INTEGRATING FUNCTIONAL LOGIC AND OBJECTORIENTED PROGRAMMING LANGUAGE PARADIGMS WITH APPLICATION TO DEDUCTIVE DATABASES

by

ZEKI OGUZ BAYRAM

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree
of Doctor of Philosophy in the Department of Computer and
Information Sciences in the Graduate School,
The University of Alabama at Birmingham

BIRMINGHAM ALABAMA

ABSTRACT

This dissertation is about the ultimate integration of the functional, logic and object-oriented programming language paradigms and the application of such a combined framework to the area of object-oriented deductive databases. The significance of such an integration is that it permits declarative, symbolic manipulation of complex objects, a major step in the advancement of software engineering.

Towards that goal, we first describe a higher order functional/logic language, ROSE, that is implemented in the Prolog language. Programs of ROSE consist of conditional rewrite rules with optional committing guards. The operational semantics of ROSE is *conditional narrowing* and *e-unification*, augmented to deal with committing guards. Committing guards, combined with non-determinism in rewrite rules, give the programmer extra-logical control over the execution of programs in a sequential environment. A consequence of this is the possibility of defining the *not* function which implements the *negation as finite failure* rule in the context of functional/logic programming.

DataFunLog (DFL for short) is a deductive database model that adapts functional/logic programming to the area of databases. A DFL database is defined by a set of conditional rewrite rules. DFL subsumes the logic and functional data models. One major achievement in DFL is the development of a query evaluation algorithm that terminates even if the rules defining the database are non-terminating.

We then describe FLOOP, a language integrating functional, logic and object-oriented programming language paradigms. Programs of FLOOP consist of conditional rewrite rules, augmented to permit object-expressions to appear in the place of constants. The operational semantics is e-unification through *transformations*, and calls to an underlying object-expression evaluator. FLOOP, which is implemented in Smalltalk, achieves the ultimate integration of the three paradigms we mentioned above.

Next, we describe an object-oriented database system, *Next Generation Opal (NGO)*, implemented in Smalltalk, which, in combination with FLOOP, results in an object-oriented deductive database model with object-oriented deduction. In this database model, inter-object relationships are described using conditional rewrite rules and queries are posed declaratively in the form of equations to be solved.

DEDICATION

I dedicate the work reported in this dissertation to my beloved father.

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, Dr. Barrett Bryant, for being a constant source of encouragement and guidance during my six years of graduate studies, for supporting me with a research assistantship, but most importantly for being my good friend and mentor. I owe a great deal to Dr. Warren Jones for helping me stay focussed on my area of research at crucial points by his very helpful suggestions. I thank Drs. Edwin Battistella, Christer Bennewitz, Barrett Bryant, Warren Jones and Alan Sprague for graciously accepting to serve in my dissertation committee and making very helpful suggestions regarding my research direction. I thank Mr. Hisao Tamaki at Ibaraki University in Japan for the very fruitful discussions we had regarding my research, and the Japanese government for awarding me a scholarship which made my visit to Japan possible. My appreciation also goes out to our secretaries Ms. Barbara Rivers, Ms. Joan Smith and Ms. Jennifer Williams, and to our system administrator Mr. Bruce Williams for always holding out a hand when I needed them.

Contents

ABSTRACT iii

DEDICATION v

ACKNOWLEDGEMENTS vi

LIST OF FIGURES ix

CHAPTER 1: INTRODUCTION 1
- 1.1 An Overview of Functional, Logic, and Object-Oriented Programming . 1
  - 1.1.1 Functional Programming 2
  - 1.1.2 Logic Programming 2
  - 1.1.3 Object-Oriented Programming 3
  - 1.1.4 Integration of the Three Paradigms 4
  - 1.1.5 Application Areas for the Integrated Functional/Logic/Object-Oriented Programming Paradigms 5
- 1.2 Database Models Compared 5
  - 1.2.1 The Relational Data Model 5
  - 1.2.2 The Functional Data Model 6
  - 1.2.3 The Logic Data Model 6
  - 1.2.4 The Object-Oriented Data Model 7
  - 1.2.5 The *DataFunLog* Data Model 7
  - 1.2.6 Next Generation Opal (NGO) 8
  - 1.2.7 Combining NGO and FLOOP to get a Deductive Object-Oriented Database Model with Functional Syntax 8
- 1.3 Looking Ahead 8

CHAPTER 2: THE FUNCTIONAL/LOGIC PROGRAMMING LANGUAGE ROSE 10
- 2.1 A Brief Description of ROSE 11
- 2.2 ROSE In Detail 12
- 2.3 Operational Semantics of ROSE 14
  - 2.3.1 (Unconditional) Narrowing 15
  - 2.3.2 Conditional Narrowing 16
  - 2.3.3 Narrowing Strategies 17

2.3.4 E-Unification 17
2.3.5 Committing Guards 18
2.3.6 Using Conditional Guards to Define the Function *not* 19
2.3.7 Implementation of ROSE 19
2.3.7.1 (Lack of) Parsing 19
2.3.7.2 Representation of Functions in the Prolog Database 20
2.3.7.3 Predicates that Do the Narrowing 21
2.4 Sample Programs in ROSE 24
2.5 Related Work 28
2.6 Conclusion 29

CHAPTER 3: A DEDUCTIVE DATABASE MODEL BASED ON CONDITIONAL TERM REWRITING SYSTEMS 31
3.1 The *DataFunLog* (DFL) Data Model 35
3.1.1 The Meaning of DataFunLog Programs Without Negation 36
3.1.2 An Algorithm for Computing the Answer to a Given Query 40
3.1.3 Examples 42
3.1.4 Termination of the Query Evaluation Algorithm and the Nature of the Computed Answer 44
3.1.5 Correctness and Completeness of the Query Evaluation Algorithm 46
3.1.5.1 Databases Consisting of Unconditional Rules 47
3.1.5.2 Databases Consisting of Conditional Rules 51
3.1.6 Translation of DataLog to DFL 55
3.2 DataFunLog With Negation 58
3.2.1 Stratification 58
3.2.2 Query Evaluation Algorithm for DFLN Programs 61
3.3 Related Work 64
3.4 Summary, Conclusions and Future Work 65

CHAPTER 4: INTEGRATING FUNCTIONAL, LOGIC AND OBJECT-ORIENTED PROGRAMMING PARADIGMS: THE LANGUAGE FLOOP 68
4.1 FLOOP: A Brief Introduction to the Language 70
4.2 Operational Semantics Through Transformations 73
4.2.1 The Transformation Algorithm 74
4.2.2 Transformations 75
4.3 Implementation 78
4.3.1 Smalltalk Implementation 78
4.3.2 ML Implementation 80
4.4 FLOOP Through Examples 83
4.5 Related Work 90
4.6 Conclusion and Future Research Directions 92

CHAPTER 5: AN OBJECT-ORIENTED DEDUCTIVE DATABASE MODEL BASED ON CONDITIONAL REWRITE RULES 94
5.1 Next Generation Opal 96
5.2 Problem Definition 98

5.3 Solution Using NGO (Purely Object-Oriented) 98
5.4 Solving the Problem Declaratively 103
5.5 Discussion 107
5.6 Related Work 108
5.7 Conclusion and Future Research Directions 110

CHAPTER 6: CONCLUSION AND FUTURE RESEARCH DIRECTIONS . . 111

BIBLIOGRAPHY 114

APPENDIX A: SAMPLE ROSE PROGRAMS 118
A.1 Predefined Functions 118
A.2 User Defined Modules and Functions 120

APPENDIX B: SAMPLE FLOOP PROGRAMS 132

List of Figures

2.1 Narrowing Algorithm (Using Unconditional Rules) 16
2.2 Narrowing Algorithm (Using Conditional Rules) 17
2.3 Defining A Logically Wrong Function That Exhibits Correct Behavior . . . 19
2.4 Defining The *not* Function Using Committing Guards 19
2.5 Rules For Calculating *factorial* And *fibonacci* 24
2.6 Some List Functions 25
2.7 Query With Unbound Variables 25
2.8 Query For *revstrings* 26
2.9 Higher-Order Functions 26
2.10 Query For Some Higher-Order Functions 27
2.11 Function Definitions Demonstrating Infinite Lists 28
2.12 Query For Generating The First Three *lucky* Numbers 28
3.1 Algorithm for Computing the Rule Set to Answer a Query 41
3.2 Algorithm to Find a Stratification for a DFLN Program 60
3.3 Finding the Answer to a Given Query for DFLN Programs 62
5.1 Definition Of *Tuple* and *MySet* 97

CHAPTER

INTRODUCTION

This dissertation is about the ultimate integration of the functional, logic and object-oriented programming (OOP) paradigms and the application of such a combined framework to the area of object-oriented deductive databases. The significance of such an integration is that it will permit declarative, symbolic manipulation of complex objects, resulting in programs that are not only expressive, easy to write, understand and analyze (a trait of functional/logic programming) but also reusable, and possess very powerful modeling abilities of the problem domain (a trait of object-oriented programming). Towards that goal, we first describe a higher order functional/logic language, ROSE, that is implemented in the Prolog language, and a deductive database model, *DataFunLog*, that is derived directly from it. We then describe FLOOP, a language integrating all of the three paradigms mentioned above, implemented in Smalltalk. Next, we describe an object-oriented database system, *Next Generation Opal (NGO)*, also implemented in Smalltalk, which, in combination with FLOOP, results in an object-oriented deductive database model with object-oriented deduction.

An Overview of Functional Logic and ObjectOriented Programming

In integrating different programming language paradigms, we strive to preserve the "best" features of each paradigm involved, and incorporate those features in a coherent framework in the new language we are designing. In the next few sections, we shall try to outline the distinguishing characteristics of the functional, logic and object-oriented programming language paradigms that are involved in the integration we are proposing.

Functional Programming

Functional programming is based on the mathematical notion of *functions* whereby a function defines a mapping from the elements of one domain to elements of another domain. A functional program consists of a set of *function definitions*.

Computation in functional programming consists of finding the value *denoted* by a particular expression, given the function definitions that make up the program. This is achieved through repeated simplification of the given expression through the use of the function definitions until simplification is no longer possible, in which case the expression obtained is the value *denoted* by the original expression.

The features characteristic of modern functional languages include *set abstraction*, *lazy evaluation*, *strong type checking*, *higher order functions*, *pattern matching*, *polymorphism*, and *polymorphic type inference* [40, 52].

Logic Programming

A logic program consists of a set of *axioms* as well as a *goal statement* whose validity we are trying to prove given the axioms. In the process, we are also trying to find values for the *uninstantiated variables* in it that would indeed make the goal statement a *logical consequence* of the axioms. Thus logic programming (restricted to first order relational Horn Clauses with resolution as the operational semantics) includes the concepts of *logical variables*, *constraint satisfaction* ("for what values of the logical variables in the goal statement is the goal statement true ?") and a *declarative style of programming*, since no information is present in the program itself as to how the values for the variables in the goal statement will be found which will make the goal statement a logical consequence of the axioms that make up the logic program. We refer the interested reader to [35] for a detailed description of and the theoretical basis for logic programming.

Logic programming has been popularized by the Prolog [21] programming language.

ObjectOriented Programming

The real world is composed of distinct objects interacting with each other in specific ways. The object-oriented paradigm tries to capture that reality as authentically as possible through the use of *classes*, *objects* that are *instances* of classes, *class hierarchies*, *inheritance*, *methods* that define the *behavior* of objects and *messages* that *activate* methods.

As software systems became larger and larger and more and more complex, the need to divide a program into meaningful units of data and operations with a well-defined interface between the units has led to the development of *modules*. Modules have *local state* in the form of *global variables* that can be seen only by operations defined in the module, as well as an *interface* part and *implementation* part. The interface defines which operations and data in the module will be seen from the *outside*.

An *abstract data type* can be seen as a specialized use of a module, where the aim of the module is not necessarily to help in the manageability of a large program by breaking it up into smaller units, but rather to define a set of data types that define a certain kind of an object, and operations on that kind of object. The only operations allowed on the abstract data type are those defined in the module. An example of the abstract data type is the well-known *stack* with operations such as *pop*, *push*, *empty*. The *packages* of Ada [1] and the *abstype* and *local* declarations of ML [40] are good examples of modules and abstract data types as realized in concrete programming languages.

The object-oriented paradigm can be seen as an evolution from the concepts of *modules* and *abstract data types*. An object has *local state*, as in the case of modules, and operations (called *methods*) that operate only on the object for which they are defined (as in the case of abstract data types). In addition, however, there is the notion of *inheritance* whereby methods and structure can be inherited from an existing object type (i.e., *class*). Furthermore, due to the state changing nature of objects, we can claim that the object-oriented paradigm is an extension of the *imperative* style of programming.

The advantages of object-oriented programming then include *encapsulation* (i.e., close coupling between data and operations on that data), *authentic representation* of objects as well as the *interaction between objects* in a problem domain, *code reusability* and *specialization* through the inheritance mechanism and *polymorphism*. As a result of all of these features, programs written in an OOP language are extensible, reusable, maintainable, and robust [48].

Object-oriented programming was pioneered in the Simula [13] programming language and popularized by Smalltalk [2]

Integration of the Three Paradigms

Obviously, each paradigm has its application area where it is most suited: OOP mainly in *simulation* and problem areas where there is a need to model a real world system, such as a machine, and functional and logic paradigms in symbolic manipulation, which is the backbone of *artificial intelligence* and *automated theorem proving (ATP)*. By integrating the three paradigms we shall make symbolic manipulation at the object level possible. This will allow us to write programs that are not only semantically very expressive, but also represent the program domain very authentically and are reusable and robust. We believe this will be a solid contribution to the area of software engineering and a positive step in addressing the *software crisis*.

We have implemented a combined functional/logic/object-oriented language called FLOOP.

Application Areas for the Integrated FunctionalLogicObject Oriented Programming Paradigms

We expect that our combined functional/logic/object-oriented language, FLOOP, will find application mainly in the areas of *knowledge representation*, *expert systems*, *rapid prototyping*, and *deductive databases*. The combined abstraction facilities of object-oriented programming and the high level symbolic manipulation offered by functional/logic programming make it suitable for knowledge representation. Its deductive nature makes it suitable for expert system development and deductive databases, and the fact that it is *high-level* makes it suitable for rapid prototyping of large and complex software.

Database Models Compared

In the real world, there are *objects* or *entities*, and relationships between objects. Such relationships, rather than being explicitly stated, are sometimes required to be computed using *existing relationships* and *rules for generating new relationships*.

Various database models use different schemes for representing objects. Below we give some of the more widely used data models and describe briefly how they represent objects and the relationships between objects.

The Relational Data Model

The *relational model* simulates objects through the use of *relations*. Relationships *between* objects are described through relating the *keys* of objects by using a separate relation. Consequently, the concept of *keys* is needed to identify objects. Keys used this way effectively act as *object identifiers*.

The method of computation in the relational model is through the use of a fixed set of algebraic operations which do not permit recursion. This restriction precludes the computation of the *transitive closure* of a relation, a serious drawback of the relational model.

The Functional Data Model

The *functional* data model is based on the concept of the application of *functions* to *datatypes* [33]. Functions are used both to represent inter-object relationships, as well as attributes of objects [30]. Functions (sometimes called *data* functions) are necessarily multi-valued. This model supports intuitive queries through composition of functions and inverse functions [42]. One of its drawbacks, however, is its inability to represent directly *n-ary* relationships where $n \geq 3$, in which case an entity type participating in n binary relationships is required [43].

The functional data model was pioneered by Shipman [46] in the language *Daplex*.

The Logic Data Model

The *logic* model of data uses first order predicate calculus to describe objects and relationships between objects. As realized in the *Datalog* language [53], only universally quantified *clauses* are permitted to be part of a logic program defining a database. A logic program then basically is a conjunction of *facts* and *implications*.

The logic data model is certainly more powerful than the relational model in that recursion is allowed. The logic model also provides a declarative, very powerful and simple query language (queries are existentially quantified statements in first order logic). The problem of representation of complex objects still remains however, with the same kinds of techniques as in the relational model being used for representing objects and relationships between objects.

The ObjectOriented Data Model

In the object-oriented model, real world objects whose representation is needed in the database are directly represented as *database objects* that are *instances* of some *class*. An object has *attributes* and a unique *object identifier*.

Although in this model real world objects are represented very authentically, *external* relationships among objects are necessarily represented through the use of pointer valued attributes, which necessitates the physical traversal of pointers representing relationships in answering a query. This is less than ideal, since external relationships among objects are declarative, and should be handled in a declarative way. Ideally, the only pointer valued attributes in an object should be those which relate a *part* to its component *subparts*.

The *DataFunLog* Data Model

DataFunLog (described in chapter 3 of this dissertation) is a deductive database model with functional syntax. It adapts functional/logic programming to the area of databases. It can be seen as the functional data model with the addition of logical variables and the capability to represent arbitrary n-ary relations using functions. DataFunLog thus subsumes both the functional data model and the *logic* data model, keeping the "best" aspects of each (*functional notation* and *function composition* from the functional data model, the *logical variable* and *function inversion* from the logic data model).

Next Generation Opal NGO

NGO, described later in this dissertation in detail, is a pure object-oriented database system that is modeled after the Opal database language [16]. It is implemented in Smalltalk. It provides two subclasses, *Tuple* and *MySet*. *Tuple* is used for creating classes whose instances have typed attribute names, and *MySet* is used for creating sets whose contents must be of a specific type. Its main utility is that in combination with the programming language FLOOP (also to be described in detail later) it gives a deductive object-oriented database model.

Combining NGO and FLOOP to get a Deductive Object Oriented Database Model with Functional Syntax

Since both FLOOP and NGO are implemented in Smalltalk, their integration comes naturally. Since any object-expression can appear inside a FLOOP program, expressions denoting objects created as instances of a subclass of *Tuple* can also appear inside a FLOOP program. This combination provides for us the declarative creation of (complex) database objects, their symbolic manipulation, as well as a declarative query language. Although not a direct extension of DataFunLog, in essence this model combines all the features of DataFunLog with object orientation.

Looking Ahead

The remainder of this thesis is as follows. Chapter 2 gives a detailed description of functional/logic programming as it is realized in the ROSE programming language. Chapter 3 develops the DataFunLog database model, and gives a formal semantics for it based on term rewriting. DataFunLog is a direct adaptation of functional/logic programming to the area of databases. Chapter 4 introduces the FLOOP language and its implementation in Smalltalk. FLOOP combines all the three paradigms of functional, logic and object-oriented programming into a simple and coherent framework. Chapter 5 describes a purely object-oriented database system, NGO, which, in combination with FLOOP, gives a deductive, object-oriented database model. Chapter 6 is the summary and future research directions. Appendix A contains sample ROSE programs and Appendix B contains sample FLOOP programs.

CHAPTER

THE FUNCTIONALLOGIC PROGRAMMING LANGUAGE ROSE

We present a higher-order functional/logic language, ROSE. The programs of ROSE are made up of conditional constructor based term rewriting systems. The conditions (guards) in the rules can optionally be committing. The operational semantics of the language is conditional narrowing, augmented to deal with committing guards. The major innovation of the language is the use of committing guards and backtracking to make possible very practical, operationally oriented programs. We show many practical examples, one of which is the definition of the extra-logical function *not*, which implements the *negation-as-finite-failure* rule in the context of functional/logic programming (a *first* in this area).

ROSE is a powerful functional/logic programming language. In designing ROSE, we intended to create a practical language with as simple a syntax as possible, but at the same time one that gave the programmer as much flexibility as possible in controlling the execution of his programs. Thus, we have adopted *conditional constructor based term rewriting systems* augmented with *higher-order constructs* and *committing guards* as our language, and an integrated *e-unification/narrowing* algorithm that can employ either the *innermost* or the *outermost* narrowing strategy (controlled by the programmer) as the operational semantics.

Many efforts have been made for the integration of logic and functional programming paradigms in a common framework (see, for example [14, 22, 24, 26, 34, 45, 47]). Some form of *narrowing* lies at the heart of the operational semantics of most of the functional/logic languages proposed so far. These languages are *pure* in that they have no facility, similar to the *cut* predicate in Prolog, for controlling the execution of the program in a sequential environment. We present here an *impure* functional/logic programming language, ROSE, which permits *committing guards* in rewrite rules, which together with non-determinism give the functionality of *cut* in the context of functional/logic programming.

A Brief Description of ROSE

The language adopted for ROSE is *constructor based conditional term rewriting systems* with optional *committing guards (conditions)*. Functions can be defined very elegantly using conditional rewrite rules [23], and allowing committing guards (we use the terms *guard* and *condition* interchangeably) not only gives more control of program execution in a sequential environment to the programmer, but also facilitates the writing of total functions, eliminating one of the causes of the incompleteness of the innermost narrowing strategy. It is in effect the counterpart of the *cut* predicate of Prolog in an *equational* setting.

ROSE has higher-order features, such as the passing of functions as arguments and returning them as values. To facilitate this, *nameless functions* in the form of *conditional lambda abstractions* are part of ROSE syntax. This is the counterpart of lambda abstractions in functional languages. Also, *apply* is a built-in function of the language.

The operational semantics makes use of two narrowing strategies, as specified by the programmer: the default strategy can be *innermost*, which results in an *eager* evaluation of expressions, or *outermost* which results in a *lazy* evaluation of expressions and allows the handling of infinite data structures. Individual subterms of a term can be marked for execution using either strategy, which gives the programmer a lot of flexibility in determining the strategy to be used in evaluating expressions.

The *e-unification/narrowing* algorithm employed can solve equations using either the *(leftmost) innermost* or *(leftmost) outermost* narrowing strategies. Although incomplete in general, this kind of e-unification solves many practical problems, but it is the responsibility of the programmer to predict the outcome of his programs based on the evaluation strategy he chooses. Thus, not all the answers dictated by a declarative reading of the programs shall be generated by the interpreter. We do not think this is a great handicap, since many programmers think operationally, rather than declaratively (Prolog [21] after all, does not employ a complete *resolution* strategy, but this has not prevented it from becoming the most widely used logic programming language).

Other researchers (see, for example [23, 25, 28, 32]) have investigated the conditions under which narrowing and/or conditional narrowing is complete. Some sort of restriction is usually imposed on the set of rewrite rules to achieve the completeness results.

ROSE In Detail

In *constructor based term rewriting systems*, there are two kinds of function symbols: *data constructors*, or just *constructors*, and *defined function symbols*, also called *function names*. Constructors are used to represent data, much in the same way of the list constructor "." in Lisp. Zero-ary constructors are also called *constants*.

Below we give a recursive definition of *term* as it applies to ROSE (in this presentation the terms *term* and *expression* are used interchangeably).

- A constant (zero-ary constructor) is a term.
- A variable is a term. Every variable starts with a capital letter.

- If $s_1, \ldots, s_n$ are terms and f is a function name, then $f(s_1, \ldots, s_n)$ is a term, also called a *function application.*
- If $s_1, \ldots, s_n$ are terms and c is a constructor, then $c(s_1, \ldots, s_n)$ is a term, also called a *constructed expression.*
- $(lambda, [s_1, \ldots, s_n], condition, body)$ is a term, called a *conditional lambda abstraction* (or just *lambda abstraction*), if $s_1, \ldots, s_n$ are terms *in normal form* (see below), and *condition* and *body* are terms. $s_1, \ldots, s_n$ are the *formal parameters* of the lambda abstraction.
- $apply(s_1, \ldots, s_n)$ is a term, called and *apply expression*, if $s_1, \ldots, s_n$ are terms. s_1 should be an expression that evaluates to a lambda abstraction and $s_2, \ldots, s_n$ are the actual parameters s_1 will be applied to.

The *scope* of the variables in $s_1, \ldots, s_n$ of a lambda abstraction $(lambda, [s_1, \ldots, s_n], condition, body)$ is the *condition* and *body*. All such variables in *body* or *condition* are said to be *bound*. If a variable is not bound, then it is said to be *free*.

A term is said to be *in normal form* if it consists *only* of constructors, variables and lambda abstractions of the form $(lambda, [s_1, \ldots, s_n], guard, body)$.

A ROSE program consists of an ordered set of conditional rewrite rules of the form

$$f(s_1, \ldots, s_n) : condition \rightarrow body$$

or

$$f(s_1, \ldots, s_n)\#condition \rightarrow body$$

where $s_1, \ldots, s_n$ are all in terms in normal form, f is a function symbol, and *condition* and *body* are terms. The first definition above is a *non-committing* definition, whereas the second one is a *committing* definition. Note that unconditional rewrite rules are equivalent to conditional ones where the condition is the constant *true*.

