vdash \theta'$ and τ is a type. The *ground instances* of (θ, τ) are $\{(\sigma, \tau) \mid \sigma \in [\Gamma \vdash \theta']_D\}$, where $[\Gamma \vdash \theta']_D$ is the set of all ground instances of θ' consistent with Γ and D. We write $[\theta, \tau]_D$ for the set of ground instances of (θ, τ).

Example 4.2.7 Consider the program P with respect to the declarations D in Example 4.2.3 and the (substitution, type) pair $S = (\{x{:}psf, x'{:}psf\} \vdash \{x/x'\}, psf)$. The ground instance of S is $(\{x{:}psf, x'{:}psf\} \vdash \{x/failure, x'/failure\}, psf)$. $\square$

Definition Let P be a program with respect to a set of declarations D and $\Gamma \vdash A$ a standard positive literal. There is an atom derivation for each clause in P whose head unifies with $\Gamma \vdash A$. The goals derived are G_i, with type τ_i, and substitutions θ_i. For each goal G_i, suppose $\{\theta_{i1}, \ldots, \theta_{in_i}\}$ are the computed answers for $P \cup \{G_i\}$. Then the substitutions $\theta'_{i1} = \theta_i\theta_{i1}, \ldots, \theta'_{in_i} = \theta_i\theta_{in_i}$. Each (substitution, type) pair $[\theta'_{ij}, \tau_i]_D$ restricted to the variables of $\Gamma \vdash A$ is a *possible answer* for $P \cup \{\Gamma \vdash \leftarrow A\}$.

Definition Let P be a program with respect to a set of declarations D, $\Gamma \vdash A$ a standard positive literal, and Ψ a set of possible answers for $P \cup \{\Gamma \vdash \leftarrow A\}$. Let $A = t[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $A = t[m@t_1, \ldots, t_n \twoheadrightarrow t']$). A possible answer $\psi_1 = (\theta_1, \tau_1) \in \Psi$ is *smaller* than another possible answer $\psi_2 = (\theta_2, \tau_2) \in \Psi$ if $A\theta_1$ conflicts with $A\theta_2$, and $\tau_1 <_t \tau_2$ with respect to declarations D. In this case we write $\psi_1 < \psi_2$. We say that θ_k is *minimal* with respect to Ψ where $\psi_k = (\theta_k, \tau_k) \in \Psi$ and there is no $\psi_j \in \Psi$ such that $\psi_j < \psi_k$. Let $(\theta, \tau) \in \Psi$ be a possible answer for $P \cup \{\Gamma \vdash \leftarrow A\}$ where $\Gamma \vdash A$ is a predicate atom, θ is *minimal* with respect to Ψ.

Definition Let P be a program with respect to a set of declarations D.

- Suppose $\Gamma \vdash A$ is a positive literal in standard form. Then an *atom answer* for $P \cup \{\Gamma \vdash \leftarrow A\}$ is a minimal possible answer for $P \cup \{\Gamma \vdash \leftarrow A\}$.
- Otherwise, suppose $\Gamma \vdash A$ is a positive literal not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash A$, then an *atom answer* for $P \cup \{\Gamma \vdash \leftarrow A\}$ is a computed answer for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ restricted to the variables in $\Gamma \vdash A$.

An unrestricted answer is an atom answer in which arbitrary unifiers are used rather than most general unifiers.

First consider an example that does not involve negation.

Example 4.2.8 Assume the set of declarations D including $failure{:}psf$ and $psf < sf$. Consider program P with respect to the set of declarations D:

$\{x{:}sf\} \vdash x[treatment \twoheadrightarrow$ "check readings"$]$
$\{x{:}psf\} \vdash x[treatment \twoheadrightarrow$ "call supervisor"$]$

and the goal $G = \leftarrow failure[treatment \twoheadrightarrow$ "check readings"$]$. The atom answers for $P \cup \{G\}$ are the computed answers of $P \cup \{\{x{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow x] \wedge x =$

"check readings"}. The goal tree for $P \cup \{\{x{:}string\} \vdash\ \leftarrow failure[treatment \twoheadrightarrow x] \wedge x =$ "check readings"} via R is shown in Figure 4.3. The possible answers for $P \cup \{\{x{:}string\} \vdash failure[treatment \twoheadrightarrow x]\}$ are

$$\{(\{x{:}string\} \vdash \{x/\text{“call supervisor”}\}, psf),$$
$$(\{x{:}string\} \vdash \{x/\text{“check readings”}\}, sf)\}.$$

Obviously the minimal answer is $\{x{:}string\} \vdash \{x/$"call supervisor"$\}$, and there is no refutation of $P \cup \{G\}$. $\Box$

$$\begin{tikzcd}
\{x{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow x] \wedge x = \text{“check readings”} \arrow[d, dash, "x/\text{“call supervisor”}"] \\
\text{“call supervisor”} = \text{“check readings”}
\end{tikzcd}$$

Figure 4.3: Goal tree for $P \cup \{G\}$

Next consider an example that does involve negation.

Example 4.2.9 In Example 4.2.6, it was assumed there was a way to compute atom answers. We now describe how the atom answers are computed. Consider the declarations D and program P in Example 4.2.3, and the atom $A = \{y{:}string\} \vdash failure[treatment \twoheadrightarrow y]$.
Goal $\Box$, type sf, and substitution $\{y{:}string\} \vdash \{y/$"check readings"$\}$ are derived from A and clause 4.1.
Goal $\Box$, type sf, and substitution $\{y{:}string\} \vdash \{y/$"call supervisor"$\}$ are derived from A and clause 4.2.
Goal $\leftarrow$*pressure(failure, high)*, type *psf*, and substitution $\{y{:}string\} \vdash \{y/$"lower pressure"$\}$ are derived from A and clause 4.3.
There are no clauses in P that unify with $pressure(failure, high)$, so the computed answers for $P \cup \{\leftarrow pressure(failure, high)\}$ are $\emptyset$.
Goal $\leftarrow \neg$*pressure(failure, high)*, type *psf*, and substitution $\{y{:}string\} \vdash \{y/$"raise pressure"$\}$ are derived from A and clause 4.4.
There is a failed goal tree for $P \cup \{\leftarrow pressure(failure, high)\}$, so the computed answer for $P \cup \{\leftarrow \neg pressure(failure, high)\}$ is ε.
Goal $\Box$, type psf, and substitution $\{y{:}string\} \vdash \{y/$"call supervisor"$\}$ are derived from A and clause 4.5.
Thus, the possible answers for $P \cup \{\{y{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow y]\}$ are

$$\{(\{y{:}string\} \vdash \{y/\text{“check reading”}\}, sf), (\{y{:}string\} \vdash \{y/\text{“call supervisor”}\}, sf),$$
$$(\{y{:}string\} \vdash \{y/\text{“raise pressure”}\}, psf), (\{y{:}string\} \vdash \{y/\text{“call supervisor”}\}, psf)\}.$$

The atom answers $\{\{y{:}string\} \vdash \{y/$"raise pressure"$\}, \{y{:}string\} \vdash \{y/$"call supervisor"$\}\}$, are the minimal possible answers for $P \cup \{\{y{:}string\} \vdash \leftarrow failure[treatment \twoheadrightarrow y]\}$. $\Box$

In the theorem that proves the soundness of this top-down query evaluation procedure, we use proof by induction on the rank of a goal tree. We first define the rank of a goal tree. Note that this definition for the rank of a goal tree is different from the definition for the rank of a failed goal tree. However, both definitions capture the notion of depth of subtrees.

Definition Let P be a program with respect to a set of declarations D, and G a goal. The *rank* of a goal tree for $P \cup \{G\}$ is 0 if $G = \Box$.

Definition Let P be a program with respect to a set of declarations D, $G = \Gamma \vdash\ \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_n$ be a goal, and $\Gamma \vdash L_m$ be the selected literal. Suppose $\Gamma \vdash L_m$ is a positive literal in standard form that does not unify with any clause in P then the *rank* of a goal tree for $P \cup \{G\}$ is 1.

Definition Let P be a program with respect to a set of declarations D, and G a goal. The *rank* of a goal tree for $P \cup \{G\}$ is $j + 1$ if

- The root node is G.
- For every non-leaf node $\Gamma_i \vdash\ \leftarrow L_{i1} \wedge \cdots \wedge L_{ik_i}$, with selected literal $\Gamma_i \vdash L_{in}$ where $1 \leq n \leq k_i$, either
  - $\Gamma_i \vdash L_{in}$ is not in standard form. Let the standard form of $\Gamma_i \vdash L_{in}$ be $\Gamma' \vdash L'_1 \wedge L'_2$. Then the rank of the goal tree for $P \cup \{\Gamma' \vdash\ \leftarrow L'_1 \wedge L'_2\}$ is j_{in}.
  - $\Gamma_i \vdash L_{in}$ is a positive literal in standard form and the maximum rank of the resulting goal trees for clauses with which $\Gamma_i \vdash L_{in}$ unifies is j_{in}. Or
  - $\Gamma_i \vdash L_{in}$ is a negative literal $\Gamma_i \vdash \neg A_{in}$ and the rank of the goal tree for $P \cup \{\Gamma_i \vdash\ \leftarrow A_{in}\}$ is j_{in}.

  Then j is the maximum j_{in}.

Example 4.2.10 Consider the declarations and program P in Example 4.2.8 and the goal $G = \leftarrow failure[treatment \twoheadrightarrow \text{“check readings”}]$. The trees used to find the computed answers for $P \cup \{G\}$ are shown in Figure 4.3 and Figure 4.4.

First consider the tree in Figure 4.3. The atom $\{x{:}string\} \vdash failure[treatment \twoheadrightarrow x]$ unifies with both $\{x{:}sf\} \vdash x[treatment \twoheadrightarrow \text{“check readings”}]$ and $\{x{:}psf\} \vdash x[treatment \twoheadrightarrow \text{“call supervisor”}]$. The maximum rank of the trees resulting from these unifications is 0. Thus the rank of the goal tree for $P \cup \{\{x{:}string\} \vdash\ \leftarrow failure[treatment \twoheadrightarrow x] \wedge x = \text{“check readings”}\}$ is 1.

Now, consider the tree in Figure 4.4. The standard form of G is $G' = \{x{:}string\} \vdash\ \leftarrow failure[treatment \twoheadrightarrow x] \wedge x = \text{“check readings”}$. As we have just shown, the rank of the goal tree for $P \cup \{G'\}$ is 1, so the rank of the goal tree for $P \cup \{G\}$ is 2. $\Box$

$$\leftarrow failure[treatment \twoheadrightarrow \text{“check readings”}]$$

Figure 4.4: Goal tree for $P \cup \{\leftarrow failure[treatment \twoheadrightarrow \text{“check readings”}]\}$

Lemma 4.2.1 *Let P be a simple program with respect to a set of declarations D, and G a goal. Suppose the selected literal in G is positive.*

1. *If the set $\{G_1, \ldots, G_r\}$ of derived goals is non-empty, then $G \leftrightarrow G_1 \vee \cdots \vee G_r$ is true in M_P.*
2. *If there are no derived goals, then $\neg G$ is true in M_P.*

Proof Let G be the goal $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_n$ and $\Gamma \vdash L_m$ be the selected literal. Consider part 1. First, we want to prove that if G_i is true in M_P for any i, then G is true. There is a clause $\Gamma' \vdash L' \leftarrow L'_1 \wedge \cdots \wedge L'_j$ in P such that $(\Gamma' \vdash L')\theta = (\Gamma \vdash L_m)\theta$ that is not overridden by an instance of another clause in P with respect to D. Then $\Gamma \vdash L_1 \wedge \cdots \wedge L_n \leftarrow (\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L'_1 \wedge \cdots \wedge L'_j \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta$ is true in M_P. This result can be applied r times. Thus proving that if G_i is true in M_P for any i, then G is true.

Next, we want to prove that if G is true in M_P, then G_i is true for some i. If G is true in M_P then there is a clause $\Gamma' \vdash L' \leftarrow L'_1 \wedge \cdots \wedge L'_j$ in P such that $(\Gamma' \vdash L')\theta = (\Gamma \vdash L_m)\theta$ that is not overridden by an instance of another clause in P with respect to D and $(\Gamma' \vdash L'_1 \wedge \cdots \wedge L'_j)\theta$ is true in M_P. If this were not true, then we could remove $(\Gamma' \vdash L')\theta$ from M_P and still have a model. As M_P is minimal, this is a contradiction. Hence, G_i is true for some i.

Now consider part 2. We prove this by induction on the rank of the goal tree for $P \cup \{G\}$. Suppose the rank of the goal tree for $P \cup \{G\}$ is 1. As there are no derived goals, there are no clauses in P whose head unifies with $\Gamma \vdash L_m$ so $\neg G$ is true in M_P.

Assume the result holds for goal trees of rank $\leq k$.

Suppose the rank of the goal tree for $P \cup \{G\}$ is $k+1$. Either

- $\Gamma \vdash L_m$ is not in standard form. The standard form of $\Gamma \vdash L_m$ is $G' = \Gamma' \vdash A_1 \wedge A_2$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ which has rank $\leq k$. Using the induction hypothesis and part 1, $\neg G'$ is true in M_P, thus $\neg G$ is true in M_P. Or

- $\Gamma \vdash L_m$ is in standard form. For every clause with which $\Gamma \vdash L_m$ unifies, $\Gamma' \vdash L' \leftarrow L'_1 \wedge \cdots \wedge L'_j$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_j\}$ that has rank $\leq k$. Using the induction hypothesis and part 1, $\neg\exists(\Gamma' \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_j)$ is true in M_P, thus $\neg G$ is true in M_P. $\Box$

Lemma 4.2.2 *Let P be a simple program with respect to a set of declarations D, and G a goal. If there is a failed goal tree for $P \cup \{G\}$, then $\neg G$ is true in M_P.*

Proof The proof is by induction on the rank of the failed goal tree for $P \cup \{G\}$. Every leaf node in the tree for $P \cup \{G\}$ is non-empty. Let G be the goal $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_n$.

Suppose first that the rank of the failed goal tree is 0. Then only positive literals are selected. The result follows by induction on the depth of the tree, using Lemma 4.2.1.

Assume the result holds for failed goal trees of rank k.

Consider a failed goal tree of rank $k+1$ for $P \cup \{G\}$. We now do induction on the depth of the failed goal tree for $P \cup \{G\}$. Suppose first that the depth of this tree is 1. Either

- the selected literal in G is positive. Then the result follows from Lemma 4.2.1, part 2. Or

- the selected literal $\Gamma \vdash L_m$ is the ground negative literal, $\Gamma \vdash \neg A_m$. Since the depth of the goal tree is 1, there is a refutation of $P \cup \{\Gamma \vdash \leftarrow A_m\}$. Using Lemma 4.2.1, part 1 and applying the induction hypothesis on any failed goal trees of rank k in this refutation, we obtain that $\Gamma \vdash A_m$ is true in M_P. Hence $\neg\exists(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)$ is true in M_P. (This last step uses the fact that $\Gamma \vdash A_m$ is ground.)

Assume that the result holds for failed goal trees of depth d. Now suppose that the failed goal tree for $P \cup \{G\}$ has depth $d+1$. Either

- the selected literal in G is positive. The result follows from Lemma 4.2.1, part 1 and the induction hypothesis. Or

- the selected literal $\Gamma \vdash L_m$ is the ground negative literal, $\Gamma \vdash \neg A_m$. By the inner induction hypothesis, we obtain that $\neg\exists(\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)$ is true in M_P. Hence $\neg\exists(\Gamma \vdash L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_n)$ is true in M_P. $\square$

Theorem 4.2.1 *(Soundness) Let P be a simple program with respect to a set of declarations D, and G a goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

Proof Let G be the goal $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_n$ and $\theta_1, \ldots, \theta_n$ be the sequence of atom answers used in a goal refutation of $P \cup \{G\}$ via R. Let M_P be the preferred model of P. We want to show that $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P.

We use induction on the rank of the goal tree for $P \cup \{G\}$. Suppose the rank is 1. Then the restriction of P contains only facts, the literals in the tree are positive and are in standard form.

We use induction on the length of the goal refutation of $P \cup \{G\}$ via R. Suppose that the length is 1. Then G is the positive literal $\Gamma \vdash \leftarrow L_1$. The program has a clause $\Gamma \vdash L$ such that $(\Gamma \vdash L)\theta_1 = (\Gamma \vdash L_1)\theta_1$ where $(\Gamma \vdash L)\theta_1$ is not overridden by an instance of another clause in P. It follows that $(\Gamma \vdash L_1)\theta_1$ is true in M_P.

Assume the result holds where the length of the goal refutation of $P \cup \{G\}$ via R is $n-1$.

Suppose the length of the goal refutation of $P \cup \{G\}$ via R is n. Let $\Gamma \vdash L_m$ be the selected literal. Then $\Gamma \vdash L_m$ is a positive literal. There is an instance of a clause $\Gamma \vdash L$ in P with which it unifies that is not overridden by an instance of another clause in P. By the induction hypothesis $(\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P. Also $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n = (\Gamma \vdash L)\theta_1 \ldots \theta_n$ is true in M_P. Thus $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P.

Assume the result holds for goal trees for $P \cup \{G\}$ of rank $\leq q-1$.

Suppose the the rank of the goal tree for $P \cup \{G\}$ is q. Again we use induction on the length of the refutation of $P \cup \{G\}$ via R. Suppose the length is 1. Then G has the form $\Gamma \vdash \leftarrow L_1$. Either

- $\Gamma \vdash L_1$ is a positive literal. Either
  - $\Gamma \vdash L_1$ is a standard positive literal. Then there is an instance of a clause $\Gamma \vdash L \leftarrow L'_1 \wedge \cdots \wedge L'_k$ in P such that $(\Gamma \vdash L_1)\theta' = (\Gamma \vdash L)\theta'$ that is not overridden by an instance of another clause in P. Then either
    - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $< q-1$. Let $\theta_1 = \theta'$. Then by the main induction hypothesis $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
    - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $q-1$. Then there is a refutation of $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \ldots \wedge L'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_k$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_k$. By the main induction hypothesis, $(\Gamma \vdash L'_1 \wedge \cdots \wedge L'_k)\theta_1$ is true in M_P, thus $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
  - $\Gamma \vdash L_1$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L_1$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ of rank $\leq q-1$. There is a refutation of $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ via R with computed answer θ_1. By the main induction hypothesis, $(\Gamma' \vdash A_1 \wedge A_2)\theta_1$ is true in M_P, thus $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
- $\Gamma \vdash L_1$ is a negative literal, $\Gamma \vdash \neg A$. There is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A\}$ with substitution $\theta_1 = \varepsilon$. By Lemma 4.2.2, it follows that $(\Gamma \vdash L_1)\theta_1$ is true in M_P.

Assume the result holds for refutations of $P \cup \{G\}$ via R of length $n-1$.

Suppose the length of the refutation of $P \cup \{G\}$ via R is n. Then G has the form $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_n$. Let $\Gamma \vdash L_m$ be the selected literal. Either

- $\Gamma \vdash L_m$ is a positive literal. Either
  - $\Gamma \vdash L_m$ is a standard positive literal. Then there is an instance of a clause $\Gamma \vdash L \leftarrow L'_1 \wedge \cdots \wedge L'_k$ in P such that $(\Gamma \vdash L_m)\theta' = (\Gamma \vdash L)\theta'$ that is not overridden by an instance of another clause in P. Then either
    - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $< q-1$. Let $\theta_1 = \theta'$. Then by the main induction hypothesis $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P. Or
    - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $q-1$. Then there is a refutation of $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_j$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_j$. By the main induction hypothesis $(\Gamma \vdash \leftarrow L'_1 \wedge \ldots \wedge L'_k)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P.
  - $\Gamma \vdash L_m$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L_m$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ of rank $\leq q-1$. There is a refutation of $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ via R with computed answer θ_1. By the induction hypothesis, $(\Gamma' \vdash A_1 \wedge A_2)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P. Or
- $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A$. There is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A\}$ with substitution $\theta_1 = \varepsilon$. From Lemma 4.2.2, it follows that $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P.

In either case, by the induction hypothesis $(\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P. □

The top-down evaluation procedure is not complete because it does not always terminate. There is a less restrictive condition called partial completeness [62], in which all correct answers are computed but the procedure may not terminate. We conjecture that the top-down evaluation procedure is partially complete.

Example 4.2.11 Assume a set of declarations D. Consider the following program P with respect to the declarations D:

$$edge(a,b)$$
$$edge(b,c)$$
$$\{x{:}node, y{:}node\} \vdash path(x,y) \leftarrow edge(x,y)$$
$$\{x{:}node, y{:}node, z{:}node\} \vdash path(x,y) \leftarrow edge(x,z) \wedge path(z,y).$$

Consider the goal $\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)$. There is an atom derivation for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)\}$ where $G = \{\{x'{:}node, y'{:}node\} \vdash \leftarrow edge(x',y')\}$, $\theta = \{x'{:}node, y'{:}node, x{:}node, y{:}node\} \vdash \{x/x', y/y'\}$, $\tau = undefined$. There is another atom derivation for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)\}$ where $G = \{\{x'{:}node, y'{:}node, z{:}node\} \vdash \leftarrow edge(x',z) \wedge path(z,y')\}$, $\theta = \{x'{:}node, y'{:}node, x{:}node, y{:}node\} \vdash \{x/x', y/y'\}$, $\tau = undefined$. In the goal tree for $P \cup \{\{x'{:}node, y'{:}node, z{:}node\} \vdash \leftarrow edge(x',z) \wedge path(z,y')\}$, let the selected atom be $\{y'{:}node, z{:}node\} \vdash path(z,y')$. There are again two atom derivations for $P \cup \{\{z{:}node, y'{:}node\} \vdash \leftarrow path(z,y')\}$. They are variants of the above. The goal trees are shown in Figure 4.5. Obviously if the *path* atom is selected each time, there are an infinite number of trees. □

In this procedure all the trees are finite but there may be an infinite number of them. This is different from an SLDNF-tree, which may be infinite.

This evaluation procedure is similar to the SLDNF procedure. The SLDNF procedure with a post check to find the minimal conflicting answer would provide correct answers, but

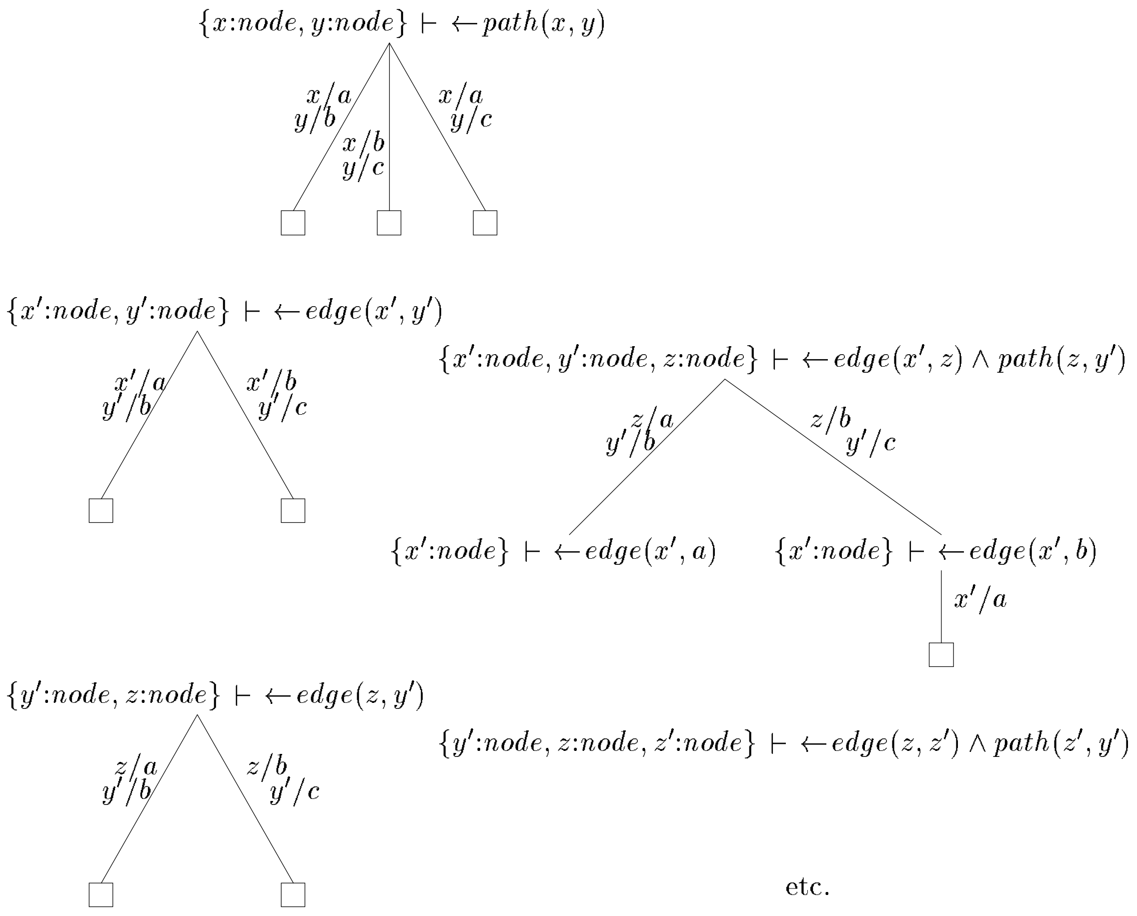
$\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)$
x/a y/b
x/b y/c
x/a y/c
$\{x'{:}node, y'{:}node\} \vdash \leftarrow edge(x',y')$
x'/a y'/b
x'/b y'/c
$\{x'{:}node, y'{:}node, z{:}node\} \vdash \leftarrow edge(x',z) \wedge path(z,y')$
z/a y'/b
z/b y'/c
$\{x'{:}node\} \vdash \leftarrow edge(x',a)$
$\{x'{:}node\} \vdash \leftarrow edge(x',b)$
x'/a
$\{y'{:}node, z{:}node\} \vdash \leftarrow edge(z,y')$
z/a y'/b
z/b y'/c
$\{y'{:}node, z{:}node, z'{:}node\} \vdash \leftarrow edge(z,z') \wedge path(z',y')$
etc.

Figure 4.5: Goal trees for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)\}$

would be less efficient than the procedure we propose because possible answers would not be eliminated until the post check is done.

There are various ways in which the procedure described in this section can be optimized. In the atom derivation, rather than unifying the atom with all the matching clauses in the program, it is possible to unify the atom with the clause of the least type. If this does not lead to a refutation, then the atom should be unified with the clause of the next smallest type until it leads to a refutation.

4.3 Top-down evaluation with tabling

Currently the top-down evaluation of a recursive program that has a finite number of solutions may enter an infinite loop by searching an infinite path. Consider Example 4.2.11.

In logic programming, techniques have been introduced to alter the top-down evaluation procedure to handle recursive programs. Two similar techniques are Extension Tables [42] and QSQR [18, 107]. QSQR has been extended to QSQR/SLS [62] for normal queries and stratified normal databases. Our procedure is based on QSQR/SLS. It differs from QSQR/SLS because our underlying top-down evaluation procedure is different from SLDNF-resolution due to inheritance and overriding.

In this procedure, as in QSQR/SLS, we identify recursive atoms. However, because of the presence of inheritance and overriding, we also define "subatom". The presence of subatoms makes our definition of recursive atoms different from the definition in [62]. We now define the terms *subatom* and *recursive atom* before describing our procedure.

Definition Given a set of declarations D, an atom $\Gamma \vdash t[m@s_1,\ldots,s_n \rightarrow s]$ is a *subatom* of $\Gamma' \vdash t'[m@s'_1,\ldots,s'_n \rightarrow s']$ (respectively, $\Gamma \vdash t[m@s_1,\ldots,s_n \twoheadrightarrow s]$ is a *subatom* of $\Gamma' \vdash t'[m@s'_1,\ldots,s'_n \twoheadrightarrow s']$) if $\Gamma \vdash t{:}\tau_t$, $\Gamma \vdash s_1{:}\tau_{s_1},\ldots,\Gamma \vdash s_n{:}\tau_{s_n}$, $\Gamma \vdash s{:}\tau_s$, $\Gamma' \vdash t'{:}\tau_{t'}$, $\Gamma' \vdash s'_1{:}\tau_{s'_1},\ldots,\Gamma' \vdash s'_n{:}\tau_{s'_n}$, and $\Gamma' \vdash s'{:}\tau_{s'}$ are terms and $\tau_t <_t \tau_{t'}, \tau_{s_1} \leq_t \tau_{s'_1},\ldots,\tau_{s_n} \leq_t \tau_{s'_n}, \tau_s \leq_t \tau_{s'}$ with respect to D.

If atom a_1 is a subatom of a_2 then atom a_2 is a *superatom* of a_1.

Example 4.3.1 Assume a set of declarations D including *special_node*<*node*, *d*:*special_node*, and *a*:*node*. The atom $\{x : \mathit{special\_node}\} \vdash x[\mathit{path} \twoheadrightarrow d]$ is a subatom of $\{x : \mathit{node}\} \vdash x[\mathit{path} \twoheadrightarrow a]$. The atom $\{x : \mathit{special\_node}\} \vdash x[\mathit{path} \twoheadrightarrow a]$ is not a subatom or a superatom of $\{x : \mathit{special\_node}\} \vdash x[\mathit{path} \twoheadrightarrow d]$. □

Recall that expressions E and F are variants if there are substitutions θ and σ such that $E = F\theta$ and $F = E\sigma$. Note that if atom B is a subatom of atom C then B and C are not variants.

A dependency graph can be used to identify recursive atoms. The dependency graph defined below is similar to that presented in [12]. However the nodes in this graph are atoms rather than predicate names. This is necessary because we consider typed methods as well as predicates.

Definition Given a program P with respect to a set of declarations D, we define a *dependency graph* as follows. The nodes of the graph correspond to the atoms that appear in P. Variables are renamed so that variant atoms correspond to a single node. A positive arc $\phi \overset{+}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there is a clause $\phi \leftarrow \cdots \wedge \psi \wedge \cdots$ in P, or if $\psi \leftarrow \cdots \wedge \phi \wedge \cdots$ in P, and ψ is a subatom of ϕ in P. A negative arc $\phi \overset{-}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there is a clause $\phi \leftarrow \cdots \wedge \neg\psi \wedge \cdots$ in P, or if there is a clause $\phi \leftarrow \cdots$, and another clause $\phi' \leftarrow \cdots \wedge \psi \wedge \cdots$ or $\phi' \leftarrow \ldots \wedge \neg\psi \wedge \ldots$ and ϕ' is a subatom of ϕ.

Example 4.3.2 Consider the following program that states that there is a path between a source and destination node if there is an edge between the source and destination node, or if there is an edge between the source node and another node and a path between the other node and the destination node, except in the case where the source node is a special node. In the case of a special node, there is only a path between the source and destination node if there is an edge between the source and destination node.

Assume a set of declarations D including $a{:}special\_node$, $b{:}node$, $c{:}node$, $d{:}node$ and $special\_node < node$. Consider the program P with respect to D:

$$a[edge \twoheadrightarrow b]$$
$$b[edge \twoheadrightarrow c]$$
$$c[edge \twoheadrightarrow d]$$
$$\{x{:}node, y{:}node\} \vdash x[path \twoheadrightarrow y] \leftarrow x[edge \twoheadrightarrow y]$$
$$\{x{:}node, y{:}node, z{:}node\} \vdash x[path \twoheadrightarrow y] \leftarrow x[edge \twoheadrightarrow z] \wedge z[path \twoheadrightarrow y]$$
$$\{x{:}special\_node, y{:}node\} \vdash x[path \twoheadrightarrow y] \leftarrow x[edge \twoheadrightarrow y]$$

The dependency graph is shown in Figure 4.6. □

$$\begin{tikzcd}
\{x{:}node, y{:}node\} \vdash x[path \twoheadrightarrow y] \arrow[loop left, "+"] & \{x{:}node, y{:}node\} \vdash x[edge \twoheadrightarrow y] \arrow[l, "+"'] \\
\{x{:}special\_node, y{:}node\} \vdash x[path \twoheadrightarrow y] & \{x{:}special\_node, y{:}node\} \vdash x[edge \twoheadrightarrow y] \arrow[l, "+"'] \arrow[ul, "-"']
\end{tikzcd}$$

Figure 4.6: Dependency graph for program P with respect to D

The following example illustrates that there is a cycle in the dependency graph of a program if there is a clause in the program where the atom in the head of the clause is a subatom of an atom in the body of the clause.

Example 4.3.3 Assume a set of declarations D including $a{:}special\_node$ and $special\_node < node$. Consider the program P with respect to D:

$$\{x{:}special\_node, y{:}node, z{:}node\} \vdash x[path \twoheadrightarrow z] \leftarrow y[path \twoheadrightarrow z] \wedge r(x, y)$$
$$r(a, a).$$

The dependency graph is shown in Figure 4.7. □

$$\begin{tikzcd}
\{x{:}special\_node, z{:}node\} \vdash x[path \twoheadrightarrow z] \arrow[r, bend left, "+"] & \{y{:}node, z{:}node\} \vdash y[path \twoheadrightarrow z] \arrow[l, bend left, "+"] \\
 & \{x{:}node, y{:}node\} \vdash r(x, y) \arrow[ul, "+"]
\end{tikzcd}$$

Figure 4.7: Dependency graph for program P with respect to D

Definition The set of *recursive atoms* (r-atoms) of P is the set of atoms that are involved in at least one cycle in the dependency graph for P. The atoms in P that are not in the r-atom set are in the *iterative atom* (i-atom) set. An r-atom (respectively, i-atom) is an atom in the r-atom (respectively, i-atom) set.

Example 4.3.4 First consider program P with respect to the declarations D in Example 4.3.2. The r-atom is $\{x{:}node, y{:}node\} \vdash x[path \twoheadrightarrow y]$.
The i-atoms are $\{\{x{:}node, y{:}node\} \vdash x[edge \twoheadrightarrow y], \{x{:}special\_node, y{:}node\} \vdash x[path \twoheadrightarrow y],$ $\{x{:}special\_node, y{:}node\} \vdash x[edge \twoheadrightarrow y], a[edge \twoheadrightarrow b], b[edge \twoheadrightarrow c], c[edge \twoheadrightarrow d]\}$.

Now consider program P with respect to the declarations D in Example 4.3.3. The r-atoms are $\{\{x{:}special\_node, z{:}node\} \vdash x[path \twoheadrightarrow z], \{y{:}node, z{:}node\} \vdash y[path \twoheadrightarrow z]\}$. The i-atoms are $\{\{x{:}node, y{:}node\} \vdash r(x, y), r(a, a)\}$. $\Box$

We now describe a top-down evaluation procedure with tabling. As in the top-down procedure described in Section 4.2, a forest of goal trees is used to find the computed answer for a goal with respect to a program. However, in the top-down procedure with tabling, there is more than one forest if there is a recursive atom in the program. In the first forest, all the computed answers are found for the program and goal without considering subgoals that contain a recursive atom. A table is used to store the computed answers for each atom from each iteration. In subsequent iterations, the computed answers from the table are used, in particular as answers for goals containing a recursive atom. The process is continued until there are no new computed answers. In the table, standard atoms and their computed answers with respect to the program are stored in the form of (standard atom, atom answer) pairs. In the forest, we distinguish between predecessor and successor atoms. We now define subtree, and predecessor and successor atoms. In the following definitions P is a program with respect to a set of declarations D, G is a goal, and S is a set of (standard atom, atom answer) pairs.

Definition A tree t' is a *subtree* of a tree t in a forest T if refutations from t' are used in derivations in t.

Example 4.3.5 In Figure 4.5, the tree for $P \cup \{\{x'{:}node, y'{:}node\} \vdash \leftarrow edge(x', y')\}$ is a subtree of the tree for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow path(x, y)\}$. $\Box$

In this procedure we initially recognize atoms that are called recursively. Such an atom is a predecessor atom the first time it is called. In the recursive call, it is a successor atom.

Definition Let A and B be atoms where B is a variant or a subatom of A and A and B are r-atoms with respect to P. Let t and t' be trees in a forest T where t' is a subtree of t. If A is a subgoal of a goal in t and B is a subgoal of a goal in t' then B is a *successor atom* with respect to T, and A is a *predecessor atom* with respect to T.

Example 4.3.6 Assume a set of declarations D including $special\_node < node$. Let B be the atom $\{x : special\_node, y{:}node\} \vdash x[path \twoheadrightarrow y]$ and A be the atom $\{z{:}node, y{:}node\} \vdash$ $z[path \twoheadrightarrow y]$ in P where A and B are r-atoms and B is a subatom of A. Suppose B is derived from A in T, then B is a successor atom and A is a predecessor atom. $\Box$

Recall the definitions of computed answer and possible answer from Section 4.2. In this section we redefine atom answer. Atom answers are defined with respect to a set of (atom, atom answer) pairs or table. There are two ways to derive atom answers. If the atom is an atom in the table, or a subatom or a variant of an atom in the table, then the answers are taken from the table. If the atom is not in the table, then the atom answers are derived as they are for the top-down procedure. We also define a forest with respect to a goal, a program and a set of (atom, atom answer) pairs. Then, we give a procedure for evaluating a goal with respect to a program, which involves iterating through a number of such forests with respect to a set of (atom, atom answer) pairs until no new atom answers are generated.

Definition Let $\Gamma \vdash L$ be a literal in forest T with respect to program P, a set of declarations D and a set of (atom, atom answer) pairs S. Either

- $\Gamma \vdash L$ is not a standard literal. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L$. The *atom answers* for $P \cup \{\Gamma \vdash \leftarrow L\}$ with respect to T and S are the computed answers for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$. Or
- $\Gamma \vdash L$ is in standard form. Either
  - $\Gamma \vdash L$ is not a predecessor atom and is a variant of an atom $\Gamma_B \vdash B$ in S. Then there is a pair $(\Gamma_B \vdash B, \Theta_B)$ in S, and the *atom answers* for $P \cup \{\Gamma \vdash \leftarrow L\}$ with respect to T and S are Θ_B.
  - $\Gamma \vdash L$ is not a predecessor atom and is a subatom of an atom $\Gamma_B \vdash B$ in S. Then there is a pair $(\Gamma_B \vdash B, \Theta_B)$ in S, and the *atom answers* for $P \cup \{\Gamma \vdash \leftarrow L\}$ with respect to T and S are each $\theta_B \in \Theta_B$ such that θ_B is consistent with Γ. Or
  - $\Gamma \vdash L$ is an atom that is not a variant or subatom of a literal in S, or a predecessor atom. The *atom answers* for $P \cup \{\Gamma \vdash \leftarrow L\}$ with respect to T and S are the minimal possible answers for $P \cup \{\Gamma \vdash \leftarrow L\}$.

Definition A *forest* T for goal G with respect to program P, declaration D and (atom, atom answer) pairs S is a forest of trees of goals such that

- The root of some tree t of T is G.
- Nodes that are the empty goal have no children.
- Let $G' = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_m \wedge \cdots \wedge L_k$ be a node of a tree where $\Gamma \vdash L_m$ is selected. Either
  - $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A_m$. If there is an atom answer for $P \cup \{\Gamma \vdash \leftarrow A_m\}$ with respect to T and S then G' has no children. Otherwise G' has a child $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k$. Or
  - $\Gamma \vdash L_m$ is a positive literal. Either
    * $\Gamma \vdash L_m$ is not in standard form. Let the standard form of $\Gamma \vdash L_m$ be $\Gamma' \vdash A_1 \wedge A_2$. There is a tree in T with root $\Gamma' \vdash A_1 \wedge A_2$. For each atom answer θ for $P \cup \{\Gamma \vdash \leftarrow L_m\}$ with respect to T and S, G' has a child $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$. Or
    * $\Gamma \vdash L_m$ is a standard literal. Either
      - $\Gamma \vdash L_m$ is not a predecessor atom and is not a variant or subatom of an atom in S, or it is a predecessor atom. Then there is a tree in T for every atom derivation where the head of a clause $\Gamma \vdash A \leftarrow M_1 \wedge \cdots \wedge M_p$ unifies with $\Gamma \vdash L_m$ with substitution θ_m and type τ. The root of the tree is $(\Gamma \vdash \leftarrow M_1 \wedge \cdots \wedge M_p)\theta_m$. For each atom answer θ of $P \cup \{\Gamma \vdash \leftarrow L_m\}$ with respect to T and S, G' has a child $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$. Or

  · $\Gamma \vdash L_m$ is not a predecessor atom and is a variant or subatom of an atom in S. Then for every atom answer θ of $P \cup \{\Gamma \vdash \leftarrow L_m\}$ with respect to T and S, G' has a child $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_k)\theta$. Or

– G' has no children.

Definition The procedure for evaluating goal G with respect to program P and declarations D is:

```
$n := 0$
let $S_n$ be $\emptyset$
repeat
    $n := n + 1$
    construct a forest $T_n$ for $G$ with respect to $P$, $D$ and $S_{n-1}$
    $\Delta S$ is the set of (atom, atom answer) pairs derived from forest $T_n$
    $S_n := S_{n-1} \cup \Delta S$
until $\Delta S \subseteq S_{n-1}$
Return all computed answers for $P \cup \{G\}$ from $T_n$.
```

The following example demonstrates the procedure where the program is recursive.

Example 4.3.7 Consider program P and declarations D in Example 4.2.11. The r-atom in P is $\{\{x{:}node, y{:}node\} \vdash path(x,y)\}$ and the i-atoms in P are $\{\{x{:}node, y{:}node\} \vdash edge(x,y)$, $edge(a,b), edge(b,c)\}$. Consider the goal $\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)$.
The forest T_1 is shown in Figure 4.8. The set of (atom, atom answer) pairs is

$$\begin{aligned} S_1 = \{ \quad & (\{x'{:}node, y'{:}node\} \vdash \leftarrow edge(x',y'), \{\{x'{:}node, y'{:}node\} \vdash \{x'/a, y'/b\}, \\ & \{x'{:}node, y'{:}node\} \vdash \{x'/b, y'/c\}\}), \\ & (\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y), \{\{x{:}node, y{:}node\} \vdash \{x/a, y/b\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/b, y/c\}\})\}. \end{aligned}$$

The forest T_2 is shown in Figure 4.9. The set of (atom, atom answer) pairs is

$$\begin{aligned} S_2 = \{ \quad & (\{x'{:}node, y'{:}node\} \vdash \leftarrow edge(x',y'), \{\{x'{:}node, y'{:}node\} \vdash \{x'/a, y'/b\}, \\ & \{x'{:}node, y'{:}node\} \vdash \{x'/b, y'/c\}\}), \\ & (\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y), \{\{x{:}node, y{:}node\} \vdash \{x/a, y/b\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/b, y/c\}, \{x{:}node, y{:}node\} \vdash \{x/a, y/c\}\})\}. \end{aligned}$$

The process is repeated, and no new atom answers are generated. As expected, the computed answers for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)\}$ are

$$\begin{aligned} & \{\{x{:}node, y{:}node\} \vdash \{x/a, y/b\}, \{x{:}node, y{:}node\} \vdash \{x/b, y/c\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/a, y/c\}\}. \ \Box \end{aligned}$$

The following example demonstrates the procedure where the program has a clause that possibly overrides a recursive definition.

Example 4.3.8 Consider program P and declarations D in Example 4.3.2 and its associated dependency graph. The r-atom is $\{\{x{:}node, y{:}node\} \vdash x[path \twoheadrightarrow y]\}$.
The i-atoms are $\{\{x{:}node, y{:}node\} \vdash x[edge \twoheadrightarrow y], \{x{:}special\_node, y{:}node\} \vdash x[path \twoheadrightarrow y]$, $\{x{:}special\_node, y{:}node\} \vdash x[edge \twoheadrightarrow y], a[edge \twoheadrightarrow b], b[edge \twoheadrightarrow c], c[edge \twoheadrightarrow d]\}$.
Consider the goal $\{x{:}node, y{:}node\} \vdash \leftarrow x[path \twoheadrightarrow y]$.

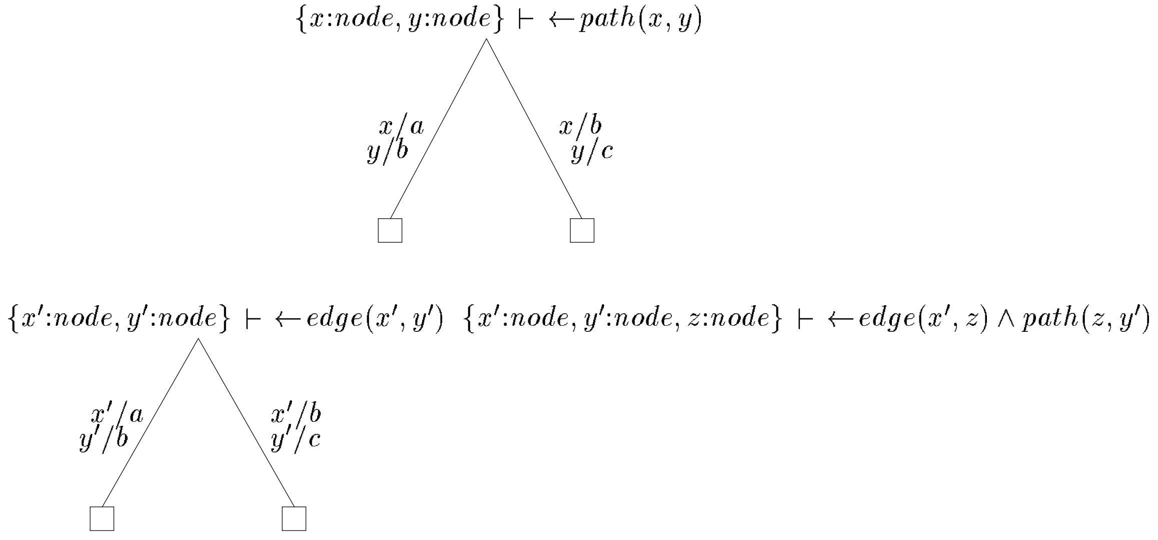
$\{x{:}node, y{:}node\} \vdash \leftarrow path(x,y)$
x/a y/b
x/b y/c
$\{x'{:}node, y'{:}node\} \vdash \leftarrow edge(x',y')$
$\{x'{:}node, y'{:}node, z{:}node\} \vdash \leftarrow edge(x',z) \wedge path(z,y')$
x'/a y'/b
x'/b y'/c

Figure 4.8: T_1 for $P \cup \{\{x{:}node, y{:}node\} \vdash\ \leftarrow path(x,y)\}$

The forest T_1 is shown in Figure 4.10. The set of (atom, atom answer) pairs is

$$\begin{aligned} S_1 = \{ \quad & (\{x{:}node, y{:}node\} \vdash\ x[path\twoheadrightarrow y], \{\{x{:}node, y{:}node\} \vdash \{x/a, y/b\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/b, y/c\}, \{x{:}node, y{:}node\} \vdash \{x/c, y/d\}\}), \\ & (\{x'{:}node, y'{:}node\} \vdash\ x'[edge\twoheadrightarrow y'], \{\{x'{:}node, y'{:}node\} \vdash \{x'/a, y'/b\}, \\ & \{x'{:}node, y'{:}node\} \vdash \{x'/b, y'/c\}, \{x'{:}node, y'{:}node\} \vdash \{x'/c, y'/d\}\}), \\ & (\{x'{:}special\_node, y'{:}node\} \vdash\ x'[edge\twoheadrightarrow y'], \{\{x'{:}special\_node, y'{:}node\} \vdash \\ & \{x'/a, y'/b\}\})\}. \end{aligned}$$

The forest T_2 is shown in Figure 4.11. The set of (atom, atom answer) pairs is

$$\begin{aligned} S_2 = \{ \quad & (\{x{:}node, y{:}node\} \vdash\ x[path\twoheadrightarrow y], \{\{x{:}node, y{:}node\} \vdash \{x/a, y/b\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/b, y/c\}, \{x{:}node, y{:}node\} \vdash \{x/c, y/d\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/b, y/d\}\}), \\ & (\{x'{:}node, y'{:}node\} \vdash\ x'[edge\twoheadrightarrow y'], \{\{x'{:}node, y'{:}node\} \vdash \{x'/a, y'/b\}, \\ & \{x'{:}node, y'{:}node\} \vdash \{x'/b, y'/c\}, \{x'{:}node, y'{:}node\} \vdash \{x'/c, y'/d\}\}) \\ & (\{x'{:}special\_node, y'{:}node\} \vdash\ x'[edge\twoheadrightarrow y'], \{\{x'{:}special\_node, y'{:}node\} \vdash \\ & \{x'/a, y'/b\}\})\}. \end{aligned}$$

Notice that the possible answer $\{x{:}node, y{:}node\} \vdash \{x/a, y/c\}$ is not minimal.

There are no further atom answers in the next iteration.

The computed answers for $P \cup \{\{x{:}node, y{:}node\} \vdash\ \leftarrow x[path\twoheadrightarrow y]\}$ are

$$\begin{aligned} & \{\{x{:}node, y{:}node\} \vdash \{x/a, y/b\}, \{x{:}node, y{:}node\} \vdash \{x/b, y/c\}, \\ & \{x{:}node, y{:}node\} \vdash \{x/c, y/d\}, \{x{:}node, y{:}node\} \vdash \{x/b, y/d\}\}. \ \square \end{aligned}$$

Theorem 4.3.1 *(Soundness) Let P be a simple program with respect to a set of declarations D, and G a goal. Then every computed answer for $P \cup \{G\}$ found using the procedure for evaluating G with respect to P and D is a correct answer for $P \cup \{G\}$.*

Proof This proof is an extension of the proof of Theorem 4.2.1.

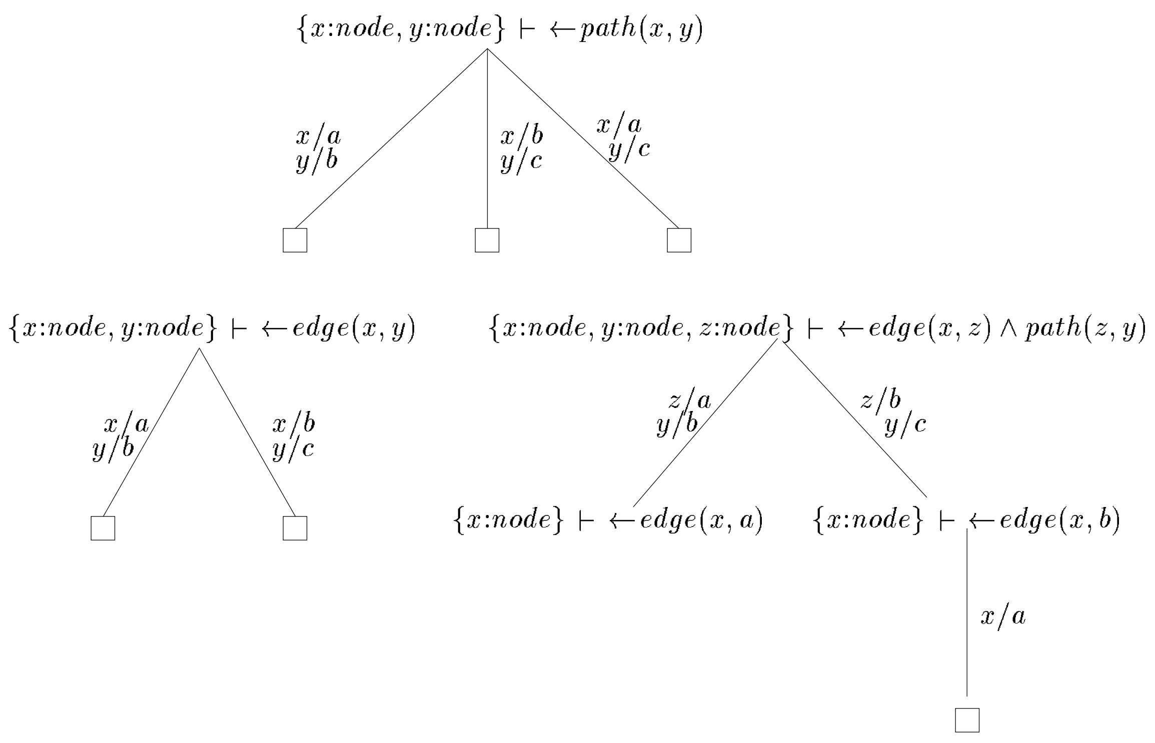
{x:node, y:node} ⊢ ←path(x, y)
x/a y/b
x/b y/c
x/a y/c
{x:node, y:node} ⊢ ←edge(x, y)
x/a y/b
x/b y/c
{x:node, y:node, z:node} ⊢ ←edge(x, z) ∧ path(z, y)
z/a y/b
z/b y/c
{x:node} ⊢ ←edge(x, a)
{x:node} ⊢ ←edge(x, b)
x/a

Figure 4.9: T_2 for $P \cup \{\{x{:}node, y{:}node\} \vdash\ \leftarrow path(x, y)\}$

Let G be the goal $\Gamma \vdash\ \leftarrow L_1 \wedge \cdots \wedge L_n$ and $\theta_1, \ldots, \theta_n$ be the sequence of atom answers used in a goal refutation of $P \cup \{G\}$ via R in a forest T_n for G with respect to P, D and S_{n-1} in the procedure for evaluating G with respect to P. Let M_P be the preferred model of P. We have to show that $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P.

We use induction on the number of the forests generated by the procedure. Initially, suppose there are no new atom answers in forests T_n where $n > 1$. Then there are no atom answers generated due to recursion. This part of the proof is the same as the proof of Theorem 4.2.1. We use induction on the rank of the goal tree for $P \cup \{G\}$ via R in forest T_1. Suppose the rank is 1. Then the restriction of P contains only facts, the literals in the tree are in standard form and are positive.

We use induction on the length of the goal refutation of $P \cup \{G\}$ via R in forest T_1. Suppose that the length of the goal refutation of $P \cup \{G\}$ via R is 1. Then G is a positive literal $\Gamma \vdash\ \leftarrow L_1$. The program has a clause $\Gamma \vdash L$ such that $(\Gamma \vdash L)\theta_1 = (\Gamma \vdash L_1)\theta_1$ where $(\Gamma \vdash L)\theta_1$ is not overridden by an instance of another clause in P. It follows that $(\Gamma \vdash L_1)\theta_1$ is true in M_P.

Assume the result holds where the length of the goal refutation of $P \cup \{G\}$ via R in T_1 is $n - 1$.

Suppose the length of the goal refutation of $P \cup \{G\}$ via R in T_1 is n. Let $\Gamma \vdash L_m$ be the selected literal. Then $\Gamma \vdash L_m$ is a positive literal. There is an instance of a clause $\Gamma \vdash L$ in P with which it unifies that is not overridden by an instance of another clause in P. By the induction hypothesis $(\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P. Also $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n = (\Gamma \vdash L)\theta_1 \ldots \theta_n$ is true in M_P. Thus $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P.

Assume the result holds for goal trees for $P \cup \{G\}$ in T_1 of rank $\leq q - 1$.