ROSE also has a module system with *public* and *private* parts in each module. Let m be the name of a module. The functions defined in a public part of m are visible to other modules that *use* m, whereas the functions in the private part of m are visible only *within* m. Any function in the public section of a module can be invoked from any module by prefixing it with the module it belongs to. For example, if $f(\ldots)$ is defined in the module m, $m :: f(\ldots)$ invokes $f(\ldots)$ in module m. A function f is *accessible* in a module n if f is defined in n, if f is in a module that n *uses*, or if f is *accessible* in some module n' and n *uses* n'.

Operational Semantics of ROSE

The operational semantics of ROSE is based on *conditional narrowing*. First, we give some terminology and notations.

We denote a substitution σ by $\{t_1/X_1, \ldots, t_n/X_n\}$ where X_i, $1 \leq i \leq n$, are called the *replaced variables*, and t_i, $1 \leq i \leq n$, are called the *replacing terms*. An application of a substitution σ to a term t, denoted by $(t)\sigma$, is the term obtained by *simultaneously* replacing all unbound variables X in t with v such that $v/X \in \sigma$. *Composition* of two substitutions σ_1 and σ_2, denoted by σ_1 o σ_2, is defined to be a substitution θ such that for any term t, $(t)\theta{=}(t)(\sigma_1 \text{ o } \sigma_2) = ((t)\sigma_1)\sigma_2$.

Let W be a set of variables, and σ a substitution. The *restriction* of σ to W, $\sigma[W]$, is defined as $\{t/X | X \in W \text{ and } t/X \in \sigma\}$. VAR_t is the set of variables in the term t, and $O(t)$ is the set of all occurrences of t (i.e., the set of addresses of all the subterms of t). $t[u]$ denotes the subterm of t at occurrence u.

A *redex* is a term of the form $f(\ldots)$ where f is a *function name*. An *innermost redex* is a redex that does not have a proper subterm that is a redex ($t[u]$ is a *proper subterm* of a term t if $t[u] \neq t$).

Unconditional Narrowing

Let R denote a set of rewrite rules. Given an expression E, let $E[u]$ denote the subterm of E at occurrence u, $E[u \leftarrow t]$ the term obtained from E by replacing $E[u]$ with the term t, $lhs \rightarrow rhs$ be a rewrite rule and σ be the *most general unifier (mgu)* of lhs and $E[u]$. If $E' = (E[u \leftarrow rhs])\sigma$ then E and E' are said to belong to the *one step narrowing relation* defined on terms that can be generated using the function symbols and constants of the language, together with a finite set of variables. For two terms E and E' as above, we denote membership in the one step narrowing relation as $E >>_\sigma E'$ which also gives information about the substitution needed for the narrowing step.

Given a term and a set of rewrite rules, through repeated applications, the one step narrowing relation generates a *tree* of terms, which we shall call the *narrowing tree*. Let E be the expression to be evaluated and P a set of unconditional rewrite rules. The narrowing algorithm in figure 2.1 creates a *labeled* narrowing tree for an expression E, where if the tree is finite, then all the leaves of the tree are labeled with either *success* or *failure*.

Let E_1 and E_2 be two nodes in the narrowing tree of some expression E, and let E_2 be a descendant of E_1. Then, we say that E_1 can be *narrowed* to E_2 with substitution θ, where θ is the composition of the substitutions that label the arcs of the path from E_1 to E_2. In a narrowing tree, we are usually interested only in paths from the root to leaves that are labeled *success*. If the *negation as failure* rule will be used to determine the validity of a negated term, though, we would also be interested in the *absence* of a path from the root

1. Create a tree T with only one node, E (the expression to be narrowed).
2. Let L be a leaf in the narrowing tree generated so far.
   - If L is in *normal form*, label it *success*.
   - If there is no rule $lhs \rightarrow rhs$ in P such that some subterm of L is unifiable with lhs, label it *failure*.
   - Otherwise, for any rule $lhs \rightarrow rhs \in P$ and any $L[u]$ such that $u \in O[L]$: if $L[u]$ is unifiable with lhs with mgu σ, create a child L' of L such that $L' = (L[u \leftarrow rhs])\sigma$, and label the arc between L and L' as σ (this is a one step narrowing operation applied at every possible occurrence of L using every possible rule).
3. Repeat 2 until all leaves marked, or forever!

Figure 2.1: Narrowing Algorithm (Using Unconditional Rules)

to a *success* leaf.

Conditional Narrowing

Given a term E and a set of (conditional) rewrite rules, the algorithm in figure 2.2 generates a *conditional narrowing tree* for E. This algorithm performs conditional narrowing essentially equivalent to that described in [28], but generates the narrowing tree explicitly. Note that a *forest* of conditional narrowing trees is generated during the execution of the algorithm due to the recursive calls for evaluating the conditions. For two expressions E and E' that are in the conditional narrowing tree for some expression, if E' is a *direct descendant* of E (i.e., there are no other nodes between E and E') and the arc between them is labeled with σ we say that E and E' belong to the *one step conditional narrowing relation* and denote this fact as $E >>>_\sigma E'$.

Notice that conditional narrowing subsumes *standard* or *unconditional* narrowing in that all unconditional rewrite rules are trivially conditional, where the condition is the constant *true*. Note also that conditional narrowing is a semantically much more complex operation than simple narrowing, which is what gives the conditional rewrite rules their

1. Create a tree T with only one node, E (the expression to be narrowed)
2. Let L be a leaf in the narrowing tree generated so far.
   - If L is in *normal form*, label it *success*.
   - For any rule $lhs : cond \rightarrow rhs \in P$ and any $L[u]$ such that $u \in O[L]$: if $L[u]$ is unifiable with lhs with *mgu* σ and the conditional narrowing tree for $(cond)\sigma$ has a path from the root to a leaf node that is the constant *true* with the composition of substitutions along the path being θ, create a child L' of L such that $L' = ((L[u \leftarrow rhs])\sigma)\theta)$, and label the arc between L and L' as $\sigma \circ \theta$ (this is a one step conditional narrowing operation applied at every possible occurrence of L using every possible rule). If there is no such rule, label the node *failure*.
3. Repeat 2 until all leaves marked, or forever!

Figure 2.2: Narrowing Algorithm (Using Conditional Rules)

expressive power.

Narrowing Strategies

A *narrowing strategy* determines which of the redexes of a leaf node in a narrowing tree will be used in a narrowing step. Two obvious strategies are the *innermost* strategy, where an innermost redex is selected, and the *outermost* strategy. In the outermost strategy, let L be a leaf node, and $L[u]$ be the redex selected. Then, it must be the case that there is no redex $L[w]$ of L such that $L[u]$ is a *proper* subterm of $L[w]$ *and* $L[w]$ can participate in a narrowing step. In other words, the inner subterms are evaluated *just enough* so that outer redexes can participate in a narrowing step.

EUnication

The narrowing algorithms in figures 2.1 and 2.2 can also be used to solve equations, an operation called *e-unification*. Suppose for two terms t_1 and t_2, we want to find a substitution σ such that $(t_1)\sigma = (t_2)\sigma$ can be shown to be true using the rewrite rules in a program. To find such a σ, we need a rule $X = X \rightarrow true$ to be part of the program. We then generate the narrowing tree for the expression $t_1 = t_2$. If there is a leaf in this tree which is the constant *true*, the composition of substitutions on the path leading from the root to this success node, say θ, gives us a substitution that is *more general* than σ, i.e., $\sigma = \theta \circ \phi$ for some ϕ [54].

Committing Guards

Committing guards are meaningful only if the narrowing tree for an expression is generated in a *left-to-right*, *depth-first* fashion and the rules in a program have an ordering imposed on them by their physical location in a program. Let E_1 and E_2 be related with the *one step conditional narrowing relation*, E_2 being the child of E_1. If $f(\ldots)\#guard \rightarrow body$ is the rule used in the narrowing tree to get from E_1 to E_2, the semantics of the committing guard dictates that there will be no other child of E_1 to the right of the E_2 that is obtained through the use of another definition of f and the same subterm of E_1 that was used in obtaining E_2. This restriction prunes the full conditional narrowing tree of some of its branches, because the programmer has given further information by using committing guards that those branches are not useful in finding a solution.

The committing guard allows the programmer to specify control information in a similar way to the *cut* operator in Prolog by allowing him to provide information about when remaining rules defining a function need/should not be used once the guard in one rule evaluates to *true*. The effect and convenience of this can be seen in the definition of various functions in the section on examples.

Because of the committing guard, however, it is possible to write programs that are logically wrong, but which operationally exhibit correct behavior. An example of this is given in figure 2.3.

```
smaller(X,Y)# X<Y -> true.
smaller(X,Y) -> false.
```

Figure 2.3: Defining A Logically Wrong Function That Exhibits Correct Behavior

```
not(X)# X -> false.
not(X) -> true.
```

Figure 2.4: Defining The *not* Function Using Committing Guards

Using Conditional Guards to Dene the Function *not*

In *Horn clause logic programming*, since no negative information can be represented [35], negative information has to be deduced from the unprovability of positive information. Thus, if $p(a)$ cannot be shown to be $true$, then it is assumed to be $false$. This has been termed the *negation as (finite) failure* rule. The same idea can be implemented in the context of functional/logic programming using the idea of committing guards. Figure 2.4 depicts a definition of the *not* function using committing guards.

Implementation of ROSE

Prolog has been used as the implementation language of ROSE. The intrinsic unification facilities of Prolog and the fact that all the *data* we are dealing with, including the rules defining functions, can be coded as first order terms, the native data structure of Prolog, have made our job relatively easy. Currently, we have an interpreter for ROSE running on the IBM PC.

Lack of Parsing

Prolog has a facility for defining *functors* to be *infix* as well as assigning them a *precedence* number (in the range 1 to 255), and an *associativity*. The *op* predicate is used for doing this. The lower the precedence number given to a functor, the higher its precedence. This facility has been used to declare the functors `->` , `#` and `:` as infix, as well as others. We thus avoided the need for lexical analysis and parsing the input program file, since our rules are nothing but first order terms where the outermost functor is `->` .

Representation of Functions in the Prolog Database

Each rule, as we mentioned before, belongs to either the public or private section of a module.

To avoid the runtime determination of where the body of a function is kept, each function definition is given a unique name, generated by the system, and function bodies are accessed by using that unique name. This necessitates the *transformation* of all expressions in a rule definition so that the original function names are replaced with the system-generated ones. This transformation is done at *compile time*.

To give an example, assume that the function `f(a):g(a)->h(a)` is declared in the private section of module m, g and h are functions declared in the module n and that m *uses* n. The following *facts* (in addition to others for the definitions of g and h) would be asserted to the database.

```
function(f,m,private,id1).
function(g,n,public,id2).
function(h,n,public,id3).
user_defined(id1,(lambda,[a],id2(a),no_commit,id3(a))).
```

We note here that the function definitions are converted to *lambda expressions*. A lambda expression used for representing the code for a rewrite rule has five parts: the constant *lambda*, a list of formal arguments to the function, the guard, the constant *commit* or *no_commit* depending on whether the rule is committing or non-committing, and then the body of the rule.

This representation brings all functions, no matter where they are defined, into one *level*, although the semantics of the modules and the visibility of functions are preserved.

Another detail we need to consider is that in order to speed up recognition of what kind of a term any term is, terms are labeled with functors such as *is_constructor*, *is_function_complex* in order to avoid searching the database at runtime to determine whether a function symbol is a *function name* or a *constructor*. For example if c is defined to be a *constructor* and f is the name of a function, the term $f(c(a))$ would be represented as

$$is\_function\_complex(f(is\_constructor(c(a)))).$$

The actual representation of the above example would then be

```
function(f,m,private,id1).
function(g,n,public,id2).
function(h,n,public,id3).
user_defined(id1,(lambda,
                  [a],
                  is_function_complex(id2(a)),
                  no_commit,
                  is_function_complex(id3(a)))).
```

Predicates that Do the Narrowing

There are two main predicates,

$$eval(Mode, Expression, Result)$$

and

$$eval3(Mode, LambdaExpression, ActualParameters, Synchronization, Result)$$

that perform narrowing.

eval narrows *Expression* using strategy *Mode* (which is one of the six given below) to give the answer *Result*.

The main function of *eval*3 is to *e-unify* actual parameters with the formal parameters of a function definition, and call *eval* to evaluate the function body. The meaning of the parameters of *eval*3 (called that way due to *historical* reasons: at one point, it had only three arguments!) are as follows. *Mode* has the same meaning as in *eval*. *LambdaExpression* is a lambda expression with five components (as described in the previous section). *ActualParameters* is the actual parameters the lambda expression is to be applied to. *Synchronization* is a unique symbol that is given to both *eval*3 and the predicate that extracts from the database the function definitions. *Result* is a term in normal form that represents the value denoted by the *body* of the function definition. *eval*3, depending upon *Mode*, evaluates the actual arguments, binds the formal parameters to the actuals, calls *eval* to evaluate the condition part of the lambda expression, and calls *eval* again to evaluate the *body* of the lambda expression. If the lambda expression is *committing*, and the condition in the lambda expression evaluates to *true*, *eval*3 asserts the symbol *Synchronization* into the database, which informs the predicate extracting function definitions not to extract any more function definitions for the same function name. Note that *eval* and *eval*3 are mutually recursive.

The available modes for evaluation are:

- Mode=1: *lazy-eager narrow.* For *eval*3: in binding a variable to an expression, if the expression contains as a subterm a function applied to a variable, bind the variable to the unevaluated expression. (Actually, evaluate fully the parts of the expression that do not contain such a function application, but leave only the subterm containing the *risky* function application unevaluated. This implements the *heuristic* that if a function application does not contain a variable, it can possibly be evaluated fully without giving an infinite number of answers as its value.) For *eval*: evaluate *Expression* as much as possible until a term in normal form is reached.

- Mode=2: *eager narrow*. For *eval*3: when binding a variable to an expression, always evaluate the expression fully. For *eval*: same as mode 1.
- Mode=3: *lazy-eager half-way-narrow*. For *eval*3: same as mode 1. For *eval*: stop evaluating *Expression* if a constructor at the outermost level is reached. This helps in terminating the evaluation if the expression is being unified with a constructed term with a different outermost constructor. Otherwise same as mode 1.
- Mode=4: *eager half-way-narrow*. For *eval*3: same as mode 2. For *eval*: stop evaluating *Expression* if a constructor at the outermost level is reached. This helps in terminating the evaluation if the expression is being unified with a constructed term with a different outermost constructor. Otherwise same as mode 2.
- Mode=5: *lazy narrow*. For *eval*3: in binding an actual parameter to a formal parameter, evaluate the actual parameter only as much as needed to make it unifiable with the formal parameter, then unify the two. This implements the *leftmost outermost* narrowing strategy. For *eval*: evaluate *Expression* as much as possible until a term in normal form is reached.
- Mode=6: *lazy half-way-narrow*. For *eval*3, same as mode 5. For *eval*: stop evaluating *Expression* if a constructor at the outermost level is reached. This helps in terminating the evaluation if the expression is being unified with a constructed term with a different outermost constructor. Otherwise same as mode 5.

```
fact(1)# true -> 1.
fact(X) -> X*fact(X-1).
fib(1)# true -> 1.
fib(2)# true ->  1.
fib(N) -> fib(N-1) + fib(N-2).
```

Figure 2.5: Rules For Calculating *factorial* And *fibonacci*

Sample Programs in ROSE

In this section, we give some function definitions to highlight the main features and capabilities of the language. We start out with two simple mathematical functions, then follow on with functions that demonstrate the equation solving capabilities, higher-order features, and lazy evaluation facilities of the language that permit the handling of infinite data structures. (Note that even though in ROSE all functions must belong to some module, here we have left out the declarations for modules for clarity and brevity.)

We have used outermost (lazy) narrowing as the default mode in executing the functions given in the examples. However, no matter which strategy we are using as the default mode, the built-in functions (labels) *lazy* and *eager* allow us to override the default strategy and execute any (sub)expression in a lazy or eager way. $lazy(exp)$ evaluates exp utilizing the outermost narrowing strategy, and $eager(exp)$ evaluates exp utilizing the innermost strategy. eq and $lazy\_eq$ are boolean valued built-in binary functions that e-unify their two arguments, eq utilizing the innermost narrowing strategy, and $lazy\_eq$ utilizing the outermost narrowing strategy.

Figure 2.5 contains the standard definition of the *factorial* and *fibonacci* number functions. Note the notation for variables, which is the same as in Prolog.

```
app([],X) -> X.
app([H|T],L) -> [H|app(T,L)].
reverse([]) ->  [].
reverse([H|T]) ->  app(reverse(T),[H]).
member(X,[X|Y])# true -> true.
member(X,[H|T])# true -> member(X,T).
member(X,Y) -> false.
palindrome(L)# L lazy_eq reverse(L) -> true.
palindrome(L) -> false.
revstrings(A,B) : A lazy_eq app(X,app([H|T],Z))  and
                  B lazy_eq app(XX,app(reverse([H|T]),ZZ))
                 ->  [H|T].
```

Figure 2.6: Some List Functions

```
>> palindrome([1,2,_,3,_,_]).
palindrome([1,2,3,3,2,1]) =>  true
another solution ? y
palindrome([1,2,_62,3,_66,_68]) has no other solution
```

Figure 2.7: Query With Unbound Variables

In figure 2.6 definitions of *append*, *reverse* and *member* are given. *palindrome* and *revstrings* demonstrate the equation solving capability of ROSE. *palindrome* tests to see whether its argument is a palindrome, i.e., if it reads the same both backwards and forwards. However, when given a partially instantiated argument, it can generate the missing values in the sequence given to it. Figure 2.7 shows a terminal session, calling the function palindrome with an argument that is not fully instantiated. (The underscore "_" has the same meaning as in Prolog: a variable whose value will not be used elsewhere.) *revstrings* finds all substrings occurring in its first argument that occur in its second argument in reverse order. A terminal session is given in figure 2.8.

The *foldr* function (adapted from [14]) is defined in figure 2.9 and demonstrates the higher-order capabilities of ROSE. *foldr* takes three arguments: a binary operation OP,

```
>> revstrings([1,2,3,4],[5,2,1,7]).
revstrings([1,2,3,4],[5,2,1,7]) => [1]
another solution ? y
revstrings([1,2,3,4],[5,2,1,7]) => [1,2]
another solution ? y
revstrings([1,2,3,4],[5,2,1,7]) => [2]
another solution ? y
revstrings([1,2,3,4],[5,2,1,7]) has no other solution
```

Figure 2.8: Query For *revstrings*

```
twice(F,A) -> apply(F,apply(F,A)).
map(F,[])# true -> [].
map(F,[H|T]) -> [apply(F,H)|map(F,T)].
foldr(OP,IDENTITY,[])# true -> IDENTITY .
foldr(OP,IDENTITY,[H|T]) -> apply(OP,H,foldr(OP,IDENTITY,T)).
list_append -> foldr(app, []).
list_and -> foldr(and,true).
list_or -> foldr(or,false).
true and true # true -> true.
X and Y -> false.
false or false # true  -> false.
X or Y -> true.
```

Figure 2.9: Higher-Order Functions

the identity element under the operation, and a list of items belonging to the domain of the binary operation.

$$foldr(OP, IDENTITY, [a_1, ..., a_n])$$

returns

$$OP(a_1, OP(a_2, \cdots OP(a_{n-1}, OP(a_n, IDENTITY)) \cdots))$$

foldr is thus implemented as a higher-order function.

Using *foldr* we can define other functions such as *list_append*, *list_and* and *list_or* by instantiating the first two arguments of foldr to a previously defined function, and an identity element under that function. Thus, *list_append* is a function that takes one argument, a

```
>> list_append([[1],[2],[3]] ).
list_append([[1],[2],[3]]) => [1,2,3]
another solution ? y
list_append([[1],[2],[3]]) has no other solution
>> map(twice(app([a])),[[b],[c],[d]]).
map(twice(app([a])),[[b],[c],[d]]) => [[a,a,b],[a,a,c],[a,a,d]]
another solution ? y
map(twice(app([a])),[[b],[c],[d]]) has no other solution
>> list\_append( map( twice(app([a])), [[c],[d,e],[f]] )).
list_append(map(twice(app([a])),[[c],[d,e],[f]])) =>
        [a,a,c,a,a,d,e,a,a,f]
another solution ? y
list_append(map(twice(app([a])),[[c],[d,e],[f]])) has no other solution
```

Figure 2.10: Query For Some Higher-Order Functions

list of lists, and appends all the lists together. *list_and* and *list_or* are defined similarly.

map and *twice* are also higher-order functions: *map* applies its first argument to all the elements of its second argument, and *twice* applies its first argument to its second argument twice. Figure 2.10 depicts some function calls using interesting combinations of the above functions.

Note that when a function is to be passed as an argument, if that function has more than one defining rule, lazy narrowing must be used, so that the name of the function, and not the individual rewrite rules defining the function, are passed into the formal arguments.

Finally, we give the definitions of functions for generating the list of *lucky* numbers (again adapted from [14]) which demonstrates the use of infinite data structures. The lucky numbers are generated as follows: We start out with 1,<u>3</u>,5,7,9,11,13,15,17,19,21,... and remove from the list every third item, which gives us 1,3,<u>7</u>,9,13,15,19,21,... Now, we remove from the resulting list every seventh item, and keep going. The numbers remaining in this sequence are the *lucky* numbers. Obviously, the list of lucky numbers is an infinite list. Figure 2.11 contains an implementation of the lucky numbers function in ROSE, using

```
lucky -> [1|lucky2(2,odds(1))] .
odds(N) -> [N|odds(eager(N+2))].
lucky2(N,List): Y eq ith(N, lazy List) ->
                [Y|lucky2(eager(N+1),knock_out(1,Y,List))].
ith(1,[H|T])# true -> H.
ith(N,[H|T]) -> ith(N-1,lazy T).
knock_out(X,Y,[H|T])# X eq Y -> knock_out(1,Y,T).
knock_out(X,Y,[H|T]) -> [H|knock_out(eager(X+1),Y,T)].
first_n_items(0,[H|T])# true -> [].
first_n_items(N,[H|T]) -> [H|first_n_items(eager(N-1),T)].
```

Figure 2.11: Function Definitions Demonstrating Infinite Lists

```
>> first_n_items(3,lucky).
first_n_items(3,lucky) =>  [1,3,7]
another solution ? y
first_n_items(3,lucky) has no other solution
```

Figure 2.12: Query For Generating The First Three *lucky* Numbers

an infinite list to hold the resulting numbers. Note that we have used the built in function *eager* to evaluate eagerly what does not need to be evaluated lazily, thus saving redundant recomputation caused by lazy evaluation. Figure 2.12 depicts a query for generating the first three lucky numbers.

Related Work

IDEAL, described in [14], is a language which combines type checking, higher-order objects, lazy evaluation, function invertibility, and non-determinism. However, no mention is made of equation solving, and the language adopted is much more complicated than conditional rewrite rules with committing guards.

In the language described in [24], infinite data structures are available not as a result of the innate execution mechanism, but by modifying the programs in a not-so-obvious way.

SLOG, described in [26], employs innermost superposition as its operational semantics and does not have any notion of infinite data structures or higher-order functions.

K-LEAF, described in [34], is a language based on Horn Clause Logic with equality, and also has no notion of higher-order functions.

The language described in [47] tries to unify the two paradigms at a semantic level using *domain theory* as a common basis for functional and logic programming and uses set abstraction in a predominantly functional language to provide logic programming capability.

The field of functional/logic programming has reached a certain maturation point, and there is indeed quite a collection functional/logic programming languages already in existence, each with a different subset of features associated with functional and logic programming. We could not hope to list all of them here. One could then ask, "What *else* can be said in this mature field that has not already been said?" Well, we argue that ROSE offers something *unique* in combining *committing* guards *and* non-determinism in rewrite rules to bring functional/logic programming to a *practical* level. This is a feature not present in any of the other functional/logic languages that we are aware of, and one we believe is crucial in making functional/logic programming appealing and practical.

Conclusion

We presented in this chapter a higher-order functional/logic programming language, ROSE, that permits committing guards in the conditional rewrite rules making up its programs. It also allows higher-order features in the rules. The operational semantics, which is conditional narrowing augmented to deal with committing guards, can be done *lazily* using the *outermost* narrowing strategy, or *eagerly* using the *innermost* narrowing strategy.

The major innovation in ROSE is the way committing guards in the rules are combined with backtracking (due to non-determinism) to permit very practical and concise programs to be written. No other functional/logic programming language that we are aware of has this feature. The practicality of combining committing guards with backtracking can be seen in the definition of the extra-logical function *not*, which implements the *negation as failure* rule in the context of functional/logic programming. So before the final word is said in the area of functional/logic programming, we propose ROSE as a demonstration of the fact that functional/logic programming can be *practical* and does not need to be confined to proving the workability of some theoretical idea.

CHAPTER

A DEDUCTIVE DATABASE MODEL BASED ON CONDITIONAL TERM REWRITING SYSTEMS

In this chapter, we describe a functional/logic deductive database model based on conditional term rewriting systems for defining logic databases with functional syntax. In this approach, we use conditional rewrite rules as a database language, and develop a bottom-up evaluation algorithm that utilizes narrowing for evaluating queries. We demonstrate the soundness and completeness of this query evaluation algorithm with respect to a more standard reduction semantics, and prove that it terminates. The model we develop subsumes the logic data model, with Datalog programs making up the database, as well as the functional data model [46]. The advantages of our model over Datalog stem mainly from the fact that our model makes available within a unified framework both relational and functional styles of programming. The availability of the functional style of programming, in addition to the relational style, can be used to good advantage to write programs that have a less ambiguous declarative reading than their purely logical counterparts, to more authentically and naturally represent data (some data can best be represented using a functional notation rather than relational) , and pose queries more intuitively through the use of function nesting.

In our method, a functional/logic program, composed of conditional rewrite rules, can be seen as defining a deductive database. We are then in effect adapting functional/logic programming to the area of deductive databases, much as logic-based deductive database languages are an adaptation of logic programming.

The bottom-up query evaluation algorithm which implements conditional narrowing is guaranteed to terminate due to the choice of rules used at every narrowing step.

The logic data model, as defined by *Datalog*, by and large is the foundation on which deductive databases are built. This model is described in detail in [53], and various aspects of it are given in-depth treatment in [18, 29, 41]. The model we are defining, which we shall call DataFunLog (DFL), subsumes the logic data model as defined by Datalog. We shall demonstrate this by giving a simple translation scheme from Datalog to DFL which translates clauses to boolean valued functions, and prove that the translated Datalog programs have the same meaning as the original Datalog program, i.e., given a query, they both generate the same set of answers. We prove this for programs without negation, but the same result can be shown for programs with negation in a similar fashion.

We note however that DFL programs are not restricted to boolean valued functions, and that we can define functions that map an arbitrary set of arguments to any set of values. This gives us much more flexibility than in Datalog in that we can use either relational style of programming by writing boolean valued functions, or functional style of programming by writing non-boolean valued functions, depending upon the nature of the problem. The availability of function nesting in DFL removes the need for extraneous logical variables under certain conditions, and leads to more natural queries, and more easily understandable programs (in fact the use of function nesting is an advantage of functional programming over logic programming in general, and DFL inherits this advantage). As an example, consider the following set of facts/rules in *Datalog* that represent a certain family relationship.

parent(john,jack).
parent(jack,mary).
grandparent(X,Y) $\leftarrow$ parent(X,Z) & parent(Z,Y).

Note the extra variable "Z" that we had to introduce in defining the clause for "grandparent," since our basic mode of thinking is relational. Also it is not at all obvious how the predicates are to be read: does "parent(john,jack)" mean that "john" is the parent of "jack," or vice versa, i.e., "jack" is the parent of "john" ?

A corresponding DFL program to the above Datalog program *could* be:

parent(john,jack) $\rightarrow$ true .
parent(jack,mary) $\rightarrow$ true .
grandparent(X,Y) $\rightarrow$ parent(X,Z) and parent(Z,Y).

This program would generate the same answers as the corresponding Datalog program for any query.

Contrast, however, the above DFL rules defining the database with the equivalent DFL program below, written with a functional style. This program does not exhibit the weaknesses of the relational style shown above. The variable "Z" does not appear anywhere, and the grandparent of any person "X" is obviously the parent of a parent of "X," captured beautifully with function nesting.

parent(john) $\rightarrow$ jack.
parent(jack) $\rightarrow$ mary.
grandparent(X) $\rightarrow$ parent(parent(X)).

The equality predicate "=" is a built in operator in DFL (more details on built-in operators, or functions, will be given in subsequent sections), which permits us to have queries of the form "grandparent(Y) = mary," for example, and we would get the answer "john" for the unbound variable Y in the query. The corresponding query for the DFL program using the relational style would be "grandparent(Y,mary)," which would give us the same answer for the unbound variable "Y." Another query might ask about the great grandparents of "john," which would simply be "parent(grandparent(john))." In the relational style, we would have to provide a variable to hold the result, and a *temporary* variable (Y), whose value we don't even care about, in order to formulate the same query: "grandparent(john,Y) and parent(Y,Z)."

The above small example and the arguments presented are intended to emphasize the point that in some cases the functional style of programming leads to a more natural representation of data, easier to read programs, and more intuitive queries than the relational style of programming. This is the main contribution of DFL in that it makes *both* styles of programming available in one unified framework *in the context of databases* (the advantages of the integration of these two paradigm have already been realized in the context of programming languages, as stated before).

The remainder of this chapter is organized as follows.

In the next section, we develop the *DataFunLog* (DFL) model which permits the definition of a database with conditional rewrite rules. An algorithm for evaluating queries is presented. We also give a reduction semantics for DFL programs, and show the correctness and completeness of the query evaluation algorithm relative to this semantics.

In section 3.2 we extend our rules to permit the "not" function to appear anywhere within the body or the condition part of a rule. We call the resulting language DFLN, *"DataFunLog with Negation."* We introduce the notion of a *stratified* database, explain why stratification is necessary in the presence of negation, and describe an extended bottom-up evaluation algorithm to handle negation for stratified DFLN programs.

Section 3.3 describes other approaches to database languages, and compares them to DFLN. The last section is the summary, conclusion and future research directions.

The *DataFunLog* DFL Data Model

Our data model is based on conditional term rewriting systems. Since we are dealing with atomic data, and following the lead of *Datalog*, we shall not allow constructors in our rules: just variables and constants. A rule will, in general, be of the form

lhs : condition $\rightarrow$ rhs

with the following restrictions:

1. All variables appearing in the *lhs* must appear in the *rhs* or in the *condition*.
2. All variables must be *restricted*: i.e. they must be an argument to a function call in the *condition* or in the *rhs*. Thus, "f(X) $\rightarrow$ X" is not allowed.
3. *lhs* can have no function applications in argument positions: only variables and constants. Thus "f(g(a),X) $\rightarrow$ rhs" is not allowed.
4. All variables occurring in the condition part must be arguments to some function: i.e., "f(...):X $\rightarrow$ rhs" is not allowed.

These restrictions on rewrite rules are necessary for the bottom-up query evaluation algorithm to terminate and have other properties that are described in later sections.

To demonstrate the elegance and power of the proposed data model, we give the following example of a *DataFunLog with Negation* (DFLN) program. Note the use of *conditional* rewrite rules, as opposed to *unconditional* ones. We also give the intuitive meaning of the rules defining the program.

The following rules define certain family relationships:

male(joe)→true.
female(mary)→true.
parent(tom)→joe.
brother(X):parent(X) = parent(Y) and male(Y) and not(X=Y) → Y.
father(X):Y=parent(X) and male(Y)→Y.

The rules defining "=" and the logical connectives are assumed. This is what makes them "built-in," as we mentioned before: the rules defining them are included by default in the database. We give the precise rules defining them and other logical "functions" later in the presentation.

The functions "male" and "female" are boolean valued functions. The "parent" function is straightforward, where "parent(X)→ Y" means "the parent of X is Y." The way we read the last two rules are as follows: "For all *constants* X and Y, the brother of X is Y if the parents of X and Y are the same, Y is male and Y is not equal to X" and " the father of X is Y if Y is the parent of X and Y is male."

Given these rules defining the database, we can, for example, ask queries of the form "Whose father is *joe*" by "father(X) = joe" where X is a logical variable.

We have developed an algorithm using bottom-up evaluation for answering such queries. For the case involving negation, we have developed a stratification algorithm and an algorithm for evaluating DFLN queries which makes use of the bottom-up evaluation algorithm developed for evaluating DFL queries.

The Meaning of DataFunLog Programs Without Negation

We partition all symbols used in a DFL program into three sets: The set of function names F, the set of variables V and the set of atomic objects (also called *constants*, or *data objects*) DOM.

We define a *term* recursively as:

- A variable is a term
- A constant is a term
- If f is a function name, and $a_i, 1 \leq i \leq n$, are terms, then $f(a_1, a_2, \cdots, a_n)$ is a term

A constant is also called a term in *normal form.*

An *occurrence* is a string of integers of the form $i.j.k\ldots$, where $1 \leq i, j, k \ldots$ We need the notion of occurrences to give an unambiguous address for a particular subterm of a term. If t is a term, we denote the subterm of t at occurrence u as t[u]. A few examples will serve best to describe the meaning of occurrences.

f(g(a,b),c)[1] = f(g(a,b),c)
f(g(a,b),c)[1.1] = g(a,b)
f(g(a,b),c)[1.1.2] = b
f(g(a,b),c)[1.2] = c

If u is pointing to a subterm of t that does not exist, we define $t[u]$ to be t itself. For example,

f(g(a,b),c)[1.2.4.2]=f(g(a,b),c).

Obtaining a subterm of a term by the above process is called an *extraction* operation. Replacing a subterm s of a term t at some occurrence u by another term w is called a *replacement* operation and is denoted by $t[u \leftarrow w]$.

Conditional rewrite rules, in the presence of *confluence*, have a natural declarative reading, in addition to a procedural one, whereby a conditional rewrite rule can be seen as conditional equality between two terms. However, we cannot use this approach to give a declarative reading to DataFunLog programs, since by necessity, we do not have confluence in the rewrite rules. For example, with a database consisting of the following rules,

f(a)→b.
f(a)→c.

where a, b and c are terms in normal form, if we interpreted the rules to mean (unconditional) equality, we would be able to prove that b and c are equal, an obvious falsity (note that the unconditional rule " f(a)$\rightarrow$b" is the same as the conditional rule "f(a):true$\rightarrow$b").

Instead, the declarative reading of conditional rewrite rules will be that they define set-valued functions. In the above example, f is a (partial) function that maps the constant "a" to the set of constants {b,c}. Answering a query which possibly can contain variables means finding the set of values denoted by the expression that makes up the query, as well as the values for the variables in the query for which the expression denotes this set of values.

In order to give meaning to rules, we need to give a meaning to expressions. Below, we define what an expression denotes, and then what each rule means. First, we define some more terms.

Let σ be a substitution. We say that if all substituted terms in σ are ground, then σ is a *ground substitution.* If all the substituted terms are in normal form, then we say σ is a *normal substitution.* We can combine these different qualifiers to describe a particular substitution.

We define R to be the set of rules that make up a DFL program, and R' to be the set {(rule)σ | rule $\in$ R and σ is a ground normal substitution and (rule)σ is ground}.

We define the *one step reduction relation*, $\Rightarrow$, as: $a \Rightarrow a[u \leftarrow (rhs)\sigma]$ iff there exists a subterm of a at occurrence u ($a[u]$), a rule $lhs \rightarrow rhs$ in the rule set, and a substitution σ such that $a[u] = (lhs)\sigma$.

We define $\Rightarrow^+$ as: $a \Rightarrow^+ b$ iff $a \Rightarrow b$ or $a \Rightarrow c$ for some term c and $c \Rightarrow^+ b$.

We define $\Rightarrow^*$ as: $a \Rightarrow^* b$ iff $a = b$ or $a \Rightarrow^+ b$.

We define the *one step conditional reduction*, $\Longrightarrow$, and its transitive closure $\Longrightarrow^*$, recursively as:

- $a \Longrightarrow a[u \leftarrow (rhs)\sigma]$ iff there exists a subterm of a at occurrence u ($a[u]$), a rule $lhs : cond \rightarrow rhs$ in the rule set, a substitution σ such that $a[u] = (lhs)\sigma$ and $(cond)\sigma \Longrightarrow^* true$,
- $c \Longrightarrow^* d$ iff $c = d$ or $c \Longrightarrow e$ for some term e and $e \Longrightarrow^* d$.

We define $\Longrightarrow^+$ as: $a \Longrightarrow^+ b$ iff $a \Longrightarrow c$ for some term c and $c \Longrightarrow^* b$.

The meaning of any ground expression exp, $Den(exp)$, is the set of constants (terms in normal form) exp can be *(conditionally) reduced* to using ONLY the rules in R', i.e.,

$$Den(exp) = \{\ a \mid exp \Longrightarrow^* a, \text{ and } a \text{ is a constant}\}.$$

The meaning of any non-ground expression exp is

$$\bigcup_\sigma Den((exp)\sigma)$$

where σ is a ground normal substitution and $(exp)\sigma$ is ground.

We can now give meaning to the rules in a DFL program: If $lhs : cond \rightarrow rhs$ is a rule, for any ground normal substitution σ that makes the rule ground, true $\in Den((cond)\sigma)$ implies $Den((rhs)\sigma) \subseteq Den((lhs)\sigma)$.

Alternatively, we can see rules as functions mapping substitutions to sets of constants. For this interpretation, we would need to standardize, or *rectify* the rules in the database. The process of *rectification* proceeds in the following manner. For any function name f, let

$$f(a_1, \ldots, a_n) : cond \rightarrow rhs$$

be one of the rules defining f. Replace this rule with

$$f(V_1, \ldots, V_n) : V_1 = a_1 \ and \ V_2 = a_2 \ and \ \ldots and \ V_n = a_n \ and \ cond \rightarrow rhs$$

where $V_1, V_2, \ldots, V_n$ are new variables not occurring in *any* defining rule of f. Repeat the same procedure for all defining rules of f. Note that the new rules defining the function f will now have the same variables in their i^{th} argument positions. Then, for any given ground normal substitution σ that makes all the rules defining f ground, we can say f maps σ to $Den((lhs)\sigma)$.

An Algorithm for Computing the Answer to a Given Query

We assume that every database includes, by default, the set of rules $\{x = x \rightarrow true \mid x \in DOM\}$, as well as the rules

true and true $\rightarrow$ true.	false or false $\rightarrow$ false.
true and false $\rightarrow$ false.	true or false $\rightarrow$ true.
false and true $\rightarrow$ false.	false or true $\rightarrow$ true.
false and false $\rightarrow$ false.	true or true $\rightarrow$ true.

Note that we have adopted an infix notation for the connectives "and" and "or," as well as the "=" function. The precedence of these operators, from highest to lowest, are: =, *and*, *or*.

In figure 3.1 we give the algorithm for computing the answer to a given query Q. Given a query Q, we generate a temporary rule of the form "$answer(Q) \rightarrow Q$," and add it to the set of rules defining the database. F is the set of function names in the database (as we mentioned before) and R is the database itself made up of (conditional) rewrite rules. R also contains the rule for "answer" generated by the query.

We can view the execution of algorithm in figure 3.1 as building a collection of trees in each "raw" set. The roots of trees in a raw set are those rules placed in the raw set in step 1 of the algorithm. Each time a new rule is added to a raw set, we can view the new rule as being a node joined to a tree in the raw set as the child of the "operated upon" rule. If a node (rule) is a leaf at the termination of the algorithm and its right hand side is a

1. For every function name f in F, create two sets: call the first set raw_f, and the second set $success_f$. Initialize: $success_f = \emptyset$ and $raw_f =$ all rules defining f.

2. Repeat steps 2a through 2d until neither of the sets $success_f$ and raw_f change any more (f is any function name occurring in the program)

   (a) Let $f(a_1, a_2, \ldots, a_n) : Cond \rightarrow rhs \in raw_f$. Call this rule the "operated upon" rule. If $Cond =$ "true" then add $f(a_1, a_2, \ldots, a_n) \rightarrow rhs$ to raw_f.

   (b) Let $f(a_1, a_2, \ldots, a_n) : Cond \rightarrow rhs \in raw_f$. Call this rule the "operated upon" rule. Let $Cond[u]$ be a nonvariable subterm of $Cond$ at occurrence u. Let $g(b_1, b_2, \ldots, b_m) \rightarrow some\_constant \in success_g$ for some function name g. Let $g(b_1, b_2, \ldots, b_m)$ be unifiable with $Cond[u]$ with most general unifier σ. Let $Cond' = Cond[u \leftarrow some\_constant]$, i.e., replace the subterm of $Cond$ at occurrence u with $some\_constant$. Add the rule $f(a_1, a_2, \ldots, a_n)\sigma : (Cond')\sigma \rightarrow (rhs)\sigma$ to raw_f. This operation is called a *narrowing operation on a condition.*

   (c) Let $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in raw_f$. Call this rule the "operated upon" rule. Let $rhs[u]$ be a nonvariable subterm of rhs at occurrence u. Let $g(b_1, b_2, \ldots, b_m) \rightarrow some\_constant \in success_g$ for some function name g. Let $g(b_1, b_2, \ldots, b_m)$ be unifiable with $rhs[u]$ with most general unifier σ. Let $rhs' = rhs[u \leftarrow some\_constant]$, i.e., replace the subterm of rhs at occurrence u with $some\_constant$. Add the rule $f(a_1, a_2, \ldots, a_n)\sigma \rightarrow (rhs')\sigma$ to raw_f. This operation is called a *narrowing operation on the right hand side.*

   (d) Let $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in raw_f$. If rhs is a constant, then add $f(a_1, a_2, \ldots, a_n) \rightarrow rhs$ to $success_f$

3. The answer is the set $success_{answer}$.

Figure 3.1: Algorithm for Computing the Rule Set to Answer a Query

constant and it has no condition part, a copy of it will exist in a success set (placed there by step (2d) of the algorithm).

Examples

We give two examples demonstrating the execution of the algorithm in figure 3.1.

EXAMPLE 1: Assume the database is made up of the rules given below. The query is "f(Z)." We show how the answer to this query is computed (using the algorithm in 3.1).

f(X) $\rightarrow$ h(g(X)).
g(a) $\rightarrow$ b.
h(b) $\rightarrow$ c.
answer(f(Z)) $\rightarrow$ f(Z).

We give the sets at each iteration as they are computed. The domain. DOM, is the set $\{a, b, c, true, false\}$.

Initialize:

$success_{answer} := success_f := success_g := success_h := success_{and}$
$:= success_{or} := success_= := \emptyset$.
$raw_f := \{f(X) \rightarrow h(g(X))\}$.
$raw_g := \{g(a) \rightarrow b\}$.
$raw_h := \{h(b) \rightarrow c\}$.
$raw_{answer} := \{answer(f(Z)) \rightarrow f(Z)\}$.
raw_{and} :={rules defining "and"}.
raw_{or} :={rules defining "or"}.
$raw_=$:={rules defining "="}.

Iteration 1:

$success_g := success_g \bigcup \{g(a) \rightarrow b\}$.
$success_h := success_h \bigcup \{h(b) \rightarrow c\}$.
$success_{and} := success_{and} \bigcup raw_{and}$.
$success_{or} := success_{or} \bigcup raw_{or}$.
$success_= := success_= \bigcup raw_=$.

Note that Iteration 1 places in the success sets the actual data from the "extensional" database, i.e., the part of the database consisting of rules of the form $f(\ldots) \rightarrow rhs$, where f is any function name, and rhs is any constant.

Iteration 2:

$raw_f := raw_f \bigcup \{f(a) \rightarrow h(b)\}$.

Iteration 3:

$raw_f := raw_f \bigcup \{f(a) \rightarrow c\}$.

Iteration 4:

$success_f := success_f \bigcup \{f(a) \rightarrow c\}$.

Iteration 5:

$raw_{answer} := raw_{answer} \bigcup \{answer(f(a)) \rightarrow c\}$.

Iteration 6:

$success_{answer} := success_{answer} \bigcup \{answer(f(a)) \rightarrow c\}$.

At this point, since none of the sets can change any further, we take the contents of $success_{answer}$ to be the answer to the query, in this case $\{answer(f(a)) \rightarrow c\}$.

EXAMPLE 2: We give another sample database below. Note that the query is "f(g(Z),W)," which caused the insertion of the rule "answer(f(g(Z),W)) → f(g(Z),W)" into the database.

f(X,Y): g(X)=h(Y) → g(Y).
g(a) → b.
g(e) → a.
g(a) → c.
h(e) → b
answer(f(g(Z),W)) → f(g(Z),W).

Initialize:

$success_{answer} := success_f := success_g := success_h := success_{and} := success_{or} :=$

$success_= := \emptyset$.

$raw_f :=$ { f(X,Y) : g(X)=h(Y) → g(Y) }.
$raw_g :=$ { g(a) → b, g(a) → c, g(e) → a }.
$raw_h :=$ { h(e) → b }.
$raw_{answer} :=$ { answer(f(g(Z),W)) → f(g(Z),W) }.
$raw_{and} :=$ { rules defining "and" }.
$raw_{or} :=$ { rules defining "or" }.
$raw_= :=$ { rules defining "=" }.

Iteration 1:

$success_g := success_g \bigcup$ { g(a) → b, g(a) → c, g(e) → a }.
$success_h := success_h \bigcup$ { h(e) → b }.
$success_{and} := success_{and} \bigcup raw_{and}$.
$success_{or} := success_{or} \bigcup raw_{or}$.
$success_= := success_= \bigcup raw_=$.

Iteration 2:

$raw_f := raw_f \bigcup \{$ f(a,Y): b=h(Y) $\rightarrow$ g(Y), f(a,Y): c=h(Y) $\rightarrow$ g(Y), f(e,Y) : a=h(Y) $\rightarrow$ g(Y), f(X,e) : g(X)=b $\rightarrow$ g(e) $\}$.

$raw_{answer} := raw_{answer} \bigcup \{$ answer(f(g(a),W)) $\rightarrow$ f(b,W), answer(f(g(a),W)) $\rightarrow$ f(c,W), answer(f(g(e),W)) $\rightarrow$ f(a,W) $\}$.

Iteration 3:

$raw_f := raw_f \bigcup \{$ f(a,e) : b=b $\rightarrow$ g(e), f(a,e) : c=b $\rightarrow$ g(e), f(e,e) : a=b $\rightarrow$ g(e) $\}$.

Iteration 4:

$raw_f := raw_f \bigcup \{$ f(a,e) : true $\rightarrow$ g(e) $\}$.

Iteration 5:

$raw_f := raw_f \bigcup \{$ f(a,e) $\rightarrow$ g(e) $\}$.

Iteration 6:

$raw_f := raw_f \bigcup \{$ f(a,e) $\rightarrow$ a $\}$.

Iteration 7:

$success_f := success_f \bigcup \{$ f(a,e) $\rightarrow$ a $\}$.

Iteration 8:

$raw_{answer} := raw_{answer} \bigcup \{$ answer(f(g(e),e) $\rightarrow$ a $\}$.

Iteration 9:

$success_{answer} := success_{answer} \bigcup \{$ answer(f(g(e),e) $\rightarrow$ a $\}$.

Since no more changes to any sets are possible, the contents of $success_{answer}$, which is $\{$ answer(f(g(e),e)) $\rightarrow$ a $\}$, gives us the required answer to the query.

Termination of the Query Evaluation Algorithm and the Na ture of the Computed Answer

The following lemma is needed to show the termination properties of the algorithm in figure 3.1.

LEMMA 1: Let f be any function name. If $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in success_f$ at any stage of the execution of the algorithm, then $a_1, a_2, \ldots, a_n$ are all constants, and so is rhs.

Proof: We shall prove the above assertion by induction on the the order in which rules are copied into success sets from raw sets, i.e., we assume that elements in raw sets that are copied to success sets by step (2d) of the algorithm are done so in a strictly sequential fashion, so that we can determine "which rule was copied first, which second, etc." This does not affect the meaning of the algorithm.

Basis: Let "lhs $\rightarrow$ rhs" be the first rule to be copied. This rule must have been copied to a success set either at the first iteration, through step (2d), or at the second iteration through the application of step (2a) at iteration 1 and step (2d) at iteration 2. In both cases the assertion holds because of the restrictions placed on the rules (please refer to section 3.1).

Inductive Hypothesis: Assume that if a rule is the N^{th} rule to be copied to a success set, or any rule that was placed before, then it satisfies the assertion. We need to prove that the $(N+1)^{th}$ rule to be added to a success set also satisfies the assertion.

Note that in the process of adding the first N rules to success sets, we have generated a certain collection of forests in the raw sets that represent the computation done so far. In generating this collection, only the first N members, which we are assuming satisfy the assertion, of the success sets (as a whole) were utilized in the narrowing operations of steps (2b) and (2c) of the algorithm. This means that every rule added to any success or raw set by the algorithm obeys the conditions imposed on rules that make up the database (described in section 3.1). Since every rule that is transferred to a success set must have rhs constant, the assertion follows. □

THEOREM 1 (Termination): The algorithm in figure 3.1 terminates.