Suppose the rank of the goal tree for $P \cup \{G\}$ in T_1 is q. Again we use induction on the

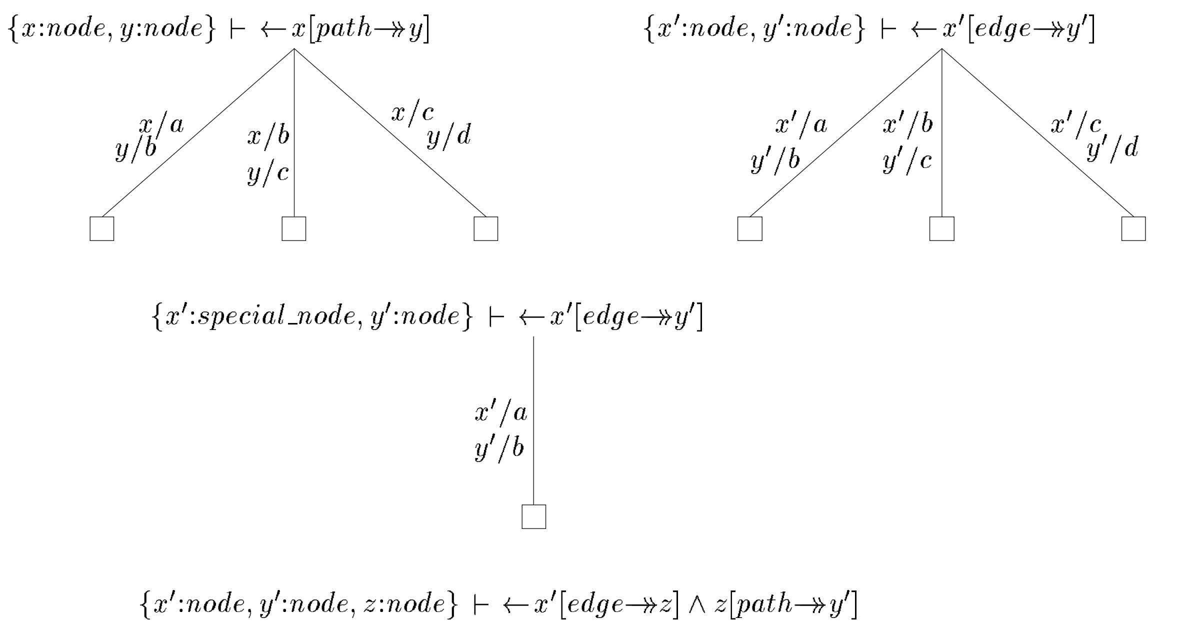
$\{x{:}node, y{:}node\} \vdash \leftarrow x[path \twoheadrightarrow y]$
x/a y/b
x/b y/c
x/c y/d
$\{x'{:}node, y'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow y']$
x'/a y'/b
x'/b y'/c
x'/c y'/d
$\{x'{:}special\_node, y'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow y']$
x'/a y'/b
$\{x'{:}node, y'{:}node, z{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow z] \wedge z[path \twoheadrightarrow y']$

Figure 4.10: T_1 for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow x[path \twoheadrightarrow y]\}$

length of the refutation of $P \cup \{G\}$ via R. Suppose the length is 1. Then G has the form $\Gamma \vdash \leftarrow L_1$. Either

- $\Gamma \vdash L_1$ is a positive literal. Either
  - $\Gamma \vdash L_1$ is a standard positive literal. Suppose $\Gamma \vdash L_1$ is not a successor atom. Then $\Gamma \vdash L_1$ unifies with the head of a clause $\Gamma \vdash L' \leftarrow L'_1 \wedge \cdots \wedge L'_k$ with unifier θ'. Either
    - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $< q-1$. Let $\theta_1 = \theta'$. Then by the induction hypothesis $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
    - the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $q-1$. Then there is a refutation of $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_k$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_k$. By the induction hypothesis $(\Gamma \vdash \leftarrow L'_1 \wedge \ldots \wedge L'_k)\theta$ is true in M_P thus $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
  - $\Gamma \vdash L_1$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L_1$. Then there is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ of rank $\leq q-1$. There is a refutation of $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ via R with computed answer θ_1. By the induction hypothesis, $(\Gamma' \vdash A_1 \wedge A_2)\theta_1$ is true in M_P, thus $(\Gamma \vdash L_1)\theta_1$ is true in M_P.
- $\Gamma \vdash L_1$ is a negative literal, $\Gamma \vdash \neg A$. Then, there is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A\}$ where $\theta_1 = \varepsilon$. By Lemma 4.2.2, it follows that $(\Gamma \vdash L_1)\theta_1$ is true in M_P.

Assume the result holds for refutations of $P \cup \{G\}$ via R in T_1 of length $n-1$.

Suppose that the length of the refutation of $P \cup \{G\}$ via R in T_1 is n. Then G has the form $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_n$. Let $\Gamma \vdash L_m$ be the selected literal. Either

- $\Gamma \vdash L_m$ is a positive literal. Either

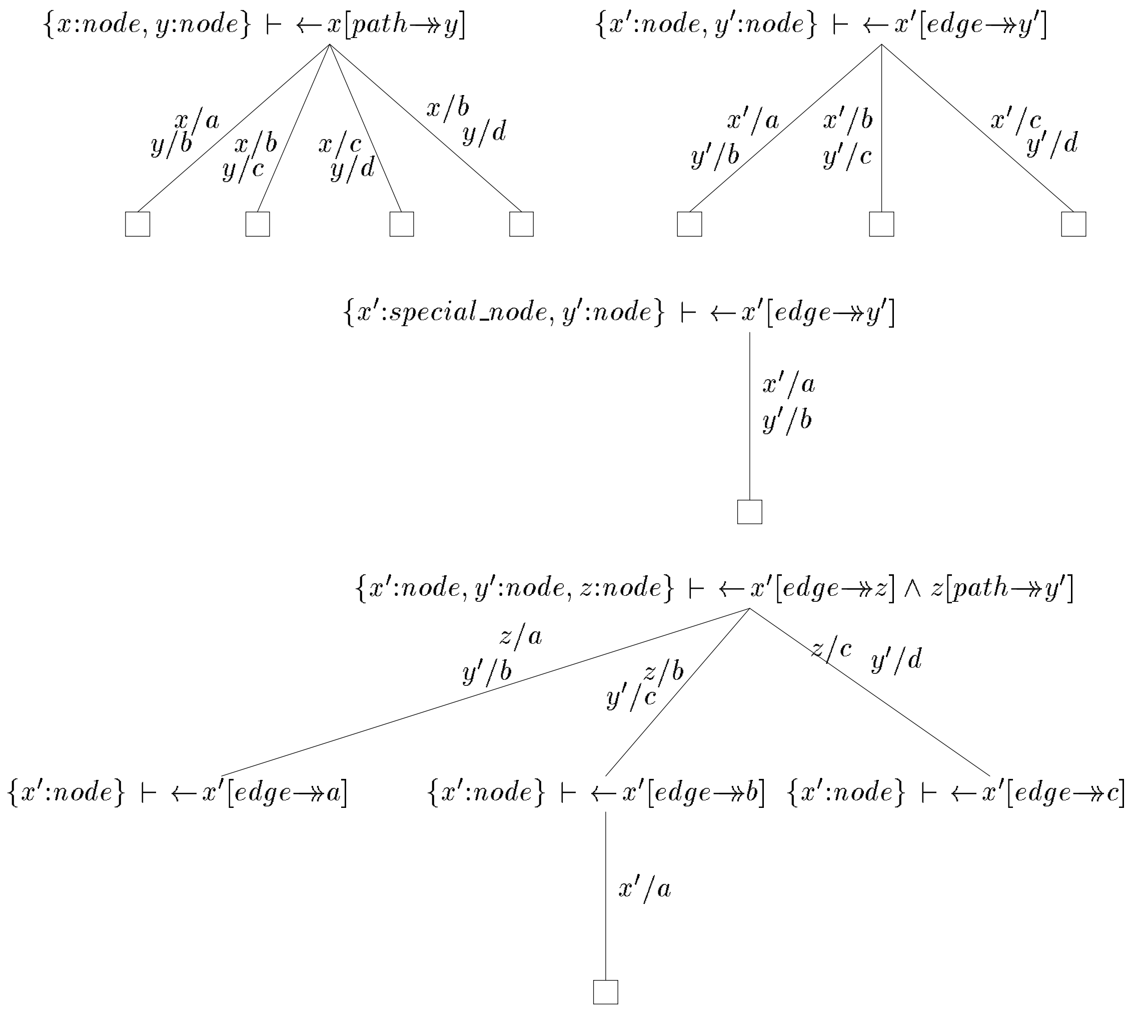
$\{x{:}node, y{:}node\} \vdash \leftarrow x[path \twoheadrightarrow y]$
x/a y/b
x/b y/c
x/c y/d
x/b y/d
$\{x'{:}node, y'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow y']$
x'/a y'/b
x'/b y'/c
x'/c y'/d
$\{x'{:}special\_node, y'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow y']$
x'/a y'/b
$\{x'{:}node, y'{:}node, z{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow z] \wedge z[path \twoheadrightarrow y']$
z/a y'/b
z/b y'/c
z/c y'/d
$\{x'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow a]$
$\{x'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow b]$
$\{x'{:}node\} \vdash \leftarrow x'[edge \twoheadrightarrow c]$
x'/a

Figure 4.11: T_2 for $P \cup \{\{x{:}node, y{:}node\} \vdash \leftarrow x[path \twoheadrightarrow y]\}$

- $\Gamma \vdash L_m$ is a standard positive literal. Suppose $\Gamma \vdash L_m$ is not a successor atom. Then $\Gamma \vdash L_m$ unifies with the head of a clause $\Gamma \vdash L' \leftarrow L'_1 \wedge \cdots \wedge L'_k$ in P that is not overridden by an instance of another clause in P such that $(\Gamma \vdash L_m)\theta' = (\Gamma \vdash L')\theta'$. Either
  * the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $< q-1$. Then by the induction hypothesis $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P. Or
  * the rank of the tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ is $q-1$. Then there is a refutation of $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_k$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_k$. By the induction hypothesis $(\Gamma \vdash \leftarrow L'_1 \wedge \ldots \wedge L'_k)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P. Or
- $\Gamma \vdash L_m$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L_m$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ of rank $\leq q-1$. There is a refutation of $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ via R with computed answer θ_1. By the induction hypothesis, $(\Gamma' \vdash A_1 \wedge A_2)\theta_1$ is true in M_P, thus $(\Gamma \vdash L_m)\theta$ is true in M_P.

By the inner induction hypothesis $(\Gamma \vdash \leftarrow L_1 \wedge \ldots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash \leftarrow L_1 \wedge \ldots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P. Suppose $\Gamma \vdash L_m$ is a successor atom, then this node has no children and there are no refutations for $P \cup \{\Gamma \vdash \leftarrow L_m\}$.

- $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A$. There is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A\}$ where $\theta_1 = \varepsilon$. By Lemma 4.2.2, it follows that $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P. By the induction hypothesis $(\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P.

The rest of this proof is an extension to the proof of Theorem 4.2.1.

Assume the result holds where the number of forests generated is $q-1$.

Suppose now that the number of forests generated is q. We use induction on the length of the refutation of $P \cup \{G\}$ via R in T_q. Suppose the length of the goal refutation of $P \cup \{G\}$ via R in T_q is 1. Then G has the form $\Gamma \vdash \leftarrow L_1$. Either

- $\Gamma \vdash L_1$ is a standard positive literal. Either
  - $\Gamma \vdash L_1$ is a standard positive literal. Suppose $\Gamma \vdash L_1$ is not a predecessor atom. Then it is a variant or subatom of an atom in S_{q-1} with atom answer θ_1. By the induction hypothesis $(\Gamma \vdash L_1)\theta_1$ is true in M_P.

    Suppose $\Gamma \vdash L_1$ is a predecessor atom. There is a ground instance of a clause $\Gamma \vdash L \leftarrow L'_1 \wedge \cdots \wedge L'_k$ in P such that $(\Gamma \vdash L_1)\theta' = (\Gamma \vdash L)\theta'$ that is not overridden by an instance of another clause in P. There is a goal tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$. There is a refutation for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_k$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_k$. By the induction hypothesis $(\Gamma \vdash L'_1 \wedge \cdots \wedge L'_k)\theta_1$ is true in M_P, thus $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
  - $\Gamma \vdash L_1$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L_1$. It follows from the above that $(\Gamma \vdash A_1)\theta_1$ and $(\Gamma \vdash A_2)\theta_1$ are true in M_P, thus $(\Gamma \vdash L_1)\theta_1$ is true in M_P. Or
- $\Gamma \vdash L_1$ is a negative literal, $\Gamma \vdash \neg A$. There is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A\}$ in T_k where $k < q$ with substitution $\theta_1 = \varepsilon$. By the induction hypothesis $(\Gamma \vdash L_1)\theta_1$ is true in M_P.

Assume the result holds for refutations of $P \cup \{G\}$ via R of length $n-1$.

Suppose that the length of the refutation of $P \cup \{G\}$ via R is n. Then G has the form $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_n$. Let $\Gamma \vdash L_m$ be the selected literal. Either

- $\Gamma \vdash L_m$ is a standard positive literal. Suppose $\Gamma \vdash L_m$ is not a predecessor atom. Then it is a variant or subatom of an atom in S_{q-1} with atom answer θ_1. By the induction hypothesis $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P.

  Suppose $\Gamma \vdash L_m$ is a predecessor atom. There is a ground instance of a clause $\Gamma \vdash L \leftarrow L'_1 \wedge \cdots \wedge L'_k$ in P such that $(\Gamma \vdash L_m)\theta' = (\Gamma \vdash L)\theta'$ that is not overridden by an instance of another clause in P. There is a goal tree for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$. There is a refutation for $P \cup \{(\Gamma \vdash \leftarrow L'_1 \wedge \cdots \wedge L'_k)\theta'\}$ with atom answers $\theta'_1, \ldots, \theta'_k$. Let $\theta_1 = \theta'\theta'_1 \ldots \theta'_k$. By the induction hypothesis $(\Gamma \vdash L'_1 \wedge \cdots \wedge L'_k)\theta_1$ is true in M_P, thus $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P.

- $\Gamma \vdash L_m$ is not in standard form. Let $\Gamma' \vdash A_1 \wedge A_2$ be the standard form of $\Gamma \vdash L_m$. It follows from the above that $(\Gamma \vdash A_1)\theta_1 \ldots \theta_n$ and $(\Gamma \vdash A_2)\theta_1 \ldots \theta_n$ are true in M_P, thus $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P. Or

- $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A$. There is a failed goal tree for $P \cup \{\Gamma \vdash \leftarrow A\}$ in T_k where $k < q$ with substitution $\theta_1 = \varepsilon$. By the induction hypothesis $(\Gamma \vdash L_m)\theta_1 \ldots \theta_n$ is true in M_P.

By the induction hypothesis $(\Gamma \vdash L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \ldots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P, thus $(\Gamma \vdash L_1 \wedge \cdots \wedge L_n)\theta_1 \ldots \theta_n$ is true in M_P. □

Lemma 4.3.1 *Let P be a simple program with respect to a set of declarations D, and G a goal. In the procedure that evaluates G with respect to P and D every tree is finite, and only finitely many subtrees are generated.*

Proof We need to show that every node in a tree has a finite number of children and generates a finite number of subtrees. Suppose $G = \Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_k$.

We use induction on the number of the forests generated by the procedure. Initially suppose new atom answers are generated only in forest T_1. Then no new atom answers are generated due to recursion.

We use induction on the rank of the goal tree for $P \cup \{G\}$ via R. Suppose the rank is 1. Then the restriction of P contains only facts, the atoms are in standard form, and there is no negation. Then the restriction of P contains only i-atoms.

Suppose $\Gamma \vdash L_m$ is the selected literal. $\Gamma \vdash L_m$ is an i-atom. Let n be the number of clauses with which $\Gamma \vdash L_m$ unifies, which are not overridden by another clause in P. This node has n children and no subtrees are generated.

Assume the result holds for goal trees for $P \cup \{G\}$ of rank $\leq p-1$.

Suppose the rank of the goal tree for $P \cup \{G\}$ is p. Let $\Gamma \vdash L_m$ be the selected literal. Either

- $\Gamma \vdash L_m$ is a positive literal. Either

  - $\Gamma \vdash L_m$ is not in standard form. Suppose $\Gamma' \vdash A_1 \wedge A_2$ is the standard form of $\Gamma \vdash L_m$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ of rank $\leq p-1$. By the induction hypothesis, in the procedure that evaluates $\Gamma' \vdash \leftarrow A_1 \wedge A_2$ with respect to P and D every tree is finite, and only finitely many subtrees are generated. Let the number of atom answers for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ be n. Then the node $\Gamma \vdash \leftarrow G$ has n children. Or

  - $\Gamma \vdash L_m$ is a standard positive literal. Either
    - $\Gamma \vdash L_m$ is not a successor atom. There is potentially a goal tree for each clause with which $\Gamma \vdash L_m$ unifies with rank $\leq p-1$. By the induction hypothesis, in the procedure that evaluates $\Gamma \vdash L_m$ with respect to P and D every tree is finite, and only finitely many subtrees are generated. Let the number of atom answers for $P \cup \{\Gamma \vdash L_m\}$ be n. Then the node $\Gamma \vdash \leftarrow G$ has n children. Or
    - $\Gamma \vdash L_m$ is a successor atom. This node has 0 children and generates 0 subtrees. Or

- $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A_m$. This node has 0 or 1 children. There is a subtree with root $\Gamma \vdash \leftarrow A_m$ with rank $\leq p-1$. By the induction hypothesis, in the procedure that evaluates $\Gamma \vdash \leftarrow A_m$ with respect to P and D every tree is finite, and only finitely many subtrees are generated.

Assume the result holds where the number of forests generated is $q-1$.

Suppose now that the number of forests generated is q. Let $\Gamma \vdash L_m$ be the selected literal. Either

- $\Gamma \vdash L_m$ is a positive literal. Either
  - $\Gamma \vdash L_m$ is not in standard form. Suppose $\Gamma' \vdash A_1 \wedge A_2$ is the standard form of $\Gamma \vdash L_m$. There is a goal tree for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ in T_{q-1}. By the induction hypothesis, in the procedure that evaluates $\Gamma' \vdash A_1 \wedge A_2$ with respect to P every tree is finite, and only finitely many subtrees are generated. Let the number of atom answers for $P \cup \{\Gamma' \vdash \leftarrow A_1 \wedge A_2\}$ be n. Then the node $\Gamma \vdash \leftarrow G$ has n children. Or
  - $\Gamma \vdash L_m$ is a standard positive literal. Either
    - $\Gamma \vdash L_m$ is an i-atom that is a variant or subatom of an atom in S_{q-1}. There is potentially a goal tree for each clause with which $\Gamma \vdash L_m$ unifies with rank $\leq p-1$. By the induction hypothesis, only finitely many subtrees are generated. Let there be n answers in S_{q-1}. Then this node has n children.
    - $\Gamma \vdash L_m$ is a predecessor atom that is a variant or subatom of an atom in S_{q-1}. There is potentially a goal tree for each clause with which $\Gamma \vdash L_m$ unifies with rank $\leq p-1$. By the induction hypothesis, only finitely many subtrees are generated. Let the number of atom answers for $P \cup \{\Gamma \vdash L_m\}$ be n. Then the node $\Gamma \vdash \leftarrow G$ has n children. Or
    - $\Gamma \vdash L_m$ is a successor atom. This node has 0 children and 0 subtrees are generated. Or

- $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A_m$. This node has 0 or 1 children. There is a goal tree with root $\Gamma \vdash \leftarrow A_m$ in forest T_{q-1}. By the induction hypothesis, in the procedure that evaluates $\Gamma \vdash \leftarrow A_m$ with respect to P and D every tree is finite, and only finitely many subtrees are generated. $\square$

Recall that an unrestricted answer is an atom answer in which arbitrary unifiers are used rather than most general unifiers. Because atom answers are ground instances of answers, the answers computed using arbitrary unifiers are also computed using most general unifiers.

Lemma 4.3.2 *Let P be a simple program with respect to a set of declarations D, and G a goal. Suppose that $P \cup \{G\}$ has an unrestricted answer θ. Then $P \cup \{G\}$ has a computed answer θ' such that $\theta = \theta'$.*

Proof Let G be $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_n$. We use induction on the rank of the tree for $P \cup \{G\}$. Assume the rank of the tree is 1. Then in the restriction of P there is no negation, all the clauses are facts, and all atoms in the tree are in standard form.

We use induction on the length of the refutation. Let the length be 1. Then G is $\Gamma \vdash \leftarrow L_1$. There is a clause $\Gamma \vdash L$ that unifies with $\Gamma \vdash L_1$ with arbitrary unifier θ. Ground clause $(\Gamma \vdash L)\theta$ is not overridden by another clause in P. Let θ' be the mgu of $\Gamma \vdash L$ and $\Gamma \vdash L_1$. For every $\theta_a \in [\theta]_D$, $\theta_a \in [\theta']_D$ and θ_a is an atom answer for $P \cup \{\Gamma \vdash \leftarrow L_1\}$.

Assume the result holds for refutations of length $n-1$.

Suppose the refutation has length n. Let $\Gamma \vdash L_m$ be the selected literal. There is a clause $\Gamma \vdash L$ that unifies with $\Gamma \vdash L_m$ with arbitrary unifier θ_1. Ground clause $(\Gamma \vdash L)\theta_1$ is not overridden by another clause in P. Let θ_1' be the mgu of $\Gamma \vdash L$ and $\Gamma \vdash L_m$. For every $\theta_a \in [\theta_1]_D$, $\theta_a \in [\theta_1']_D$. By the induction hypothesis, $P \cup \{(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_a\}$ has a refutation with computed answer $\theta_2 \ldots \theta_n$. Thus $P \cup \{G\}$ has a refutation with computed answer $\theta_a\theta_2 \ldots \theta_n$.

Assume the result holds for trees of rank $k-1$.

Suppose now that the tree for $P \cup \{G\}$ has rank k. We use induction on the length of the refutation. Let the length be 1. Then G is $\Gamma \vdash \leftarrow L_1$. Either

- $\Gamma \vdash L_1$ is a standard positive literal. Then there is a clause $\Gamma \vdash L \leftarrow L_1' \wedge \cdots \wedge L_j'$ in P such that $\Gamma \vdash L$ unifies with $\Gamma \vdash L_1$ with arbitrary unifier θ. Ground clause $(\Gamma \vdash L \leftarrow L_1' \wedge \cdots \wedge L_j')\theta$ is not overridden by another clause in P. Let θ' be the mgu of $\Gamma \vdash L$ and $\Gamma \vdash L_1$. There is a tree of rank $\leq k-1$ for $P \cup \{\Gamma \vdash \leftarrow L_1' \wedge \cdots \wedge L_j'\}$. By the induction hypothesis $P \cup \{\Gamma \vdash \leftarrow L_1' \wedge \cdots \wedge L_j'\}$ has a refutation with computed answer $\theta_j = \theta_1' \ldots \theta_j'$. There is an atom answer θ_a for $P \cup \{\Gamma \vdash \leftarrow L_m\}$ such that $\theta_a \in [\theta\theta_j]_D$ and $\theta_a \in [\theta'\theta_j]_D$.

- $\Gamma \vdash L_1$ is a non-standard positive literal. Let the standard form of $\Gamma \vdash L_1$ be $\Gamma \vdash A_1 \wedge A_2$. Then there is a tree of rank $\leq k-1$ for $P \cup \{\Gamma \vdash \leftarrow A_1 \wedge A_2\}$. By the induction hypothesis, it follows that if there is an unrestricted refutation of $P \cup \{\Gamma \vdash \leftarrow A_1 \wedge A_2\}$, then there is a refutation of $P \cup \{\Gamma \vdash \leftarrow L_1\}$. Or

- $\Gamma \vdash L_1$ is a negative literal, $\Gamma \vdash \neg A$. Then there is no unrestricted refutation for $P \cup \{\Gamma \vdash \leftarrow A_m\}$ so there is no refutation for $P \cup \{\Gamma \vdash \leftarrow A_m\}$. By the induction hypothesis, $P \cup \{\Gamma \vdash \leftarrow L_1\}$ has a refutation.

Assume the result holds for refutations of length $n-1$.
Suppose the refutation has length n. Let $\Gamma \vdash L_m$ be the selected literal. Either

- $\Gamma \vdash L_m$ is a standard positive literal. Then there is a clause $\Gamma \vdash L \leftarrow L_1' \wedge \cdots \wedge L_j'$ in P such that $\Gamma \vdash L$ unifies with $\Gamma \vdash L_m$ with arbitrary unifier θ. Ground clause $(\Gamma \vdash L \leftarrow L_1' \wedge \cdots \wedge L_j')\theta$ is not overridden by another clause in P. Let θ' be the mgu of $\Gamma \vdash L$ and $\Gamma \vdash L_1$. By the induction hypothesis $P \cup \{\Gamma \vdash \leftarrow L_1' \wedge \cdots \wedge L_j'\}$ has a refutation with computed answer $\theta_j = \theta_1' \ldots \theta_j'$. There is an atom answer θ_1 for $P \cup \{\Gamma \vdash \leftarrow L_m\}$ such that $\theta_1 \in [\theta\theta_j]_D$. It follows that every atom answer $\theta_1 \in [\theta\theta_j]_D$, $\theta_1 \in [\theta'\theta_j]_D$.

- $\Gamma \vdash L_m$ is a non-standard positive literal. Let the standard form of $\Gamma \vdash L_m$ be $\Gamma \vdash A_1 \wedge A_2$. Then there is a tree of rank $\leq k-1$ for $P \cup \{\Gamma \vdash \leftarrow A_1 \wedge A_2\}$. By the induction hypothesis, it follows that if there is an unrestricted refutation of $P \cup \{\Gamma \vdash \leftarrow A_1 \wedge A_2\}$, then there is a refutation of $P \cup \{\Gamma \vdash \leftarrow L_m\}$ with atom answer θ_1. Or

- $\Gamma \vdash L_m$ is a negative literal, $\Gamma \vdash \neg A$. Then there is no unrestricted refutation for $P \cup \{\Gamma \vdash \leftarrow A\}$ so there is no refutation for $P \cup \{\Gamma \vdash \leftarrow A\}$. By the induction hypothesis, $P \cup \{\Gamma \vdash \leftarrow L_m\}$ has a refutation with atom answer $\theta_1 = \varepsilon$.

By the induction hypothesis, $P \cup \{(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_{m-1} \wedge L_{m+1} \wedge \cdots \wedge L_n)\theta_1\}$ has a refutation with computed answer $\theta_2 \ldots \theta_n$. Thus $P \cup \{G\}$ has a refutation with computed answer $\theta_1\theta_2 \ldots \theta_n$. $\Box$

Lemma 4.3.3 *Let P be a simple program with respect to a set of declarations D, G a goal, and θ a substitution. Suppose there exists a goal refutation of $P \cup \{G\theta\}$ via R. Then there exists a goal refutation of $P \cup \{G\}$ via R. Furthermore if $\theta_1, \ldots, \theta_n$ are the atom answers from the goal refutation of $P \cup \{G\theta\}$ via R and $\theta'_1, \ldots, \theta'_n$ are the atom answers of the goal refutation of $P \cup \{G\}$ via R then $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n$.*

Proof Suppose the literal $\Gamma \vdash L_m$ is selected in $G\theta$ and suppose an atom answer for $P \cup \{\Gamma \vdash \leftarrow L_m\}$ is θ_1 and the resulting goal is G_1. Then there is a clause $C_1 = \Gamma \vdash L \leftarrow L'_1 \wedge \cdots \wedge L'_k$ such that $(\Gamma \vdash L)\theta_1 = (\Gamma \vdash L_m)\theta_1$. We assume θ does not act on any of the variables in C_1. Now $\theta\theta_1$ is a unifier for $\Gamma \vdash L$ and the atom in G that corresponds to L_m in $G\theta$. The result of resolving G and C_1 using $\theta\theta_1$ is G_1. The result is proven by applying this result n times. $\Box$

Theorem 4.3.2 *(Completeness) Let P be a simple program with respect to a set of declarations D, and G a goal. Every correct answer for $P \cup \{G\}$ is a computed answer for $P \cup \{G\}$ found using the procedure for evaluating G with respect to P and D. The procedure for evaluating G with respect to P and D terminates.*

Proof We use induction on the level of G. Suppose G has level 0. Then there is no overriding or negation in the restriction of P. Suppose G is the goal $\Gamma \vdash \leftarrow A_1 \wedge \cdots \wedge A_k$. Since θ is correct $(\Gamma \vdash A_1 \wedge \cdots \wedge A_k)\theta$ is true in M_P. Since for every i, $(\Gamma \vdash \leftarrow A_i)\theta$ is ground, there is a goal refutation of $P \cup \{(\Gamma \vdash \leftarrow A_i)\theta\}$ such that the computed answer is the identity. Suppose that the sequence of atom answers for the refutation of $P \cup \{G\theta\}$ is $\theta_1, \ldots, \theta_n$. Then $G\theta\theta_1 \ldots \theta_n = G\theta$. By Lemma 4.3.3, there is a refutation of $P \cup \{G\}$ with atom answers $\theta'_1, \ldots, \theta'_n$ such that $\theta\theta_1 \ldots \theta_n = \theta'_1 \ldots \theta'_n$. Let σ be $\theta'_1 \ldots \theta'_n$ restricted to the variables in G. Then $\theta = \sigma$, and θ is a computed answer for $P \cup \{G\}$.

Assume the result holds for all goals of level $\leq k$.

Suppose the level of G is $k+1$. Then G is the goal $\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_j$. Since θ is a correct answer, $(\Gamma \vdash \leftarrow L_1 \wedge \cdots \wedge L_j)\theta$ is true in M_P. There are three possibilities for each $\Gamma \vdash L_i$.

- $\Gamma \vdash L_i$ is a positive literal of level $\leq k$. By the induction hypothesis $P \cup \{(\Gamma \vdash \leftarrow L_i)\theta\}$ has a goal refutation.

- $\Gamma \vdash L_i$ is a negative literal, $\Gamma \vdash \neg A_i$. Then A_i has level $\leq k$ and $(\Gamma \vdash \leftarrow \neg A_i)\theta$ is true in M_P. By the induction hypothesis, the goal tree for $P \cup \{(\Gamma \vdash \leftarrow A_i)\theta\}$ failed and $(\Gamma \vdash \leftarrow L_i)\theta$ has a goal refutation.