Proof: Because DOM is finite, the total number of rules in success sets is finite. To see that the raw sets cannot grow indefinitely, note that each raw set contains a set of narrowing trees (although these trees are not *explicitly* built by the algorithm), whose nodes consist of rewrite rules. Due to lemma 1, each new leaf node that is added to any tree through steps

(2b) or (2c) contains one less function name than its parent node (i.e., the "operated-upon" rule). If a rule is added to a tree set through the application of step (2a) then its parent node is a conditional rule and the newly added leaf is unconditional. Under these conditions, it is obvious that no tree can grow indefinitely. Hence no raw set can grow indefinitely, and the algorithm stops when none of the success or raw sets can grow any more. □

THEOREM 2 (Nature of the Computed Answer): Let Q be a query, possibly containing variables. Let "$answer(Q)\sigma \rightarrow rhs$" $\in success_{answer}$ at the termination of the algorithm in figure 3.1. Then $(Q)\sigma$ is ground, rhs is a constant, and all variables in Q have been replaced by constants (terms in normal form).

Proof: The only way in which "$answer(Q) \rightarrow rhs$" is different from any other rule is that it contains nested functions in the left hand side. However, "answer" is only a label, and not a real function name (we are assuming that it is illegal to define a function called "answer"). Hence, no element of $success_{answer}$ is ever used in steps (2b) and (2c) of the algorithm. We can thus replace $answer(Q) \rightarrow Q$ with $answer(V_1, V_2, \ldots, V_n) \rightarrow Q$ where $V_1, V_2, \ldots, V_n$ are the variables occurring in Q, in the order in which they appear in Q. This rule is now no different form any other rule in the database and lemma 1 applies. □

Correctness and Completeness of the Query Evaluation Algorithm

In this section, we give correctness and completeness results for the query evaluation algorithm for DFL programs with respect to reduction semantics. First we start out with the non-conditional case, develop a proof strategy, and use this strategy as a model for proving similar results for the conditional case.

The set R, we remember, is the set of rules defining a database, and R'=$\{$(rule)σ $|$ rule $\in$ R and σ is a ground normal substitution and (rule)σ is ground$\}$. These sets are referred to frequently in the discussions that follow.

Databases Consisting of Unconditional Rules

The following lemma is used in showing the correctness (with respect to reduction semantics) of the computed result for databases consisting of unconditional rules.

LEMMA 2: Given a DFL program R which contains only unconditional rules, for any function name f in R, $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in success_f$ at the termination of the algorithm in figure 3.1 implies $f(a_1, a_2, \ldots, a_n)) \Rightarrow^* rhs$ using only the rules in R'.

Proof: We shall prove the assertion by induction on the order in which a rule is placed in a success set.

Basis: Assume the rule $lhs \rightarrow rhs$ is the first one to be placed in a success set. Then rhs is a constant, and lhs is ground (due to the restrictions on rules). Since $lhs \rightarrow rhs$ is the very first rule to be placed in any success set, $lhs \rightarrow rhs$ must also be in R, and also in R'. Hence $lhs \Rightarrow rhs$.

Inductive Hypothesis: Assume that if a rule $lhs \rightarrow rhs$ is the N^{th} rule to be placed in a success set, or is placed before the N^{th} rule, then $lhs \Rightarrow^* rhs$ using only the rules in R'. We need to show that if a rule $lhs1 \rightarrow rhs1$ is placed in a success set as the $(N+1)^{th}$ rule, then $lhs1 \Rightarrow^* rhs1$ using only the rules in R'.

How did we end up placing $lhs1 \rightarrow rhs1$ in a raw set? We started out with a rule of the form $lhs2 \rightarrow rhs2$ in the database as the root of a narrowing tree, and narrowed $rhs2$ to $rhs1$, with σ being the composition of substitutions generated along the way. Thus $lhs1 = (lhs2)\sigma$. The set of rules we used in the narrowing operations was the first N

elements of the success sets combined. Given the properties of this set of rules, we can show by induction on the number of narrowing steps performed that $(rhs2)\sigma \Rightarrow^* rhs1$ using this set of first N rules in success sets. But we can take such a reduction sequence, and expand every step of it so that we are using only rules from R', because according to the inductive hypothesis, for every one of the first N rules $lhs \rightarrow rhs$, there is a reduction sequence from lhs to rhs using only the rules in R'. For example, if $a \Rightarrow a[u \leftarrow r]$ is part of the reduction sequence using the rule, say "$l \rightarrow r$" in some success set, we know that $l \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \ldots \Rightarrow t_n \Rightarrow r$ using only the rules in R' (by the inductive hypothesis). So we would replace $a \Rightarrow a[u \leftarrow r]$ in the original reduction sequence with $a \Rightarrow a[u \leftarrow t_1] \Rightarrow a[u \leftarrow t_2] \Rightarrow \ldots \Rightarrow a[u \leftarrow t_n] \Rightarrow a[u \leftarrow r]$, which uses only rules in R'. We can repeat this process for all parts of the original reduction sequence, which gives us a reduction sequence using only the rules in R'. We conclude that $(rhs2)\sigma \Rightarrow^* rhs1$ using only the rules from R'. This means $(rhs2)\sigma$ is ground since all the rules in R' are ground. But $(lhs2)\sigma$ is ground also (since it is equal to $lhs1$, which by lemma 1 is ground), which implies $(lhs2)\sigma \rightarrow (rhs2)\sigma \in$ R' (remember that $lhs2 \rightarrow rhs2 \in R$). Hence $(lhs2)\sigma \Rightarrow (rhs2)\sigma$ using $(lhs2)\sigma \rightarrow (rhs2)\sigma$. Since we already showed that $(rhs2)\sigma \Rightarrow^* rhs1$ using only rules from R', we have $(lhs2)\sigma \Rightarrow (rhs2)\sigma \Rightarrow^* rhs1$, or equivalently $lhs1 \Rightarrow (rhs2)\sigma \Rightarrow^* rhs1$ (since $((lhs2)\sigma = lhs1)$ using only rules in R'. $\Box$

Lemma 3 will be useful in showing the completeness (with respect to reduction semantics) of the computed result for databases consisting of unconditional rules.

LEMMA 3: Given a DFL program R which contains only unconditional rules, for any function name f in P, $f(a_1, a_2, \ldots, a_n) \Rightarrow^* rhs$, where $a_1, a_2, \ldots, a_n$ are all constants, rhs is a constant, and using only the rules in R', implies $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in success_f$ at the termination of the algorithm in figure 3.1.

Proof: We shall prove the assertion by induction on the number of reduction steps, N, which takes us from $f(a_1, a_2, \ldots, a_n)$ to rhs.

Basis: N=1. There must be a rule $f(a_1, a_2, \ldots, a_n) \rightarrow rhs$ in R'. But this same rule must be in R also due to the restrictions on the rules. If it is in R, then it is also in raw_f as the root of a tree. Since rhs is a constant, this rule would be copied to $success_f$ in the first iteration.

Inductive Hypothesis: Assume the assertion is true for N=n. Prove the assertion for N=n+1. Let $f(a_1, a_2, \ldots, a_n) \Rightarrow exp \Rightarrow^* rhs$ be a reduction sequence of length n+1. Note that there must be a rule $f(a_1, a_2, \ldots, a_n) \rightarrow exp$ in R', since $a_1, a_2, \ldots, a_n$ are all constants. Without loss of generality, and in order to more concisely explain some ideas, assume exp is of the form $g(h(a), q(c))$, where g, h and q are function names and a and c are constants. In order that $g(h(a), q(c)) \Rightarrow^* rhs$ in n reduction steps using only rules in R', $h(a) \Rightarrow^* d$ for some constant d in less than $n+1$ reduction steps, $q(c) \Rightarrow^* e$ for some constant e in less than $n+1$ reduction steps, and $g(d, e) \Rightarrow^* rhs$ in less than $n+1$ reduction steps. Now, by the inductive hypothesis, $h(a) \rightarrow d \in success_h$, $q(c) \rightarrow e \in success_q$ and $g(d, e) \rightarrow rhs \in success_g$. Note that we now can have the reduction $g(h(a), q(c)) \Rightarrow_{h(a) \rightarrow d}$ $g(d, q(c)) \Rightarrow_{q(c) \rightarrow e} g(d, e) \Rightarrow_{g(d,e) \rightarrow rhs} rhs$ ($exp_1 \Rightarrow_{a \rightarrow b} exp_2$ means there is a reduction from exp_1 to exp_2 using the rule $a \rightarrow b$). Thus there is a reduction from exp to rhs using the above rules in the success sets of g, h and q. But since $f(a_1, a_2, \ldots, a_n) \rightarrow exp$ $\in$ R', there must be a rule $f(b_1, b_2, \ldots, b_n) \rightarrow exp2$ in R such that $f(a_1, a_2, \ldots, a_n) = f(b_1, b_2, \ldots, b_n)\sigma$ and $exp = (exp2)\sigma$ for some ground normal substitution σ. That means there is a reduction from $(exp2)\sigma$ to rhs using only the rules in the success sets as above. But according to Hullot [32], there is a narrowing sequence from $exp2$ to rhs with composition of substitutions used in the narrowing operations being equal to σ. This is true because

rhs is ground. Hence in the narrowing tree whose root is $f(b_1, b_2, \ldots, b_n) \to exp2$, we have a leaf $f(b_1, b_2, \ldots, b_n)\sigma \to rhs$, or equivalently $f(a_1, a_2, \ldots, a_n) \to rhs$, which would be copied into $success_f$. □

THEOREM 3 (Soundness, unconditional rules): Given a DFL program R composed only of unconditional rewrite rules, and a query Q, let σ be a normal ground substitution. Then, upon termination of the algorithm in figure 3.1, $answer((Q)\sigma) \to rhs \in success_{answer}$ implies $(Q)\sigma \Rightarrow^* rhs$ using only rules in R'.

Proof: Similar to the proof of theorem 2. Assume we are replacing $answer(Q) \to Q$ in the rules with with $answer(V_1, \ldots, V_n) \to Q$ where $V_1, \ldots, V_n$ are the variables in Q. We know that $answer((Q)\sigma) \to rhs \in success_{answer}$ iff $answer(V_1, \ldots, V_n)\sigma \to rhs \in success_{answer}$. By lemma 2 $answer(V_1, \ldots, V_n)\sigma \Rightarrow^* rhs$, or equivalently $answer(V_1, \ldots, V_n)\sigma \Rightarrow (Q)\sigma \Rightarrow^* rhs$, since $answer(V_1, \ldots, V_n)\sigma \to (Q)\sigma \in$ R'. Thus $(Q)\sigma \Rightarrow^* rhs$. □

THEOREM 4 (Completeness, unconditional rules): Given a DFL program R composed only of unconditional rewrite rules, and a query Q, let σ be a normal ground substitution. Using only rules in R', $(Q)\sigma \Rightarrow^* rhs$, and rhs is a constant implies that upon termination of the algorithm in figure 3.1, $answer((Q)\sigma) \to rhs \in success_{answer}$.

Proof: Assume we are replacing $answer(Q) \to Q$ in the rules with $answer(V_1, \ldots, V_n) \to Q$ where $V_1, \ldots, V_n$ are the variables in Q. We know that $answer(V_1, \ldots, V_n)\sigma \Rightarrow^* rhs$ because of the rule $answer(V_1, \ldots, V_n)\sigma \to (Q)\sigma$ in R'. More explicitly, $answer(V_1, \ldots, V_n)\sigma \Rightarrow (Q)\sigma \Rightarrow^* rhs$. Now we can apply lemma 3, which gives us $answer(V_1, \ldots, V_n)\sigma \to rhs \in success_{answer}$, which implies $answer((Q)\sigma) \to rhs \in success_{answer}$, at the termination of the algorithm. □

Databases Consisting of Conditional Rules

The following lemma is used in showing the correctness (with respect to conditional reduction semantics) of the computed result for databases consisting of conditional rules.

LEMMA 4: Given a DFL program R which contains conditional rules (unconditional rules are just conditional rules where the condition is the condition "true,") $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in success_f$ for any function name f in R implies $f(a_1, a_2, \ldots, a_n)) \Longrightarrow^* rhs$ using only the rules in R'.

Proof: We shall prove the assertion by induction on the order in which a rule is placed in a success set (in a way similar to the proof of lemma 1).

In order to facilitate the proof, we modify the query evaluation algorithm in figure 3.1 slightly without changing its outward behavior. First we eliminate the simplification step (2a). We modify step (2d) so that rules of the form $lhs : true \rightarrow rhs$ will cause $lhs \rightarrow rhs$ to be placed in a success set if rhs is a constant. Step (2c) will perform the *narrowing operation on the right hand side* only on rules of the form $lhs : true \rightarrow rhs$. Note that these changes do not affect the basic workings of the algorithm.

Basis: Assume the rule $lhs \rightarrow rhs$ is the first one to be placed in a success set. Then rhs is a constant, and lhs is ground (due to the restrictions on rules). Since $lhs \rightarrow rhs$ is the very first rule to be placed in any success set, $lhs : true \rightarrow rhs$ must also be in R, and also in R'. Hence, $lhs \Longrightarrow rhs$ because $true \Longrightarrow^* true$.

Inductive Hypothesis: Assume that if a rule $lhs \rightarrow rhs$ is the N^{th} rule to be placed in a success set, or is placed before the N^{th} rule, then $lhs \Longrightarrow^* rhs$ using only the rules in R'. We need to show that if a rule $lhs1 \rightarrow rhs1$ is placed in a success set as the $(N+1)^{th}$ rule, then $lhs1 \Longrightarrow^* rhs1$ using only the rules in R'.

What led to the placement of $lhs1 \rightarrow rhs1$ in a success set? We started out with a rule of the form $lhs2 : cond2 \rightarrow rhs2$ in the database as the root of a narrowing tree, narrowed $cond2$ to $true$ with composition of substitutions generated along the way being ϕ, and narrowed $(rhs2)\phi$ to $rhs1$, with δ being the composition of substitutions. If we let σ be the composition of substitutions ϕ and δ, then $lhs1 = (lhs2)\sigma$. If we look at "$cond2 \rightarrow rhs2$" as a single term, then we narrowed "$cond2 \rightarrow rhs2$" to "$true \rightarrow (rhs2)\phi$" with composition of substitutions generated along the way being ϕ, $true \rightarrow (rhs2)\phi$ to $true \rightarrow rhs1$ with composition of substitutions δ.

The set of rules we used in the narrowing operations was the first N elements of the success sets combined. We can show by induction on the number of narrowing steps that $(cond2)\phi \Rightarrow^* true$ using the same set of rules (the first N rules in success sets), and $((rhs2)\phi)\delta \Rightarrow^* rhs1$, (or just $(rhs2)\sigma \Rightarrow^* rhs1$, since σ is the composition of ϕ and δ).

We can transform the reduction sequence $(rhs2)\sigma \Rightarrow^* rhs1$ to a conditional reduction sequence that uses rules only from R' in the following fashion: If $a \Rightarrow a[u \leftarrow rhs]$ is a part of this reduction sequence, where a rule $lhs \rightarrow rhs$ is used to reduce a to $a[u \leftarrow rhs]$, by the induction hypothesis, $lhs \Longrightarrow^* rhs$, or $lhs \Longrightarrow b_1 \Longrightarrow b_2 \Longrightarrow \ldots \Longrightarrow b_n \Longrightarrow rhs$ using only rules in R'. Replace $a \Rightarrow a[u \leftarrow rhs]$ in the original reduction sequence with $a \Longrightarrow a[u \leftarrow b_1] \Longrightarrow a[u \leftarrow b_2] \Longrightarrow \ldots \Longrightarrow a[u \leftarrow b_n] \Longrightarrow a[u \leftarrow rhs]$. Thus, we have $(rhs2)\sigma \Longrightarrow^* rhs1$, using only rules in R'.

Using the same methodology, we have $(cond2)\phi \Longrightarrow^* true$ using only rules from R'.

According to the (slightly modified) query evaluation algorithm, $lhs1 : true \rightarrow rhs1$ causes the addition of $lhs \rightarrow rhs$ to a success set if $rhs1$ is a constant. We know that $lhs1$ ($= (lhs2)\sigma$) is ground (due to lemma 1). $(rhs2)\sigma$ and $(cond2)\sigma$ must also be ground since all the rules in the success sets that were used in the reduction are ground (otherwise reduction would not be possible!). Hence $(lhs2)\sigma : (cond2)\sigma \rightarrow (rhs2)\sigma \in$ R'. We then have $lhs1 (= (lhs2)\sigma) \Longrightarrow (rhs2)\sigma$ because $(cond2)\sigma \Longrightarrow^* true$ (using rules only in R') as shown above. $(rhs2)\sigma \Longrightarrow^* rhs1$ (using rules only in R') as shown above, and thus $lhs1 \Longrightarrow (rhs2)\sigma \Longrightarrow^* rhs1$, using only rules in R'. $\Box$

Lemma 5 will be useful in showing the completeness (with respect to reduction semantics) of the computed result for databases consisting of conditional rules.

LEMMA 5: Given a DFL program R which contains conditional rules, for any function name f in P, $f(a_1, a_2, \ldots, a_n) \Longrightarrow^* rhs$, where $a_1, a_2, \ldots, a_n$ are all constants, rhs is a constant, and using only the rules in R', implies $f(a_1, a_2, \ldots, a_n) \rightarrow rhs \in success_f$ at the termination of the algorithm in figure 3.1.

Proof: We shall prove the assertion by induction on the *total* number of reduction steps, N, which takes us from $f(a_1, a_2, \ldots, a_n)$ to rhs.

Basis: N=1. There must be a rule $f(a_1, a_2, \ldots, a_n) : true \rightarrow rhs$ in R'. But this same rule must be in R also, since the left hand side cannot contain any variables not occurring on the right hand side or in the condition. Since rhs is a constant, this rule would be copied to $success_f$ in the first iteration.

Inductive Hypothesis: Assume the assertion is true for N=n. Prove the assertion for N=n+1. Let $f(a_1, a_2, \ldots, a_n) \Longrightarrow exp \Longrightarrow^* rhs$ be a reduction sequence of length n+1. Note that there must be a rule $f(a_1, a_2, \ldots, a_n) : cond \rightarrow exp$ in R', since $a_1, a_2, \ldots, a_n$ are all constants. Without loss of generality, and in order to more concisely explain some ideas, assume exp is of the form $g(h(a), q(c))$, where g, h and q are function names and a and c are constants. Also assume that $cond$ is of the form $r(s(a_1), t(a_2))$, where r, s, and t are function names, and a_1 and a_2 are constants. In order that $g(h(a), q(c)) \Longrightarrow^* rhs$ in n reduction steps or less using only rules in R', $h(a) \Longrightarrow^* d$ for some constant d in less than

$n+1$ reduction steps, $q(c) \Longrightarrow^* e$ for some constant e in less than $n+1$ reduction steps, and $g(d,e) \Longrightarrow^* rhs$ in less than $n+1$ reduction steps. In addition, $r(s(a_1), t(a_2)) \Longrightarrow^* true$ implies $s(a_1) \Longrightarrow^* b_1$, $t(a_2) \Longrightarrow b_2$, and $r(b_1, b_2) \Longrightarrow true$, all reductions taking less than $n+1$ steps individually (b_1 and b_2 are constants).

Now, by the inductive hypothesis, $h(a) \rightarrow d \in success_h$, $q(c) \rightarrow e \in success_q$ and $g(d,e) \rightarrow rhs \in success_g$. Also, $s(a_1) \rightarrow b_1 \in success_s$, $t(a_2) \rightarrow b_2 \in success_t$ and $r(b_1, b_2) \rightarrow true \in success_r$. Note that we now can have the reduction $g(h(a), q(c)) \Rightarrow_{h(a)\rightarrow d}$ $g(d, q(c)) \Rightarrow_{q(c)\rightarrow e} g(d,e) \Rightarrow_{g(d,e)\rightarrow rhs} rhs$ ($exp_1 \Rightarrow_{a\rightarrow b} exp_2$ means there is a reduction from exp_1 to exp_2 using the rule $a \rightarrow b$) . Thus there is a reduction from exp to rhs using the above rules in the success sets of g, h and q. Also, there is a reduction $r(s(a_1), t(a_2)) \Rightarrow^* true$ using only the rules in the success sets of r, s, and t. More explicitly, $r(s(a_1), t(a_2)) \Rightarrow_{s(a_1)\rightarrow b_1}$ $r(b_1, t(a_2)) \Rightarrow_{t(a_2)\rightarrow b_2} r(b_1, b_2) \Rightarrow_{r(b_1,b_2)\rightarrow true} true$.

But since $f(a_1, a_2, \ldots, a_n) : cond \rightarrow exp \in$ R', there must be a rule $f(b_1, b_2, \ldots, b_n) :$ $cond2 \rightarrow exp2$ in R such that $f(a_1, a_2, \ldots, a_n) = f(b_1, b_2, \ldots, b_n)\sigma$, $cond = (cond2)\sigma$ and $exp = (exp2)\sigma$ for some ground normal substitution σ. That means there is a reduction from $(exp2)\sigma$ to rhs and $(cond2)\sigma$ to $true$ using only the rules in the success sets as above. Again considering $cond2 \rightarrow exp2$ as a single term, there is a reduction from $(cond2 \rightarrow exp2)\sigma$ to "$true \rightarrow rhs$," using the rules in success sets. But according to Hullot [32], there is a narrowing sequence from "$cond2 \rightarrow exp2$" to "$true \rightarrow rhs$" with composition of substitutions used in the narrowing operations being equal to σ. This is true because "$true \rightarrow rhs$" is ground. Hence in the narrowing tree whose root is $f(b_1, b_2, \ldots, b_n) : cond2 \rightarrow exp2$, we have a leaf $f(b_1, b_2, \ldots, b_n)\sigma : true \rightarrow rhs$, or equivalently $f(a_1, a_2, \ldots, a_n) : true \rightarrow rhs$, which would result in $f(a_1, a_2, \ldots, a_n) \rightarrow rhs$ being copied into $success_f$. $\Box$

THEOREM 5 (Soundness, conditional rules): Given a DFL program R composed of conditional rewrite rules, and a query Q, let σ be a normal ground substitution. Then, upon termination of the algorithm in figure 3.1, $answer((Q)\sigma) \rightarrow rhs \in success_{answer}$ implies $(Q)\sigma \Longrightarrow^* rhs$ using only rules in R'.

Proof: Assume we are replacing $answer(Q) \rightarrow Q$ in the rules with with $answer(V_1, \ldots, V_n) \rightarrow Q$ where $V_1, \ldots, V_n$ are the variables in Q. We know that $answer((Q)\sigma) \rightarrow rhs \in success_{answer}$ iff $answer(V_1, \ldots, V_n)\sigma \rightarrow rhs \in success_{answer}$. By lemma 4 $answer(V_1, \ldots, V_n)\sigma \Longrightarrow^* rhs$, or equivalently $answer(V_1, \ldots, V_n)\sigma \Longrightarrow (Q)\sigma \Longrightarrow^* rhs$, since $answer(V_1, \ldots, V_n)\sigma : true \rightarrow (Q)\sigma$ is the only rule in R' defining the function name "answer." We then have $(Q)\sigma \Longrightarrow^* rhs$. □

THEOREM 6 (Completeness, conditional rules): Given a DFL program R composed of conditional rewrite rules, and a query Q, let σ be a normal ground substitution. $(Q)\sigma \Longrightarrow^* rhs$, where rhs is a constant, using only rules in R' implies that upon termination of the algorithm in figure 3.1, $answer((Q)\sigma) \rightarrow rhs \in success_{answer}$.

Proof: Assume we are replacing $answer(Q) \rightarrow Q$ in the rules with with $answer(V_1, \ldots, V_n) \rightarrow Q$ where $V_1, \ldots, V_n$ are the variables in Q. We know that $answer(V_1, \ldots, V_n)\sigma \Longrightarrow^* rhs$ because of the rule $answer(V_1, \ldots, V_n)\sigma : true \rightarrow (Q)\sigma$ in R', i.e., $answer(V_1, \ldots, V_n)\sigma \Longrightarrow (Q)\sigma \Longrightarrow^* rhs$. Now we can apply lemma 5, which gives us $answer(V_1, \ldots, V_n)\sigma \rightarrow rhs \in success_{answer}$, which implies $answer((Q)\sigma) \rightarrow rhs \in success_{answer}$. □

Translation of DataLog to DFL

A Datalog program is composed of facts and implications. If "fact" is a fact, we generate the DFL rule "fact → true." If "a ← b & c & d ..." is an implication, we generate the rule "a → b and c and d ...." If Datalog programs are translated in this fashion to DFL programs, then the DFL program has the same meaning as the source Datalog program. In fact there is almost a one-to-one correspondence between bottom-up evaluation of the original Datalog program, using "naive" evaluation [53], and the execution of the algorithm in figure 3.1 on the DFL program translated from the Datalog program. Datalog programs with negation are treated similarly, with "not"s remaining intact. For example, "a $\leftarrow$ b & not(c) &d ..." would be translated as "a $\rightarrow$ b and not(c) and d ...." In the next section we discuss how we can extend DFL and the query evaluation algorithm to deal with negation.

We have the following theorem which establishes the connection between Datalog programs and their translated counterparts in DFL. Since SLD resolution is correct and complete with respect to both fixpoint semantics and least model semantics for Prolog programs [35] (and hence Datalog programs), we use SLD resolution as the operational semantics for Datalog programs in establishing this connection.

THEOREM 7 (Subsumption of Datalog by DFL): Let P be a set of clauses defining a logic program in Datalog. Let P'={(clause)σ | clause $\in$ P and σ is a normal ground substitution and (clause)σ is ground}. Let R be the corresponding set of DFL rules (translated from the Datalog program according to the rules given above). Let R'={(rule)σ | rule $\in$ R and σ is a normal ground substitution and (rule)σ is ground}. Then:

(a) $f(a_1, \ldots, a_n) \Rightarrow^* true$, where $a_1, \ldots, a_n$ are all constants, implies there is a path in the SLD resolution tree from the goal $f(a_1, \ldots, a_n)$ to the empty clause, using the clauses in P'.

(b) there is a path in the SLD resolution tree from the goal $f(a_1, \ldots, a_n)$, where $a_1, \ldots, a_n$ are all constants, to the empty clause, using the clauses in P', implies $f(a_1, \ldots, a_n) \Rightarrow^* true$.

Proof:

(a) We prove the assertion by induction on the length, N, of the reduction sequence from $f(a_1, \ldots, a_n)$ to "$true$."

Basis: N=1. There must be a rule $f(a_1, \ldots, a_n) \rightarrow true$ in R', and correspondingly a clause (fact) $f(a_1, \ldots, a_n)$ in P'. In one resolution step we reach the empty clause. Inductive Hypothesis: Assume that the assertion is true for N=n. Prove the assertion for N=n+1. Let a ==> b mean there is an *arc* in the refutation tree from a to b, and c ==>* d mean there is a *path* in the refutation tree from c to d. Assume, without loss of generality, $f(a_1, \ldots, a_n) \Rightarrow and(g(b), h(c)) \Rightarrow^* true$. This means $g(b) \Rightarrow^* true$ and $h(c) \Rightarrow^* true$ each in less than n+1 steps. We also know that $and(true, true) \Rightarrow true$. Now there must be a clause in P' $f(a_1, \ldots, a_n) \leftarrow g(b)\&h(c)$ which gave rise to the rule $f(a_1, \ldots, a_n) \rightarrow and(g(b), h(c))$ in R' which was used in the first step of the reduction sequence above. So we have the path $f(a_1, \ldots, a_n)$ ==> $g(b)\&h(c)$. But individually g(b) and h(c) have a refutation by the inductive hypothesis. Hence, $f(a_1, \ldots, a_n)$ ==> $g(b)\&h(c)$ ==>* $h(c)$ ==>* $empty\_clause$.

(b) We prove the assertion by induction on the length, N, of the path in the resolution tree from the root (goal) to the empty clause.

Basis: N=1. There must be a a clause (fact) $f(a_1, \ldots, a_n)$ in P', and correspondingly a rule $f(a_1, \ldots, a_n) \rightarrow true$ in R'. We have $f(a_1, \ldots, a_n) \Rightarrow true$.

Inductive Hypothesis: Assume the assertion is true for N=n. Prove the assertion for N=n+1. Assume, without loss of generality, $f(a_1, \ldots, a_n)$ ==> $g(b)\&h(c)$ ==>* $empty\_clause$. This means $g(b)$ ==>* $empty\_clause$ and $h(c)$ ==>* $empty\_clause$, each in less than n+1 steps. Now there must be a rule in R' $f(a_1, \ldots, a_n) \rightarrow and(g(b), h(c))$ which is the translation of the clause $f(a_1, \ldots, a_n) \leftarrow g(b)\&h(c)$ in P' which was used to generate the first arc of the resolution tree above. So we have the reduction $f(a_1, \ldots, a_n) \Rightarrow and(g(b), h(c))$ (and being used in prefix notation). But $g(b) \Rightarrow^* true$ and $h(c) \Rightarrow^* true$ due to the inductive hypothesis. Hence we have the reduction $f(a_1, \ldots, a_n) \Rightarrow and(g(b), h(c)) \Rightarrow^* and(true, h(c)) \Rightarrow^* and(true, true) \Rightarrow true$ (last reduction step due to the fact that $and(true, true) \rightarrow true$ is included in every database by default). $\square$

DataFunLog With Negation

In this section, we generalize DFL programs to allow for the function "not" to appear either in the condition or right hand side of a rule. We shall call such programs DFLN *programs*. In order to accommodate negation, we introduce a special atom "failure" into DOM.

Stratication

We start this section by defining a few terms which we shall use in describing the concept of *stratification* and *stratified databases*.

Given a term t, we say that a function f occurs *negatively* in t if $t[u] = not(s)$ for some occurrence u, term s, and subterm $f(\ldots)$ of s. If $f(\ldots)$ is a subterm of t and f does not occur negatively in t, then we say that f occurs *positively* in t.

We say that a function f *depends* on another function g if g occurs either on the right hand side or condition of a rule defining f, OR (recursively) f depends on some other function h, and h depends on g.

In order for the query evaluation algorithm for DFLN programs to work, the DFLN program needs to be *stratified*. A DFLN program is *stratified* if there are no two function names f and g such that g occurs negatively in either the condition or the right hand side of a rule defining f, and g depends on f. The reason we want to classify function names into strata is that we want to put an order on the computation of functions. Suppose we are given the set of rules $\{f(\ldots) : \ldots not(g(b)) \ldots \rightarrow rhs,\ g(a) \rightarrow true\}$. DOM=$\{a, b, true, false, failure\}$. We want to deduce that $g(b)$ must be $false$, since it is not provable from the given facts (i.e., we want to make the so-called *closed world assumption*). We can achieve this by computing all the values X for which $g(X)$ is true, and setting to $false$ ($failure$) all other instantiations of X in $g(X)$. This process would result in our adding "$g(b) \rightarrow failure$" to the provable facts (as well as the rules $g(true) \rightarrow failure$, $g(false) \rightarrow failure$ and $g(failure) \rightarrow failure$) so that when we need to evaluate "$not(g(b))$," we will have a value for $g(b)$. Of course this should be done BEFORE we try to compute f. Since the computation of f needs g to be already computed before it can proceed, the computation of g should not depend, even indirectly, on f.

The rules making up a DFLN program have a *stratification* if all the function names can be assigned integer values such that for any two functions f and g, if there exists the rule "f(...) : cond → rhs," then:

- stratum(f) ≥ stratum(g) if g occurs positively in "cond" or in "rhs"
- stratum(f) > stratum(g) if g occurs negatively in "cond" or in "rhs"

In figure 3.2 we give an algorithm which decides whether a DFLN program is stratifiable or not, and if stratifiable gives a stratification. We assume the existence of an integer array, *stratum*, indexed by function names.

It is the case that a *Datalog* program is *stratified* iff it has a *stratification*. For a proof of this, please refer to [53]. However, we can use similar reasoning to show that this result holds for DFLN programs. In the same way, we can show the correctness of algorithm given in figure 3.2, which is basically an adaptation of the stratification algorithm for Datalog programs given in [53].

1. stratum(f) := 1 for all function names f.
2. Repeat
   (a) For any two function names g and h: If stratum(g) $\leq$ stratum(h) and h occurs negatively in a defining rule of g then stratum(g) := stratum(h)+1
   (b) For any two function names g and h: If stratum(g) $<$ stratum(h) and h occurs positively in a defining rule of g then stratum(g) := stratum(h)
3. Until for any function name f, stratum(f) $>$ the total number of function names, in which case return "NO STRATIFICATION" OR none of the above conditions apply, in which case return $\{< f, stratum(f) > \mid f$ is a user defined function name in the DFLN program }

Figure 3.2: Algorithm to Find a Stratification for a DFLN Program

EXAMPLE 3: Suppose we are given the following DFLN program.

f(X): not(g(h(X))) → k(X).   (1)
h(X) → k(X).   (2)
g(a) → true.   (3)
k(b) → a.   (4)
k(c) → d.   (5)

There are four user defined function names in this program. The algorithm in figure 3.2 proceeds as follows:

stratum(and) := stratum(or) := stratum(not) := stratum(=) := stratum(f) :=
stratum(g) := stratum(h) := stratum(k) := 1.
stratum(f):=2 because of rule 1.
return {<f,2>,<g,1>,<h,1>,<k,1> <and,1>,<or,1>,<not,1>,<=,1> }.

EXAMPLE 4: Here we give a program that has no stratification.

f(X): not(g(h(X),Y)) → k(X).   (1)
g(b,c) → true.   (2)
f(a) → true.   (3)
h(b) → true.   (4)
h(X) → f(X).   (5)
h(a) → b.   (6)
k(d) → e.   (7)

stratum(and) := stratum(or) := stratum(not) := stratum(=) := 1
stratum(f) := stratum(g) := stratum(h) := stratum(k) := 1.
stratum(f):=2. (due to rule 1)
stratum(h):=2. (due to rule 5)
stratum(f):=3. (due to rule 1)
stratum(h):=3. (due to rule 5)
stratum(f):=4. (due to rule 1)
stratum(h):=4. (due to rule 5)
stratum(f):=5. (due to rule 1)
Algorithm stops with "NO STRATIFICATION" as the answer.

Query Evaluation Algorithm for DFLN Programs

The algorithm for computing the result of a query is given in figure 3.3.

Lemma 1 and theorems 1 and 2 hold true for DFLN programs by similar reasoning as for DFL programs.

We assume the set of rules { not(true) $\rightarrow$ false, not(false) $\rightarrow$ true, not(failure) $\rightarrow$ true } will be part of every database, in addition to the rules defining "and" and "or" in the previous sections. We also assume the set of rules { x = x $\rightarrow$ true | x $\in$ DOM } $\bigcup$ { x = y $\rightarrow$ false | x $\in$ DOM and y $\in$ DOM and x $\neq$ y } to be part of every database.

This particular set of rules for the logical functions (connectives) capture nicely their intended meaning as boolean operators. Special care is needed for the "=" relation though. Because there is no guarantee of confluence in the rules, the precise meaning of "=" can be given by: if exp_1 and exp_2 are ground, then $true \in Den(exp_1 = exp_2)$ iff $Den(exp_1) \bigcap Den(exp_2) \neq \emptyset$, i.e., two ground expressions are "equal" if there is a common constant to which they can be reduced. If any of the expressions contains unbound variables, then the query evaluation algorithm finds values (constants) for those variables such that the when those values are substituted into the original expressions, the two expressions can be reduced to a common constant.

1. Let Q be the query. Find a stratification for the DFLN program using the algorithm in figure 3.2. Let S[Y] = the set of function names at stratum Y.
2. For X:=1 to stratum(answer) do
   - Compute the success sets for the functions in S[X], using the success sets of functions in S[1] $\bigcup$ S[2] $\bigcup \ldots \bigcup$ S[X] and the raw sets of functions in S[X] by executing the algorithm in figure 3.1
   - Let f be an n-ary function name in S[X] such that f occurs *negatively* in some rule in R
   - Let $temp_f = \{\ f(a_1, \ldots, a_n) \rightarrow failure \mid a_1, \ldots, a_n \in DOM$ and $f(a_1, \ldots, a_n) \rightarrow rhs \notin success_f$ for any constant $rhs\ \}$
   - $success_f := success_f \bigcup temp_f$
3. Endfor
4. The answer is the contents of $success_{answer}$

Figure 3.3: Finding the Answer to a Given Query for DFLN Programs

Another issue is that due to the extra-logical treatment of negation, some anomalies arise. The primary consequence of the anomalies is that "failure" has the meaning "false" only when it is the argument to the function "not." At other times it is just another constant.

To give an example of the fact that "failure" does not have the intended meaning of "false" unless it is the argument of "not," after the first iteration of the algorithm in figure 3.3, a rule "failure or true $\rightarrow$ failure" will be added to $success_{or}$, if "or" occurs negatively in some function definition. *If* we give the meaning "false" to "failure," this implies the rule "false or true $\rightarrow$ false," obviously a quite counter-intuitive, and erroneous result.

Another anomaly which illustrates the same point: the query "false = failure" results in "answer(false = failure) $\rightarrow$ false" $\in success_{answer}$; however, the query "false = not(not(failure))," which is *logically* equivalent to the previous one, results in "answer(false = not(not(failure))) $\rightarrow$ true " $\in success_{answer}$.

But there seems to be no easy solution to the problem.

Assume we were to include the following set of rules in every database:

true and failure $\rightarrow$ false.	true or failure $\rightarrow$ true.
failure and true $\rightarrow$ false.	failure or true $\rightarrow$ true.
failure and false $\rightarrow$ false.	failure or false $\rightarrow$ false.
false and failure $\rightarrow$ false.	false or failure $\rightarrow$ false.
failure and failure $\rightarrow$ false.	failure or failure $\rightarrow$ false.

in addition to the rules defining the functions "and" and "or" in section 3.1.

For the (perfectly valid) DFLN program

f(a) $\rightarrow$ f(b) or true.
answer(f(a)) $\rightarrow$ f(a).

(where the query is "f(a)"), even the above set of rules fail to generate the required answer, i.e., to place "answer(f(a)) $\rightarrow$ true" into $success_{answer}$, which logically we would expect to be the case. This is because the rule "f(b) $\rightarrow$ failure" is not generated in time (in this case it is not generated at all) for it to be used by the query answering algorithm.

We have a similar problem (due to the *same* reason as above) involving the function "and." Assume we have the following DFLN program:

f(a) $\rightarrow$ f(b) and false.
answer(f(a)) $\rightarrow$ f(a).

(The query is "f(a).") The required answer, i.e., placing "answer(f(a)) $\rightarrow$ false" into $success_{answer}$, does not happen.

These observations lead us to conclude that "failure" has the meaning "false" *only* when it is the argument of the function "not." This is the price we pay, i.e., not being able to use "failure" and "false" interchangeably everywhere, for dealing with the problem of negation in an extra-logical manner.

EXAMPLE 5: Let us trace the execution of the algorithm in 3.3 for the program below.

f(X):not(g(X))→:h(X).
g(a)→true.
h(b)→c.
answer(f(Y)) → f(Y).

- DOM = {a,b,c,true,false,failure}
- Stratify rules: S[1] = {h,g,or,and,not,=} and S[2] = {f,answer}
- Stratum 1:Computing the success sets of functions in stratum 1
  - $success_h$:={ h(b) → c }
  - $success_g$:={ g(a) → true }
  - $temp_g$:= { g(b) → failure, g(c) → failure, g(true) → failure , g(false) → failure , g(failure) → failure }
  - $success_g$:= $success_g \bigcup temp_g$
  - $success_{and}$:= all the rules associated with *and*
  - $success_{or}$:= all the rules associated with *or*
  - $success_{not}$:= all the rules associated with *not*
- Stratum 2: Computing $success_f$ and $answer_f$
  - $success_f$:= {f(b) → c }
  - $success_{answer}$:= {answer(f(b)) → c }
- Return $success_{answer}$, which is {answer(f(b)) → c } as the answer.

Related Work

The logic data model has been extended to include objects [3, 8, 36], higher order logic that supports structured data, object identity and sets [37] , higher order syntax [19] and various other features to improve its expressiveness. Chomicki [20] extends the logic model by allowing function symbols to appear in logic programs in a restricted way. Besides these, there are countless others which extend the logic data model in various ways, and we couldn't hope to cite them all here.

However, we shall not compare DFL with any of these models which extend the logic model. We shall instead compare it with Datalog, and assert that it acts as a good a platform as Datalog for the extensions made to Datalog that are cited above.

DFL, as we have seen, subsumes the logic model as defined by Datalog. As such, DFL shares all the advantages of the Datalog model, such as treating procedural and factual information in a uniform manner. Its syntax is simple and uniform, just like Datalog's. In addition, however, DFL gives us much more flexibility than Datalog in that we have a choice between the relational and functional styles of programming in one unified framework, which permits us to write easier to understand programs and pose more intuitive queries. The examples we have given in this presentation help to demonstrate this point.

Along similar lines with DFL, Poulovassilis [43] adapts a functional language to the area of deductive databases. The language developed (PFL) has features in common with both logic based deductive database languages, and with functional programming languages. The data model developed there is elegant. However, the syntax and semantics of PFL is not as simple and uniform as in Datalog or DFL, a major strength of both models.

Another approach to deductive databases is given by Bryant and Pan in [15] whereby the *Two-Level Grammar* specification language is adapted to the area of deductive databases. This scheme also has some of the advantages of logic and functional programming, in addition to a natural-language like syntax.

Summary Conclusions and Future Work

We defined a data model (called *DataFunLog*, or DFL for short) that adapts functional/logic programming to the area of deductive databases. We developed a bottom-up query evaluation algorithm for answering queries, and demonstrated the soundness and completeness of this algorithm with respect to a more standard reduction semantics. We also showed that the algorithm terminates. We then extended the DFL model to the DFLN model which deals with negation by explicitly making the *closed world assumption.* We introduced the notion of a *stratified* DFLN program, and explained its desirability. We developed an extended query evaluation algorithm which can answer queries on DFLN programs.

Our model subsumes the logic model as defined by Datalog. The availability of both the functional and relational (logic) styles of programming in one unified framework facilitates more natural representation of data, both procedural and factual, and more intuitive formulation of queries.

DFL (also DFLN) currently is at the conceptual level; we have not implemented the bottom-up query evaluation algorithms. In an actual implementation, the query evaluation algorithm for DFL programs (which is the counterpart of the *naive* evaluation algorithm for Datalog programs [53]) would need to be modified so that unnecessary computation is avoided (much in the spirit of the *semi-naive* evaluation algorithm for Datalog programs [53]) and also made more goal-directed by making use of the function dependencies. These modifications are relatively straightforward. Other database issues that would need to be considered in an actual implementation include updates, deletions, integrity constraints and concurrency control.

Future directions for further developing the DFL model include the incorporation of higher order functions (a very distinctive characteristic of functional programming), complex objects, which would allow more authentic representation of certain kinds of information, and program transformation techniques (similar to the *magic sets* transformation for

Datalog programs [9, 12, 44]) that preserve the meaning of the original program but result in more efficient execution of the bottom-up evaluation procedures.

CHAPTER

INTEGRATING FUNCTIONAL LOGIC AND OBJECTORIENTED PROGRAMMING PARADIGMS THE LANGUAGE FLOOP

Motivated by the goal of being able to manipulate complex objects symbolically, we propose a method of integrating functional, logic and object-oriented programming paradigms. Our method assumes the existence of an object-expression evaluator and relies on transformations and calls to this object-expression evaluator as its means of computation. Programs of the combined paradigm consist of conditional rewrite rules augmented to incorporate object expressions.

Integrating the three different styles of programming, when successful, has the natural consequence of permitting the programmer to approach different aspects of a problem in ways that are most convenient for those aspects, resulting in maximum program development convenience and productivity. Through the integration, the advantages of the component styles of programming naturally carry over. This ultimately leads to greater reliability in the software which is produced.

The advantages of integrating functional and logic programming have been well demonstrated in [14, 22, 24, 23, 26, 34, 45, 47] and chapter 2 of this dissertation. We are taking those advantages one step further in FLOOP by incorporating complex objects into the combined functional/logic paradigm. FLOOP is a (new) language that achieves the integration of functional, logic and object-oriented programming paradigms through a functional/logic interpreter based on transformations, an independent "object-evaluator" for evaluating object-oriented expressions (e.g. Smalltalk [2]), and an interface between the two. A FLOOP program consists of a set of *conditional rewrite rules*. The incorporation of the object-oriented functionality is achieved through allowing *object expressions* to occur anywhere constants can in the rewrite rules. Object expressions can also contain logical variables.

The expressiveness of conditional term rewriting systems is well demonstrated [23]. In FLOOP, this expressiveness is taken to greater heights since any object-oriented expression (more specifically, any *Smalltalk* [2] expression) is allowed to appear anywhere in a rule where a constant can and logical variables can be bound to objects. Naturally, the definition of a term also has to be changed to accommodate object-oriented expressions to be part of terms. Complex objects that are created in Smalltalk can thus be manipulated symbolically just as ordinary constants can. Moreover, since almost everything in Smalltalk is an object of some kind, including *messages* to objects, and a logical variable in a rule can be bound to an object, we get some unexpected (but welcome!) higher-order features in the language FLOOP.

The proposed method of integration dictates the use of conditional rewrite rules to describe relationships between objects *externally*, but a different imperative language (namely, Smalltalk) to create classes, class hierarchies, objects, and methods. To justify this separation of the object-oriented component from the functional/logic component, we need to consider that in a language like Smalltalk, the main duty of methods is to *change* the internal state of the objects they *belong to*, and to *inform* the "outside" world of the internal state of the objects. Methods, then, need to be performing *imperative* actions when they are used for changing the internal state of objects, and an imperative style of programming is most convenient for that role. On the other hand, the relationships of objects to one another is *declarative*, and can be concisely expressed using conditional equality in the form of rewrite rules.

The remainder of this chapter is organized as follows. Section 4.1 gives a brief introduction to the syntax and semantics of FLOOP. Section 4.2 describes the set of transformations used as the operational semantics. Section 4.3 discusses the implementation of FLOOP. Section 4.4 gives sample programs of the language, demonstrating the possibilities of being able to manipulate complex objects symbolically. Section 4.5 compares other approaches to the integration of the three paradigms to the transformations approach developed in this chapter, and section 4.6 is the conclusion and future research directions.

FLOOP A Brief Introduction to the Language

In this section we give an informal introduction to the syntax and semantics of FLOOP programs. The language of implementation, as well as the underlying object-oriented component, is Smalltalk (we are assuming that the reader is familiar with Smalltalk, as well as basic term rewriting terminology). The operational semantics, which relies on transformations, is described in detail in the next section.

In order that we may give a declarative reading to FLOOP programs, we need to place certain restrictions on how methods should be written. These restrictions are:

- Let `m` be a method defined for the instances of a class `C`. If `m` changes the the internal state of an object it *belongs to*, then its last statement should be `^self`; conversely if it returns some other object, it must NOT change the internal state of the object it *belongs to*.

- If a message has arguments, the method activated by the message must not send state changing messages to any of the arguments. Hence a method can change the state only of the object it *belongs to.*

These restrictions allow us to regard methods as pure (polymorphic) functions without any side effects.

Further restrictions on the object-oriented component are that each object must be able to respond to the `deepCopy` message, returning a structurally equivalent copy of itself, as well as the `=` message upon the receipt of which returns `true` if the argument to the message is structurally equivalent to itself, and `false` otherwise.

In the following, we assume the existence of two disjoint sets of function symbols: the set of *(data) constructors*, and the set of *defined* function symbols (also to be called *function names*).

We define an *(augmented) term* recursively as:

- A variable is a term (variables start with capital letters).
- A constant is a term (constants start with a small letter, and end with a delimiter like white space, comma, or a right parenthesis).
- If f is a function name, and $a_i, 1 \leq i \leq n$, are terms, then $f(a_1, a_2, \cdots, a_n)$ is a term (called a *function application*).
- If c is a constructor, and $a_i, 1 \leq i \leq n$, are terms, then $c(a_1, a_2, \cdots, a_n)$ is a term (called a *constructor term*).
- If $anyExpression$ is a valid Smalltalk expression, then $\{anyExpression\}$ is a term (called an *object term*).

- If $a_i, 1 \leq i \leq n$, are terms, then $MSG(a_1, a_2, \cdots, a_n)$ is a term (called a *message term*). a_1 should eventually evaluate to an object, a_2 should evaluate to a valid Smalltalk message understood by a_1, and $a_3, \cdots, a_n$ should eventually evaluate to objects to be passed as arguments to the method a_2.
- If $a_i, b_i, 1 \leq i \leq n$, are terms, then $NOT(a_1 = b_1, a_2 = b_2, \cdots, a_n = b_n)$ is a term (called a *negation term*). Informally, if the equations in the argument of the negated term cannot be shown to be true, then the negated term is true.

A *rewrite rule* is of the form

$$f(a_1, a_2, \ldots, a_n) : (b_1 = d_1, b_2 = d_2, \ldots, b_m = d_m) \rightarrow rhs$$

where $a_1, a_2, \ldots, a_n, b_1, d_1, b_2, d_2, \ldots, b_m, d_m$ and rhs are all terms, and f is a function name. $a_1, a_2, \ldots, a_n$ are not allowed to contain any function names. $f(a_1, a_2, \ldots, a_n)$ is called the *left hand side* of the rewrite rule, $(b_1 = d_1, b_2 = d_2, \ldots, b_m = d_m)$ is called the *condition,* and rhs is called the *right hand side* of the rewrite rule.