- $\Gamma \vdash L_i$ is a positive literal of level $k+1$. By Theorem 4.1.1, $(\Gamma \vdash L_i)\theta \in T_{P_{k+1}} \uparrow n(M_{P_k})$ for some $n \geq 0$.

  We prove by induction on n that for every ground atom A of level $k+1$ in $T_{P_{k+1}} \uparrow n(M_{P_k})$, $P \cup \{\leftarrow A\}$ has a goal refutation.

  Suppose first that $n = 1$. Then there exists a ground instance $A \leftarrow L'_1 \wedge \cdots \wedge L'_q$ of a clause of type σ in P such that $L'_1 \wedge \cdots \wedge L'_q$ is true in M_P, the level of $L'_1 \wedge \cdots \wedge L'_q$

is $\leq k$, and there is no instance $A \leftarrow L_1'' \wedge \cdots \wedge L_p''$ of a clause of type τ in P such that $L_1'' \wedge \cdots \wedge L_p''$ is true in M_P where $\tau <_t \sigma$. By the main induction hypothesis, $\leftarrow L_1' \wedge \cdots \wedge L_q'$ has a goal refutation and hence $P \cup \{\leftarrow A\}$ has an unrestricted answer. By Lemma 4.3.2, $P \cup \{\leftarrow A\}$ has an atom answer and a goal refutation.

Suppose that $n > 1$. Then there exists a ground instance $A \leftarrow L_1' \wedge \cdots \wedge L_q' \wedge A_1 \wedge \cdots \wedge A_r$ where the A_j are positive literals such that $L_1' \wedge \cdots \wedge L_q' \wedge A_1 \wedge \cdots \wedge A_r$ is true in M_P, the level of $L_1' \wedge \cdots \wedge L_q'$ is $\leq k$ and the level of $A_1 \wedge \cdots \wedge A_r$ is $k+1$. By the induction hypothesis on k, $\leftarrow L_1' \wedge \cdots \wedge L_q'$ has a goal refutation. By the induction hypothesis on n, $P \cup \{\leftarrow A_i\}$ has a goal refutation for each $\leftarrow A_i$. As each L_i' and A_j is ground, these refutations may be combined and $P \cup \{\leftarrow A\}$ has an unrestricted answer. By Lemma 4.3.2, $P \cup \{\leftarrow A\}$ has an atom answer and a goal refutation.

Thus each $P \cup \{(\Gamma \vdash \leftarrow L_i)\theta\}$ has a goal refutation. The goal refutations of the $P \cup \{(\Gamma \vdash \leftarrow L_i)\theta\}$ may also be combined to give a goal refutation of $P \cup \{G\theta\}$ with computed answer ε.

By Lemma 4.3.3, there exists a goal refutation of $P \cup \{G\}$ with computed answer θ' such that $\theta' = \theta\varepsilon$. By Lemma 4.3.1, in the procedure that evaluates G with respect to P and D every tree is finite and only infinitely many subtrees are generated. Thus θ is a computed answer for $P \cup \{G\}$ and the procedure terminates. □

4.4 Summary

In this chapter we describe bottom-up and top-down query evaluation procedures for a deductive object-oriented language, Gulog.

We describe a bottom-up procedure and prove it is sound and complete with respect to the declarative semantics described in Section 3.5. Because this procedure is not goal directed, it is inefficient when computing the answers to some queries, such as the value of a method when the method is applied to a particular object. We then describe a goal directed top-down procedure. This procedure is sound with respect to the declarative semantics, but not complete because it does not always terminate. We extend this procedure with a tabling mechanism, and prove that the new procedure is sound and complete with respect to the declarative semantics.

The bottom-up procedure that we describe is very similar to the procedure that computes the standard model of a stratified logic program in [12]. It differs in the following ways: it is based on the possibility of overriding, as well as the presence of negation; the immediate consequence operator is a mapping between typed interpretations rather than interpretations.

Brass and Lipeck [23] describe a bottom-up evaluation procedure for deductive object-oriented database specifications consisting of overridable rules (defaults) and non-overridable rules (axioms). They determine "conflicts" due to overriding and add their complements to the original rule set. Then they draw conclusions from the enlarged rule set using traditional bottom-up evaluation. They include rules with disjunction in the head, which provides another mechanism for dealing with conflicts due to multiple inheritance. However allowing disjunction complicates the semantics and procedures. Their semantics for overriding is different from ours, and changing their definition to deal with dynamic overriding would not be simple. It would involve recognizing the truth of the bodies of rules when extending the program.

Laenens et al. describe a bottom-up query evaluation procedure for ordered logic in [72]. Although their language does not differentiate between objects, classes and attributes, or the subclass and instance relationship, the procedures they describe are still interesting. Taking the approach the authors describe, an ordered logic program has several meanings, one for each object. Their stratification condition is similar to ours but they are dealing with ground clauses and consider stratification with respect to an object. Their immediate consequence operator is also like ours. Because the clauses are ground, they do not need to consider clauses that possibly override other clauses.

In [65], the authors describe the fixpoint semantics of F-logic, on which a bottom-up evaluation procedure can be based. The authors describe triggers that are fired to activate inheritance. A more complex evaluation procedure is needed because it is possible to alter the inheritance hierarchy dynamically. Thus the inheritance hierarchy is not known in advance and it is not possible to define levels of the program. We believe that our procedure is simple and provides an understanding of the interaction of inheritance, overriding and deduction that is easier to comprehend than the more complex fixpoint semantics provided in [65].

The top-down procedure described in this chapter is based on typed unification and a variant of SLDNF-resolution [77]. The main difference between this procedure and SLDNF-resolution is that the answer for the selected literal must be computed before another literal is selected. It is like SLDNF-resolution with a locality restriction where the answers to the selected literal are computed before the next literal is selected. The difference between SLDNF-resolution with a locality restriction and the procedure we propose lies in eliminating overridden answers. An alternative approach is the SLDNF procedure with a post check to find the minimal conflicting answers. This approach would provide correct answers but the tree it generates will usually have more branches than the tree generated by the procedure we propose. Thus, the SLDNF procedure with a post check would be less efficient.

In [65], the authors describe stratification, unification and a proof theory for F-logic. However these definitions do not take overriding into account. Also the authors consider the unification of molecules, rather than atoms, which makes the algorithm asymmetric and more complex than it needs to be.

In [8], Ait-Kaci and Nasr define a unification procedure that is similar to ours. However you will recall from Section 2.3 that in LOGIN there is no distinction between classes and objects, nor functional and multi-valued methods. Part of the contribution of LOGIN was ψ-terms that are like molecules in F-logic. The authors describe a procedure for unifying ψ-terms. The procedure deals with the subparts of the ψ-terms occurring in any order. This procedure could again be simpler if it was not concerned with the ordering, and if it unified the parts that make up the ψ-terms.

In [73], the authors describe a partial model and a proof theory for an ordered logic program which is defined on ground programs. In the language they do not differentiate between objects, classes and attributes, or the subclass and instance relationships. In another proof theory for ordered logic [72], the authors extend their language with a fail operator that allows them to compile all information pertaining to an object and its position in the program hierarchy into a single extended program for which a proof theory is easily provided. This procedure requires program rewriting.

The tabling mechanism described in this chapter is based on QSQR/SLS [62]. It differs from QSQR/SLS and other tabling mechanisms for deductive databases because the underlying top-down evaluation procedure is concerned with inheritance and overriding. The outer part of the algorithm, which iterates through forests, is the same as the outer part of the algorithm for QSQR/SLS. The part of the algorithm that finds computed answers is different. Also the definitions of recursive atoms, predecessor atoms and successor atoms consider subatoms because of the presence of inheritance and overriding.

Chapter 5

Translation to Datalog

In this chapter, we give a translation from Gulog programs to Datalog (with negation) programs. This translation enables techniques developed for Datalog programs to be applied to Gulog programs. It extends the translation from C-logic to first-order logic [36], by including the translation of overriding. A similar translation is given by McCabe for *L&O* in [81]. The semantics of inheritance with overriding in *L&O* is different from the semantics of inheritance with overriding in Gulog and this is reflected in the translations. The translation we describe is similar to dynamic rewriting described in [7] except that we consider multiple inheritance and distinguish between functional and multi-valued methods.

5.1 Translation

The object and hierarchy declarations of a Gulog program are included in the translated program. An object declaration $a{:}\tau$ is translated to an atom $\$\tau(a)$. A type hierarchy declaration $\sigma < \tau$ is translated to a clause $\$\tau(x) \leftarrow \$\sigma(x)$. A variable typing $\{x_1{:}\sigma_1, \cdots, x_n{:}\sigma_n\}$ is translated to a conjunction of atoms $\$\sigma_1(x_1) \wedge \ldots \wedge \$\sigma_n(x_n)$. A predicate atom $p(t_1, \ldots, t_n)$ is left unchanged in the translation. A functional method atom with arity n, $t[m@t_1, \ldots, t_n \rightarrow t']$ is translated to an $(n+2) - ary$ predicate $\$m(t, t_1, \ldots, t_n, t')$. A multi-valued method atom with arity n, $t[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is translated to $\$\$m(t, t_1, \ldots, t_n, t')$. Negative literals retain their negation in the translation, that is, $\neg A$ is translated to $\neg A'$, where A' is the translation of A.

Let $\Gamma \vdash A$ be a predicate atom, and B the body of a clause. The clause $\Gamma \vdash A \leftarrow B$, is translated to the clause $A' \leftarrow \Gamma' \wedge B'$, where A' is the translation of A, Γ' is the translation of Γ, and B' is the translation of B.

If a clause defines a functional method on a type, and another defines the same functional method on a subtype, these clauses must be considered together. Let $\Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t']$ be a method atom of type τ. A clause defining this method is $C = \Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t'] \leftarrow B$. Let $\sigma_1, \ldots, \sigma_n$ be all the (direct or indirect) subtypes of τ that have functional method m of arity k defined on their instances t_i of type σ_i by clauses: $C_i = \Gamma_i \vdash t_i[m@t_{i1}, \ldots, t_{ik} \rightarrow t_i'] \leftarrow B_i$, for $1 \leq i \leq n$. The clause defining functional method m of arity k on instances of type τ is translated to the Datalog clauses:

$$\begin{array}{l} app_{\tau,\$m}(t, t_1, \ldots, t_k) \leftarrow \Gamma' \wedge B'. \\ \$m(t, t_1, \ldots, t_k, t') \leftarrow \Gamma' \wedge B' \wedge \neg app_{\sigma_1,\$m}(t, t_1, \ldots, t_k) \wedge \cdots \wedge \neg app_{\sigma_n,\$m}(t, t_1, \ldots, t_k), \end{array}$$

where $\$m(t, t_1, \ldots, t_k, t')$ is the translation of $t[m@t_1, \ldots, t_k \rightarrow t']$, Γ' is the translation of Γ, and B' is the translation of B. The translation of clauses with multi-valued methods in the head follows from the translation of clauses with functional methods in the head.

The predicate $app_{\tau,\$m}(t, t_1, \ldots, t_k)$ will be true when functional method m has been *applied* to instances t of type τ with arguments $t_1, \ldots, t_k$. That is, predicate $app_{\tau,\$m}(t, t_1, \ldots, t_k)$ will be true if the body of the rule for functional method m of arity k on instances t of type τ with arguments $t_1, \ldots, t_k$ is true.

Example 5.1.1 In this example, we identify if there is a malfunction in the temperature sensor and identify the problem. Assume a set of declarations D including:

$temp\_sensor{:}part$	$low{:}level$
$high{:}level$	$failure{:}tf$
$csf < sf$	$cf < csf$
$fsf < csf$	$tf < cf$
$tf < fsf.$	

Consider the Gulog program P with respect to the set of declarations D:

$\{x{:}sf\} \vdash x[problem \rightarrow \text{“system failure”}]$
$\{x{:}cf\} \vdash x[problem \rightarrow \text{“over efficient condenser”}] \leftarrow x[condenser \rightarrow low]$
$\{x{:}csf\} \vdash x[problem \rightarrow \text{“cooling system failure”}] \leftarrow x[temp \rightarrow high]$
$\{x{:}fsf\} \vdash x[problem \rightarrow \text{“cooling system flow high”}] \leftarrow x[flow \rightarrow high]$
$\{x{:}tf\} \vdash x[problem \rightarrow \text{“total failure”}]$
$\{x{:}cf\} \vdash x[malfunction \rightarrow temp\_sensor] \leftarrow \neg x[temp \rightarrow high] \wedge \neg x[temp \rightarrow low].$

This is translated to the Datalog program P':

$\$part(temp\_sensor).$	$\$level(low).$
$\$level(high).$	$\$tf(failure).$
$\$sf(x) \leftarrow \$csf(x).$	$\$csf(x) \leftarrow \$cf(x).$
$\$csf(x) \leftarrow \$fsf(x).$	$\$cf(x) \leftarrow \$tf(x).$
$\$fsf(x) \leftarrow \$tf(x).$	

$\$problem(x, \text{“system failure”}) \leftarrow \$sf(x) \wedge \neg app_{cf,\$problem}(x) \wedge \neg app_{csf,\$problem}(x) \wedge$
$\quad \neg app_{fsf,\$problem}(x) \wedge \neg app_{tf,\$problem}(x).$
$\$problem(x, \text{“over efficient condenser”}) \leftarrow \$cf(x) \wedge \$condenser(x, low) \wedge$
$\quad \neg app_{tf,\$problem}(x).$

$\$problem(x, \text{“cooling system failure”}) \leftarrow \$csf(x) \wedge \$temp(x, high) \wedge$
$\quad \neg app_{fsf,\$problem}(x) \wedge \neg app_{cf,\$problem}(x) \wedge \neg app_{tf,\$problem}(x).$
$\$problem(x, \text{“cooling system flow high”}) \leftarrow \$fsf(x) \wedge \$flow(x, high) \wedge$
$\quad \neg app_{tf,\$problem}(x).$
$\$problem(x, \text{“total failure”}) \leftarrow \$tf(x).$
$\$malfunction(x, temp\_sensor) \leftarrow \$cf(x) \wedge \neg\$temp(x, high) \wedge \neg\$temp(x, low).$

$app_{sf,\$problem}(x) \leftarrow \$sf(x).$
$app_{cf,\$problem}(x) \leftarrow \$cf(x) \wedge \$condenser(x, low).$
$app_{csf,\$problem}(x) \leftarrow \$csf(x) \wedge \$temp(x, high).$
$app_{fsf,\$problem}(x) \leftarrow \$fsf(x) \wedge \$flow(x, high).$
$app_{tf,\$problem}(x) \leftarrow \$tf(x).$
$app_{cf,\$malfunction}(x) \leftarrow \$cf(x) \wedge \neg\$temp(x, high) \wedge \neg\$temp(x, low).$ $\Box$

Inheritance with overriding in a Gulog program can thus easily be translated to inference with negation in a Datalog program.

In order to investigate the properties of the translation, we also require a translation from the interpretations of a Gulog program to an interpretation of the corresponding Datalog program. We now describe such a translation. Given an interpretation I of program P with respect to a set of declarations D the translation I' of I (with respect to P and D) is constructed as follows. Each predicate or method atom in I is translated to a predicate atom in I' as given above. Every object in D is assigned atoms in I' that are consistent with the object and type hierarchy declarations in D. We call these atoms *type atoms* and the related predicates *type predicates.* Let $C = \Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t'] \leftarrow B$ (respectively, $C = \Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t'] \leftarrow B$) be a clause in P defining functional (respectively, multi-valued) method m of arity n on instances t of type τ. For every instance $C\theta$ of C consistent with the typing Γ, $app_{\tau,\$m}(t, t_1, \ldots, t_n)\theta \in I'$ (respectively, $app_{\tau,\$\$m}(t, t_1, \ldots, t_n)\theta \in I'$) if for every literal B_i in B, B_i is a positive literal and $B_i\theta \in I$ or B_i is a negative literal $\neg B_{in}$ and $B_{in}\theta \notin I$. We call these atoms *app-atoms* and the related predicates *app-predicates.*

The translation from Gulog to Datalog programs described in this chapter is like the dynamic rewriting from $datalog^{meth}$ to Datalog described by Abiteboul et al. in [7]. In their rewriting, redundant subgoals are generated in the clauses that define the applicable predicates (*app*-predicates). They claim that their rewriting is well-founded for positive $datalog^{meth}$ programs. They do not have a similar result for $datalog^{meth}$ programs that contain negative literals. We show below that any i-stratified Gulog program is translated to a locally stratified Datalog program.

We now study the relationships between the original Gulog program P with respect to a set of declarations D and its Datalog translation P', between the preferred model of P with respect to D and the perfect model of P', and between computed answers with respect to P' and correct answers with respect to P and D.

Lemma 5.1.1 *Let P be an i-stratified program with respect to a set of declarations D and P' be the Datalog translation of P with respect to D. Then P' is locally stratified.*

Proof As P with respect to D is i-stratified, a mapping μ consistent with the definition of i-stratification exists on P. We now demonstrate that a mapping μ' consistent with the definition of local stratification exists on P'. We start by defining the construction of the mapping μ' based on the mapping μ, the program P with respect to D, and its translation P'. Let θ be a substitution.

- In μ' each ground type atom of P' is assigned to the lowest level.
- For each ground atom $(\Gamma \vdash B)\theta$, which is in the body of a clause $C\theta$, and not in the head of a clause $C'\theta'$ in P, and is assigned level i in μ, the translation of $(\Gamma \vdash B)\theta$ is assigned level i in μ'.
- For each ground atom $(\Gamma \vdash A)\theta$, which is the head of a clause $C\theta$ in P, and is assigned level i in μ, the translation of $(\Gamma \vdash A)\theta$ is assigned level $i + 1$ in μ'.
- For each ground instance of a clause $(app_{\tau,m}(t, t_1, \ldots, t_k) \leftarrow B)\theta$ in P', let $B_i\theta$ (respectively, $\neg B_i\theta$) be a literal in $B\theta$ with the greatest level j with respect to μ. If $\neg B_i\theta$ is a negative literal then $app_{\tau,m}(t, t_1, \ldots, t_k)\theta$ is assigned level $j + 1$ in μ', else if $B_i\theta$ is a positive literal then $app_{\tau,m}(t, t_1, \ldots, t_k)\theta$ is assigned level j in μ'.

We now show that if there is a mapping μ on P consistent with the definition of i-stratification then μ', a mapping constructed using the procedure above, is a mapping on P' consistent with the definition of local stratification. We have to show that if $\mu(\alpha) < \mu(\beta)$ then $\mu'(\alpha') < \mu'(\beta')$. Suppose $C = \Gamma \vdash A \leftarrow B$ is a clause in P, where $\Gamma \vdash A$ is a typed predicate atom. This clause is translated to the clause $C' = A' \leftarrow \Gamma' \wedge B'$ in P', where A' is the translation of A, Γ' is the translation of Γ, and B' is the translation of B. Based on the above construction, for every ground instance $C'\theta$ of C', for each positive literal B'_i in B', $\mu'(B'_i\theta) \leq \mu'(A'\theta)$, for each negative literal $\neg B'_i$ in B', $\mu'(B'_i\theta) < \mu'(A'\theta)$, and for each atom Γ'_i in Γ', $\mu'(\Gamma'_i\theta) < \mu'(A'\theta)$.

Suppose $C = \Gamma \vdash t[m@t_1, \ldots, t_k \rightarrow t'] \leftarrow B$ is a clause defining a functional method m of arity k on instances t of type τ in P with respect to D. Suppose also that $C_i = \Gamma_i \vdash t_i[m@t_{i1}, \ldots, t_{ik} \rightarrow t'_i] \leftarrow B_i$, for $1 \leq i \leq n$ are clauses in P with respect to D for method m of arity k defined on instances t_i of types $\sigma_1, \ldots, \sigma_n$, which are all (direct or indirect) subtypes of τ. Clause C is translated to the following clauses in P':

$$app_{\tau,\$m}(t, t_1, \ldots, t_k) \leftarrow \Gamma' \wedge B'.$$
$$\$m(t, t_1, \ldots, t_k, t') \leftarrow \Gamma' \wedge B' \wedge \neg app_{\sigma_1,\$m}(t, t_1, \ldots, t_k) \wedge \cdots \wedge \neg app_{\sigma_n,\$m}(t, t_1, \ldots, t_k),$$

where $\$m(t, t_1, \ldots, t_k, t')$ is the translation of $t[m@t_1, \ldots, t_k \rightarrow t']$, Γ' is the translation of Γ and B' is the translation of B. In the level mapping μ, $\mu((t_i[m@t_{i1}, \ldots, t_{ik} \rightarrow t'_i])\theta_i) \leq \mu((t[m@t_1, \ldots, t_k \rightarrow t'])\theta)$, where $\{t_i/a, t_{ij}/a_j, t'_i/a'_i\} \in \theta_i$ and $\{t/a, t_j/a_j, t'/a'\} \in \theta$ and $a'_i \neq a'$, for $1 \leq j \leq k$. Based on the above construction of μ',

$$\mu'(app_{\sigma_i,\$m}(t, t_1, \ldots, t_k)\theta) < \mu'(\$m(t, t_1, \ldots, t_k, t')\theta),$$
$$\mu'(app_{\tau,\$m}(t, t_1, \ldots, t_k)\theta) \leq \mu'(\$m(t, t_1, \ldots, t_k, t')\theta),$$
$$\mu'(\Gamma'\theta) \leq \mu'(app_{\tau,\$m}(t, t_1, \ldots, t_k)\theta),$$

for each positive literal B'_i in B', $\mu'(B'_i\theta) \leq \mu'(app_{\tau,\$m}(t, t_1, \ldots, t_k)\theta)$, and for each negative literal B'_i in B', $\mu'(B'_i\theta) < \mu'(app_{\tau,\$m}(t, t_1, \ldots, t_k)\theta)$. The results for clauses with multi-valued methods in the head follows from those for clauses with functional methods in the head.

We have demonstrated that if there is a mapping on P consistent with the definition of i-stratification then there is a mapping on P' consistent with the definition of local stratification. □

In the above construction of μ', a level mapping on P', it is possible to assign all the type atoms to the lowest level because they depend only on other type atoms.

The following example illustrates the construction of a level mapping for the translated program based on the level mapping of the original program as described in the proof above.

Example 5.1.2 Consider program P in Example 5.1.1. A level mapping μ on P assigns atoms $failure[flow \rightarrow high]$, $failure[temp \rightarrow high]$, $failure[temp \rightarrow low]$, $failure[condenser \rightarrow low]$, $failure[problem \rightarrow$ "total failure"$]$, $failure[problem \rightarrow$ "cooling system flow high"$]$ to 1; and $failure[problem \rightarrow$ "system failure"$]$, $failure[problem \rightarrow$ "over efficient condenser"$]$, $failure[malfunction \rightarrow temp\_sensor]$, $failure[problem \rightarrow$ "cooling system failure"$]$ to 2.

Consider the program P' in Example 5.1.1. A level mapping μ' on P' can be constructed as given in the proof of Lemma 5.1.1. This level mapping is given in Table 5.1. □

Lemma 5.1.2 *Let P be a program with respect to a set of declarations D and M a model of P with respect to D. Let P' be the translation of P with respect to D and M' the translation of M with respect to P and D. Then M' is a model of P'.*

Proof We need to show that every ground instance of a clause C' in P' is true in M'.

Suppose $C = \Gamma \vdash A \leftarrow B$ is a clause in P with respect to D, where $\Gamma \vdash A$ is a typed predicate atom. Then there is a clause $C' = A' \leftarrow \Gamma' \wedge B'$ in P', where A' is the translation of A, Γ' is the translation of Γ, and B' is the translation of B. Because M is a model of P with respect to D, $C\theta$ is true in M for all ground instances $C\theta$ of C. As every atom in M is

level	atoms
1	$\$fsf(failure), \$cf(failure), \$csf(failure), \$sf(failure),$ $\$flow(failure, high), \$temp(failure, high), \$temp(failure, low),$ $\$part(temp\_sensor), \$string($"total failure"$), \$string($"system failure"$),$ $\$string($"cooling system flow high"$), \$string($"cooling system failure"$),$ $app_{sf,\$problem}(failure), app_{fsf,\$problem}(failure), app_{csf,\$problem}(failure),$ $app_{tf,\$problem}(failure), \$tf(failure), \$condenser(failure, low),$ $app_{cf,\$problem}(failure), \$problem(failure,$ "total failure"$)$
2	$\$problem(failure,$ "cooling system flow high"$)$ $\$malfunction(failure, temp\_sensor)$
3	$\$problem(failure,$ "over efficient condenser"$),$ $\$problem(failure,$ "cooling system failure"$),$ $\$problem(failure,$ "system failure"$)$

Table 5.1: A level mapping on P'

mapped to an atom in M', $M' \models C'\theta$ for every θ for which $C\theta$ is true in M. For every other ground instance of C', $C'\theta'$, $M' \models C'\theta'$ because $\Gamma'\theta'$ is false in M'.

Now suppose $C = \Gamma \vdash t[m@t_1, \ldots, t_p \rightarrow t'] \leftarrow B$ is a clause defining a functional method m of arity p on instances t of type τ in P with respect to D. Suppose also that $C_i = \Gamma_i \vdash t_i[m@t_{i1}, \ldots, t_{ip} \rightarrow t'_i] \leftarrow B_i$, for $1 \leq i \leq n$, are clauses in P for method m of arity p defined on instances t_i of types σ_1, ... , σ_n, which are all (direct or indirect) subtypes of τ. Clause C is translated to the following clauses in P':

$$C'_\tau = app_{\tau,\$m}(t, t_1, \ldots, t_p) \leftarrow \Gamma' \wedge B'.$$
$$C' = \$m(t, t_1, \ldots, t_p, t') \leftarrow \Gamma' \wedge B' \wedge \neg app_{\sigma_1,\$m}(t, t_1, \ldots, t_p) \wedge \cdots \wedge \neg app_{\sigma_n,\$m}(t, t_1, \ldots, t_p),$$

where $\$m(t, t_1, \ldots, t_p, t')$ is the translation of $t[m@t_1, \ldots, t_p, t']$, Γ' is the translation of Γ and B' is the translation of B.

We first consider substitutions that do not satisfy Γ. As every atom in M is mapped to an atom in M', $M' \models C'_\tau\theta$ and $M' \models C'\theta$, for all instances of $C'_\tau\theta$ and $C'\theta$, where θ does not satisfy Γ because $\Gamma'\theta$ is false in M'.

We next consider substitutions that do satisfy Γ. For all ground instances $C\theta$ of C, there are two cases we must consider. Either $C\theta$ is true in M or there is an instance $C_k\theta_k$ of a clause C_k that overrides $C\theta$.

In the first case, $C\theta$ is true in M and there is no θ_i such that $(\Gamma_i \vdash B_i)\theta_i$ is true in M, where $\{t_i/a, t_{ij}/a_j, t'_i/a'_i\} \in \theta_i$, $\{t/a, t_j/a_j, t'/a'\} \in \theta$ and $a'_i \neq a'$, for $1 \leq j \leq p$. Then $M' \models C'_\tau\theta$ and $M' \not\models app_{\sigma_i,\$m}(t, t_1, \ldots, t_p)\theta$, for $1 \leq i \leq n$, so $M' \models C'\theta$.

In the second case, $C_k\theta_k$ is true in M and $t[m@t_1, \ldots, t_p, t']\theta$ is not true in M, where $\{t_k/a, t_{kj}/a_j, t'_k/a'_k\} \in \theta_k$, $\{t/a, t_j/a_j, t'/a'\} \in \theta$ and $a'_k \neq a'$, for $1 \leq j \leq p$. Then $M' \models app_{\sigma_k,\$m}(t, t_1, \ldots, t_p)\theta$, and $\$m(t, t_1, \ldots, t_p, t')\theta$ is not true in M', so $M' \models C'\theta$, and because of the way M' is constructed, $M' \models C'_\tau\theta$. The results for clauses with multi-valued methods in the head follows from those for clauses with functional methods in the head.

We have shown that M' is a model of P'. □

The next lemma, which demonstrates a relationship between the preferred model of a program and the perfect model of its translation, uses the concept of "negative dependency". The literals in the body of a ground clause C, which is possibly overridden by a ground clause C', depend negatively on the head of the clause C'. As in logic programming, the negative literals in the body of a clause also depend negatively on the head of the clause. It is easier to define "negative dependency" using a dependency graph. In this dependency graph, the nodes are ground atoms.