The meaning of a rewrite rule such as the above is as follows. Let σ be a substitution that makes the rule ground by substituting terms that contain only constructors, constants, or Smalltalk objects for each variable in the rule. We say that for any such substitution σ, $f(a_1, a_2, \ldots, a_n)\sigma$ is equal to $(rhs)\sigma$ iff $(b_1 = d_1)\sigma$, $(b_2 = d_2)\sigma$, ..., $(b_m = d_m)\sigma$ can all be shown to be true using the equality axioms and all the other (conditional) rules in the rulebase *after* all message terms have been simplified to single objects through being evaluated by the Smalltalk interpreter. Note that because of the limitations placed on methods, they can be seen in purely functional terms, taking in a set of objects as arguments, including the object to which they were sent, and returning another object as an answer (which may very well be the object to which they were sent, possibly with a different internal state). Thus we can give a fully declarative reading to rewrite rules. The declarative reading of a FLOOP *programs*, then, is the conjunction of the declarative reading of the individual rewrite rules in the program.

Operational Semantics Through Transformations

In this section, we give the set of transformations that form the core of the FLOOP interpreter. The method of transformations for solving unification problems both in the first order and higher-order cases, either with or without equality, has been developed and advocated strongly by [27, 31, 38, 49, 51, 50]. Indeed, the method provides "an abstract and mathematically elegant means of analyzing the invariant properties of unification in various settings by providing a clean separation of the logical issues from the specification of procedural information" [50]. The usefulness of transformations is that they can be used (as we have done, though in some restricted form due to efficiency considerations) as the operational semantics of functional/logic languages in a very straightforward manner.

Our transformations, which are based on those in [31], force the binding of variables *only* to terms that denote a *value* (i.e., a term that contains no defined function symbols and no subterm of the form $MSG(\ldots)$). This implements an *eager* strategy for parameter passing. Because of this eager strategy, our transformations are not complete. The alternative of trying to achieve completeness would have resulted in much more nondeterminism and inefficiency, and would probably disqualify FLOOP as a programming language! The transformations (excluding the ones that deal with the object-oriented extensions to keep the discussion confined to first order equational logic) are sound, however, since any rule that we use is either a restricted version of a rule in [31], or can be simulated by *narrowing* and *reflection* as defined in [31].

The Transformation Algorithm

A FLOOP program consists of a set of rules of the form

$$f(t_1, t_2, \ldots, t_n) : (r_1 = s_1, r_2 = s_2, \ldots r_m = s_m) \rightarrow t.$$

The goal of the transformation algorithm is to solve a set of equations of the form $(e_1 = d_1, e_2 = d_2, \ldots, e_p = d_p)$, i.e., find values (terms that consist only of variables, constants, constructors or objects) for variables in the equations such that the equalities can be shown to be true (using the rules in the program) when the values are substituted in the place of variables in the equations. The methodology described below returns answers in the form of *sequential* substitutions. We define a *sequential* substitution as a list of pairs of terms, $< a_1/X_1, a_2/X_2, \ldots, a_n/X_n >$, where a_i is a term that contains no function applications, and X_j is any variable. Let the notation $t[a/X]$ mean the term obtained when all occurrences of the variable X in t are replaced by the term a (if a is an object term, then by a copy of a). Then the *application* of a sequential substitution σ (e.g. $< a_1/X_1, a_2/X_2, \ldots, a_n/X_n >$) to a term t, $(t)\sigma$, is the term $(((t)[a_1/X_1])[a_2/X_2]) \ldots [a_n/X_n]$. This is the same as applying γ (in the conventional sense) to t where γ is the composition of the substitutions $\{a_1/X_1\}$, $\{a_2/X_2\}$, ..., $\{a_n/X_n\}$.

The Algorithm: We start out with a *pair* of ordered lists of equations (F, S). F, which will eventually become the basis for a solution, is empty. S is the original list of equations to be solved (i.e., the *goal*). If S ever becomes empty as a result of the transformations, F will give us the desired solution. We perform the transformations, always choosing the leftmost equation in S to *operate* on, until S becomes empty, in which case we terminate successfully, or until no transformation applies, in which case we stop with failure. Upon successful termination of the transformation sequence, if F is $< X_1 = t_1, X_2 = t_2, \ldots, X_q = t_q >$, then the sequential substitution $< t_1/X_1, t_2/X_2, \ldots, t_q/X_q >$ is an answer.

Transformations

Below is the set of transformations (described using terminology as consistent with [31] as possible) used by the FLOOP interpreter. Lists are represented as $< \ldots >$ and $\bullet$ will mean list concatenation. To avoid duplication, we shall regard $a = b$ as representing $b = a$ also. The pair of lists will be represented by (L_1, L_2) etc.

Removal of Trivial Equations:

$$(L_1, < a = a > \bullet L_2) \Rightarrow_t (L_1, L_2)$$

if a is a variable, a constant or an object.

Variable elimination 1:

$$(L_1, < X = a > \bullet L_2) \Rightarrow_{ve1} (L_1 \bullet < X = a >, L_2[a/X])$$

if X is a variable, and a is a variable, or a constant and $X \neq a$.

Variable elimination 2 (objects):

$$(L_1, < X = a > \bullet L_2) \Rightarrow_{ve2} (L_1 \bullet < X = copy(a) >, L_2[a/X])$$

if X is a variable, and a is an object. Note that a copy of a is placed in F. Furthermore, each occurrence of X in L_2 is replaced by a distinct copy of a.

Imitation:

$$(L_1, < X = c(s_1, \ldots, s_n) > \bullet L_2)$$

$$\Rightarrow_i (L_1 \bullet < X = c(Y_1, \ldots, Y_n) >, < Y_1 = s_1, \ldots, Y_n = s_n > \bullet (L_2[t/X]))$$

where c is a constructor, X is a variable, X does *not* occur in $c(s_1, \ldots, s_n)$, $c(s_1, \ldots, s_n)$

does not contain any defined function symbols, $t = c(Y_1, \ldots, Y_n)$, and $Y_1, \ldots, Y_n$ are all new variables. This rule helps incrementally bind a variable to its final value.

Exposure 1:

$$(L_1, < X = t > \bullet L_2) \Rightarrow_{e1} (L_1, < Y = t[\alpha], X = t[\alpha \leftarrow Y] > \bullet L_2)$$

where X is a variable, Y is a new variable, $t = c(a_1, \ldots, a_n)$, c is a constructor, $t[\alpha]$ is the subterm of t at address α, $t[\alpha \leftarrow Y]$ is the term obtained by replacing the subterm of t at address α with the term (in this case variable) Y, $t[\alpha] = f(\ldots)$, f is a defined function symbol, and $f(\ldots)$ is not a proper subterm of any $g(\ldots)$ where g is a defined function symbol and $g(\ldots) = t[\gamma]$. This rule exposes a reducible expression in a goal equation so that lazy narrowing can be applied to it. The restriction that X be a variable ensures that the *imitation* rule and this rule are not simultaneously applicable (reducing nondeterminism).

Exposure 2:

$$(L_1, < s = t > \bullet L_2) \Rightarrow_{e2} (L_1, < Y = t[\alpha], s = t[\alpha \leftarrow Y] > \bullet L_2 >)$$

where Y is a new variable, $t = MSG(a_1, a_2, \ldots, a_{n'})$, s is *any* term and the rest is the same as in *Exposure 1*. Note that there is no restriction on s.

Decomposition:

$$(L_1, < c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n) > \bullet L_2) \Rightarrow_d (L_1, < s_1 = t_1, \ldots, s_n = t_n > \bullet L_2)$$

where c is a constructor, and $s_1, \ldots, s_n, t_1, \ldots, t_n$ are all terms.

Lazy Narrowing:

$$(L_1, < t = f(s_1, \ldots, s_n) > \bullet L_2)$$

$$\Rightarrow_{ln} (L_1, < a_1 = s_1, \ldots, a_n = s_n, b_1 = d_1, \ldots, b_m = d_m, t = rhs > \bullet L_2)$$

where $f(a_1, \ldots, a_n) : (b_1 = d_1, \ldots, b_m = d_m) \rightarrow rhs$ is a rewrite rule where all variables have been renamed so that they are different from the variables occurring in the two lists.

Simplification:

$$(L_1, < t = MSG(arg_1, arg_2, \ldots, arg_n) > \bullet L_2) \Rightarrow_s (L_1, < t = s > \bullet L_2)$$

where $arg_1, arg_2, \ldots, arg_n$ are all either message terms or object terms. Let obj be the object returned by the Smalltalk interpreter upon the evaluation of the message contained in arg_2 to the object contained in arg_1 with the arguments $arg_3 \ldots arg_n$; then s is the object term containing obj. If any arg_i is a message term, then it is (recursively) simplified to obtain an object term to be used as described above.

Negation:

$$(L_1, < t = NOT(a_1 = b_1, \ldots, a_n = b_n) > \bullet L_2) \Rightarrow_{neg} (L_1, < t = bool > \bullet L_2)$$

where if $< a_1 = b_1, \ldots, a_n = b_n >$ can be shown to have at least one solution (using the transformations recursively) then *bool* is the constant $false$, otherwise, it is the constant $true$. This is a version of the *closed world assumption* in the context of (conditional) equality theories.

This concludes the transformations. An observation about the first list is that at all times it contains equations only of the form $X = t$ where X is a variable and t is either a variable, a constant, an object, or a term of the form $c(Y_1, \ldots, Y_n)$ for some n, and each Y_i is a variable. Consequently, variables in the original set of equations will not be bound to terms containing any defined function symbols, or terms of the form $MSG(\ldots)$.

EXAMPLE 6: To demonstrate how the transformations work, let $P = \{f(X) : () \rightarrow a\}$ be a FLOOP program, and $E = \{Z = c(f(Z))\}$ be the set of equations to be solved. We have

$$\begin{aligned}
&(<>, < Z = c(f(Z)) >)\\
&\Rightarrow_{e1} (<>, < W_1 = f(Z), Z = c(W_1) >)\\
&\Rightarrow_{ln} (<>, < X_1 = Z, W_1 = a, Z = c(W_1) >)\\
&\Rightarrow_{ve1} (< X_1 = Z >, < W_1 = a, Z = c(W_1) >)\\
&\Rightarrow_{ve1} (< X_1 = Z, W_1 = a >, < Z = c(a) >)\\
&\Rightarrow_{i} (< X_1 = Z, W_1 = a, Z = c(W_2) >, < W_2 = a >)\\
&\Rightarrow_{ve1} (< X_1 = Z, W_1 = a, Z = c(W_2), W_2 = a >, <>)
\end{aligned}$$

The answer substitution σ is obtained from the first list, i.e., $\sigma = < Z/X_1, a/W_1, c(W_2)/Z, a/W_2 >$ (remember that σ is a *sequential* substitution). We confirm that

$$\begin{aligned}
&(Z)\sigma \\
&= (Z) < Z/X_1, a/W_1, c(W_2)/Z, a/W_2 > \\
&= (Z) < a/W_1, c(W_2)/Z, a/W_2 > \\
&= (Z) < c(W_2)/Z, a/W_2 > \\
&= (c(W_2)) < a/W_2 > \\
&= (c(a))
\end{aligned}$$

as we had expected.

Implementation

Smalltalk Implementation

FLOOP has been fully implemented in Smalltalk. The transformations that form the operational semantics of the language have been faithfully followed in the implementation. Below, we give a high level description of the more important classes involved in the implementation.

The two most important classes that are needed by the interpreter are $Term$ and $RuleBase$. Instances of $Term$ are augmented terms described before. $RuleBase$ is the depository of rules that make up a FLOOP program. The declarations for the class $Term$ is as follows:

```
Object variableSubclass: #Term
  instanceVariableNames:
    'msg isVar isConstant isNot eqList isMsg isFunApp isObjApp
     isConstructor isObject name args theObject originalName '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

The instance variables in the class definitions are used in the following way (in case of similar variables, only one is explained). `msg` contains the *message* if `self` is a *message* term. `isVar` is boolean depicting whether `self` is a variable. Similarly for `isConstant`, `isNot`, `isMsg`, `isFunApp`, `isConstructor` and `isObject`. `eqList` holds the list of equations to be proven false. `name` contains the constructor, function name, variable name or constant, depending upon the type of `self`. `args` holds function arguments, constructor arguments or message arguments, again depending upon the type of `self`. `object` holds a Smalltalk object. Finally, `originalName` is used to remember the original name of a variable to be able to show what variable was bound to what term at the end of a query.

The class `Term` responds (among others) to messages for creating an instance of itself with a specific type (e.g. a constant) and parsing a stream to extract a term from it. Instances of class `Term` respond to messages for substituting a term for a variable in themselves, giving a structural copy of themselves, and renaming themselves.

The definition of class `RuleBase` is as follows.

```
Set variableSubclass: #RuleBase
  instanceVariableNames:
    'dummy '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

Note that `RuleBase` is declared as a subclass of `Set` and that it has no instance variables (except `dummy` which does not serve any useful purpose except to work around a bug in the Smalltalk interpreter).

`RuleBase` holds the rules that make up a FLOOP program, and contains the methods for actually finding the answers to a query. It then answers to messages for adding a rule to itself, and solving a set of equations.

The other classes are centered around these two.

ML Implementation

FLOOP has also been implemented in ML, although the implementation is rudimentary since it lacks a front-end parser for the language. Programs then need to be given in the form of an explicit syntax tree to the interpreter. Implementing FLOOP in ML, however, has been of great instructional value, due to the fact that a whole object-oriented engine had to be built from scratch. In addition, the ML implementation sheds light onto how we can give a (partially) denotational semantics to FLOOP programs.

Below, we give the `datatype` definitions needed in the ML implementation. They should give the reader an idea about the structure of the interpreter in ML.

```
datatype
(* first the object-oriented types *)
expr = NUM of int | VAR of varname | BOOL of bool | BNOT of expr |
       BAND of expr*expr | BOR of expr*expr |  != of expr*expr |
       !+ of expr*expr  | !- of expr*expr  | !< of expr*expr |
       MESSAGE of expr*methodname*(expr list) | NEW of classname |
       (* SELF_MSG of methodname * (expr list) | *) SELF |
       FL_EXP of term | VALUE of tp*int

and

comm =   IFCOMM of expr * (comm list) * (comm list) |
 !:= of varname*expr | WHILEDO of expr * (comm list)

and

tp = INT | OBJECT of classname | BOOLEAN

and

store = STORE of varname * tp * int

and

var_def = TYP of varname * tp   (* variable declarations *)

and
```

```
method = METHOD of
          classname*methodname * (var_def list) * (var_def list) *
                      (comm list) * expr

and

tr_method   = TR_METHOD of
          classname*methodname * (var_def list) * (var_def list) *
        ( (
            int *
            ( (object list) *
              (store list) *
              (store list) ) *
            code_type ) ->
            ( int *
              ( (object list) *
                (store list) *
                (store list) ) )) *

        ( (
            int *
            ( (object list) *
              (store list) *
              (store list) ) *
            code_type ) ->
            ( int *
              ( (object list) *
                (store list) *
                (store list) ) *
              expr  ))

and

classdef = CLASS of classname * (var_def list) * ( methodname list )

and

subclass = SUBCLASS of classname * classname

and

program = PROGRAM of (classdef list) * (subclass list) *
                     (method list) * (var_def list) * (comm list) *
                     (rule list) *
                     ( term list )
and
```

```
object = OBJ of index*classname*(store list)

and

code_type = MAIN_CODE | METHOD_CODE of index

and (******** functional-logic starts **********)

state = STATE of  int *
                  ( (object list) *
                    ( store list) *
                    ( store list) ) *
                  code_type

and

term = VAR_FL of string |
       CONSTANT_FL of string |
       CONS_FL of string * (term list) |
       FUN_APP of string * (term list) |
       EQ of term * term |
       OO_EXP of expr |
       OO_VALUE of tp * int


and

rule = RULE of  rule_name * (definition list)

and

definition = DEF of ( term list ) * (term list) * term

and

triple = TRIPLE of ( term  list ) * ( term  list ) * state

and

pair = PAIR of ( term list) * state
       (* solution set will be a list of PAIRs*)

and

term_state_pair = TSP of term * state;
```

```
type object_store = object list;
type environment = var_def list;
type var_store = store list;
type rule_list = rule list;
```

FLOOP Through Examples

In this section we give several examples of FLOOP programs (using the Smalltalk implementation) to demonstrate the key features of the language.

EXAMPLE 7: The following is the insertion sort algorithm that sorts *any* list of objects that accept the messages $<$ and $>=$. Each line in the program is numbered so that they can be explained in detail.

```
(1)  | rb answer |
(2)  rb := RuleBase new2.
(3)  rb addConstructor: 'c'.
(4)  rb addRules: '
(5)     iSort(nil):()->nil.
(6)     iSort(c(H,T)):()->insert(H,iSort(T)).
(7)     insert(X,nil):()->c(X,nil).
(8)     insert(X,c(H,T)):(MSG(X,{#<},H)={true})->c(X,c(H,T)).
(9)     insert(X,c(H,T)):(MSG(X,{#>=},H)={true})->c(H,insert(X,T)).
(10)    list1():()->c({4},c({2},c({5},c({3},nil)))).'.
(11) answer := rb solveEquations: ' ( X = iSort(list1()) ) '
                   instantiating:'(X)'.
(12) ^ answer.
```

Explanation:

(1) Declare the local variables `rb` and `answer`. Note that the FLOOP interpreter is implemented in Smalltalk, and the underlying object expression evaluator is also Smalltalk, hence the Smalltalk code in "setting up" the FLOOP program.

(2) Assign a new instance of Rulebase to `rb`. `rb` will hold the rewrite rules that make up the FLOOP program. `new2` is a class method for `Rulebase` that returns an instance of `Rulebase`.

(3) Declare `c` to be a constructor.

(4-7) Self explanatory.

(8) `X` is being sent the message `#<`. Note the free mixing of logical variables and messages to objects that the variables will be instantiated to.

(9) Similar to (7), except the message now is `#>=`.

(10) A constant function that evaluates to a list of Smalltalk objects is defined (note the curly brackets around the numbers which signify that their contents are Smalltalk expressions to be evaluated).

(11) The rulebase (`rb`) is given the message to solve a set of equations, and instantiate the variable `X` with the solutions found to the equations. In general, if '$(X_1, X_2, \ldots, X_n)$' is the list of variables to be instantiated, then $\{\ (X_1, X_2, \ldots, X_n)\sigma \mid \sigma$ is a substitution found by the transformations $\}$ is what is returned as an answer.

(12) Return the answer, in this case

```
``X:c( 2 c( 3 c( 4 c( 5 nil ) ) ) )´´
```

In the Smalltalk implementation of FLOOP, every FLOOP program has to be in the format shown in example 7. In this scheme an advantage that we notice is that we can have more than one rule base at any given time, each containing a different set of rules. This amounts to having a module system of rewrite rules.

EXAMPLE 8: The following program demonstrates the true modeling capabilities of the combined functional/logic/object-oriented paradigms. We have again labeled the program for easy explanation. This program represents information about four cities, including the distances between them, and computes the minimum traveling distance between two of the cities.

```
(1)   | rb answer aStream smallestDistance aNumber |

(2)   Tuple           subclass: #City
             attributeNames: ´name population´
             attributeTypes: ´Symbol Integer´.

(3)   rb := RuleBase new2.

(4)   rb addConstructor: ´c´.

(5)   rb addRules: ´

(6)        city(a):()->{ (City new)  name:#Atlanta;
                                population: 1600 }.

(7)        city(b):()->{ (City new)  name:#Birmingham;
                                population: 1000 }.

(8)        city(c):()->{ (City new)  name:#Cambridge;
                                population: 3000 }.

(9)        city(d):()->{ (City new)  name:#Denver;
                               population: 200 }.

(10)       dist(a,b):()->{10}.
           dist(a,d):()->{50}.
           dist(b,c):()->{20}.
           dist(b,d):()->{30}.
           dist(c,d):()->{5}.

(11)       twoWayDist(X,Y):()->dist(X,Y).
           twoWayDist(X,Y):()->dist(Y,X).

(12)       member(X,nil):()-> false.
           member(X,c(X,Z)):()->true.
           member(X,c(Y,Z)):( NOT(X=Y)=true )->member(X,Z).

(13)       comp(Name1, Name2):( city(A)=C1,
                               MSG(C1,{#name}) = Name1,
                               city(B)=C2,
                               MSG(C2,{#name}) = Name2
                               )
```

```
                          ->  computeDistance( c(A,nil), A, B).

(14)      computeDistance(L,A,B):()->twoWayDist(A,B).
(15)      computeDistance(L,A,B):
                  ( twoWayDist(A,C)=D,
                    NOT(C=B)=true,
                    member(C,L)=false,
                    computeDistance(c(C,L),C,B)=E
                  )
                 -> MSG(D,{#+},E) ´.

(16)  answer  := rb solveEquations: ´( W=comp({#Denver},{#Atlanta}))´
                    instantiating: ´(W)´.

(17)  answer do:
        [ :anOrderedCollection |
              aStream := ReadStream on: anOrderedCollection.
              aNumber := (aStream peek) object.
              (smallestDistance isNil)
                    ifTrue: [ smallestDistance := aNumber ]
                    ifFalse: [ (smallestDistance > aNumber)
                                 ifTrue: [ smallestDistance := aNumber ]
                             ]
        ].

(18) ^ smallestDistance
```

Explanation:

(1) Local variables for the Smalltalk code that follows.

(2) `Tuple` is a class whose instances respond to the `=` and `deepCopy` messages (as described in the text). Subclasses of `Tuple` that are created with this message have attribute names with types, and each instance of such a class responds to the message `x` and `x:` if `x` is an instance variable of its class. Here we are creating `City` as a subclass of `Tuple` with instance variable names `name` and `population` whose types are `Symbol` and `Integer` respectively.

(3) Get a new instance of class `Rulebase` which will hold the rewrite rules defining the FLOOP program.

(4) Define `c` to be a constructor.

(5) Add the rules contained in the string to the rulebase.

(6) The rule evaluates to the `City` object whose `name` attribute is `Atlanta` and `population` attribute is `1600` (numbers made up!). `a` (which is a plain constant) in the argument of the rewrite rule will be used to easily refer to this object, and act as a kind of object identifier for it.

(7-9) Similar to (6)

(10) Distance between any two cities that are directly connected.

(11) Distance between any two cities without directionality.

(12) The familiar `member` function. Note that the condition part of a rule can contain *only* equations.

(13) Compute the distance between two cities whose names are `Name1` and `Name2`. In the condition part, we find the constants that are the object identifiers for the cities involved, and use them to compute the distance in the body of the rule. This is a perfect example of the separation of the internals of objects, and their relationships to one another.

(14) (15) `L` is the list of cities visited so far (to avoid infinite loops). These rules implement a graph traversal algorithm, keeping track of the nodes visited so far in `L`. The underlying object evaluator (i.e., Smalltalk) is used to evaluate the numerical expressions.

(16) Ask the rulebase to solve the given goal, instantiating only the variables specified as the answer.

(17) answer is assigned a Set of OrderedCollections of Terms. Each term, if it contains an object, responds to the message object and returns the object it contains. This piece of code checks each of the values W was bound to, and returns the smallest of them (could have been done in many other ways too).

(18) Return smallestDistance as the answer, in this case 35.

In the following examples, we have omitted explanations since the programs are pretty much self explanatory.

EXAMPLE 9: This example demonstrates some of the higher-order capabilities of FLOOP that are due to the higher-order features available in Smalltalk. Contrast this with higher-order features based on lambda calculus.

```
| rb answer |

 rb := RuleBase new2.
 rb addConstructor: ´c´.
 rb addRules: ´
   foldr(nil, Message, Default):()->Default ´.
   foldr( c(H,T), Message, Default ):
            (R=foldr(T,Message,Default)) -> MSG(H,Message,R).
   list1():()->c({4},c({2},c({5},c({3},nil))))´.
 answer := rb solveEquations:´(Z=foldr(list1(),{#+},{0}))´
               instantiating:´(Z)´.
 ^ answer.
```

EXAMPLE 10: Below is the permute function that permutes a list of objects.

```
 | rb answer |
 rb := RuleBase new2.
 rb addConstructor: ´c´.
 rb addRules: ´
    permute(nil):()->nil.
    permute(c(H,T)):()->insert(H,permute(T)).
    insert(X,L):()-> c(X,L).
    insert(X,c(H,T)):()->c(H,insert(X,T)).
```

```
answer := rb solveEquations:´(Z =permute(c({0},c({1},c({2},nil)))))´
              instantiating:´(Z)´.
^answer .
```

EXAMPLE 11: Below we define the `map` function that seems to be the "litmus test" for languages claiming to be higher-order. `map` (here) takes a message as its first argument, a list of objects as its second argument, and sends the message to each object in the list. The resulting list contains objects that were returned by the objects that received the messages. In this case the message is simple: the objects that receive it return themselves.

```
| rb answer |

rb := RuleBase new2.
rb addConstructor: ´c´.
rb addRules: ´
  map(M,nil):()->nil.
  map(M,c(H,T)):()->c( MSG(H,M), map(M,T))´.

answer := rb solveEquations:
                ´(Z=map({#yourself},c({0},c({1},c({2},nil)))))´
             instantiating:´(Z)´.
^ answer .
```

EXAMPLE 12: This example shows a more general `map` than the previous one in that it takes a *block* (the equivalent of which would be a lambda abstraction in lambda calculus) as an argument and a list of objects and passes the objects one by one to the block. The resulting list consists of the values returned by the block when it was given the objects in the original list. In this instance, the block adds one to every element in the list.

```
| rb answer |

 rb := RuleBase new2.
 rb addConstructor: ´c´.
 rb addRules: ´
    map1(Block,nil):()->nil.
    map1(Block,c(H,T)):() -> c( MSG(Block,{#value:},H),map1(Block,T))´.

 answer := rb solveEquations:
                  ´(Z=map1( { [:x|x+1] }, c({0},c({1},c({2},nil)))))´
              instantiating:´(Z)´.

 ^ answer .
```

Related Work

LIFE [7] and *MaudeLog* [39] are languages that represent significant efforts in joining the three paradigms of functional, logic and object-oriented programming.

LIFE (Logic, Inheritance, Functions, Equations) is based on the idea of generalizing the notion of a term to an *Order Sorted Feature (OSF for short) term* which permits self reference, has attributes and belongs to a *sort*. A hierarchy of sorts is possible, consequently LIFE has *structured type inheritance*. LIFE programs consist of Prolog-style clauses and function definitions which are basically rewrite rules.

It is possible to write very expressive programs in LIFE mainly due to the OSF terms which allow for complex constraints to be placed on their attributes, and an (obviously powerful) OSF term unifier that unifies two OSF terms. However, to what degree does structured type inheritance (the only object-oriented feature in LIFE) qualify LIFE to be an object-oriented language, when such ideas as *local state in an object*, *method definitions*, *inheritance of methods*, *method overriding* and *method refinement* which are so central to the object-oriented paradigm are missing? These very features are what make object-oriented programs reusable, maintainable and robust. FLOOP, on the other hand, has all the facilities of the underlying object-oriented component (i.e., Smalltalk), in addition everything we have come to expect from functional/logic programming, such as the *logical variable*, *non-determinism* and *higher-order functions*, in a coherent framework.

MaudeLog (an extension of *Maude*) which is described in [39] is based on *rewriting logic*. *MaudeLog* unifies the paradigms of functional, relational and *concurrent* object-oriented programming. *MaudeLog* programs consist of *rewrite rules* in *system modules*, *equations* in *functional modules* and *methods* in *object-oriented modules*. Object-oriented modules define the classes and the attributes of instances of the class. The idea of a method belonging to an object and modifying only that object is generalized to a method not belonging to any object, but rather a "set" of objects that it can operate upon. Methods are then rewrite rules that expect to be passed as one of their parameters each object they "belong to" and which they can modify. The result of a rewrite rule being applied to a set of objects is the same set of objects, but possibly with different internal states. The whole process is dynamic, with a common store of objects and messages to objects "floating in a vacuum." If a message is applicable to a set of objects, the corresponding rewrite rule is activated, possibly changing the internal state of the objects it operated upon, hence the concurrency.

MaudeLog is not directly comparable to FLOOP. It incorporates concurrency and a new way of looking at method definitions. Disregarding the concurrency aspect, however, despite the fact that the ideas behind "rewriting logic" are simple, the language itself is very involved, and does not seem to be suitable for use as a general purpose programming language. Another question is whether *MaudeLog* can be implemented with reasonable efficiency: according to [39] its implementation is still far away. We do have for FLOOP, however, a working prototype, and the language FLOOP was designed with practicality in mind.

Conclusion and Future Research Directions

In this chapter, we described a way of integrating the three paradigms of functional, logic and object-oriented programming without sacrificing any of the desirable aspects of each component paradigm. The language we defined has conditional term rewriting systems as its programs, and employs a fixed set of transformations for its operational semantics. An underlying object expression evaluator is assumed. The main highlights of the language are logical variables, all the facilities provided by the underlying object expression evaluator such as creating classes, class hierarchies, and objects, programs composed of conditional rewrite rules that can contain arbitrary object expressions, and higher-order features (dependent upon higher-order features allowed in the object-oriented component). The programs of the combined paradigm still maintain a declarative reading, however one that takes into account the operational aspect introduced by the calls for evaluation of object-oriented expressions to the underlying object expression evaluator.

FLOOP, a language which is a specific implementation of our scheme, was also described. In the Smalltalk implementation, FLOOP exhibits higher-order functionality due to the fact that messages are also objects in Smalltalk. We gave examples of FLOOP programs that highlighted the main aspects of the language.

Seen in a greater perspective, we believe that our method, by providing for declarative manipulation of complex objects, lays the framework for maximum program development convenience and productivity. This is aided by the fact that FLOOP has a simple syntax and very intuitive semantics (i.e., everything works the way you would expect it to work).

The programmer, using FLOOP, can approach different aspects of a problem in ways that are most convenient for those aspects.

Future work in this direction would include developing a more rigorous declarative semantics for FLOOP programs, as well as investigating the possibility of parallel execution.

CHAPTER

AN OBJECTORIENTED DEDUCTIVE DATABASE MODEL BASED ON CONDITIONAL REWRITE RULES

The pure object-oriented data model is not convenient for describing inter-object relationships and can result in very involved queries. We show in this chapter that augmenting the object-oriented model with declarative description of the relationships between objects using conditional rewrite rules is a convenient way to address the problem. Under this scheme, inter-object relationships are described declaratively using rewrite rules, and queries are stated declaratively in the form of equations to be solved.

In the object-oriented model of data, real world entities whose representation is needed in the database are directly represented as *database objects* that are *instances* of some *class*. Each object has a certain set of *attributes* and a unique *object identifier*.

Although in this model real world objects are represented very authentically, *external* relationships among objects are necessarily represented through the use of pointer valued attributes, which necessitates the physical traversal of pointers representing relationships in answering a query. This is less than ideal, since external relationships among objects are declarative and should be handled that way. In fact, ideally the only pointer valued attributes in an object should be those which relate a *part* to its component *subparts*.

To give an example, assume there is a one-to-many relationship between a set of objects S_1 and another set of objects S_2. If $O_1 \in S_1$ and $O_2 \in S_2$, and O_1 and O_2 are related, O_2 needs a pointer valued attribute which will point to O_1, and O_1 needs a set valued attribute that points to a set of objects that are in S_2, one of which is O_2. We could get by with fewer pointers in a case like this by having only objects in S_1 point to objects in S_2 or vice versa. However, there is a tradeoff involved in this decision: more pointers mean writing queries will be easier, but initially setting up relationships among objects (and also deleting object later) more difficult. Less pointers have the opposite effect.

Even though this is a perfect representation of the real world, in querying the database, we have to physically navigate through objects in the database. This has to be done through the definition of methods for objects, and the writing of object-oriented code even for simple queries.

We present a solution to this problem which utilizes conditional rewrite rules to represent relationships among objects and a separate object-oriented module for dealing with objects. In chapter 3 of this dissertation and also in [11] we defined a deductive database model (*DataFunLog*) based on conditional rewrite rules and showed how conditional rewrite rules can be used to declaratively define and query deductive databases. There was no object-oriented functionality in *DataFunLog* though. The model we develop here, similar to DataFunLog in that both use conditional rewrite rules to define deductive databases, *does* have object-oriented functionality, permits the posing of queries in the form of equations to be solved, and inter-object relationships can be described through conditional rewrite rules in the model. The operational semantics for answering queries is narrowing and e-unification through transformations.

In terms of implementation, this model is the result of a synthesis between the functional/logic/object-oriented programming language FLOOP, described in the previous chapter, and a pure object-oriented database system, NGO (for *Next Generation Opal*).

In the next section we give a brief introduction to NGO. In section 5.2 we define a very simple but revealing database schema, and describe some queries we would like to pose. In section 5.3 we develop a purely object-oriented solution to the problem which serves to illustrate and emphasize the fact that the pure object-oriented model is unsuitable for describing inter-object relationships from a query point of view. In section 5.4 we develop a solution based on the combined functional/logic/object-oriented model which contrasts sharply with the pure object-oriented model in the ease with which inter-object relationships are created and queries posed. Section 5.5 is a discussion of the transition from the pure object-oriented data model to the functional/logic/object-oriented data model. Section 5.6 describes other approaches to object-oriented deductive databases, and section 5.7 is the conclusion.

Next Generation Opal

Next Generation Opal (NGO) is a research tool that we developed to act as a minimal object-oriented database system. It is modeled after the *Gemstone* object oriented database system [16] (which uses the *Opal* query language). NGO provides a type system for the attributes of objects, as well as sets whose contents can only be of a certain type. The system automatically defines methods to access the attributes of objects and set those attributes to certain values. A subset of the attributes of objects belonging to some class can be specified to form a *key*. If a set contains objects of a given type, and these objects have a certain set of attributes defined as a key for the objects, this information causes the (automatic) definition of a method for the set which, when given a set of values for keys, associatively searches for an object in the set whose key matches the argument given to the method. This is not necessarily part of the object-oriented data model, but it facilitates

```
Object subclass: #Tuple
  instanceVariableNames:
    'attributeValuePairs '
  classVariableNames:
    'KeyDictionary TypeDictionary '
  poolDictionaries:
    'CharacterConstants '

Set variableSubclass: #MySet
  instanceVariableNames: ''
  classVariableNames:
    'SetTypeDictionary '
  poolDictionaries:
    'CharacterConstants '
```

Figure 5.1: Definition Of *Tuple* and *MySet*

writing queries. Of course, the user can define any other methods for any class.

NGO is implemented through the definition of two classes in Smalltalk. These are `Tuple` and `MySet`. They act as *abstract superclasses*: they are not meant to have instances of their own, but rather provide the methods their subclasses need. `Tuple` and `MySet` are defined in figure 5.1.

The instance variable `attributeValuePairs` is a `Dictionary` of *attribute-name/value* pairs. The *meta-information* about each subclass of `Tuple` is kept in the class variables `KeyDictionary` and `TypeDictionary`.

`TypeDictionary` is a dictionary, indexed by the names of the subclasses. Each element in the `TypeDictionary` is also a dictionary, whose contents are *attribute-name/attribute-type* pairs.

`KeyDictionary` is a dictionary, again indexed by subclass names of `Tuple`, whose values are arrays, each array containing the set of attributes that form the keys of a particular class.

In the definition of MySet, there is only one class variable, SetTypeDictionary, which is a dictionary indexed by subclasses of MySet. The values of SetTypeDictionary are types (i.e., class names).

Problem Denition

The problem is to represent three sets of objects, namely *courses*, *students* and *teachers*, the relationships among them and answer queries about the objects. There are only two explicit relationships among objects: teachers *teach* courses and students *take* courses. The relationship between teachers and courses is *one-to-many* and the relationship between students and courses is *many-to-many*.

The attributes of a teacher are *name*, *age*, *sex* and *salary*. The attributes of a student are *name*, *age*, *sex* and *GPA*. The attributes of a course are *name* and *timeAt* (the time the course is taught at).

Given this schema, we would like to pose the following queries:

- What courses does student s1 take that are taught by teacher t2 ?
- Who are all the students of teacher t1 ?
- What other teachers teach a course at the same time that teacher t1 teaches a course ?

Solution Using NGO Purely ObjectOriented

In this section, we show a purely object-oriented solution to the problem described in the previous section. We explain each section of the code as we proceed.

```
Tuple           subclass: #Person
          attributeNames: ´name   age      sex´
          attributeTypes: ´Symbol Integer  Character´
                     key: ´name´.
```

Create a subclass of Tuple, called Person, with attributes name, age and sex. The attributes have type (class) Symbol, Integer and Character respectively (note that the hash mark in front of a sequence of characters makes it a *symbol* in Smalltalk). Objects of type Person have a key field, name. Person is an abstract superclass, since we shall not use it for creating objects, but rather for defining the attributes that are common to both Teacher and Student classes.

```
Tuple          subclass: #Course
         attributeNames: 'name     timeAt'
         attributeTypes: 'Symbol   Integer'
                    key: 'name'.
```

Similar to the definition of Course.

```
Person         subclass: #Teacher
         attributeNames: 'teaches    salary'
         attributeTypes: 'CourseSet  Integer'
                    key: ''.

Person         subclass: #Student
         attributeNames: 'takes      gpa'
         attributeTypes: 'CourseSet  Float'
                    key: ''.
```

Define Teacher and Student as a subclass of Person. Here we see that the key attributes for Teacher and Student are empty. This is because both Teacher objects and Student objects inherit the key of Person objects, i.e., name. In general, the key attributes inherited from a parent are appended to the list of the key attributes defined in a class. In this case, no attribute is defined as a key for Teacher, so the set of attributes comprising the key for a teacher is consists only of name which is inherited from Person. Also note that an attribute called teaches was necessary in order that the relationship between teachers and the courses could be established (the chosen scheme of representing the relationships among different objects is by no means the only one possible, but we believe it is a reasonable one).

For `Student` objects, an attribute `takes` of type `CourseSet` has been defined which will contain the `Course` objects that a student takes.

```
MySet subclass: #CourseSet
        ofType: #Course.

MySet subclass: #StudentSet
        ofType: #Student.

MySet subclass: #TeacherSet
        ofType: #Teacher.
```

These definitions define an instance of `CourseSet` to be a set containing `Course` objects, an instance of `StudentSet` to be set containing `Student` objects and an instance of `TeacherSet` to be a set of `Teacher` objects. Instances of these classes will be depositories for the objects we shall create in the database, as well as being used in setting up the relationships among objects.

```
  | teachers students courses  answer1 answer2 answer3  |
```

Declare the local variables to be used in the ensuing program segment.

```
   teachers := TeacherSet new.
   students := StudentSet new.
   courses  := CourseSet new.
```

Create the *containers* for objects.

```
   students add: ( (Student new)
                    name: #s1;
                    age:20;
                    sex:$M;
                    takes:(CourseSet new);
                    gpa: (3.5) ).

           ......
```

Create a student object, and add it to the `students` set. Note that attribute names are used by NGO to automatically define methods by the same names for instances of a class.

As an example, the methods `age` and `age:` are defined for instances of class `Person`. The first method returns the `age` attribute of the receiver, whereas the second one sets the `age` attribute to some value (of type `Integer`, or an object that is an instance of a subclass of `Integer`). These methods are inherited by subclasses of `Person`, that is why a `Student` object understands the messages given to it above.

```
teachers add: ( (Teacher new)
                 name: #t1;
                 age:37;
                 sex:$F;
                 teaches: (CourseSet new);
                 salary: 10000 ).

      ......
```

Add the teacher objects to the depository for such objects (i.e., the set `teachers`).

```
courses add: ( (Course new)
                name: #c1;
                timeAt:10 ).

      ......
```

This is similar to `Student` and `Teacher` objects.

Object creation has now been completed. Now, we define the relationships among objects.

```
((students name: #s1) takes)  add: (courses name:#c1).
((students name: #s1) takes)  add: (courses name:#c4).

 ......
```

The student whose name is `s1` takes the course with the name `c1`. Note that the method `name:` has been automatically defined for instances of class `StudentSet` since `name` is the key of a `Student` object. This method `name:anObject`, which belongs to the `students` set, returns an object that `students` contains whose `name` attribute matches `anObject`. Similarly with the set `courses`.

More generally, if a set `S1` (an instance of a subclass of `MySet`) contains objects that are instances of some class `C1` that is a descendant of `Tuple` and `C1` has a key consisting of attributes $a_1, \ldots, a_n$, then the method $a_1 : \ldots a_n :$ is defined for S_1 automatically by the system. It returns the object in S_1 whose key matches the argument of $a_1 : \ldots a_n$ (if one exists).

```
((teachers name: #t1) teaches) add: (courses name:#c1).
((teachers name: #t1) teaches) add: (courses name:#c3).

 ......
```

Teacher `t1` teaches courses `c1` and `c3` etc.

Now, we come to the queries.

Query: What courses does `s1` take that are taught by `t2` ?

```
answer1 := CourseSet new.
((teachers name: #t2)  teaches)
       do: [ :course1 |
                ((students name: #s1) takes)
                    do: [ :course2 | (course1=course2)
                                        ifTrue:
                                           [ answer1 add: course1 ]

                         ]
           ].
```

Query: Who are the students of `t1` ?

```
answer2 := StudentSet new.
students
  do:[ :aStudent |
         ((teachers name: #t1) teaches)
             do: [ :course1 |
                     (aStudent takes)
                        do: [ :course2 |
                                 (course1 =  course2)
                                     ifTrue: [ answer2 add:  aStudent
                                              ]]]].
```

Query: What other teachers teach a course at the same time that `t1` teaches a course ?

```
answer3 := TeacherSet new.
teachers do:
   [ :aTeacher|
       (aTeacher teaches) do:
          [ :course1|
               ((teachers name:#t1) teaches) do:
                  [ :course2|
                       ((course1 timeAt)=(course2 timeAt)
                            and: [(((aTeacher name)=#t1) not)
                                 ])
                        ifTrue:[answer3 add: aTeacher]]]].
```

We note that in all these queries, we had to traverse sets of objects, perform tests on objects, and place the resulting objects in an answer set. It is obvious that in all the above cases, the intended meaning of the query is not readily observable from the code implementing the query.

Solving the Problem Declaratively

In this section we give a declarative solution to the specified problem.

We first proceed with the definition of `Teacher`, `Student` and `Course` classes, as in the NGO solution. Note that the *set valued* attributes `teaches` and `takes` are no longer needed, neither do we need to define `keys`.

```
Tuple          subclass: #Person
        attributeNames: ´name   age      sex´
        attributeTypes: ´Symbol Integer  Character´.


Tuple          subclass: #Course
        attributeNames: ´name     timeAt´
        attributeTypes: ´Symbol   Integer´.


Person         subclass: #Teacher
        attributeNames: ´salary´
        attributeTypes: ´Integer´.
```

```
Person          subclass: #Student
         attributeNames: 'gpa'
         attributeTypes: 'Float'.
```

We then declare the local variables for the Smalltalk code to follow.

```
|    rb  answer1 answer2 answer3 |
```

Below is the actual code, creating objects and relationships among them.

```
rb := RuleBase new2.
```

Create an instance of a rule base that will hold the conditional rewrite rules.

```
rb addRules: '
 students({#s1}):() ->
          {  (Student new)
                  name: #s1;
                  age:20;
                  sex:$M;
                  gpa: (3.5) }.

           .......
```

Create the Student objects. Each student object is given a *tag* or an *object identifier* that identifies the object uniquely. For example, the student whose name is s1 is given the tag s1. The tags shall be used in relating objects to one another. Note that we did not have to use student names as object identifiers; we could have used any other symbol instead. But under the circumstances, using them is more convenient and suggestive of the object the tags represent.

```
 teachers({#t1}):() ->
         {  (Teacher new)
                  name: #t1;
                  age:37;
                  sex:$F;
                  salary: 10000 }.

            ......
```

Create the Teacher objects.

```
courses({#c1}):() ->
        {  (Course new)
                name: #c1;
                timeAt:10  }.

        ......
```

Create the Course objects.

We now start describing the *relationships* among objects.

```
takes({#s1}):()->{#c1}.
takes({#s1}):()->{#c4}.
......
```

Define the takes relationship. For example, the first two rules above describe the fact that Student object s1 takes Course objects c1 and c4.

```
teaches({#t1}):()->{#c1}.
teaches({#t1}):()->{#c3}.
......
```

´

Define the teaches relationship between Teacher objects and Course objects. The first two rules, for example, describe the fact that Teacher object t1 teaches Course objects c1 and c3.

This is all that is required to create objects and relationships between objects. Next, we come to the queries. This is where we shall see the real difference between stating the query *declaratively* and *imperatively* (as was the case in the last section).

Query: What courses does s1 take that are taught by t2 ?

```
answer1 := rb solveEquations:´ ( takes({#s1}) = C,
                                 teaches({#t2}) = C,
                                 courses(C)=C2 ) ´
             instantiating:´(C2)´.
```

The rule base `rb` that contains the rules making up the database is sent the message `solveEquations:instantiating:` which causes the invocation of a method that solves the equations given in the first argument and instantiates the variables specified in the second argument with the computed substitution(s). The answer returned is a `Set` of `OrderedLists` of `Terms`. Terms in this context are *object terms*.

Note in the query the use of logical variables `C` and `C2`. Logical variables always start with a capital letter in FLOOP. In the above query, we are interested only in the values that `C2` will be bound to, so that is the only variable that is specified in the argument to `instantiating`.

In plain English, we could read the query as: Find all `C2` such that `C` is (the tag of) a course taken by `s1`, `C` is (the tag of) a course taught by `t2` and `C2` is the actual course object represented by `C`.

Query: Who are the students of `t1` ?.

```
answer2 := rb solveEquations: ´( teaches({#t1})=C1,
                                 takes(S)=C1,
                                 students(S)=S2 )´
                instantiating: ´( S2 )´.
```

In plain English, the query says: Find all `S2` such that `t1` teaches `C1`, `S` takes `C1` and `S2` is the actual object represented by `S`.

Query: What other teachers teach a course at the same time that `t1` teaches a course ?

```
answer3 := rb solveEquations:´ ( teaches({#t1}) = C1,
                                 courses(C1)=C2,
                                 teaches(T)=C3,
                                 NOT(T={#t1})=true,
                                 courses(C3)=C4,
                                 MSG(C2,{#timeAt}) = MSG(C4,{#timeAt}),
                                 teachers(T)=T2
                               ) ´
            instantiating: ´(T2)´.
```

This query can be stated in English as: Find all `T2` such that `C1` is a course taught by `t1`, `C2` is the actual `Course` object represented by `C1`, another teacher `T` teaches `C3`, `T` is different from `t1`, `C4` is the `Course` object represented by `C3`, `C2` and `C4` are taught at the same time, and `T2` is the `Teacher` object represented by `T`.

Discussion

We see that in the transition from the pure object-oriented data model to a combined functional/logic/object-oriented model, a few things have happened.

First, sets as depositories of objects are no longer needed. Function names act as classifiers of objects. For example, the *set* `teachers` has been replaced by the *(multi-valued) function* `teachers`.

Explicit object identifiers permit the declarative definition of the relationship among objects. The object identifier is in the form of a *constant* in the argument of the function whose body creates the object.

There is no need to specify keys. The availability of user defined object identifiers makes them unnecessary. However, if need be, the object identifier of any (set of) object(s) can be found through a simple query. Say we want to find the object identifier of any `Course` objects that are taught at 10 o'clock. We could do this through the query

```
answer := rb solveEquations:´(courses(ObjectId) = C,
                              MSG(C,#timeAt) = 10 )´
          instantiating:´ObjectId´.
```

This query would be read as: Find all `ObjectId` such that `ObjectId` is in the argument of the function `courses`, the body of this function is `C`, and `C` responds to the message `timeAt` with the response `10`.

We believe that the clarity of the queries expressed in the combined functional/logic/object-oriented data model, in comparison to the same queries expressed in the pure object-oriented data model, is indisputable. This should be obvious from the examples presented in the previous section. We should also note that in the combined scheme, we have lost nothing of the advantages that the pure object-oriented model offers (classes, class hierarchies, inheritance, encapsulation, methods, etc.). We have taken these features, and put them in a context where declarative description of inter-object relationships and declarative posing of queries are possible.

Related Work

Deductive object-oriented database languages is an area of active research. Below we review some of the proposals and how they compare with our model.

COL (Complex Object Language) [3, 6, 5, 4] is a rule-based database language for complex objects. It is an extension of Datalog [53]. Complex objects in COL are typed trees constructed recursively using tuple and set constructors. A distinguishing characteristic of COL is the availability of *data functions*.

Lou and Ozsoyoglu [36] propose extending Horn clause logic languages with object-oriented features in the language LLO. LLO uses "meta variables as an abstract mechanism to build type/class hierarchies. Methods are defined by rules and method inheritance is achieved through typing and unification."

LOGRES [17] is a database system which supports classes of objects, with "generalization hierarchies" and object sharing. It is rule-based, extends Datalog [53] to support sets, multisets, sequences and "controlled forms of negation." It permits queries and updates through the rule-based paradigm. Object identifier unification is possible.

A different approach to incorporating object-oriented functionality into deductive databases is taken in [19, 37] where *higher-order logic*, which can be mapped into first-order logic, is proposed. These schemes are based upon extending predicate calculus with higher-order syntax which increases the modeling and programming capabilities of deductive databases.

The MOOD project, described in [10], attempts to integrate concepts of object-oriented and deductive databases by "providing powerful knowledge modeling techniques via class hierarchies of complex objects, dealing with object-bases primarily in a non-procedural, declarative way, handling large objectbases efficiently."