Definition We define a *dependency graph* for a program P with respect to a set of declarations D as follows. Let $[P]_D$ denote the set of all ground instances of clauses in P with respect to D. The nodes in the graph correspond to the ground atoms in the Herbrand base of P with respect to D. A positive arc $\phi \overset{+}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there is a clause $\phi \leftarrow \ldots \wedge \psi \wedge \ldots$ in $[P]_D$. A negative arc $\phi \overset{-}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there is a clause $\phi \leftarrow \ldots \wedge \neg\psi \wedge \ldots$ in $[P]_D$. A positive arc $\phi \overset{+}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there are clauses $C_2 = \phi \leftarrow B_2$ and $C_1 = \psi \leftarrow B_1$ in $[P]_D$, where C_1 possibly overrides C_2. A negative arc $\phi \overset{-}{\leftarrow} \psi$ connects a pair of nodes ϕ, ψ if there are clauses $C_2 = \phi \leftarrow B$ and $C_1 = A \leftarrow \ldots \wedge \psi \wedge \ldots$ or $C_1 = A \leftarrow \ldots \wedge \neg\psi \wedge \ldots$ in $[P]_D$, where C_1 possibly overrides C_2. We say a ground atom A_1 *depends on* a ground atom A_2 if there is a path from A_2 to A_1 in the dependency graph. We say a ground atom A_1 *depends negatively on* a ground atom A_2 if there is a path from A_2 to A_1 and at least one of the arcs in the path is a negative arc.

Lemma 5.1.3 *Let P be an i-stratified program with respect to a set of declarations D and P' the Datalog translation of P with respect to D. Let M be the preferred model of P with respect to D and M' the translation of M with respect to D. Then M' is the perfect model of P'.*

Proof Suppose M' is not the perfect model of P'. Then there is a model $N' \ll M'$ such that $N' \models P'$.

Consider the translation from N' to N defined as follows.

- Remove all type atoms and *app*-atoms from N', and
- translate the remaining atoms in N' to predicate atoms, functional and multi-valued method atoms in N.

The resulting interpretation N is clearly a model of P with respect to D. We now show $N' \ll M'$ implies that $N \ll M$, which contradicts the assumption that M' is not perfect. Assume there is an $A \in N - M$ then there is an $A' \in N' - M'$ (where A' is the translation of A) and there is a $B' \in M' - N'$ such that $A' < B'$.

We need to show that if $B' \in M' - N'$ then there is a $B \in M - N$. That is, B' is not a type atom and if B' is a *app*-atom then there is another atom B^* in $M' - N'$ that is not a type or *app*-atom. Due to the construction of M', M' contains the minimal model of the type atoms. This part of the interpretation is based on the declarations D so any type atom in M' is also in N'. If B' is a *app*-atom then either there is a type atom B^* in $M' - N'$ or there is a predicate or method atom B^* in $M' - N'$ on which B' depends in P'. We proved above that there are no type atoms in $M' - N'$. Thus B^* is a predicate or method atom so we can assign B^* to B'. As B' is the image of a predicate or method atom B then $B \in M - N$. So there is a $B \in M - N$.

We now need to show that if $A' < B'$ then $A <_p B$. The relationship $A <_p B$ holds if A is negatively dependent on B in P with respect to D. Due to the way P' is constructed from P with respect to D, if $A' < B'$ in P' then $A <_p B$.

With these results, if $N' \ll M'$ then $N \ll M$. As M is a preferred model of P with respect to D, this is a contradiction so M' is the perfect model of P'. □

Proposition 5.1.1 *Let P be an i-stratified program with respect to a set of declarations D. Then P with respect to D has a unique preferred model.*

Proof Suppose M and N are preferred models of P. Let M', N' and P' be translations of M, N and P respectively. Then the translations M' and N' are perfect models of P' (by Lemma 5.1.3). Because P' is locally stratified (Lemma 5.1.1), P' has a unique perfect model so P must have a unique preferred model. □

In Section 3.5, we showed that simple Gulog programs have unique preferred models. We have given an alternative proof above.

Proposition 5.1.2 *Let P with respect to a set of declarations D be an i-stratified program and Q an atomic query $\Gamma \vdash \ \leftarrow A$. Let P' be the translation of P with respect to D and Q' the translation of Q with respect to D. If θ is a computed answer for $P' \cup \{Q'\}$, then θ is a correct answer for $P \cup \{Q\}$.*

We postpone the proof of this proposition to Section 6.1 where we prove a more general result.

In Chapter 4, we described bottom-up and top-down query evaluation procedures for this language. Proposition 5.1.2 provides an alternative procedure for evaluating Gulog queries.

5.2 Summary

In this chapter we presented a translation from Gulog to Datalog with negation. Similar translations are described in [7, 36, 65, 72, 81]. The translation presented in this chapter is a generalization of the translation from C-logic to first-order logic [36]. In particular, there is no concept of overriding in C-logic. Similarly, the translation from F-logic to first-order logic that is sketched in [65] covers only the monotonic aspects of F-logic and doesn't include overriding. The semantics of inheritance with overriding in *L&O* is different to the semantics of inheritance with overriding in Gulog, so the translation described by McCabe is different from the translation presented in this chapter. McCabe's translation is similar to the static rewriting in [7]. The translation presented in this chapter is a generalization of dynamic rewriting described in [7]. The authors of [7] do not distinguish between functional and multi-valued methods, and do not consider multiple inheritance. They do claim that their rewriting is well-founded for positive programs. However they have no results for programs with negation. The translation described in [72] is different from the translation we describe because they allow negative literals in the heads of rules. However the principle of the translation is very similar. The body of the translation of a possibly overridden clause contains the translation of the body of the possibly overridden clause and the negation of the translation of the body of any possibly overriding clauses. We show that any i-stratified Gulog program (which includes programs with negative literals in the body of the rules) is translated to a locally stratified Datalog program.

We extend the translation described in this chapter to a less restricted class of programs in Section 6.1.

Chapter 6

Extensions

In this chapter we describe three directions in which Gulog can be extended. These extensions are unrelated. However they illustrate that it is possible to give a mathematical description of a wider range of object-oriented and deductive features. The first extension is a generalization of the class of simple programs. We show that there is a translation from a more general class of Gulog programs, m-simple programs, to modularly stratified programs. Hence, there is an efficient evaluation procedure for m-simple programs.

In the second extension we include some type information in the program in a very restricted way. This provides another way of dealing with conflicts due to multiple inheritance and a way of modeling monotonic inheritance. We again describe a translation to Datalog and prove it is correct.

The third extension is the addition of aggregate and arithmetic operators to Gulog. We describe an evaluation procedure using a translation from a Gulog program with aggregate and arithmetic operators to a modularly stratified program with aggregates.

6.1 Generalizations

Some programs have intuitively well-defined meanings, but no or several preferred models. In this section we identify a class of programs that includes the simple programs, and have an intuitive and well-defined semantics, which we present. This class of programs takes computation into account. It bears the same relationship to simple programs as modularly stratified Datalog programs bears to locally stratified Datalog programs.

In the following example we present a program that is not simple and describe the intuitive meaning of this program.

Example 6.1.1 Assume a set of declarations D including $failure{:}rvf$, $rvf < psf$ and $psf < sf$. Let P with respect to D be:

$$\{x{:}sf\} \vdash x[treatment \twoheadrightarrow \text{“call supervisor”}]$$
$$\{x{:}psf\} \vdash x[treatment \twoheadrightarrow \text{“lower pressure”}] \leftarrow x[pressure \rightarrow high]$$
$$\{x{:}rvf\} \vdash x[treatment \twoheadrightarrow \text{“open relief valve”}] \leftarrow x[treatment \twoheadrightarrow \text{“lower pressure”}].$$

Program P with respect to D is not i-stratified because the relationship of the ground atoms of $[P]_D$ requires

$$\mu(failure[treatment \twoheadrightarrow \text{“lower pressure”}]) < \mu(failure[treatment \twoheadrightarrow \text{“lower pressure”}]).$$

However we believe it has an intuitive meaning.
There is no reason to believe that $failure[pressure \rightarrow high]$ is true thus there is no reason to believe that $failure[treatment \twoheadrightarrow$ “lower pressure”$]$ or $failure[treatment \twoheadrightarrow$ “open re-

lief valve"] is true. So the natural meaning of the program is $failure[treatment \twoheadrightarrow$ "call supervisor"]. □

To introduce this class of programs we first need some preliminary definitions. Our definition of m-stratified is based on the definition of modular stratification [93]. We give the relevant revised definitions here to show how they are altered to deal with inheritance with overriding in Gulog rather than negation in Datalog.

Definition A predicate or method p *depends on* a predicate or method q if there is a sequence of clauses $r_0, \ldots, r_{n-1}$ with methods or predicates $p_0, \ldots, p_{n-1}$ in the head respectively such that:

- $p = p_0$ and $q = p_n$, and
- for $i = 1, \ldots, n$, p_i appears in the body of r_{i-1}.

A predicate or method p is *recursive* in a predicate or method q if p depends on q and q depends on p.

Definition Let F be a component (that is, a subset of clauses) of a program P with respect to a set of declarations D. The component F is a *complete component* if for every predicate or method p appearing in the head of a clause in F,

- all clauses in P with respect to D with head p are in F, and
- if p is recursive in a predicate or method q, then all clauses in P with respect to D with head q are in F.

If the predicate or method p appears in the head of a clause in F then we say p *belongs to* F. If the predicate q appears in the body of a clause in F, but does not belong to F, then we say q is *used by* F.

Recursiveness is an equivalence relation between predicates and methods. Every predicate and method has a unique minimal complete component to which it belongs. The minimum complete components have a natural relation associated with them: $F_1 \sqsubset F_2$ if some predicate belonging to F_1 is used by F_2. The transitive closure of $\sqsubset$ is called the *dependency* relation between components and is denoted $\prec$. From now on when we refer to a component of a program, we mean a minimal complete component.

We say a component F_k has *level* 1 if there is no component F such that $F \prec F_k$. A component F_k has *level* $n+1$ if there is a component F with level n such that $F \sqsubset F_k$.

Definition Let F be a program component, and S be the set of predicates and methods used by F. Suppose S is fully defined by an (typed Herbrand) interpretation M, that is, every typed ground instance of a predicate or method in S is either true or false in M. Form $I(F)$, the typed instantiation of F, by substituting typed objects for all typed variables in the clauses of F. Delete from $I(F)$ all clauses having a subgoal whose predicate or method is in S but which is false in M. From the remaining clauses, delete all subgoals having predicates or methods in S (which must be true) to leave a set of instantiated clauses $R_M(F)$, which is the *reduction of F modulo M*.

We now give a constructive definition of the class of m-stratified Gulog programs. It is based on the construction of the well-founded model of a program that is briefly described in [93].

Definition Let $\prec$ be the dependency relation between components. We say that program P with respect to a set of declarations D is *m-stratified with associated model* M_P if,

- for every component F_i of P with respect to D,
  - $F_i' = \bigcup_{F \prec F_i} F$ is m-stratified with associated model M_i', and
  - $R_{M_i'}(F_i)$ is i-stratified with (unique) preferred model M_i, and
- $M_P = \bigcup\{M_i \mid F_i$ is a component of P with respect to D with associated model $M_i\}$.

The next example demonstrates the construction of the associated model of a program.

Example 6.1.2 Consider program P with respect to the set of declarations D in Example 6.1.1. Component $F_1 = \emptyset$ defines the method *pressure*. Component F_2 defines the method *treatment*:

$\{x{:}sf\} \vdash x[treatment \twoheadrightarrow$ "call supervisor"$]$
$\{x{:}psf\} \vdash x[treatment \twoheadrightarrow$ "lower pressure"$] \leftarrow x[pressure \rightarrow high]$
$\{x{:}rvf\} \vdash x[treatment \twoheadrightarrow$ "open relief valve"$] \leftarrow x[treatment \twoheadrightarrow$ "lower pressure"$]$.

The methods used by F_2 are $S = \{pressure\}$. The set of methods, S, is fully defined by the Herbrand interpretation $M_1 = \emptyset$. The typed instantiation of F_2 is:

$\{failure{:}sf\} \vdash failure[treatment \twoheadrightarrow$ "call supervisor"$]$
$\{failure{:}psf\} \vdash failure[treatment \twoheadrightarrow$ "lower pressure"$] \leftarrow$
$\quad failure[pressure \rightarrow high]$
$\{failure{:}rvf\} \vdash failure[treatment \twoheadrightarrow$ "open relief valve"$] \leftarrow$
$\quad failure[treatment \twoheadrightarrow$ "lower pressure"$]$.

The reduction $R_{M_2'}(F_2)$ is:

$\{failure{:}sf\} \vdash failure[treatment \twoheadrightarrow$ "call supervisor"$]$
$\{failure{:}rvf\} \vdash failure[treatment \twoheadrightarrow$ "open relief valve"$] \leftarrow$
$\quad failure[treatment \twoheadrightarrow$ "lower pressure"$]$.

The Herbrand interpretation $M_2 = \{failure[treatment \twoheadrightarrow$ "call supervisor"$]\}$. The reductions of each of the components of program P is i-stratified so P is m-stratified. The associated model of P is $\{failure[treatment \twoheadrightarrow$ "call supervisor"$]\}$. $\square$

We now define the class of m-simple programs that generalize simple programs in the same way that m-stratification generalizes i-stratification.

Definition A program P with respect to a set of declarations D is *m-simple* if the following conditions hold:

- P with respect to D is m-stratified, and
- P with respect to D is well-defined.

Lemma 6.1.1 *Let P with respect to a set of declarations D be an i-stratified program. Then P with respect to D is m-stratified.*

Proof As program P with respect to D is i-stratified, each component F of program P with respect to D is i-stratified. The reduction of each component is also i-stratified so program P with respect to D is m-stratified. $\square$

Lemma 6.1.2 *Let P with respect to a set of declarations D be an m-stratified program. Then P with respect to D has a unique associated model of P with respect to D.*

Proof This result follows from the construction of the associated model and Lemma 3.5.1. □

It follows that m-stratified programs strictly generalize i-stratified programs and that the associated model of an i-stratified program P with respect to a set of declarations D is the unique preferred model of P with respect to D.

The class of m-simple programs extends the class of simple programs. It is an interesting class of programs because each m-simple program has a unique associated model, and query evaluation procedures can be defined for this class of programs.

The remaining results describe the relationship between m-simple programs and their translations. The first lemma, which draws a parallel between the associated model of an m-stratified program and the well-founded model of its translation, uses the concept of "equivalence" of models.

Definition Let M_P be a model of a Gulog program and $M_{P'}$ be a model of a Datalog program. We say M_P is *equivalent* to $M_{P'}$ if the translation of every atom in M_P is in $M_{P'}$ and, for every atom A' in $M_{P'}$, which is the translation of a Gulog method or predicate atom A, A is in M_P.

Example 6.1.3 Consider program P with respect to the set of declarations D, and its associated model in Examples 6.1.1 and 6.1.2. Consider P', the translation of P with respect to D:

$\$rvf(failure).$
$\$psf(x) \leftarrow \$rvf(x).$
$\$sf(x) \leftarrow \$psf(x).$
$\$\$treatment(x, \text{“call supervisor”}) \leftarrow$
  $\$sf(x) \wedge \neg app_{psf,\$\$treatment}(x) \wedge \neg app_{rvf,\$\$treatment}(x).$
$\$\$treatment(x, \text{“lower pressure”}) \leftarrow$
  $\$psf(x) \wedge \$pressure(x, high) \wedge \neg app_{rvf,\$\$treatment}(x).$
$\$\$treatment(x, \text{“open relief valve”}) \leftarrow \$rvf(x) \wedge \$\$treatment(x, \text{“lower pressure”}).$
$app_{sf,\$\$treatment}(x) \leftarrow \$sf(x).$
$app_{psf,\$\$treatment}(x) \leftarrow \$psf(x) \wedge \$pressure(x, high).$
$app_{rvf,\$\$treatment}(x) \leftarrow \$rvf(x) \wedge \$\$treatment(x, \text{“lower pressure”}).$

The associated model of program P with respect to D is $M_P = \{failure[treatment \twoheadrightarrow \text{“call supervisor”}]\}$. and the well-founded model of P' is $M_{P'} = \{\$rvf(failure), \$psf(failure), \$sf(failure), \$\$treatment(failure, \text{“call supervisor”})\}$. The translation of every atom in M_P is in $M_{P'}$. The only atom in $M_{P'}$ which is a translation of a Gulog method or predicate is $A' = \$\$treatment(failure, \text{“call supervisor”})$. The translation of A' is in M_P. Thus M_P is equivalent to $M_{P'}$. □

Lemma 6.1.3 *Let P with respect to a set of declarations D be an m-stratified program with associated model M_P and P' the Datalog translation of P with respect to D. Then P' is modularly stratified with well-founded model $N_{P'}$, which is equivalent to M_P.*

Proof We need to show that P' is modularly stratified. To do this we must show for every component G_j of P':

- $G'_j = \bigcup_{G \prec G_j} G$ is modularly stratified with well-founded model N'_j, and

- $R_{N'_j}(G_j)$ is locally stratified with (unique) perfect model N_j.

We also need to show that M_P is equivalent to $N_{P'}$, where $N_{P'} = \bigcup\{N_j \mid G_j$ is a component of P' with well-founded model $N_j\}$.

A component F in P with respect to D is translated to one or more components G_i, $i \geq 1$ in P'. One component G_k of G_i will define predicates $m_1, \ldots, m_n$, which are translations of the predicates and methods defined in F. We say that G_k *corresponds to* F. The other components G_i, $i \neq k$, will define type and *app*-predicates and $G_i \prec G_k$. (Recall the description of *app*-predicates in Chapter 5. A predicate $app_{\tau,\$m}(t, t_1, \ldots, t_k)$ will be true when functional method m has been *applied* to instances t of type τ with arguments $t_1, \ldots, t_k$.)

We now prove by induction on the maximum level of components in P with respect to D that P' is modularly stratified with well-founded model $N_{P'}$ that is equivalent to M_P.

Assume the maximum level of components in P with respect to D is 1 and F_1 is a component at this level then $M'_1 = \emptyset$, $R_{M'_1}(F_1) = F_1$, and F_1 is i-stratified with preferred model M_1. There is a corresponding component G_{k_1} of P'. Obviously $G'_{k_1} = \bigcup_{G \prec G_{k_1}} G$ is locally stratified (by Lemma 5.1.1), hence modularly stratified and has a well-founded model $N'_{k_1} = \bigcup_{G_i \prec G_{k_1}} N_i$. Also because F_1 is i-stratified, $R_{N'_i}(G_i)$, for $i \leq k_1$, is locally stratified with perfect model N_i. The preferred model of F_1 is equivalent to the perfect model of $\bigcup_{G \preceq G_{k_1}} G$ (by Lemma 5.1.3). Hence the associated model of F_1 is equivalent to the well-founded model of $\bigcup_{G \preceq G_{k_1}} G$.

Assume the above results hold where the maximum level of components of P with respect to D is $n-1$. Assume F_{n-1} is a component at this level of P with respect to D, where F_{n-1} corresponds to a component G_{k_2} of P'. The program $\bigcup_{G \preceq G_{k_2}} G$ is modularly stratified and its well-founded model $\bigcup\{N_j \mid G_j \preceq G_{k_2}, G_j$ has well-founded model $N_j\}$ is equivalent to $\bigcup\{M_i \mid F_i \preceq F_{n-1}, F_i$ has associated model $M_i\}$.

Assume the maximum level of components in P with respect to D is n and F_n is a component at this level. There is a component G_{k_3} of P' that corresponds to F_n. The components G_i, for $k_2 < i < k_3$, define type and *app*-predicates. These components are locally stratified with perfect models N_i. By the induction hypothesis and because the components G_i, for $k_2 < i < k_3$, are locally stratified with perfect models N_i, $G'_{k_3} = \bigcup_{G \prec G_{k_3}} G$ is modularly stratified with well-founded model $N'_{k_3} = \bigcup_{G_i \prec G_{k_3}} N_i$. Model N'_{k_3} is equivalent to M'_n so $R_{N'_{k_3}}(G_{k_3})$ is a translation of $R_{M'_n}(F_n)$ and N_{k_3} is equivalent to M_n.

Because the associated model of a component of a Gulog program is equivalent to the well-founded model of the corresponding Datalog component the associated model M_P is equivalent to $N_{P'}$. $\square$

The next example shows that the associated model of program P in Example 6.1.1 is equivalent to the well-founded model of the Datalog translation of P.

Example 6.1.4 Consider program P with respect to the set of declarations D, and its associated model in Examples 6.1.1 and 6.1.2. Consider P', the translation of P with respect to D, given in Example 6.1.3. Component G_1 defines predicate $\$rvf$. The model of $R_\emptyset(G_1)$ is $N_1 = \{\$rvf(failure)\}$. Component G_2 defines predicate $\$psf$. The model of $R_{N'_2}(G_2)$ is $N_2 = \{\$psf(failure)\}$. Component G_3 defines the predicate $\$sf$. The model of $R_{N'_3}(G_3)$ is $N_3 = \{\$sf(failure)\}$. Component G_4 defines predicate $\$pressure$. The model of $R_\emptyset(G_4)$ is $N_4 = \emptyset$.
Component G_5 defines predicate $app_{sf,\$\$treatment}$. The model of $R_{N'_5}(G_5)$ is
$N_5 = \{app_{sf,\$\$treatment}(failure)\}$.
Component G_6 defines predicate $app_{psf,\$\$treatment}$. The model of $R_{N'_6}(G_6)$ is $N_6 = \emptyset$.
Component G_7 defines predicates $\$\$treatment$ and $app_{rvf,\$\$treatment}$:

$\$\$treatment(x,$ "call supervisor"$) \leftarrow$
$\quad \$sf(x) \wedge \neg app_{psf,\$\$treatment}(x) \wedge \neg app_{rvf,\$\$treatment}(x).$
$\$\$treatment(x,$ "lower pressure"$) \leftarrow$
$\quad \$psf(x) \wedge \$pressure(x, high) \wedge \neg app_{rvf,\$\$treatment}(x).$
$\$\$treatment(x,$ "open relief valve"$) \leftarrow \$rvf(x) \wedge \$\$treatment(x,$ "lower pressure"$).$
$app_{rvf,\$\$treatment}(x) \leftarrow \$rvf(x) \wedge \$\$treatment(x,$ "lower pressure"$).$

Component G_7 corresponds to component F_2. The typed instantiation of G_7 is:

$\$\$treatment(failure,$ "call supervisor"$) \leftarrow$
$\quad \$sf(failure) \wedge \neg app_{psf,\$\$treatment}(failure) \wedge \neg app_{rvf,\$\$treatment}(failure).$
$\$\$treatment(failure,$ "lower pressure"$) \leftarrow$
$\quad \$psf(failure) \wedge \$pressure(failure, high) \wedge \neg app_{rvf,\$\$treatment}(failure).$
$\$\$treatment(failure,$ "open relief valve"$) \leftarrow$
$\quad \$rvf(failure) \wedge \$\$treatment(failure,$ "lower pressure"$).$
$app_{rvf,\$\$treatment}(failure) \leftarrow \$rvf(failure) \wedge \$\$treatment(failure,$ "lower pressure"$).$

The reduction $R_{N_7'}(G_7)$ is:

$\$\$treatment(failure,$ "call supervisor"$) \leftarrow \neg app_{rvf,\$\$treatment}(failure).$
$\$\$treatment(failure,$ "open relief valve"$) \leftarrow app_{rvf,\$\$treatment}(failure).$
$app_{rvf,\$\$treatment}(failure) \leftarrow \$\$treatment(failure,$ "lower pressure"$).$

The model of $R_{N_7'}(G_7)$ is $N_7 = \{\$\$treatment(failure,$ "call supervisor"$)\}$. The reduction of each of the components of P' is locally stratified so P' is modularly stratified. The well-founded model of P' is $\{\$rvf(failure), \$psf(failure), \$sf(failure), \$\$treatment(failure,$ "call supervisor"$)\}$. Clearly the associated model of P with respect to D and the well-founded model of P' are "equivalent". $\square$

The construction of the associated model of a Gulog program is analogous to the construction of the well-founded model of a Datalog program. This analogy allows us to claim that this is an appropriate intuitive semantics for Gulog programs.

The definition of "answer" and "correct answer" for m-stratified programs is a natural extension of the definition for i-stratified programs.

Proposition 6.1.1 *Let P with respect to a set of declarations D be an m-stratified program and Q an atomic query $\Gamma \vdash \; \leftarrow A$. Let P' be the translation of P and Q' the translation of Q with respect to D. If θ is a computed answer for $P' \cup \{Q'\}$, then θ is a correct answer for $P \cup \{Q\}$ with respect to D.*

Proof The query Q is translated to query $Q' = \leftarrow \Gamma' \wedge A'$, where Γ' is the translation of Γ and A' is the translation of A. Let M_P be the associated model of P with respect to D and M_P' the well-founded model of P'. If θ is the computed answer for $P' \cup \{Q'\}$ then $M_P' \models (\Gamma' \wedge A')\theta$ and so $(\Gamma \vdash A)\theta$ is true in M_P (by Lemma 6.1.3). $\square$

We illustrate this with an example.

Example 6.1.5 Consider program P with respect to the declarations D in Example 6.1.1 and the query $Q = \{x{:}rvf, y{:}string\} \vdash \; \leftarrow x[treatment \twoheadrightarrow y]$. The program is translated to P' in Example 6.1.4 and Q is translated to $Q' = \leftarrow \$rvf(x) \wedge \$string(y) \wedge \$\$treatment(x, y)$. The computed answer for $P' \cup \{Q'\}$ is $\{x/failure, y/$"call supervisor"$\}$, which corresponds to $\{x{:}rvf, y{:}string\} \vdash \{x/failure, y/$"call supervisor"$\}$, the correct answer for $P \cup \{Q\}$ with respect to D. $\square$

To summarize, we have introduced a more general class of Gulog programs, and Proposition 6.1.1 provides an efficient evaluation procedure based on evaluation procedures developed for Datalog. It is a subject for future research to determine whether the bottom-up query evaluation procedure and the top-down query evaluation procedure with tabling described in Chapter 4 are sound and complete with respect to the declarative semantics for m-stratified programs.

6.2 Roles

In the language we have defined, the schema declarations are defined separately from the data definitions. We have shown that such a language has a clean semantics similar to that of logic programming. A model contains information about the data only and no information about the typing. In this section we show it is possible to include some type information in a program. We discuss other ways of including type information in Section 7.3. In this section, we introduce a roled-language that allows typed or roled atoms rather than atoms. A consequence is that an interpretation and model then contains roled atoms. This extension provides another way of dealing with ambiguities due to conflicts with multiple inheritance. It also provides a way to model monotonic inheritance [7, 65], that is, inheritance without overriding. Generally, roles provide a means of explicitly controlling from which type a method is inherited. We also discuss the use of the word *super* to refer to the direct supertype of the type of an object.

The following examples motivate the introduction of roled atoms. The first example illustrates how it is possible to use roles to eliminate ambiguities due to conflicts with multiple inheritance. In this example $\{x{:}ta, y{:}integer\} \vdash x{::}student[phone\_number \twoheadrightarrow y]$ is a *roled atom*. This roled atom states that the definition of 0-arity multi-valued method *phone_number* from type *student* is to be applied to objects of type *ta*.

Example 6.2.1 Assume a set of declarations D including $ta < student$, $ta < emp$, and $bob{:}ta$. Consider the following program P with respect to the set of declarations D that states that the phone number of any object of type ta is inherited from type *student*:

$$\{x{:}student, y{:}integer\} \vdash x[phone\_number \twoheadrightarrow y] \leftarrow x[home\_number \rightarrow y]$$
$$\{x{:}emp, y{:}integer\} \vdash x[phone\_number \twoheadrightarrow y] \leftarrow x[work\_number \rightarrow y]$$
$$\{x{:}ta, y{:}integer\} \vdash x[phone\_number \twoheadrightarrow y] \leftarrow x{::}student[phone\_number \twoheadrightarrow y]$$
$$bob[home\_number \rightarrow 3872651]$$
$$bob[work\_number \rightarrow 2822493]$$

In program P, the atom $\{x{:}ta, y{:}integer\} \vdash x{::}student[phone\_number \twoheadrightarrow y]$ is a *roled atom*. Consider declarations D, program P and query $\{y{:}integer\} \vdash \leftarrow bob[phone\_number \twoheadrightarrow y]$. The expected answer is $\{y{:}integer\} \vdash \{y/3872651\}$. □

The second example is taken from [65]. It illustrates the non-monotonic inheritance with which we are familiar.

Example 6.2.2 Assume a set of declarations D including: $female < person$, $writer < person$ and $mary{:}female$. Let P_{nm} be a program with respect to the set of declarations D, in which overriding does occur where P_{nm} is:

$$\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[maiden\_name \rightarrow z]$$
$$\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[pen\_name@y \rightarrow z]$$
$$\{x{:}person, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \rightarrow z]$$
$$mary[last\_name@1992 \rightarrow jones; maiden\_name \rightarrow smith].$$

Consider the query $\{y{:}string\} \vdash\ \leftarrow mary[legal\_names@1992 \twoheadrightarrow y]$ and program P_{nm} with respect to D. The expected answer is $\{y{:}string\} \vdash \{y/smith\}$. □

Recall from Section 2.3 that monotonic inheritance is inheritance without overriding. Without roles, the only way to model monotonic inheritance in Gulog would be to state the body of the inherited clause explicitly as part of the body of the inheriting clause.

Example 6.2.3 Let P_{ex} be the following program with respect to the declarations in Example 6.2.2. In P_{ex}, clause bodies are repeated explicitly to model monotonic inheritance.

$$\begin{array}{l}
\{x{:}person, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \rightarrow z] \\
\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[maiden\_name \rightarrow z] \\
\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \rightarrow z] \\
\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[pen\_name@y \rightarrow z] \\
\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \rightarrow z] \\
mary[last\_name@1992 \rightarrow jones; maiden\_name \rightarrow smith]
\end{array}$$

Consider the query $\{y{:}string\} \vdash\ \leftarrow mary[legal\_names@1992 \twoheadrightarrow y]$ and program P_{ex} with respect to D. The expected answers are $\{\{y{:}string\} \vdash \{y/smith\}, \{y{:}string\} \vdash \{y/jones\}\}$. □

This is not satisfactory. When a definition changes, the program could have to be changed in more than one place. For example, in Example 6.2.3, if you changed the body of the clause for method *legal_names* of arity 1 applied to an object of type *person*, then it would be necessary to change three clauses.

Example 6.2.4 Let P_m be a program with respect to the declarations in Example 6.2.2, in which the values of the set *legal_names* is inherited monotonically. Program P_m is:

$$\begin{array}{l}
\{x{:}person, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \rightarrow z] \\
\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[maiden\_name \rightarrow z] \\
\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow \\
\quad x{::}person[legal\_names@y \twoheadrightarrow z] \\
\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[pen\_name@y \rightarrow z] \\
\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow \\
\quad x{::}person[legal\_names@y \twoheadrightarrow z] \\
mary[last\_name@1992 \rightarrow jones; maiden\_name \rightarrow smith]
\end{array}$$

Consider the query $\{y{:}string\} \vdash \leftarrow mary[legal\_names@1992 \twoheadrightarrow y]$ and the program P_m with respect to D. The expected answers are $\{\{y{:}string\} \vdash \{y/smith\}, \{y{:}string\} \vdash \{y/jones\}\}$. □

The final motivating example illustrates that, using roles, it is possible to inherit from more than one supertype where there are conflicting definitions due to multiple inheritance.

Example 6.2.5 Consider again the declarations of Example 6.2.2. To accumulate the *legal_names* of female writers we introduce another type for female writers in the inheritance hierarchy:

$$\begin{array}{l}
female\_writer < female \\
female\_writer < writer
\end{array}$$

The following clauses added to P_m allow the monotonic accumulation for this new type:

$\{x{:}female\_writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$
$\quad x{::}female[legal\_names@y \twoheadrightarrow z]$
$\{x{:}female\_writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$
$\quad x{::}writer[legal\_names@y \twoheadrightarrow z]$

Consider the query $\{y{:}string\} \vdash \leftarrow mary[legal\_names@1992 \twoheadrightarrow y]$ and the extended program with repect to the extended set of declarations. The expected answers are $\{\{y{:}string\} \vdash \{y/smith\}, \{y{:}string\} \vdash \{y/jones\}, \{y{:}string\} \vdash \{y/brown\}\}$. □

To extend Gulog to include roles many of the definitions need to change. In this section, we consider only positive programs in order to keep the definitions and semantics clear and simple. Adding negation has the usual consequences.

Definition Given a variable typing Γ, a *roled atom* is defined as follows:

- If p is an n-ary predicate symbol with signature $\tau_1 \times \cdots \times \tau_n$ and Γ contains the typings for the variables in t_i, and $\Gamma \vdash t_i{:}\sigma_i$ is a term, where $\sigma_i \leq \tau_i$ $(1 \leq i \leq n)$, then $\Gamma \vdash p(t_1, \ldots, t_n)$ is a *roled predicate atom.*

- If m is a functional n-ary method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \Rightarrow \tau'$ and Γ contains the typings of all the variables in t, t_i, and t', and $\Gamma \vdash t{:}\sigma, \Gamma \vdash t_i{:}\sigma_i$ $(1 \leq i \leq n)$, and $\Gamma \vdash t'{:}\sigma'$ are terms, where $\sigma \leq \sigma_s \leq \tau$, $\sigma_i \leq \tau_i$, and $\sigma' \leq \tau'$, then $\Gamma \vdash t{::}\sigma_s[m@t_1, \ldots, t_n \rightarrow t']$ is a *roled functional method atom.*

- If m is a multi-valued n-ary method symbol with signature $\tau \times \tau_1 \times \cdots \times \tau_n \Rrightarrow \tau'$ and Γ contains the typings of all the variables in t, t_i, and t', and $\Gamma \vdash t{:}\sigma, \Gamma \vdash t_i{:}\sigma_i$ $(1 \leq i \leq n)$, and $\Gamma \vdash t'{:}\sigma'$ are terms where $\sigma \leq \sigma_s \leq \tau, \sigma_i \leq \tau_i$, and $\sigma' \leq \tau'$ then $\Gamma \vdash t{::}\sigma_s[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is a *roled multi-valued method atom.*

The atom $\{t{:}\sigma, \ldots\} \vdash t[m@t_1, \ldots, t_p \rightarrow t']$ (respectively, $\{t{:}\sigma, \ldots\} \vdash t[m@t_1, \ldots, t_p \twoheadrightarrow t']$) is a shorthand for $\{t{:}\sigma, \ldots\} \vdash t{::}\sigma[m@t_1, \ldots, t_p \rightarrow t']$ (respectively, $\{t{:}\sigma, \ldots\} \vdash t{::}\sigma[m@t_1, \ldots, t_p \twoheadrightarrow t']$). The type of atom $\{t{:}\sigma, \ldots\} \vdash t{::}\tau[m@t_1, \ldots, t_p \rightarrow t']$ (respectively, $\{t{:}\sigma, \ldots\} \vdash t{::}\tau[m@t_1, \ldots, t_p \twoheadrightarrow t']$) is τ.

The word *super* can be used to refer to the direct supertype of the type of an object. Let D be a set of declarations. The atom $\Gamma \vdash t{::}super[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $\Gamma \vdash t{::}super[m@t_1, \ldots, t_n \twoheadrightarrow t']$) with respect to D is a shorthand for $\Gamma \vdash t{::}\tau[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $\Gamma \vdash t{::}\tau[m@t_1, \ldots, t_n \twoheadrightarrow t']$) with respect to D where $\Gamma \vdash t{:}\sigma$ is a term and $\sigma < \tau \in D$.

Example 6.2.6 Consider the program in Example 6.2.4. With the keyword *super* it can be written more clearly as:

$\{x{:}person, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[last\_name@y \rightarrow z]$
$\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[maiden\_name \rightarrow z]$
$\{x{:}female, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$
$\quad x{::}super[legal\_names@y \twoheadrightarrow z]$
$\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow x[pen\_name@y \rightarrow z]$
$\{x{:}writer, y{:}year, z{:}string\} \vdash x[legal\_names@y \twoheadrightarrow z] \leftarrow$
$\quad x{::}super[legal\_names@y \twoheadrightarrow z]$
$mary[last\_name@1992 \rightarrow jones; maiden\_name \rightarrow smith]$. □

Definition Given a variable typing Γ, roled atoms $\Gamma \vdash A$, $\Gamma \vdash B_1, \ldots, \Gamma \vdash B_n$, $\Gamma \vdash A \leftarrow B_1 \wedge \cdots \wedge B_n$ is a *roled clause*. The roled atom $\Gamma \vdash A$ is the *head* of the clause and $\Gamma \vdash B_1 \wedge \cdots \wedge B_n$ is the *body*.

The definition for roled goal follows from this.

Definition A *roled goal* is a roled clause of the form $\Gamma \vdash \ \leftarrow B_1 \wedge \cdots \wedge B_n$, that is, a roled clause with an empty head. If Γ is empty we write $\leftarrow B_1 \wedge \cdots \wedge B_n$.

Definition A *roled program* with respect to a set of declarations D is a finite set of roled clauses with respect to the set of declarations, D.

Roled substitutions are like substitutions described in Section 3.3. The ground atom $t[m@t_1, \ldots, t_p \rightarrow t']$ (respectively, $t[m@t_1, \ldots, t_p \twoheadrightarrow t']$), where $\Gamma \vdash t{:}\sigma$ is a term, is a shorthand for $t{::}\sigma[m@t_1, \ldots, t_p \rightarrow t']$ (respectively, $t{::}\sigma[m@t_1, \ldots, t_p \twoheadrightarrow t']$). More importantly, the atoms in an interpretation are now all ground roled atoms. Throughout the remainder of this section we will indicate whether we are referring to roled interpretations or interpretations if it is not obvious from the context.

We now define a class of roled programs which have a natural meaning. The definitions differ from those in Section 3.4 because we are now dealing with roled atoms. There is a small extension to the definition of i-stratified programs. In the following definition we refer to the unroled equivalent of a ground roled atom. The *unroled equivalent* of a roled method atom $\Gamma \vdash t{::}\sigma_s[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $\Gamma \vdash t{::}\sigma_s[m@t_1, \ldots, t_n \twoheadrightarrow t']$) is $\Gamma \vdash t[m@t_1, \ldots, t_n \rightarrow t']$ (respectively, $\Gamma \vdash t[m@t_1, \ldots, t_n \twoheadrightarrow t']$). The *unroled equivalent* of a roled predicate atom is itself. For example, the unroled equivalent of $failure{::}sf[problem \rightarrow$ "system failure"$]$ is $failure[problem \rightarrow$ "system failure"$]$.

Definition A roled program P with respect to a set of declarations D is *inheritance-stratified* (or *i-stratified*) if there exists a mapping μ from the set of unroled equivalents of ground roled atoms to the set of non-negative integers such that, for every ground instance $C\theta$ of every clause C in P,

- $\mu(B^*) \leq \mu(A^*)$, where B is an atom in the body of the clause instance $C\theta$, A is the head of the clause instance $C\theta$, B^* is the unroled equivalent of B, and A^* is the unroled equivalent of A,
- $\mu(B^*) < \mu(A^*)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where B' is a roled atom in the body of $C'\theta'$, A is the head of the clause instance $C\theta$, $A \neq B'$, B^* is the unroled equivalent of B', and A^* is the unroled equivalent of A,
- $\mu(A^{**}) \leq \mu(A^*)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where A' is the head of $C'\theta'$, A is the head of $C\theta$, A^* is the unroled equivalent of A, and A^{**} is the unroled equivalent of A'.

This definition differs from the definition of i-stratified programs given in Section 3.4 where a ground instance possibly overrides another ground instance. Informally, programs in which a value of a method depends on the inherited value of the same method have no natural meaning. However, programs in which a value of a method depends on the value of the same method applied to the same object where the type is explicitly stated does have a natural meaning. Throughout the remainder of this section, we will indicate whether we are referring to the definition of i-stratified programs in this section or that given in Section 3.2 if it is not obvious from the context.

Example 6.2.7 Consider the program P and the set of declarations D in Example 3.4.2. The new definition for programs with roled atoms requires:

$$\begin{aligned}
&\mu(failure[problem \rightarrow \text{“system failure”}]) \leq \\
&\quad \mu(failure[problem \rightarrow \text{“cooling system failure”}]) \\
&\mu(failure[problem \rightarrow \text{“cooling system failure”}]) \leq \\
&\quad \mu(failure[problem \rightarrow \text{“system failure”}]) \\
&\mu(failure[problem \rightarrow \text{“system failure”}]) < \\
&\quad \mu(failure[problem \rightarrow \text{“system failure”}])
\end{aligned}$$

which is unsatisfiable. As expected, using the new definition for programs with roled atoms, P with respect to D is not i-stratified.

Now, consider the program P' with respect to D:

$$\begin{aligned}
&\{x{:}sf\} \vdash x[problem \rightarrow \text{“system failure”}] \\
&\{x{:}csf\} \vdash x[problem \rightarrow \text{“cooling system failure”}] \leftarrow \\
&\quad x{::}sf[problem \rightarrow \text{“system failure”}].
\end{aligned}$$

Then $[P']_D$ is:

$$\{failure{:}sf\} \vdash failure[problem \rightarrow \text{“system failure”}] \tag{6.1}$$

$$\begin{aligned}
&\{failure{:}csf\} \vdash failure[problem \rightarrow \text{“cooling system failure”}] \leftarrow \\
&\quad failure{::}sf[problem \rightarrow \text{“system failure”}]
\end{aligned} \tag{6.2}$$

The i-stratification for programs with roled atoms of $[P']_D$ requires:

$$\begin{aligned}
&\mu(failure[problem \rightarrow \text{“system failure”}]) \leq \\
&\quad \mu(failure[problem \rightarrow \text{“cooling system failure”}]) \\
&\mu(failure[problem \rightarrow \text{“cooling system failure”}]) \leq \\
&\quad \mu(failure[problem \rightarrow \text{“system failure”}]),
\end{aligned}$$

which is satisfiable. Hence P' with respect to D is i-stratified using the new definition for programs with roled atoms. $\square$

We now describe what an unambiguous program is. This definition is similar to that in Section 3.4 except there is now a tuple for every method atom in the body of a clause defining each method.

Definition Let P be a program with respect to a set of declarations D. For each type σ in D, let $R(\sigma)$ be $\{< m_{\rightarrow}, n, \tau, m_{\leadsto}, n', \tau' >|$ for each method atom $m_{\leadsto}$ of arity n' of type τ' in the body of every clause that defines functional method $m_{\rightarrow}$ of arity n on type τ, $\sigma \leq_t \tau$, and $m_{\leadsto} = \_$, $n' = \_$, and $\tau' = \_$ if there are no method atoms in the body of the definition $\} \cup \{< m_{\twoheadrightarrow}, n, \tau, m_{\leadsto}, n', \tau' >|$ for each method atom $m_{\leadsto}$ of arity n' of type τ' in the body of every clause that defines multi-valued method $m_{\twoheadrightarrow}$ of arity n on type τ, $\sigma \leq_t \tau$, and $m_{\leadsto} = \_$, $n' = \_$, and $\tau' = \_$ if there are no method atoms in the body of the definition$\}$. Program P is *unambiguous* with respect to multiple inheritance if and only if for every type σ in D, if $< m_{\rightarrow}, n, \tau_1, m_{1,\leadsto}, n'_1, \tau'_1 > \in R(\sigma)$ (respectively, $< m_{\twoheadrightarrow}, n, \tau_1, m_{1,\leadsto}, n'_1, \tau'_1 > \in R(\sigma)$) and $< m_{\rightarrow}, n, \tau_2, m_{2,\leadsto}, n'_2, \tau'_2 > \in R(\sigma)$ (respectively, $< m_{\twoheadrightarrow}, n, \tau_2, m_{2,\leadsto}, n'_2, \tau'_2 > \in R(\sigma)$), where $\tau_1 \neq \tau_2$, then either $\tau_1 <_t \tau_2$ or $\tau_2 <_t \tau_1$, or if $\tau_1 \not<_t \tau_2$ and $\tau_2 \not<_t \tau_1$ then there is a tuple $< m_{\rightarrow}, n, \sigma, m_{\rightarrow}, n, \tau >$ (respectively, $< m_{\twoheadrightarrow}, n, \sigma, m_{\twoheadrightarrow}, n, \tau >$) such that $\sigma <_t \tau$ and there is a tuple $< m_{\rightarrow}, n, \tau, m_{\leadsto}, n', \tau' > \in R(\sigma)$ (respectively, $< m_{\twoheadrightarrow}, n, \tau, m_{\leadsto}, n', \tau' > \in R(\sigma)$).

Example 6.2.8 Consider the declarations and program given in Example 3.4.4. Based on the new definition of unambiguous, $R(tf) = \{< problem, 0, cf, \_, \_, \_ >, < problem, 0, fsf, \_, \_, \_ >\}$. In the type lattice $cf \not<_t fsf$ and $fsf \not<_t cf$. As expected this program is ambiguous.

Now consider the following program with respect to the declarations given in Example 3.4.4.

$\{x{:}cf\} \vdash x[problem \rightarrow \text{"call supervisor"}]$
$\{x{:}fsf\} \vdash x[problem \rightarrow \text{"check history"}]$
$\{x{:}tf, y{:}string\} \vdash x[problem \rightarrow y] \leftarrow x{::}cf[problem \rightarrow y]$

Based on the above definition, $R(tf) = \{< problem, 0, cf, \_, \_, \_ >, < problem, 0, fsf, \_, \_, \_ >, < problem, 0, tf, problem, 0, cf >\}$. Now, in the type lattice $cf \not<_t fsf$ and $fsf \not<_t cf$ but there is the tuple $< problem, 0, tf, problem, 0, cf >$ such that $tf <_t cf$ and $< problem, 0, cf, \_, \_, \_ > \in R(tf)$. As the condition also holds for the other types in D, this program is unambiguous. □

The definitions of "restricted" and "well-defined" now follow from Section 3.4, using the new definition of unambiguous. Similarly the definition of "simple" uses the definitions of "unambiguous" and "well-defined" given in this section.

Definition A program P with respect to a set of declarations D is *r-simple* if the following two conditions hold:

- P with respect to D is i-stratified, and
- P with respect to D is well-defined.

The interpretation assigns a truth value to every ground instance of a clause in a program. We first describe the assignment of a truth value to an atom.

Definition Let I be an interpretation, A a roled atom, and $\mathcal{T}$ a term assignment (with respect to Γ). Then A is given a *truth value* in I (with respect to $\mathcal{T}$ and Γ) as follows:

- If A is $\Gamma \vdash p(t_1, \ldots, t_n)$, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $p(t'_1, \ldots, t'_n) \in I$.
- If A is $t{::}\sigma[m@t_1, \ldots, t_k \rightarrow t_n]$, and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'{::}\tau[m@t'_1, \ldots, t'_k \rightarrow t'_n] \in I$, where $\sigma \leq \tau$, and there is no atom $t'{::}\sigma'[m@t'_1, \ldots, t'_k \rightarrow t''_n] \in I$ such that $t''_n \neq t'_n$ and $\sigma \leq \sigma' < \tau$.
- If A is $t{::}\sigma[m@t_1, \ldots, t_k {\rightarrow\!\!\!\rightarrow} t_n]$ and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'{::}\tau[m@t'_1, \ldots, t'_k {\rightarrow\!\!\!\rightarrow} t'_n] \in I$, where $\sigma \leq \tau$, and there is no atom $t'{::}\sigma'[m@t'_1, \ldots, t'_k \rightarrow t''_n] \in I$ such that $t''_n \neq t'_n$ and $\sigma \leq \sigma' < \tau$.

That is, a ground method atom is now true in an interpretation if it is in the interpretation or the atom in the interpretation for that object and arguments with the next greatest type has the same value.

Example 6.2.9 Suppose $cf < csf$, $csf < sf$ and $I = \{failure{::}cf[treatment \rightarrow \text{"lower temperature"}], failure{::}sf[treatment \rightarrow \text{"tell supervisor"}]\}$.
The ground atom $failure{::}cf[treatment \rightarrow \text{"tell supervisor"}]$ is not true in I. Whereas the ground instance $failure{::}csf[treatment \rightarrow \text{"tell supervisor"}]$ is true in I. □

A ground instance of a clause is given a truth value in I (with respect to $\mathcal{T}$ and Γ) in the normal way.

We now define the "model" and "preferred model" of a roled program.

Definition Let P be a roled program and I an interpretation. We say that I is a *roled model* of P if, for every ground clause $C \in [P]$, C is true in I.

Because the type of the object is specified for each atom it is not necessary to consider overriding in the definition of "roled model". It is considered in the definition of truth value.

Definition A roled method atom $t{::}\tau[m@t_1, \ldots, t_k \rightarrow t_n]$ (respectively, $t{::}\tau[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$) in an interpretation I is *minimal* if there is no other atom $t{::}\sigma[m@t_1, \ldots, t_k \rightarrow t'_n]$ (respectively, $t{::}\sigma[m@t_1, \ldots, t_k {\twoheadrightarrow} t'_n]$ in I such that $\sigma < \tau$. Every predicate atom in an interpretation is *minimal*.

Example 6.2.10 Assume a set of declarations D including $failure{:}cf$, $cf < csf$ and $csf < sf$. Consider the program P with respect to D:

$$\{x{:}sf\} \vdash x[treatment \rightarrow \text{"tell supervisor"}]$$
$$\{x{:}cf\} \vdash x[treatment \rightarrow \text{"lower temperature"}]$$

The two interpretations
$I_1 = \{failure{::}cf[treatment \rightarrow \text{"lower temperature"}],$
$failure{::}sf[treatment \rightarrow \text{"tell supervisor"}]\}$ and
$I_2 = \{failure{::}csf[treatment \rightarrow \text{"lower temperature"}],$
$failure{::}sf[treatment \rightarrow \text{"tell supervisor"}]\}$ are models of P.
In model I_1, $failure{::}cf[treatment \rightarrow \text{"lower temperature"}]$ is a minimal atom. In model I_2, $failure{::}csf[treatment \rightarrow \text{"lower temperature"}]$ is a minimal atom. These atoms are the only minimal atoms in I_1 and I_2 (respectively). $\Box$

The goal of the following preference relation is to give priority to minimal atoms.

Definition Suppose that M and N are two distinct models of a roled program P with respect to a set of declarations D. Then N is *r-preferable* to M $(N \ll M)$ if, for every atom A in $N - M$, there is an atom B in M such that $A = t{::}\sigma[m@t_1, \ldots, t_k \rightarrow t_n]$ (respectively, $A = t{::}\sigma[m@t_1, \ldots, t_k {\twoheadrightarrow} t_n]$), $B = t{::}\tau[m@t_1, \ldots, t_k \rightarrow t_n]$ (respectively, $B = t{::}\tau[m@t_1, \ldots, t_k {\twoheadrightarrow} t_n]$), and $\sigma < \tau$. We write $N \leq M$ if $N \ll M$ or $N = M$. We say model N is an *r-preferred* model of P if there are no models of P r-preferable to N.

Example 6.2.11 Consider program P with respect to the declarations D of Example 6.2.10 that has two models. Because $failure{::}cf[treatment \rightarrow \text{"lower temperature"}] \in I_1 - I_2$ and $failure{::}csf[treatment \rightarrow \text{"lower temperature"}] \in I_2$, and this condition does not hold for any atoms in $I_2 - I_1$, model I_1 is r-preferable to I_2. $\Box$

Lemma 6.2.1 *Let D be a set of declarations, and P with respect to D be an r-simple program. Then P with respect to D has exactly one r-preferred model, which we denote M_P. For every model M, we have $M_P \ll M$.*

Proof Let $\{M_i\}$ be the set of models of P with respect to D. The preferred model must be a set with the smallest cardinality. Then the sets with smallest cardinality, $\{S_j\} \subseteq \{M_i\}$. Let $I = \cap\{M_i\}$ which is unique, and $R = \{a \mid a \in C - \cap\{M_i\}, C \in \{S_i\}, a$ is minimal in $\{M_i\}\}$ which is also unique. The preferred model $M_P = I \cup R$ thus P with respect to D has exactly one preferred model.

Consider a model $M \in \{M_i\}$ such that $M \leq M_P$. Then there is an atom $A \in M - M_P$ and there is an atom $B \in M_P$ such that $A = t{::}\sigma[m@t_1, \ldots, t_k \rightarrow t_n]$ (respectively, $A = t{::}\sigma[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$), $B = t{::}\tau[m@t_1, \ldots, t_k \rightarrow t_n]$ (respectively, $B = t{::}\tau[m@t_1, \ldots, t_k \twoheadrightarrow t_n]$), and $\sigma < \tau$. This is a contradiction, so for every model M, we have $M_P \ll M$. □

We now show that roled programs are a generalization of Gulog programs (without negation). Using the shorthands described previously in this chapter, any simple positive Gulog program can be written as a roled program, and any positive Gulog atom can be written as a roled atom.

Theorem 6.2.1 *Let D be a set of declarations, and P with respect to D be a simple positive Gulog program. Let M_P be the (unique) preferred model of P with respect to D, M'_P the (unique) r-preferred model of P with respect to D. Then*

- *every ground atom A in M_P is true in M'_P, and*
- *every minimal atom $A' \in M'_P$ has an unroled equivalent $A \in M_P$.*

Proof Suppose $A \leftarrow B$ is a ground instance of a clause in $[P]_D$. As A is true in M_P, B is true in M_P, and $A \leftarrow B$ is not overridden by another clause in $[P]_D$. As M'_P is the r-preferred model of P, and $A \leftarrow B$ is not overridden by another clause in $[P]_D$ then $A \leftarrow B$ is true in M'_P. If A were not true in M'_P, then B would not be true in M'_P. If B is true in M_P but not true in M'_P, then there is a contradiction. Hence, A is true in M'_P.

Now we want to show that every minimal atom $A' \in M'_P$ has an unroled equivalent A such that $A \in M_P$. Suppose there is a ground clause $A \leftarrow B$ in $[P]_D$ where A is $\{t{:}\sigma \ldots\} \vdash t[m@t_1, \ldots t_k \rightarrow t_n]$ (respectively, A is $\{t{:}\sigma\} \vdash t[m@t_1, \ldots t_k \twoheadrightarrow t_n]$). Then $A' = t{::}\sigma[m@t_1, \ldots t_k \rightarrow t_n]$ (respectively, $A' = t{::}\sigma[m@t_1, \ldots t_k \twoheadrightarrow t_n]$). As $A' \in M'_P$ and because it is a minimal atom, there is no clause in $[P]_D$ that overrides $A' \leftarrow B$ so $A \in M_P$. Obviously this result holds where A' is a roled predicate atom. □

We now give a translation from roled programs to Datalog, thus providing an evaluation procedure for roled goals with respect to roled programs. This translation is different from that given in Chapter 5. It is necessary to include type information in atoms that represent method atoms.

An object declaration $a{:}\tau$ is translated to an atom $\$type(a, \tau)$. A type hierarchy declaration $\sigma < \tau$ is translated to $\$sub(\sigma, \tau)$. There are clauses included in the translation that represent the transitive closure of the type lattice

$$\begin{aligned}
&\$sub^+(x, y) \leftarrow \$sub(x, y).\\
&\$sub^+(x, y) \leftarrow \$sub(x, z) \wedge \$sub^+(z, y).\\
&\$sub^*(x, x) \leftarrow \$type(y, x).\\
&\$sub^*(x, y) \leftarrow \$sub^+(x, y).
\end{aligned}$$

A variable typing $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ is translated to a conjunction of atoms $\Gamma' = \$type(x_1, y_1) \wedge \$sub^*(y_1, \sigma_1) \wedge \cdots \wedge \$type(x_n, y_n) \wedge \$sub^*(y_n, \sigma_n)$. A predicate atom is left unchanged in the translation. A functional method atom $t{::}\sigma[m@t_1, \ldots, t_n \rightarrow t']$ is translated to $\$m(\sigma, t, t_1, \ldots, t_n, t')$. A multi-valued method atom $t{::}\sigma[m@t_1, \ldots, t_n \twoheadrightarrow t']$ is translated to $\$\$m(\sigma, t, t_1, \ldots, t_n, t')$. A clause $\Gamma \vdash A \leftarrow B$ where A is a roled predicate atom is translated to $A' \leftarrow \Gamma' \wedge B'$ where A' is the translation of A, Γ' is the translation of Γ and B' is the translation of B.

For any set of clauses that define a functional method m of arity k

$$\{t_p{:}\tau_p\} \cup \Gamma_p \ \vdash \ t_p[m@t_{1_p}, \ldots, t_{k_p} \to t_{n_p}] \leftarrow B_p$$

$$\vdots$$

$$\{t_1{:}\tau_1\} \cup \Gamma_1 \ \vdash \ t_1[m@t_{1_1}, \ldots, t_{k_1} \to t_{n_1}] \leftarrow B_1$$

$$\{t{:}\sigma\} \cup \Gamma \ \vdash \ t[m@t_{1_\sigma}, \ldots, t_{k_\sigma} \to t_{n_\sigma}] \leftarrow B$$

where $\sigma < \tau_1 < \cdots < \tau_p$, there are clauses in the translation

$$\$m(\sigma, t, t_{1_\sigma}, \ldots, t_{k_\sigma}, t_{n_\sigma}) \leftarrow B' \wedge \$type(t, y) \wedge \$sub^*(y, \sigma) \wedge \Gamma' \tag{6.3}$$

$$app_{\$m}(\sigma, t, t_{1_\sigma}, \ldots, t_{k_\sigma}) \leftarrow B' \wedge \$type(t, y) \wedge \$sub^*(y, \sigma) \wedge \Gamma' \tag{6.4}$$

$$\$m(\tau_1, t_1, t_{1_1}, \ldots, t_{k_1}, t_{n_1}) \leftarrow B'_1 \wedge \$type(t_1, y) \wedge \$sub^*(y, \tau_1) \wedge \Gamma'_1 \tag{6.5}$$

$$app_{\$m}(\tau_1, t_1, t_{1_1}, \ldots, t_{k_1}) \leftarrow B'_1 \wedge \$type(t_1, y) \wedge \$sub^*(y, \tau_1) \wedge \Gamma'_1 \tag{6.6}$$

$$\vdots$$

$$\$m(\tau_p, t_p, t_{1_p}, \ldots, t_{k_p}, t_{n_p}) \leftarrow B'_p \wedge \$type(t_p, y) \wedge \$sub^*(y, \tau_p) \wedge \Gamma'_p \tag{6.7}$$

$$app_{\$m}(\tau_p, t_p, t_{1_p}, \ldots, t_{k_p},) \leftarrow B'_p \wedge \$type(t_p, y) \wedge \$sub^*(y, \tau_p) \wedge \Gamma'_p \tag{6.8}$$

$$\$m(\sigma, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_1, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, y) \wedge \$sub^*(y, \sigma) \wedge \Gamma' \wedge \neg app_{\$m}(\sigma, t, t_1, \ldots, t_k) \tag{6.9}$$

$$\$m(\tau_1, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_2, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, y) \wedge \$sub^*(y, \tau_1) \wedge \Gamma'_1 \wedge \neg app_{\$m}(\sigma, t, t_1, \ldots, t_k) \wedge \neg app_{\$m}(\tau_1, t, t_1, \ldots, t_k) \tag{6.10}$$

$$\vdots$$

$$\$m(\tau_{p-1} t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_p, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, y) \wedge \$sub^*(y, \tau_{p-1}) \wedge \Gamma'_{p-1} \wedge \neg app_{\$m}(\sigma, t, t_1, \ldots, t_k) \wedge \neg app_{\$m}(\tau_1, t, t_1, \ldots, t_k) \wedge \cdots \wedge \neg app_{\$m}(\tau_{p-1}, t, t_1, \ldots, t_k) \tag{6.11}$$

$$\$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\sigma, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \$sub^+(z, \sigma) \tag{6.12}$$

$$\$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_1, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \$sub^+(\sigma, z) \wedge \$sub^+(z, \tau_1) \tag{6.13}$$

$$\vdots$$

$$\$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau_p, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \$sub^+(\tau_{p-1}, z) \wedge \$sub^+(z, \tau_p), \tag{6.14}$$

where B' is the translation of B, B'_1 is the translation of B_1, B'_p is the translation of B_p, Γ' is the translation of Γ, Γ'_1 is the translation of Γ_1, Γ'_p is the translation of Γ_p. The translation of clauses with multi-valued methods in the head is similar. Notice that variables x, y and z are bound to "types" rather than "objects" in the translation.

The translation of clauses with methods in the head is interesting. This is where inheritance is considered in the translation. Clauses 6.3 to 6.8 give the value of the method when applied to an object of the specified type, including subtypes. Clauses 6.9 to 6.11 deal with overriding and clauses 6.12 to 6.14 incorporate inheritance. The value of a method for an object of a type which is smaller than any which have the method defined on them, must inherit from the smallest type on which the method is defined. The value of a method for an object of a type between two types on which the method has been defined inherits from the greater of the two types.

We illustrate the translation in the following example.

Example 6.2.12 Assume a set of declarations D including $failure{:}cf$, $csf < sf$, and $cf < csf$. Consider the program P with respect to the declarations D:

$\{x{:}sf\} \vdash x[problem \rightarrow \text{“system failure”}]$
$\{x{:}cf\} \vdash x[problem \rightarrow \text{“over efficient condenser”}] \leftarrow x[condenser \rightarrow low]$
$failure[condenser \rightarrow low]$.

This is translated to the Datalog program P':

$\$type(failure, cf)$.
$\$sub(cf, csf)$.
$\$sub(csf, sf)$.
$\$sub^+(x, y) \leftarrow \$sub(x, y)$.
$\$sub^+(x, y) \leftarrow \$sub(x, z) \wedge \$sub^+(z, y)$.
$\$sub^*(x, x) \leftarrow \$type(y, x)$.
$\$sub^*(x, y) \leftarrow \$sub^+(x, y)$.
$\$problem(sf, x, \text{“system failure”}) \leftarrow \$type(x, y) \wedge \$sub^*(y, sf)$.
$app_{\$problem}(sf, x) \leftarrow \$type(x, y) \wedge \$sub^*(y, sf)$.
$\$problem(cf, x, \text{“over efficient condenser”}) \leftarrow \$type(x, y) \wedge \$sub^*(y, cf) \wedge$
$\quad \$condenser(cf, x, low)$.
$app_{\$problem}(cf, x) \leftarrow \$type(x, y) \wedge \$sub^*(y, cf) \wedge \$condenser(cf, x, low)$.
$\$problem(cf, x, y) \leftarrow \$problem(sf, x, y) \wedge \$type(x, y) \wedge \$sub^*(y, cf) \wedge$
$\quad \neg app_{\$problem}(cf, x)$
$\$condenser(cf, failure, low) \leftarrow \$type(failure, y) \wedge \$sub^*(y, cf)$.
$app_{\$condenser}(cf, failure) \leftarrow \$type(failure, y) \wedge \$sub^*(y, cf)$.

$\$problem(z, x, y) \leftarrow \$problem(sf, x, y) \wedge \$type(x, z) \wedge \$sub^+(cf, z) \wedge \$sub^+(z, sf)$.
$\$problem(z, x, y) \leftarrow \$problem(cf, x, y) \wedge \$type(x, z) \wedge \$sub^+(z, cf)$.
$\$condenser(z, failure, y) \leftarrow \$condenser(cf, failure, y) \wedge \$type(failure, z) \wedge$
$\quad \$sub^+(z, cf)$. $\square$

We now describe a translation from an interpretation of a roled program to an interpretation of a Datalog program. Given an interpretation I of a roled program P (with respect to a set of declarations D) the translation I' of I (with respect to P) is constructed as follows. Each predicate or method atom in I is translated to a predicate or method atom as given above. For any set of atoms which are translations of method atoms for functional method m of arity k $\{\$m(\tau_1, t, t_1, \ldots, t_k, t_{\tau_1}), \$m(\tau_2, t, t_1, \ldots, t_k, t_{\tau_2}), \ldots, \$m(\tau_p, t, t_1, \ldots, t_k, t_{\tau_p})\}$ in I' such that there are no atoms $\$m(\sigma_i, t, t_1, \ldots, t_k, t_{\sigma_i})$ in I', where $1 \leq i \leq p+1$ and $\sigma_1 < \tau_1 < \sigma_2 < \tau_2 < \cdots < \sigma_p < \tau_p < \sigma_{p+1}$, there is an atom $\$m(\sigma_i, t, t_1, \ldots, t_k, t_{\tau_i})$ included in I' for every $t{:}\sigma_i$ in D, $1 \leq i \leq p$. The same kind of translation applies for sets of atoms which are translations of method atoms for multi-valued method m of arity k. For every declaration $o{:}\tau$ in D there is an atom $\$type(o, \tau)$ included in I'. For every declaration $\tau < \sigma$ in D there is an atom $\$sub(\tau, \sigma)$ in I'. The transitive closure of $\$sub$ is described by $\$sub^*$ and $\$sub^+$ in I'. The applied clauses are described by *app*-predicates in I'.

Example 6.2.13 Consider program P and the set of declarations D given in Example 6.2.12. An interpretation of P with respect to D is $I = \{failure{::}sf[problem \rightarrow \text{“system failure”}],$ $failure{::}cf[problem \rightarrow \text{“over efficient condenser”}], failure{::}cf[condenser \rightarrow low]\}$.

The translation of these atoms gives $I_t = \{\$problem(sf, failure, \text{“system failure”}),$ $\$problem(cf, failure, \text{“over efficient condenser”}), \$condenser(cf, failure, low)\}$. Considering the types between cf and sf gives $I_b = \{\$problem(csf, failure, \text{“system failure”})\}$. Considering the declarations gives $I_D = \{\$type(failure, cf), \$sub(csf, sf), \$sub(cf, csf), \$sub^*(csf, sf),$

$\$sub^*(cf, csf), \$sub^*(cf, sf)\}$. Finally, considering the clauses which are applied gives $I_A = \{app_{\$problem}(cf, failure), app_{\$problem}(sf, failure), app_{\$condenser}\ (cf, failure)\}$.

Then I', the translation of I with respect to D, is $I_t \cup I_b \cup I_D \cup I_A$. □

We now prove the correctness of this translation.

Lemma 6.2.2 *Let D be a set of declarations, P with respect to D be a positive i-stratified roled program and P' be the Datalog translation of P with respect to D. Then P' is locally stratified.*

Proof The program P with respect to D contains no negation. Negation is added in the translation only through the *app*-predicates. So the resulting program, P', is locally stratified. □

The following lemma is used in the proof of proposition 6.2.1.

Lemma 6.2.3 *Let D be a set of declarations, P with respect to D be a positive i-stratified roled program and M the r-preferred model of P with respect to D. Let P' be the translation of P with respect to D and M' the translation of M (with respect to P and D). Then M' is the least model of P'.*

Proof We are required to show that every ground instance of a clause in P' is true in M' and that M' is the least model of P'.

Each type clause in P' is a translation with respect to D and each type atom in M' is a translation with respect to D. Let $C'\theta$ be a ground instance of a type clause in P'. Then $M' \models C'\theta$.

Suppose that $C = \Gamma \vdash A \leftarrow B$ is a clause in P with respect to D that contains only predicate atoms. Then there is a clause $C' = A \leftarrow \Gamma' \wedge B$ in P' where Γ' is the translation of Γ. Because M is a model of P with respect to D, $C\theta$ is true in M for all ground instances $C\theta$ of C. As every predicate atom in M is included in M', $M' \models C'\theta$ for every θ for which $C\theta$ is true in M. We next consider substitutions θ' that do not satisfy Γ, $M' \models C'\theta'$ because $\Gamma'\theta'$ is false in M'.

Suppose $C = \Gamma \vdash t{::}\tau[m@t_1, \ldots, t_p \rightarrow t_n] \leftarrow B$ is a clause in P with respect to D. There is a clause $C'' = \$m(\tau, t, t_1, \ldots, t_p, t_n) \leftarrow type(t, y) \wedge \$sub^*(y, \tau) \wedge \Gamma' \wedge B'$ where Γ' is the translation of Γ and B' is a translation of B with respect to D. Every atom $t{::}\tau[m@t_1, \ldots, t_p \rightarrow t_n]\theta$ in M is translated to an atom $m(\tau, t, t_1, \ldots, t_p, t_n)\theta$ in M', so for every $C\theta$ which is true in M, $M' \models C'\theta$ and $M' \models C''\theta$. We now consider substitutions to atoms in P' that would cause type inconsistencies in P. For substitutions θ' which do not satisfy Γ, $M' \models C'\theta'$ and $M' \models C''\theta'$ because $\Gamma'\theta'$ is false in M'. If C is a clause in a set of clauses in P:

$$\begin{array}{l}
\{t_p{:}\tau_p\} \cup \Gamma_p \vdash t_p[m@t_{1_p}, \ldots, t_{k_p} \rightarrow t_{n_p}] \leftarrow B_p \\
\vdots \\
\{t_1{:}\tau_1\} \cup \Gamma_1 \vdash t_1[m@t_{1_1}, \ldots, t_{k_1} \rightarrow t_{n_1}] \leftarrow B_1 \\
\{t{:}\tau\} \cup \Gamma \vdash t[m@t_{1_\tau}, \ldots, t_{k_\tau} \rightarrow t_{n_\tau}] \leftarrow B,
\end{array}$$

where $\tau < \tau_1 < \cdots < \tau_p$. Then there is a clause $C' = \$m(z, t, t_1, \ldots, t_k, x_n) \leftarrow \$m(\tau, t, t_1, \ldots, t_k, x_n) \wedge \$type(t, z) \wedge \$sub^*(z, \tau)$ in P'. In the translation from M to M' there is an atom included for every $o{:}\sigma$ in D, where $\sigma < \tau$. Thus for any θ which satisfies Γ, $M' \models C'\theta$. For substitutions θ' which do not satisfy Γ, $M \models C'\theta'$ and $M \models C''\theta'$ because $\Gamma'\theta'$ is false in M'.

This also follows where C is a clause in $\{t_1{:}\tau_1\} \cup \Gamma_1 \vdash t_1[m@t_{1_1}, \ldots, t_{k_1} \rightarrow t_{n_1}] \leftarrow B_1, \ \ldots, \ \{t_p{:}\tau_p\} \cup \Gamma_p \vdash t_p[m@t_{1_p}, \ldots, t_{k_p} \rightarrow t_{n_p}] \leftarrow B_p$.

This also applies where the head of C is a multi-valued method atom.

As M is the r-preferred model, there are no unsupported atoms and the most specific atoms are part of the model. Therefore there can be no unsupported facts in the translation and the values of methods apply to the correct types so M' is a least model of P'. $\Box$

The following proposition shows that standard query evaluation can be used to compute answers for roled queries on roled programs after they have been rewritten and the expected answer is computed.

Proposition 6.2.1 *Let D be a set of declarations, P with respect to D be a positive i-stratified roled program and Q an atomic query $\Gamma \vdash \leftarrow A$. Let P' be the translation of P with respect to D and Q' be the translation of Q with respect to D. If θ is a computed answer for $P' \cup \{Q'\}$, then θ is a correct answer for $P \cup \{Q\}$.*

Proof The query Q with respect to D is translated to the query $Q' = \leftarrow \Gamma' \wedge A'$, where Γ' is the translation of Γ and A' is the translation of A with respect to D. Let M be the preferred model of P with respect to D. Then, the translation M' of M with respect to D is the least model of P' (by Lemma 6.2.3). If θ is a computed answer for $P' \cup \{Q'\}$ then $M' \models (\Gamma' \wedge A')\theta$ and so $(\Gamma \vdash A)\theta$ is true in M. $\Box$

6.3 Aggregate and arithmetic operators

Aggregate operators [33] and arithmetic operators [104] have been added to logic programming and deductive databases in order to make the language more expressive. Harel discusses recursive languages, which are a generalization of languages with arithmetic [51]. In this section we describe how arithmetic and aggregate operators can be included in Gulog. Again we consider only positive programs.

Aggregate atoms and arithmetic operators are allowed in the body of a clause. We illustrate the use of an aggregate atom in the following example.

Example 6.3.1 Clause 6.15 counts the number of failures that are recorded for each reactor and stores the reactor identifier and count in a relation $failure\_count$.

$$\begin{aligned}\{x{:}reactor, y{:}date, z{:}sf, c{:}integer\} \vdash\ & failure\_count(x, c) \leftarrow \\ & group\_by(x[failure@y \twoheadrightarrow z], [x], c = count)\end{aligned} \tag{6.15}$$

The $group\_by$ atom groups the $failure$ method on the object x to which it is applied and counts the number of $failure$s for each x. The aggregate operator $count$ does not take an argument as it is the count of the number of $failure$s for each x. $\Box$

Arithmetic operators are modeled as predicate atoms. There are built-in predicates for equality, arithmetic comparisons, and arithmetic operators that operate on integers. Aggregate operators are sum, $count$, max, min, and $average$. An aggregate atom is defined as follows: $\Gamma \vdash group\_by(A, [x_1, \ldots, x_n], y = f(x_1, \ldots, x_p, z_1, \ldots, z_q))$ is an aggregate atom where A is an atom involving variables $(x_1, \ldots, x_n, z_1, \ldots, z_m)$, $x_1, \ldots, x_n$ are the group_by variables, f is an aggregate operator, $(x_1, \ldots, x_p, z_1, \ldots, z_q)$ is a list of terms involving only some subset of the variables $(x_1, \ldots, x_n, z_1, \ldots, z_m)$. The list may be empty. The variables $z_1, \ldots, z_m$ may not appear outside the aggregate atom. A ground instance of an aggregate atom consists of replacing the variables $x_1, \ldots, x_n, y$ by ground terms $x_{1_0}, \ldots, x_{n_0}, y_0$. The variables $z_1, \ldots, z_m$ are not instantiated in such a ground instance.

Definition Given a variable typing Γ, atom $\Gamma \vdash A$, and atoms, aggregate atoms or arithmetic atoms $\Gamma \vdash B_1, \ldots, \Gamma \vdash B_n$, $\Gamma \vdash A \leftarrow B_1 \wedge \cdots \wedge B_n$ is an *aggregate clause.* Atom $\Gamma \vdash A$ is the *head atom* of the clause. Atoms $\Gamma \vdash B_1, \ldots, \Gamma \vdash B_n$ are the *body atoms.*

Definition An *aggregate program* with respect to a set of declarations D is a finite set of aggregate clauses with respect to the set of declarations, D.

Example 6.3.2 Assume a set of declarations including $r1$:*reactor*.
The atom $A = \{y{:}integer, x{:}part\} \vdash group\_by(r1[parts \twoheadrightarrow x], [], y = count)$ gives the cardinality of the set of *parts* for reactor $r1$. The atom $group\_by(r1[parts \twoheadrightarrow x], [], 100 = count)$ is a ground instance of A. Note that variable x is not instantiated in this ground instance of A. $\square$

Definition An aggregate program P with respect to a set of declarations D is *inheritance-stratified* (or *i-stratified*) if there exists a mapping μ from the set of ground atoms to the set of non-negative integers such that, for every ground instance $C\theta$ of every clause C in P with respect to D,

- $\mu(A') \leq \mu(A)$, where A' is an atom or arithmetic atom in the body of the clause instance $C\theta$, and A is the head of the clause instance $C\theta$,
- $\mu(B') < \mu(A)$ for every ground instance B' of A' where A' is the atom in the ground aggregation atom in $(\Gamma \vdash group\_by(A',$ $[x_1, \ldots, x_n], y_0 = f(x_1, \ldots, x_p, z_1, \ldots, z_p)))\theta$ in the body of the ground clause $C\theta$ and A is the head of the ground clause $C\theta$,
- $\mu(B') < \mu(A)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where B' is an atom or arithmetic atom in the body of $C'\theta'$, or for every ground instance B' of A' where A' is the atom in the ground aggregation atom in $(\Gamma \vdash group\_by(A',$ $[x_1, \ldots, x_n], y_0 = f(x_1, \ldots, x_p, z_1, \ldots, z_p)))\theta$ in the body of $C'\theta'$, and A is the head of $C\theta$,
- $\mu(A') \leq \mu(A)$ for every ground instance $C'\theta'$ that possibly overrides $C\theta$, where A' is the head of $C'\theta'$, and A is the head of $C\theta$.

The class of simple aggregate programs is a generalization of the class of simple Gulog programs. It is necessary to extend the syntactic condition which ensures that functional methods are single-valued to include arithmetic predicates. Such an extension involves the definition of "finiteness dependency". The following definition and example are from [100].

Definition A *finiteness dependency* on a predicate R is a pair $S \rightarrow j$, where j and each element of S are argument positions of R. A database B *satisfies* a finiteness dependency $S \rightarrow j$ for a predicate R if, for every tuple t in R, the set of tuples $\{s[j] \mid s \in R$ and $s[S] = t[S]\}$ is finite (where $t[S]$ is the projection of t onto the positions in S.)

Example 6.3.3 The arithmetic relations $=$ and $+$ have the following finiteness dependencies.

$$= \ \{1\} \rightarrow 2, \{2\} \rightarrow 1$$
$$+ \ \{1,2\} \rightarrow 3, \{2,3\} \rightarrow 1, \{3,1\} \rightarrow 2$$

Comparison relations do not have any finiteness dependencies. Each (finite) k-ary base relation has k dependencies $\emptyset \rightarrow j$, for $1 \leq j \leq k$. $\square$

Definition A variable z is *restricted* with respect to the terms $\{t, t_1, \ldots, t_n\}$ in the conjunction C if

- C contains a method atom $t'[m'@t'_1, \ldots, t'_k \to z]$ and each variable in t', $t'_1, \ldots, t'_k$ either occurs in t, $t_1, \ldots, t_n$ or is itself restricted with respect to $\{t, t_1, \ldots, t_n\}$ in the remainder of C. For example, the variable z is restricted with respect to $\{a, x\}$ in the conjunction $a[m'@x \to y] \wedge a[m''@y \to z]$.
- C contains an arithmetic predicate $p(t'_1, \ldots, t'_n)$, some t'_j is z, p has a finiteness dependency $S \to j$, and every variable in $\{t'_i \mid i \in S\}$ is restricted with respect to $\{t, t_1, \ldots, t_n, z\}$ in the remainder of C.

Definition An aggregate program P with respect to a set of declarations D is *well-defined* if the following conditions hold:

- For each clause $\Gamma \vdash t[m@t_1, \ldots, t_n \to t'] \leftarrow C$ in P with respect to D, t' is either an object symbol or is a variable that is restricted with respect to $\{t, t_1, \ldots, t_n\}$ in C.
- For each functional method m with signature $\tau \times \tau_1 \times \cdots \times \tau_n \to \tau'$, and for each type $\sigma \leq \tau$, program P with respect to D does *not* contain two clauses

  $$\Gamma \vdash t[m@t_1, \ldots, t_n \to s] \leftarrow C$$
  $$\Gamma \vdash t'[m@t'_1, \ldots, t'_n \to s'] \leftarrow C'$$

  such that $\Gamma \vdash t{:}\sigma$, $\Gamma \vdash t'{:}\sigma$, and atoms $t[m@t_1, \ldots, t_n \to x]$ and $t'[m@t'_1, \ldots, t'_n \to x']$ (where x and x' are new variables) are unifiable. This together with the first condition ensures that functional methods really are single-valued.
- P with respect to D is unambiguous with respect to multiple inheritance.
- For each clause $\Gamma \vdash A \leftarrow \cdots \wedge B \wedge \cdots$ where B is an arithmetic atom, every variable t_i in B is restricted with respect to $\emptyset$ in the remainder of B.

Example 6.3.4 Assume appropriate declarations. Consider the aggregate program:

$$\{x{:}integer, y{:}integer\} \vdash p(x, y) \leftarrow x + y = 5$$

Because x and y are not restricted with respect to $\emptyset$ in the remainder of this clause, this program is not well-defined. □

Definition Let I be an interpretation, L be an atom, and $\mathcal{T}$ a term assignment (with respect to Γ). Then an atom, arithmetic atom or aggregate atom, A, is given a *truth value* in I (with respect to $\mathcal{T}$ and Γ) as follows:

- If A is $\Gamma \vdash p(t_1, \ldots, t_n)$, and $t'_1, \ldots, t'_n$ are the term assignments of $t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $p(t'_1, \ldots, t'_n) \in I$.
- If A is $\Gamma \vdash t[m@t_1, \ldots, t_k \to t_n]$, and $t', t'_1, \ldots, t'_k, t'_n$ are the term assignments of $t, t_1, \ldots, t_k, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'[m@t'_1, \ldots, t'_k \to t'_n] \in I$.
- If A is $\Gamma \vdash t[m@t_1, \ldots, t_k \twoheadrightarrow s]$, and $t', t'_1, \ldots, t'_k, s'$ are the term assignments of $t, t_1, \ldots, t_k, s$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $t'[m@t'_1, \ldots, t'_k \twoheadrightarrow s'] \in I$.
- If A is $\Gamma \vdash group\_by(A'(t, t_1, \ldots, t_n), [t], y = f(t, t_1, \ldots, t_n))$ and $t', t'_1, \ldots, t'_n$ are the term assignments of $t, t_1, \ldots, t_n$ (with respect to $\mathcal{T}$ and Γ), then A is true in I (with respect to $\mathcal{T}$ and Γ) if $A'(t', t'_1, \ldots, t'_n) \in I$ and $f(S)$ is defined for the set $S = \{(t', t'_1, \ldots, t'_n) \mid A(t', t'_1, \ldots, t'_n) \in I\}$ and $y = f(S)$.

Example 6.3.5 Suppose $I = \{r1[parts \twoheadrightarrow valve], r1[parts \twoheadrightarrow heat\_exchanger]\}$. Then the following ground instance is true in I:

$$group\_by(r1[parts \twoheadrightarrow x], [], 2 = count).$$

□

Definition Let P be a program with respect to a set of declarations D. We define an *aggregate priority relation* $<_{ap}$ based on P between ground atoms with respect to D. We write $a \leq b$ to denote $a <_{ap} b$ or $a = b$. We define the relation $<_{ap}$ by the following rules, for every ground instance $C\theta$ of a clause C in P with respect to D:

- $A \leq_{ap} A'$, where A is the head of $C\theta$, and A' is an atom or arithmetic atom in the body of $C\theta$,
- $A <_{ap} B'$ for every ground instance $C'\theta'$ of C' in P with respect to D, that possibly overrides $C\theta$, where A is the head of $C\theta$ and B' is an atom or arithmetic atom in the body of $C'\theta'$,
- $A \leq_{ap} A'$ for every ground instance $C'\theta'$ of C' in P with respect to D, that possibly overrides $C\theta$, where A is the head of $C\theta$ and A' is the head of $C'\theta'$, and
- $A <_{ap} A'$ where A is the head of C and A' is the ground instance of an atom in the ground aggregation atom in the body of C.

It is obvious that this language is a generalization of Gulog. We now describe an evaluation procedure for a query in this language by translating the program and query to Datalog.

An aggregate atom $\Gamma \vdash group\_by(A, [x_1, \ldots, x_n], y = f(x_1, \ldots, x_p, z_1, \ldots, z_q))$ is translated to $\Gamma' \wedge group\_by(A', [x_1, \ldots, x_n], y = f(x_1, \ldots, x_p, z_1, \ldots, z_q))$ in the translation to Datalog, where Γ' is the translation of Γ and A' is the translation of A. The rest of the translation follows that described in Chapter 5.

This conjecture and the following discussion describes a possible evaluation procedure.

Conjecture 6.3.1 *Let P with respect to a set of declarations D be a simple aggregate program and Q a query. There is a query evaluation procedure for $P \cup \{Q\}$ that is sound with respect to the semantics described in this section.*

The translation above and the definitions in this section treat aggregate atoms in the same way as negative literals are treated in the previous sections. In Lemma 6.1.1, we proved that if a program P with respect to D is i-stratified then P with respect to D is m-stratified. In Lemma 6.1.3, we proved that if P with respect to D is m-stratified then its translation P' is modularly stratified, and that the well-founded model of P' is equivalent to the associated model of P with respect to D. Ross proves, in [93], that any modularly stratified program can be rewritten as a left-to-right modularly stratified program. Lastly, Ramakrishnan et al. provide a procedure that evaluates programs with left-to-right modularly stratified negation and aggregation in [92] that is sound with respect to well-founded semantics. We believe that these steps provide an evaluation procedure for aggregate programs and queries which are sound and complete with respect to the semantics described in this section. Proving this belief is a subject for further research.

It is important to realize not only what can be expressed but also what cannot be expressed in a language. We now discuss what can and cannot be expressed in Gulog with arithmetic and aggregates. The addition of arithmetic and aggregates to Gulog allows queries that involve counting to be expressed for a first time. With counting it is possible to express a queries such as the one in the following example that determines if the size of a relation is even or odd.

Example 6.3.6 Let $p(0)$ represent true and $p(1)$ represent false. Consider a query that returns true if the size of a relation q is even and false otherwise. This is expressed in the following clause:

$$\{x{:}integer, y{:}integer, z{:}integer\} \vdash p(x) \leftarrow group\_by(q(z), [], y = count) \wedge x = y \ mod \ 2.$$

□

Some queries cannot be expressed even with arithmetic and aggregates. Consider a query whose input is a relation, r, which has n tuples, and whose output is the cross product of r with itself, n times. As the width of the output relation depends on the data, it is not possible to express such queries in Gulog with arithmetic and aggregates. We conjecture in Gulog with arithmetic and aggregates it is possible to express all computable queries whose answers have a fixed or known width.

6.4 Summary

In this chapter we described three extensions to Gulog. The first described a more general class of program. This is similar to generalizing the class of locally stratified programs to modularly stratified programs. We proved that the same relationships exist between i-stratified and m-stratified programs as between locally stratified and modularly stratified programs.

The second described how roled atoms can be included in clauses. Roles allow the modeling of monotonic inheritance as described in [65] and provide user-controlled inheritance [65]. User-controlled inheritance provides another mechanism for dealing with conflicts due to multiple inheritance. This mechanism allows the user to write programs that determine which supertype to inherit from as described in Section 2.2.

In F-logic it is possible to model user-controlled inheritance and monotonic inheritance without introducing roled atom because classes can be variables and method names can be function symbols. Roles are analogous to the use of scope resolution operators in object-oriented languages like Eiffel [82] and C++ [76].

The third extension described how aggregate and arithmetic atoms can be included in the bodies of clauses. The result of adding these atoms to an object-oriented logic language is a more expressive language.

Chapter 7

Conclusion and Future Directions

7.1 Conclusion

This thesis proposed a deductive object-oriented language, called Gulog, that captures objects, types, functional and multi-valued methods, predicates, inheritance and multiple inheritance, overriding, deduction and negation. This language provides a mathematical foundation for deductive object-oriented database systems, leading to a better understanding of how to effectively apply and implement these concepts.

In Gulog, as in the relational data model, the data is separated from the schema. The language is strongly typed, with signatures of the methods, attributes, predicates, the subtype relation and the instance relation declared in the schema. There is a distinction between types and objects, and between the subtype and the instance relationships. Overriding is dynamic. Conflicts due to multiple inheritance are dealt with using the skeptical approach.

Chapter 1 provided the motivation for this work and Chapter 2 introduced concepts and reviewed related work that has been referred to throughout the thesis.

Chapter 3 introduced the context in which the remaining work was done. First there was an informal introduction to the logic language, Gulog. This was followed by a formal introduction. The formal introduction included syntactic restrictions on the language that are used later to define classes of programs with interesting properties. One of the restrictions is i-stratification. I-stratification is similar to local stratification in logic programming, which disallows programs with cycles due to recursion through negation. I-stratification disallows programs with cycles in method definitions which are due to a combination of inheritance, negation and deduction. The class of programs of interest are called simple programs. In this class, functional methods are single-valued, there are no ambiguities caused by conflicts due to multiple inheritance, and the programs are i-stratified. Disallowing programs with ambiguities caused by conflicts due to multiple inheritance is one approach to dealing with multiple inheritance. We discussed two other approaches in Sections 6.2 and 7.3. Using the first approach, we specified a syntactic condition that disallows programs in which such conflicts could occur while still allowing multiple inheritance. Then, the declarative or model-theoretic semantics of the language, which captures the natural or intended meaning of deductive object-oriented databases, is described. The definition of model differs from the definition of model in logic programming. However, it becomes evident that our definition has an interesting and useful theory associated with it. We proved that each simple Gulog program has a unique intuitive model. Finally, we investigated the expressive power of Gulog, proving that it is possible to model negation in a language without negation using inheritance with overriding.

Chapter 4 proposed and examined a bottom-up query evaluation procedure, and top-down query evaluation procedures with and without tabling. The bottom-up procedure is similar to the procedure that computes the standard model of stratified logic programs except it is based on the possibility of overriding, as well as the presence of negation, and the immediate consequence operator is a mapping between typed interpretations rather than interpretations. We initially described a top-down procedure without tabling and showed that it was not complete with respect to the declarative semantics. However, this procedure is important as it forms the basis for the top-down procedure with tabling. The top-down procedure is based on typed unification and a variant of SLDNF-resolution. Due to overriding, it is necessary to compute all the possible answers for an atom, and apply only those which are minimal (due to typing) to the rest of the query. With tabling, atom answers are stored in a table when the minimal answers are computed and can be reused later in the procedure. This is like QSQR/SLS except inheritance and overriding had to be considered in the definitions of recursive atoms, predecessor and successor atoms. Each of the procedures is defined directly on the queries and the programs. We proved that the bottom-up procedure and the top-down procedures are sound, and the bottom-up procedure and the top-down procedure with tabling are complete with respect to the declarative semantics described in Chapter 3.

Chapter 5 described a translation from Gulog to Datalog with negation. We proved that any i-stratified Gulog program (which includes programs with negative literals in the body of the clauses) is translated to a locally stratified Datalog program. This provides an alternative query evaluation procedure to those presented in Chapter 4. It also provides the relationship between a language for deductive object-oriented database systems and a well-understood language for deductive database systems.

Chapter 6 proposed extensions to Gulog that made it more expressive. This is important as it shows that it is possible to give a mathematical description of a wider range of object-oriented and deductive features. The first extension described a syntactic restriction that generalized the restriction in Chapter 3. In this description, we investigated m-stratified programs and the class of m-simple programs. The class of m-simple programs is a larger class than the class of simple programs, and includes the class of simple programs. We proved that any m-stratified program can be translated to a modularly stratified logic program. The translation provides a query evaluation procedure for m-simple programs.

The second extension allows inheritance to be altered in the program or database. This is done by allowing the specification of the type of an atom in the body of a clause. We called atoms that can have their type specified "roled atoms". This extension provides an alternative when dealing with conflicts due to multiple inheritance and a way to model monotonic inheritance. This approach to conflicts due to multiple inheritance allows the user to control which definition is inherited. Interpretations are now composed of roled atoms and inheritance is dealt with in the definition of the truth value of an atom in an interpretation rather than in the definition of model. We provided a translation to Datalog with negation that requires type information in the translations of method atoms. This was not necessary in the translation given in Chapter 5. Then, we proved that the translation is correct. It provides an evaluation procedure for queries and programs in this extended language.

The third extension allows aggregate and arithmetic operators in the bodies of clauses. With these operators it is possible to express all computable queries where answers have a fixed or known width. We provided a query evaluation procedure by translating Gulog programs with arithmetic and aggregate operators to modularly stratified programs with arithmetic and aggregate operators.

We now have a better understanding of query languages for deductive object-oriented databases, and have demonstrated that such systems can be based on a firm mathematical foundation, offering prospects of easier use, simpler development and greater reliability, without affecting its applicability to a range of purposes.

7.2 Comparison with related work

The combination of object-orientation and logic has been investigated for different reasons using different approaches. The approaches can be categorized as follows:

- combining object-orientation with logic to build a programming language,
- studying object-oriented concepts in a mathematical framework, with the aim of formalizing the notion underlying object-oriented systems,
- studying object-oriented concepts in a mathematical framework, with the aim of building object-oriented databases or query languages.

Although there is an overlap between the combination of object-orientation and logic to build a programming language and the combination of object-orientation and logic to build a database and query language, the concerns in these areas are quite different. The main concerns in the first area are software engineering issues, whereas, the main concerns in the third area are database issues. Due to these different concerns, different choices have been made in the languages in the areas.

In the second category, F-logic is the best developed logic with the aim of formalizing the notion underlying object-oriented systems. There are many things that can be expressed in F-logic that cannot be expressed in Gulog. They include:

- querying the schema,
- including object declaration and hierarchy declaration in the head of clauses in the program,
- distinguishing between inheritable and non-inheritable properties, and
- using function symbols as constructors of objects and class ids.

However, it was demonstrated in [74] that F-logic is not easy to implement. So while F-logic provides a general mathematical foundation for deductive object-oriented languages, Gulog provides a tractable foundation by limiting the scope of the language and by introducing strict typing.

The work described in this thesis fits best in the third category. In the design of many of the logics in this category overriding was not considered at all, was considered only in part of the description, or was added later. Overriding has been central in the design of Gulog from the beginning. We believe that using this approach one gains a better understanding of how overriding interacts with other features of the language. We also believe that dealing with conflicts due to multiple inheritance is an important issue and its importance has been played down in much of the related work.

Like us, Abiteboul investigated the issues in stages [1, 4, 6]. At each stage he altered the syntax and semantics of his languages. Although our language was designed in stages, we retained a uniform framework throughout the design.

The semantics of our language is first-order and the language is closely related to logic programming so we can extend the results from that area. We kept the semantics as simple as possible, thus as in logic programming, for a certain class of programs we can prove particular results. We believe that our model is simpler than many of those discussed in Section 2.3 and therefore could be implemented more efficiently.

7.3 Future directions

There are many possible extensions to Gulog. In this section we describe some of them.

In Section 2.2, we outlined another way of dealing with conflicts due to multiple inheritance that we have not yet investigated. This method involves imposing a static ordering on supertypes where there is no obvious ordering. In the schema it should be possible to specify an ordering on classes for a particular method. This ordering is called a hierarchy list. A method definition would then be inherited in the order specified in the hierarchy list. If the hierarchy list is empty, the ordering given in the type hierarchy declarations is assumed. Providing static ordering would involve very few changes. There would be a default hierarchy, and a method specific hierarchy would have priority over the default if specified.

We could also consider complex values in Gulog. Complex values are structured values based on set and tuple constructors [20]. The example in [65] taken from [39] could be written in Gulog as:
$\mathit{eiffelTower}[\mathit{name} \rightarrow$ "Eiffel Tower"; $\mathit{address} \rightarrow [\mathit{city} \rightarrow \mathit{paris}; \mathit{street} \rightarrow \mathit{champDeMars}]]$.

The complex value is $[\mathit{city} \rightarrow \mathit{paris}; \mathit{street} \rightarrow \mathit{champDeMars}]$. Practically it would be preferable to allow a very general approach but this would lead to higher order semantics [4]. However, it has been suggested that complex values are usually accessed only through other objects, as in the example above. In [65], Kifer et al. propose two approaches where a complex value is represented by an object whose object identifier is unknown to the user. In the first approach, the object identifier can depend on the position of the complex value. In the second approach, the object identifier depends on the values of properties of the complex value. The first approach could easily be adopted in Gulog. However, as the authors of [65] point out, something else is required if complex values are allowed as set elements because set elements do not have unique positions. The second approach could also be adopted in Gulog. In F-logic, the object identifier is a function symbol with relevant arguments. Because function symbols are not part of Gulog, the object identifier would be a function on those arguments.

It would also be useful to be able to express parametric polymorphism in Gulog. Parametric polymorphism allows a procedure to have different types in different calls, which aids code reuse. For example, there may be a procedure that counts the number of times each letter occurs in a piece of text. To implement a procedure that counts the number of times each word occurs in a piece of text, it should be possible to use the same procedure telling it that we are now counting words rather than letters. In C++, this is possible using inheritance and virtual functions. There are two approaches to extending logic to include parametric polymorphism. One approach is described by Kifer et al. in [65] proposed extending F-logic using the Hilog approach. The Hilog approach described in [35] is a popular approach in logic programming because although it has higher order syntax, it has first-order semantics. Another approach is described by Hill and Topor in [54] for a strongly typed language. In [54], the authors define a typed first-order logic on which typed logic programs are based. Using this approach types can be variables enabling parametric polymorphism. The authors have defined declarative and procedural semantics for the typed first-order language they define. Because Gulog is a typed language and the logic has first-order semantics, this approach could be adopted in Gulog while retaining the underlying principles of Gulog. The issue of how parametric polymorphism fits in with other object-oriented features needs to be explored.

We could also consider the consequences and implications of allowing type information in the heads of clauses. This would allow the modeling of views [3, 64]. The following example is taken from [7] and illustrates how virtual classes can be incorporated in our language.

Example 7.3.1 Consider the set of declarations D:

$$poor\_student < working\_student$$
$$working\_student[salary \Rightarrow integer; age \Rightarrow integer; socins \Rightarrow integer]$$
$$poor\_student[registration\_fees \Rightarrow integer; taxes \Rightarrow integer].$$

Now consider the following program with respect to the set of declarations D:

$$\{x{:}working\_student, y{:}integer, z{:}integer\} \vdash x[salary \rightarrow y] \leftarrow$$
$$x[age \rightarrow z] \wedge y = 20 \times z$$
$$\{x{:}working\_student, y{:}integer, z{:}integer\} \vdash x[socins \rightarrow y] \leftarrow$$
$$x[salary \rightarrow z] \wedge y = 0.1 \times z$$
$$\{x{:}working\_student, s{:}integer\} \vdash x{:}poor\_student \leftarrow x[salary \rightarrow s] \wedge s \leq 200$$
$$\{x{:}poor\_student, y{:}integer, z{:}integer\} \vdash x[socins \rightarrow y] \leftarrow$$
$$x[salary \rightarrow z] \wedge y = 0.01 \times z$$
$$\{x{:}poor\_student, y{:}integer, z{:}integer\} \vdash x[registration\_fees \rightarrow y] \leftarrow$$
$$x[age \rightarrow z] \wedge y = 0.1 \times z$$
$$\{x{:}poor\_student\} \vdash x[taxes \rightarrow 0].$$

There is a clause with an object declaration in the head. The virtual class *poor_student* groups some *working_student*s based on the value of the method *salary*. Method *socins* is redefined in the virtual class and other methods are introduced. □

The following example is taken from [7] and illustrates how virtual hierarchies can be incorporated in our language. In the example if the rent of the store is too high the *big_items* are sold for a reduced amount. Under these circumstances *big_item* becomes a subtype of *cheap_item*.

Example 7.3.2 Consider the set of declarations D:

$$item[price \Rightarrow integer]$$
$$r(item, integer)$$
$$s(item, integer)$$
$$rent(store\_rent)$$
$$big\_item < item$$
$$cheap\_item < item.$$

Now consider the program with respect to the declarations D:

$$\{x{:}item, y{:}integer\} \vdash x[price \rightarrow y] \leftarrow r(x, y)$$
$$\{x{:}cheap\_item, y{:}integer\} \vdash x[price \rightarrow y] \leftarrow s(x, y)$$
$$\{x{:}store\_rent\} \vdash big\_item < cheap\_item \leftarrow rent(x) \wedge x > 2000.$$

There is a clause with an hierarchy declaration in the head. □

Including virtual classes and virtual hierarchies in Gulog would involve extensive changes to the declarative semantics, and evaluation procedures.

Also, there is no concept of update or integrity constraint in Gulog. In [20], Beeri states that he is not aware of any fully satisfactory integration of updates into a logical framework. This obviously makes the addition of updates to Gulog a difficult task. Nevertheless, updates in deductive object-oriented databases have been discussed in [10, 19, 22, 80]. Updates make it possible to change connections between objects, insert objects into types or classes, create an object and delete an object. These operations can require changes to the schema as discussed below, and the creation of object identifiers. The creation of object identifiers has been addressed in [3, 6, 66, 71]. Integrity constraints ensure that the database is consistent after a transaction. As in deductive databases, integrity constraints in deductive object-oriented databases can be represented as clauses. However, the efficient checking of integrity constraints in deductive object-oriented databases is an open research problem. Triggers are an alternative approach to expressing integrity constraints [69]. Triggers are procedures that are executed when a certain event occurs. Related to updates is the problem of view updates [96]. Views are derived from a database using a clause as described above. In relational database literature, the view update problem is discussed. That is, if a value in a view object changes, this change should be propagated to the associated base object. This is a problem in relational databases since it is unclear how to trace view updates back to updates of base tuples. In Gulog, since objects have an identity independent of their associated values, some of the problems no longer exist. Which problems are alleviated is an open question.

One of the features of object-oriented databases that is ignored in most deductive object-oriented logics, but included in F-logic [65], is the ability to query and update the schema. To include this facility in Gulog, it would be necessary to include the schema declarations in the program (or database). That is, to treat the schema declarations the same way as the data is treated. The resulting language could be a generalization of Gulog. Schema updates obviously require more complicated consistency checking. Waller discusses these issues in [108].

Much of the literature about database design for object-oriented database systems is based on transforming conceptual models into an object model [53, 99]. Database design in the relational model is based on a conceptual model and normalization. Normalizing a schema reduces the possibility of update anomalies, and is based on dependencies in the data. There has been very little work done in normalization in the area of the object-oriented model. One reason could be that normalization is unnecessary. We do not believe this is the case. In [109], Weddel shows that dependencies exist in the data in the object-oriented model. Another reason may be the lack of a well-accepted mathematical foundation for the object-oriented model. We propose that Gulog provides a context in which it is possible to study normalization for the object-oriented model.

Further, in this thesis, we have proposed procedures and discussed their efficiency. However, an important question we have not considered is how to recognize simple programs efficiently. The recognition of simple programs is clearly decidable because the language is finite (that is, there are no function symbols) and the conditions for i-stratification and well-definedness depend only on the finite set of ground instances of the program's clauses. This leaves the open problem of efficient recognition.

Finally, we propose building a prototype database system and associated design tools based on Gulog. This implementation could be used to evaluate the efficiency of the basic algorithms and data representation techniques and the extent to which Gulog assists in achieving system reliability. It could then be applied to a range of simplified computer-aided design and software engineering problems to study its applicability and performance.

Bibliography

[1] S. Abiteboul. Towards a deductive object oriented database language. *Data and Knowledge Engineering*, 5:263–287, 1990.

[2] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, Institut National de Recherche en Informatique et en Automatique, 1988.

[3] S. Abiteboul and A. Bonner. Objects and views. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 238–247, 1991.

[4] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 347–374. ACM Press/Addison-Wesley (Frontier Series), New York, 1990.

[5] S. Abiteboul and R. Hull. Data-functions, datalog and negation. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 143–153, 1988.

[6] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 159–173, 1989.

[7] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and rules. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 32–41, Washington, DC, 1993.

[8] H. Ait-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[9] H. Ait-Kaci and A. Podelski. Towards a meaning of LIFE. Technical Report 11, Digital Paris Research Labs, 1991.

[10] V. Alexiev. Mutable object state for object-oriented logic programming: A survey. Technical Report 15, Department of Computer Science, University of Alberta, 1993.

[11] P. America. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP)*, 1990.

[12] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[13] M. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, 1989.

[14] R. Bal and H. Balsters. A deductive and typed object-oriented language. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, 1993.

[15] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, September 1987.

[16] F. Bancilhon. Naive evaluation of recursively defined relations. In M. Brodie and J. Mylopoulos, editors, *On knowledge-base management systems – integrating database and AI systems*, pages 165–178. Springer-Verlag, New York, 1986. Also appeared in the Proceedings of the Islamadora Conference on Databases and AI, 1985.

[17] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM PODS Symposium on Principles of Database Systems*, pages 1–15, 1986.

[18] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 16–52, 1986.

[19] C. Beeri. Data models and languages for databases. In *Proceedings of the 2nd International Conference on Database Theory*, pages 19–40, Bruges, Belgium, 1988.

[20] C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5:353–382, 1990.

[21] E. Bertino and L. Martino. *Object-Oriented Database Systems Concepts and Architectures.* Addison-Wesley, 1993.

[22] E. Bertino and D. Montesi. Towards a logical object-oriented programming language for databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the 3rd International Conference on Database Technology*, pages 168–183, Vienna, Austria, 1992.

[23] S. Brass and U. Lipeck. Semantics of inheritance in logical object specifications. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, pages 411–430, Munich, Germany, 1991.

[24] S. Brass and U. Lipeck. Bottom-up query evaluation with partially ordered defaults. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, pages 253–266, 1993.

[25] R. Breitl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. Williams, and M. Williams. The Gemstone data management system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 283–308. ACM Press Books, 1989.

[26] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *Proc. of the 1st Int. Conference on Extending Database Technology*, Venice, Italy, March 1988.

[27] M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Joint International Conference and Symposium on Logic Programming*, pages 113–127. MIT Press, 1992.

[28] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[29] M. Carey et al. A data model and query language for EXODUS. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, 1988.

[30] R. Cattell. *Object Data Management: object-oriented and extended relational systems.* Addison-Wesley, 1991.

[31] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[32] S. Ceri, K. Tanaka, and S. Tsur, editors. *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Phoenix, 1993.

[33] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, pages 156–178, 1980.

[34] B. Chandrasekaran and W. Punch. Data validation during diagnosis, a step beyond traditional sensor validation. In *Proceedings AAAI-87 6th National Conference on Artificial Intelligence*, Seattle, Washington, July, 1987.

[35] W. Chen, M. Kifer, and D. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In *Second International Workshop on Database Programming Languages*, San Mateo, CA, June, 1989. Morgan Kaufmann.

[36] W. Chen and D. Warren. C-Logic of complex objects. In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, PA, March, 1989.

[37] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[38] C. Delobel, M. Kifer, and Y. Masunaga, editors. *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, 1991. Springer-Verlag. Published as Lecture Notes in Computer Science 566 by Springer-Verlag.

[39] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[40] P. Dewan, A. Vickram, and B. Bhargava. Engineering the object-relation model of O-raid. *Proceedings of the Fourth International Conference on Foundations of Data Organisation and Algorithms*, pages 389–403, 1989. published as Lecture Notes in Computer Science 367 by Springer-Verlag.

[41] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Proc. 3rd Symposium on Logic Programming*, pages 264–272, San Francisco, California, 1987.

[42] S. W. Dietrich. Shortest path by approximation in logic programs. Technical Report TR-91-024, Department of Computer Science and Engineering, Arizona State University, 1992.

[43] G. Dobbie and R. Topor. Representing inheritance and overriding in Datalog. *Computers and AI*, 13(2-3):133–158, 1994.

[44] G. Dobbie and R. W. Topor. A model for inheritance and overriding in deductive object-oriented systems. In *Proc. 16th Australian Computer Science Conference*, pages 625–634, Brisbane, Australia, 1993.

[45] G. Dobbie and R. W. Topor. A model for sets and multiple inheritance in deductive object-oriented systems. In *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, Scottsdale, Arizona, 1993.

[46] G. Dobbie and R. W. Topor. Representing inheritance and overriding in Datalog. In *Proc. of the Deductive Database Workshop in conjunction with ICLP'93*, Budapest, Hungary, 1993.

[47] G. Dobbie and R. W. Topor. A model for sets and multiple inheritance in deductive object-oriented systems. *Journal of Intelligent Information Systems:Integrating Artificial Intelligence and Database Technologies*, 4(2):193–219, 1995.

[48] M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[49] H. Gallaire and J. Minker. *Logic and data bases.* Plenum Press, 1978.

[50] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, 1983.

[51] D. Harel and T. Hirst. Completeness results for recursive data bases. In *Proceedings of the Twelth ACM PODS Symposium on Principles of Database Systems*, 1993.

[52] J. Harland and K. Ramamohanarao. An Aditi implementation of a flights database. In *Proceedings of the Workshop on Programming with Deductive and Rule-Oriented Databases*, Vancouver, Canada, 1993.

[53] R. Herzig and M. Gogolla. Transforming conceptual data models into an object model. In *Proceedings of the 11th International conference on the Entity-Relationship Approach*, pages 280–298, Karlsruhe, Germany, 1992.

[54] P. M. Hill and R. W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, Cambridge, Massachusetts, 1992.

[55] R. Hull and R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):201–260, 1987.

[56] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proc. 16th International Conference on Very Large Data Bases*, pages 455–468, 1990.

[57] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-oriented Programming*, 1988.

[58] J.-P. Jouannaud, C. Kirchner, H. Kirchner, and A. Megrelis. Programming with equalities, subsorts, overloading, and parameterization in OBJ. *Journal of Logic Programming*, 12:257–279, 1992.

[59] D. B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998, October 1989.

[60] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In V. Saraswat and K. Ueda, editors, *Proc. 1991 International Logic Programming Symposium*, pages 387–401, San Diego, USA, 1991.

[61] D. B. Kemp, P. J. Stuckey, and D. Srivastava. Bottom-up evaluation and query optimization of well-founded models. To appear in *Theoretical Computer Science*.

[62] D. B. Kemp and R. W. Topor. Completeness of a top-down query evaluation procedure for stratified databases. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th International Conference and Symposium on Logic Programming*, Seattle, Aug, 1988.

[63] W. Kent. Limitations of record-based information models. *ACM Transactions on Database Systems*, 1(4):107–131, 1979.

[64] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 393–402, 1992.

[65] 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, 1990. Further revised as Technical Report 93/06, April 1993.

[66] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia, PA, March, 1989.

[67] W. Kim, N. Ballou, H. Chou, J. Garza, and D. Woelk. Features of the ORION object-oriented database system. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 251–282. ACM Press Books, 1989.

[68] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, 1989.

[69] H. F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, Inc., 1991.

[70] G. Kuper. Logic programming with sets. In *Proc. 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–20, 1987.

[71] G. Kuper and M. Vardi. A new approach to database logic. In *Proc. 3rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1984.

[72] E. Laenens, N. Leone, P. Rullo, and D. Vermeir. Efficient query evaluation in a language combining object-oriented and logic programming. In *DB92*, pages 96–110, 1992.

[73] E. Laenens and D. Vermeir. Assumption-free semantics for ordered logic programs: On the relationship between well-founded and stable partial models. *Journal of Logic and Computation*, 1(2):159–185, 1992.

[74] M. J. Lawley. A Prolog interpreter for F-logic. Unpublished report, Griffith University, 1993.

[75] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases, 1993. To appear in the *New Generation of Computing Journal*. An earlier version appeared in the proceedings of the FGCS'92 conference.

[76] S. B. Lippman. *C++ Primer*. Addison Wesley, 1991.

[77] J. W. Lloyd. *Foundations of Logic Programming (Second, Extended Edition)*. Springer Series in Symbolic Computation. Springer-Verlag, New York, 1987.

[78] J. W. Lloyd, E. Sonenberg, and R. Topor. Integrity Constraint Checking in Stratified Databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[79] Y. Lou and Z. M. Ozsoyoglu. LLO: An object-oriented deductive language with methods and method inheritance. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 198–207, 1991.

[80] D. Maier. A logic for objects. Technical Report CS/E-86-012, Oregon Graduate Center, Beaverton, OR, 1986.

[81] F. G. McCabe. *Logic and Objects*. Prentice Hall, 1992.

[82] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[83] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc., 1989.

[84] D. A. Moon. The COMMON LISP object-oriented programming language. In W. Kim and F. H. Lochovsky, editors, *Object-oriented concepts, Databases, and Applications*, pages 49–78. ACM Press Books, 1989.

[85] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 264–277, 1990.

[86] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[87] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York, 1989.

[88] Ontos. *ONTOS Reference Manual*. Ontos, Inc., Billerica, Massachusetts, 1989.

[89] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1992.

[90] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988. Further revised as [91].

[91] T. C. Przymusinski. Every logic program has a natural stratification and an iterated fixed point model. In *Proceedings of the Eighth ACM PODS Symposium on Principles of Database Systems*, pages 11–21, 1989.

[92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the Search in Bottom-Up Evaluation. In *Proc. of the Joint Int. Conference and Symposium on Logic Programming*, pages 273–287, Washington DC, November 1992.

[93] K. A. Ross. Modular stratification and magic sets for DATALOG programs with negation. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, pages 161–171, 1990.

[94] K. A. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *Proc. 11th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992.

[95] M. Roth and H. Korth. The design of non-1NF relational databases into nested normal form. In *Proc. of the ACM SIGMOD International Conference on the Management of Data*, 1987.

[96] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, pages 189–207, Munich, Germany, 1991.

[97] G. Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–97, 1992.

[98] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, July 1986.

[99] Z. Tari. On the design of object-oriented databases. In *Proceedings of the 11th International conference on the Entity-Relationship Approach*, pages 389–405, Karlsruhe, Germany, 1992.

[100] R. W. Topor. Safe database queries with arithmetic relations. In *Proc. 14th Australian Computer Science Conference*, pages 02–1–02–13, 1991.

[101] D. S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, Los Altos, CA, 1986.

[102] S. Tsur. Applications of Deductive Database Systems. In *Proc. of 35th IEEE Computer Society Int. Conf. (COMPCON)*, pages 511–518, Spring 1990.

[103] J. D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

[104] A. Van Gelder. Foundations of aggregation in deductive databases. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the Third International Conference on Deductive and Object-Oriented Databases*, 1993.

[105] A. Van Gelder, K. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the Seventh ACM PODS Symposium on Principles of Database Systems*, 1988.

[106] L. Vieille. Recursive axioms in deductive databases: The Query/Subquery approach. In L. Kerschberg, editor, *Proceedings of the Second International Conference on Expert Database Systems*, pages 179–194, April 1986.

[107] L. Vieille. Database-complete proof procedures based on SLD resolution. In *Proceedings of the 4th International Conference on Logic Programming*, pages 74–103, 1987.

[108] E. Waller. Schema updates and consistency. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases*, pages 167–188, Munich, Germany, 1991.

[109] G. E. Weddel. A theory of functional dependencies for object-oriented data models. In W. Kim, J.-M. Nicholas, and S. Nishio, editors, *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 150–169, Kyoto, Japan, 1989.

[110] K. Wilkinson, P. Lyngboek, and W. Hassan. The Iris architecture and implementation. *IEEE Transactions of Data and Knowledge Engineering*, 2(1):63–75, 1990.