We have taken a different approach to the inclusion of object-oriented functionality to deductive databases. Our method can best be described as *embedding* object-oriented expressions for object creation and modification into functional/logic programs composed of conditional rewrite rules. We thus do not strive to *extend* the (declarative) functional/logic data model with object-oriented features.

We justify our approach as follows. Object-oriented programming and the object-oriented data model has certain distinguishing characteristics such as *state modification* in objects, *classes* and *class hierarchies*, *inheritance of structure and methods* and *object identity*. Trying to simulate *all* these features by *extending* a declarative logic-based model with object-oriented features (as is the case in most of the above mentioned systems) is bound to leave out at least *some* of them or implement them in an awkward and unintuitive fashion. This is because *logic based deductive* database models and the *object-oriented* model are so *fundamentally different*.

Our model then differs from all the above proposals in that it *separates* the object-oriented component from the functional/logic component and provides a natural interface between the two. Consequently, none of the functionality of the pure object-oriented model is lost, inter-object relationships are represented naturally and uniformly using conditional rewrite rules, and queries are posed *declaratively* in the form of *equations to be solved.*

Conclusion and Future Research Directions

We described a combined functional/logic/object-oriented database model that provides for the declarative description of inter-object relationships through conditional rewrite rules and permits queries to be posed declaratively in the form of a set of equations to be solved.

We demonstrated the utility of our approach in an example. Our example made clear that in the pure object-oriented model, the necessity to represent inter-object relationships with pointer valued attributes makes the *representation of inter-object relationships* and *posing of queries* difficult to write and understand. On the other hand, representing inter-object relationships using conditional rewrite rules and posing queries declaratively in the form of equations to be solved was seen to be very natural and convenient.

Our system at this point is a research tool, designed to demonstrate the workability of our idea. As such, it lacks many of the features needed in a full featured database system (persistent objects, concurrency control, integrity constraints, schema evolution, etc.). Any future work would include the incorporation of at least some of these features in our system without changing the declarative nature of the representation of inter-object relationships and declarative posing of queries.

CHAPTER

CONCLUSION AND FUTURE RESEARCH DIRECTIONS

The accomplishments of the work reported in this dissertation can be broadly divided into two parts: those relating to the development of multi-paradigm programming languages, and those relating to the development of database models. On the programming language side, we reported work on two programming languages, ROSE and FLOOP. ROSE integrates functional and logic programming styles, and FLOOP integrates the functional, logic and object-oriented programming styles. On the database side, we developed fully a functional/logic data model, called *DataFunLog*, which subsumes the functional and logic data models. We then described how a pure object-oriented database system, in unison with FLOOP, can yield a deductive object-oriented data model which permits declarative description of inter-object relationships and declarative posing of queries in the form of equations to be solved.

The language ROSE, described in chapter 2, introduced the innovative notion of using optional guards in rewrite rules, which, together with non-determinism and sequential execution of programs, makes functional/logic programming practical. This is due to the ability of the programmer to specify *control* information, using committing guards, in his program. The availability of committing guards in ROSE can be likened to the availability of the *cut* predicate in Prolog. Other features of the language include a choice of *lazy* or *eager* evaluation, which can be specified at the term level by the programmer (i.e., individual subterms of a term can be marked for either eager or lazy evaluation), higher-order features, including nameless functions in the form of *conditional lambda abstractions*, and a module system for organizing a large program into more manageable units.

DataFunLog (DFL) is a functional/logic data model. DFL subsumes the functional and logic data models. It permits the definition of deductive functional/logic databases using conditional rewrite rules. The availability of the *logic variable* and function composition makes the posing of very natural and expressive queries possible. In chapter 3 we developed the DFL data model fully and gave a formal semantics for it based on conditional term rewriting. A significant advance in the DFL model was the development of a query answering algorithm that terminates even for non-terminating term rewriting systems that make up the database.

In chapter 4, we described the functional/logic/object-oriented programming language FLOOP, which brings complex objects to the realm of symbolic manipulation. Many elegant functional/logic programs can then be used to manipulate complex objects, instead of just simple symbols. The operational semantics of FLOOP is e-unification and narrowing through *transformations*. This approach eliminates the need for an explicit term unification algorithm, and is relatively easy to implement. In addition, an object-expression evaluator (in this case, the underlying Smalltalk interpreter) is used to evaluate object expressions. FLOOP then is a very high level programming language which permits symbolic manipulation of complex objects without compromising any of the features of the object-oriented paradigm. As such, we expect that it will find use in artificial intelligence and rapid prototyping of complex software systems.

Next Generation Opal (NGO), described in chapter 5, is a pure but minimal object-oriented database system. It permits the definition of a type structure for objects and their attributes. In combination with FLOOP, it gives a deductive object-oriented database model, which permits the formulation of queries declaratively in the form of equations to be solved, and the description of inter-object relationships using conditional rewrite rules.

The research reported in this dissertation can be followed up in many ways. In the area of databases, investigations into how the query evaluation algorithm in DFL can be improved to avoid unnecessary computation in answering a query would yield an algorithm more amenable for actual implementation. For the functional/logic/object-oriented data model developed in chapter 5, more research is needed to determine how features like concurrency control, integrity constraints, schema evolution, etc., can be incorporated into the model without changing the declarative nature of the representation of inter-object relationships and declarative posing of queries. In the area of programming languages, we believe that ROSE has reached a certain maturation point and we do not foresee much possibility for further significant improvement in the language. For FLOOP, on the other hand, it would be worth investigating parallel execution, since the transformation algorithm implicitly generates a tree, whose branches can be traversed in parallel. Parallel execution of ROSE programs, in principle, is possible, but would be complicated by the presence of committing guards.

Bibliography

[1] *The Ada Programming Language Reference Manual.* United States Department of Defense, Washington, D. C., 1983.

[2] The Smalltalk/V tutorial and programming handbook. Digitalk Inc., 1986.

[3] Serge Abiteboul. Towards a deductive object-oriented database language. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases.* North Holland, 1989.

[4] Serge Abiteboul and Stephane Grumbach. COL: A logic-based language for complex objects. In Francois Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages*, pages 347–374. ACM Press, 1990.

[5] Serge Abiteboul and Stephane Grumbach. A rule-based language with functions and sets. *ACM Transactions on Database Systems*, 16(1):1–30, 1991.

[6] Serge Abiteboul, Stephane Grumbach, Agnes Voisard, and Emmanuel Waller. An extensible rule-based language with complex objects and data-functions. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 298–314, 1989.

[7] Hassan Ait-Kaci. An overview of LIFE. In *Proceedings of the First International Database Workshop*, pages 42–48. Springer-Verlag, LNCS 504, 1991.

[8] A.M. Alashqur, S.Y.W. Su, and H. Lam. A rule-based language for deductive object-oriented databases. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 58–67, 1991.

[9] Francois Bancilhon et al. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth Principles of Database Systems Symposium*, pages 1–5, 1986.

[10] R. Bayer. MOOD: a knowledge-base system with object-oriented deduction. In *Database Systems For Advanced Applications*, pages 320–329. 1991.

[11] Zeki O. Bayram and Barrett R. Bryant. Conditional term rewriting as a deductive database language. In *Proceedings of the Deductive Databases Workshop, 1992 Joint International Conference and Symposium on Logic Programming*, pages 126–135, 1992.

[12] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth Principles of Database Systems Symposium*, pages 269–283, 1987.

[13] G. M. Birtwistle, O-J Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA Begin*. Van Nostrand Reinhold, 1979.

[14] P.G. Bosco and E. Giovannetti. IDEAL: An Ideal Deductive Applicative Language. In *Proceedings of the 1986 Symposium on Logic Programming*, pages 89–94, 1986.

[15] Barrett R. Bryant and Aiqin Pan. Two-level grammar: A functional/logic query language for database and knowledge-base systems. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 78–83. Springer-Verlag, LNAI 624, 1992.

[16] P. Butterworth, A. Otis, and J. Stein. The Gemstone object database management system. *Communications of the ACM*, 34(10):64–77, 1991.

[17] F. Cacace, S.Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modelling with a rule-based programming paradigm. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 225–236, 1990.

[18] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions On Database Systems*, (2):162–207, 1990.

[19] Weidong Chen, Michael Kifer, and David S. Warren. Hilog as a platform for database languages. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 315–329, 1989.

[20] Jan O. Chomicki. *Functional Deductive Databases: Query Processing in the Presence of Limited Function Symbols*. PhD thesis, Rutgers, the State University of New Jersey, 1990.

[21] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer-Verlag, 1987.

[22] D. DeGroot and G. Lindstrom, editors. *Logic Programming: Functions, Equations, and Relations*. Prentice-Hall, 1985.

[23] N. Dershowitz and M. Okada. Conditional equational programming and the theory of conditional term rewriting. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 337–346, 1988.

[24] N. Dershowitz and D. Plaisted. Logic programming cum applicative programming. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 54–67, 1985.

[25] M. Fay. First order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167. Springer-Verlag, 1979.

[26] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 172–184, 1985.

[27] J.H. Gallier and W. Snyder. A general complete E-Unification procedure. In *Rewriting Techniques and Applications*, pages 216–227. Springer-Verlag, LNCS 256, 1987.

[28] E. Giovannetti and C. Moiso. A completeness result for E-Unification algorithms based on conditional narrowing. In *Foundations of Logic and Functional Programming Workshop*, pages 157–167. Springer-Verlag, LNCS 306, 1986.

[29] J. Grant and J. Minker. Deductive database theories. *Knowledge Engineering Reviews*, (4):267–304, 1989.

[30] P. M. D. Gray. Combining functional and logic programming in expert database systems. In Keith G. Jeffery, editor, *Expert Database Systems*, pages 55–82. Academic Press, 1992.

[31] Steffen Holldobler. Conditional equational theories and complete sets of transformations. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 405–412, 1988.

[32] J. M. Hullot. Canonical forms and unification. In *Proceedings of the Fifth Conference on Automated Deduction*, pages 318–334. Springer-Verlag, LNCS 87, 1980.

[33] Keith G. Jeffery. The expert database jungle. In Keith G. Jeffery, editor, *Expert Database Systems*, pages 1–28. Academic Press, 1992.

[34] Giorgio Levi et al. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 318–327, 1987.

[35] W. L. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[36] Yanjun Lou and Z. Meral Ozsoyoglu. LLO: an object-oriented deductive language with methods and method inheritance. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 198–207, 1991.

[37] Sanjay Manchanda. Higher-order logic as a data model. In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 330–341, 1989.

[38] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[39] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. Technical Report SRI-CSL-92-08, SRI, July 1992.

[40] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[41] J. Minker, editor. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufman, 1988.

[42] Sitansu S. Mittra. *Principles of Relational Database Systems.* Prentice Hall, 1991.

[43] Alexandra Poulovassilis and Carol Small. A functional programming approach to deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 491–500, 1991.

[44] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, (3):189–216, 1991.

[45] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 138–151, 1985.

[46] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[47] Frank S. K. Silberman and Baharat Jayaraman. A domain-theoretic approach to functional and logic programming. Technical Report TUTR 91-109, Tulane University, 1991.

[48] A. U. Sinduh. Object-oriented programming cuts development time, boosts reliability. *Industrial and Process Control Magazine*, 64(3):53–55, 1991.

[49] Wayne Snyder. Higher order E-Unification. Technical Report BU-CS TR 90-008, Boston University, 1990.

[50] Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, (8):101–140, 1989.

[51] Wayne Snyder and Christopher Lynch. An inference system for horn clause logic with equality: A foundation for conditional E-Unification and for logic programming in the presence of equality. Technical Report BU-CS TR 90-014, Boston University, 1990.

[52] David A. Turner. An overview of Miranda. In David A Turner, editor, *Research Topics in Functional Programming*, pages 1–16. Addison-Wesley, 1990.

[53] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1.* Computer Science Press, 1988.

[54] Akihiro Yamamoto. Completeness of extended unification based on basic narrowing. In *Proceedings of the Logic Programming Conference '88*, pages 19–28, 1988.

APPENDIX A

SAMPLE ROSE PROGRAMS

A Predened Functions

```
module system.
public.

goal call( op(253,xfy,or) ).
goal call( op(252,xfy,and) ).
goal call( op(250,xfy,´:=´) ).

true and true # true  -> true.
X and Y  -> false.

false or false # true -> false.
X or Y -> true.

not true # true -> false.
not false # true -> true.

if(C,A,B) # C  -> A.
if(C,A,B) # true -> B.

X > Y #  call((eval(2,X,X2),eval(2,Y,Y2),X2>Y2)) -> true.
X > Y -> false.

X < Y #  call( (eval(2,X,X2),eval(2,Y,Y2),X2<Y2)) -> true.
X < Y -> false.

X = Y #  call( (eval(2,X,X2),eval(2,Y,Y2),X2=Y2) ) -> true.
X = Y -> false.

X := Y : call(var(X)) and X eq Y -> X.

X + Y #  call( (eval(2,X,X2),eval(2,Y,Y2), Z is X2+Y2) ) -> Z.
X - Y #  call( (eval(2,X,X2),eval(2,Y,Y2), Z is X2-Y2) ) -> Z.
X * Y #  call( (eval(2,X,X2),eval(2,Y,Y2), Z is X2*Y2) ) -> Z.
X / Y #  call( (eval(2,X,X2),eval(2,Y,Y2), Z is X2/Y2) ) -> Z.
X // Y  # call( (eval(2,X,X2),eval(2,Y,Y2), Z is X2 // Y2 ) ) -> Z.
```

```
X mod Y #  call( (eval(2,X,X2),eval(2,Y,Y2),Z is X2 mod Y2) ) -> Z.

writeln(X) : X2 eq X and call(write(X2)) and call(nl) -> X.
readln: call( read(X) ) and call( nl ) -> X.
```

A User Dened Modules and Functions

```
/* Othello programs should all follow the following format !! */

/* the root module is defined to have access to all other modules */

module root.

uses system.
uses test_cut.
uses list_functions.
uses numbers.
uses sorter.
uses arithmetic.
uses split_function.
uses distribute.
uses number_generator.
uses higher.
uses lazy_functions.
uses foldr.
```

```
module foldr.
uses system.
uses list_functions.

public.

list_sum -> foldr( plus , 0 ).
list_product -> foldr( times , 1 ).
list_append -> foldr( app , [] ).
list_and -> foldr( and , true ).
list_or -> foldr( or, false ).

private.

foldr(OP,IDENTITY,[]) # true -> IDENTITY .
foldr(OP,IDENTITY,[H|T]) -> apply( OP , H , foldr(OP,IDENTITY,T) ).

times(X,Y) -> X*Y.
plus(X,Y) -> X+Y.
```

```
module higher.
uses system.

public.

add1(X) -> X+1.

add(X,Y) -> X+Y .   /* add(3) is a function that adds 3 to its argument !!*/

twice(F,A) -> apply( F , apply( F, A )).

map(F,[]) # true -> [].
map(F,[H|T]) -> [ apply(F,H) | map(F,T) ].


/* e.g. try  >> map( app([a]) , [ [1] , [2] ] ) */

first_n_primes(N)
          -> [2 | primes(N,
                          (lambda,[X],true,((X mod 2) eq 0 ) ),
                          3 ) ] .



primes(1,Filter,Next) # true -> [].

primes( Remaining,  Filter , Next ) # not apply(Filter,Next) ->
 [ Next | primes( Remaining-1,
                 (lambda,[X],true,((X mod Next) eq 0) or apply(Filter,X) ) ,
                      Next + 1 ) ].

primes( Remaining, Filter , Next ) -> primes(Remaining, Filter, Next+1).
```

```
module lazy_functions.
uses system.

public.

first_n_items(0,[H|T]) # true -> [].
first_n_items(N,[H|T]) -> [H | first_n_items( N-1 , T ) ] .

primes -> list_of_primes(2).

private.

if(C,A,B) # C -> A.
if(C,A,B) -> B.

list_of_primes(N) -> [ N | filter( N, list_of_primes(N+1) ) ].

filter( N , [H|T] ) -> if( ((H mod N) eq 0) ,
                               filter(N,T),
                               [ H | filter(N,T) ] ).
```

```
module list_functions.

uses system.

public.

app([],X) -> X.
app([H|T],L) -> [H|app(T,L)].

reverse([])  -> [].
reverse([H|T]) -> app(reverse(T),[H]).

member(X,[]) # true -> false.
member(X,[X|Y]) # true -> true.
member(X,[H|T]) -> member(X,T).


palindrome(L) # L equals reverse(L) -> true.
palindrome(L) -> false.

revstrings(A,B) : A lazy_eq app(X,app([H|T],Z))  and
                  B lazy_eq app(XX,app(reverse([H|T]),ZZ)) -> [H|T].

mapfun(F,[]) # true -> [].
mapfun(F,[H|T]) -> [apply(F,H)|mapfun(F,T)].



divide(L,2) # app(X,Y) eq L -> [X,Y].
divide(L,N) : divide(L,N-1) eq [H|T]
                  -> app( divide(H,2),T).


divide_x(L,2) # app(X,Y) eq L -> [X,Y].
divide_x(L,N) : app(X,Y) eq L -> [X| divide_x(Y,N-1)].


/* useful for finding repeated words, for example !!! */
doubles(L) : divide_x(L,4) eq [X,[A|B],[A|B],Y] -> [A|B].
```

```
module test_cut.
uses system.

public.
a1(X,Y) #  cut( b(X) and c(Y) ) -> d(X,Y).
a1(X,Y) -> e(X,Y).
a2(X,Y) #  b(X) and c(Y)  -> d(X,Y).
a2(X,Y) -> e(X,Y).

b(1) -> true.
b(2) -> true.
b(3) -> true.
c(1) -> true.
c(2) -> true.
c(3) -> true.

d(X,Y) -> true.

e(e1,e1) -> true.
e(e2,e2) -> true.
```

```
module arithmetic.

uses system.

public.

fact( 1 ) # true -> 1.
fact( X ) -> X * fact(X-1).

fib(1) #  true -> 1.
fib(2) # true -> 1.
fib(N) -> fib(N-1) + fib(N-2).
```

```
module split_function.

uses list_functions.
uses system.

public.

split( Atom, D) : app(First, app([D], app([H|T], app([D],Last) ) ) )
                       lazy_eq
                   app([D], app( atom_to_list(Atom) ,[D])) and
                    not member(D,[H|T])
                  -> list_to_atom([H|T]).


private.
atom_to_list(A) : call(name(A,L)) -> L.
list_to_atom(L) : call(name(A,L)) -> A.
```

```
module distribute.

public.
distribute([]) # true -> [].
distribute([H|T]) # call(atomic(H)) -> [H|distribute(T)].
distribute([H|T]) -> [some_member(H)|distribute(T)].


private.

some_member([H|T]) -> H.
some_member([H|T]) -> some_member(T).


public.
permute([]) # true -> [].
permute([H|T]) -> insert(H,permute(T)).

private.
insert(X,L) -> [X|L].
insert(X,[H|T]) -> [H|insert(X,T)].
```

```
module sorter.

uses system.

public.
isort([]) # true -> [].
isort([H|T]) -> insert(H,isort(T)).

private.
insert(X,[]) # true -> [X].
insert(X , [H|T] ) # X<H  -> [X,H|T].
insert(X, [H|T] ) -> [H|insert(X,T)].

public.
uses list_functions.
constructor pair.

qsort([]) # true -> [].
qsort([H|T]) : partition(H,T) eq pair(Lower,Higher) ->
                app( qsort(Lower), [H | qsort(Higher)] ).


private.
partition(H,[]) # true -> pair([],[]).
partition( Atom , [F|R] ) # F < Atom  and
                           partition(Atom,R) eq pair(L,H)
                           ->
                           pair([F|L],H).

partition( Atom, [F|R] ) : partition(Atom,R) eq pair(L,H)
                          ->
                          pair(L,[F|H]).
```

```
module number_generator.

uses system.

public.

/* increasing */
numbers(X,Y) : X<Y or X=Y -> X.
numbers(X,Y) : X < Y -> numbers(X+1,Y).

/* decreasing */
numbers2(X,Y) : X < Y -> numbers2(X+1,Y).
numbers2(X,Y) : X<Y or X=Y -> X.
```

```
module numbers.

/* O.K. NOW !!! let us represent positive numbers by s(s(..)) and negative
   numbers by p(p(..)) and NO EXCEPTIONS !! WE will use the FUNCTIONS prev
   and succ for previous and successor !!!!! */

constructor s.
constructor p.
constructor pair.

prev(0) # true -> p(0).
prev(s(X)) # true -> X.
prev(p(X)) # true -> p(p(X)).

succ(0) # true -> s(0).
succ(p(X)) # true -> X.
succ(s(X)) # true -> s(s(X)).


Y + 0 # true -> Y.
0 + Y # true -> Y.
s(X) + s(Y)  # true -> succ(succ( X+Y ) ).
s(X) + p(Y)  # true ->   X+Y.
p(X) + s(Y)  # true ->  X+Y.
p(X) + p(Y)  # true ->  prev( prev(  X + Y ) ).


0 * Y # true -> 0.
Y * 0 # true -> 0.
s(X) * s(Y) # true ->  X * s(Y) +  s(Y).
s(X) * p(Y) # true ->  neg( s(X) * neg(p(Y)) ).
p(X) * s(Y) # true ->  neg( neg(p(X)) * s(Y) ).
p(X) * p(Y) # true ->  neg(p(X)) * neg(p(Y)) .

neg(0) # true -> 0.
neg(p(X)) # true -> s(neg(X)).
neg(s(X)) # true -> p(neg(X)).
```

APPENDIX B

SAMPLE FLOOP PROGRAMS

```
" File: program.0a
  The functions append and reverse "

| rb answer list0 list1 list2 list4|
rb := RuleBase new2.

rb addConstructor:´c´.

rb addRules: ´
    app(nil,X): () -> X.
    app(c(H,T),X) : () -> c(H,app(T,X)).
    rev( nil ): () -> nil .
    rev( c(H,T) ): () -> app( rev(T), c(H,nil) ).
    list0():()->c(a1,c(b1,nil)).
    list1():()->c(a2,c(b2,nil)) ´.



answer := rb  solveEquations: ´( app(X,Y) = rev(list1()) )´
              instantiating: ´(X,Y)´.

^ answer.
```

```
"File: program.3a
 insertion sort algorithm"

| rb answer |
rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´

 iSort(nil):()->nil.
 iSort(c(H,T)):()->insert(H,iSort(T)).

 insert(X,nil):()->c(X,nil).
 insert(X,c(H,T)):(MSG(X,{#<},H)={true})->c(X,c(H,T)).
 insert(X,c(H,T)):(MSG(X,{#>=},H)={true})->c(H,insert(X,T)).

 list1():()->c({4}, c({2}, c({5}, c({3},nil) ) ) )

 ´.

answer := rb solveEquations: ´ ( X = iSort(list1()) ) ´
              instantiating: ´(X)´.

^ answer.
```

```
"File: program.4a
 The higher order function foldr"

| rb answer |
rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´

 foldr(nil, Message, Default):()->Default.

 foldr( c(H,T), Message, Default ):
                  (R=foldr(T,Message,Default)) -> MSG(H,Message,R).


 list1():()->c({4}, c({2}, c({5}, c({3},nil) ) ) )

´.

answer := rb solveEquations:´ ( Z = foldr(list1(),{#+},{0}))´
              instantiating:´ (Z) ´.

^ answer.
```

```
" File: program.5a
  Testing the ability to transfer answers into the underlying smalltalk
  interpreter so that F/L part can be used inside a Smalltalk program.
"


| rb answer |
rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRule:  ´f({1}):()->{10}´.

rb addRule:  ´f({2}):()->{20}´.


rb addRule:  ´g({10}):()->{100}´.

rb addRule:  ´g({20}):()->{200}´.



answer := rb solveEquations:´ ( Z=f(X), W=g(Z))´.

^ answer getObjects inspect.
```

```
"File: program.6a
 The  permute function"

| rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´

  permute(nil):()->nil.
  permute(c(H,T)):()->insert(H,permute(T)).

  insert(X,L):()-> c(X,L).
  insert(X,c(H,T)):()->c(H,insert(X,T))

´.

answer := rb solveEquations:´ (  Z = permute(c({0},c({1},c({2},nil)) ) ) )´
              instantiating:´ (Z) ´.

^ answer .
```

```
"File: program.7a
 The higher order function ´map´ "

| rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.



rb addRules: ´
 map(M,nil):()->nil.
 map(M,c(H,T)):()->c( MSG(H,M), map(M,T))

´.

answer := rb solveEquations:´(Z=map({#yourself},c({0},c({1},c({2},nil)))))´
              instantiating:´ (Z) ´.


^ answer .
```

```
"File: program.8a
 A variation on the higher-order ´map´ function"

| rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´

 map1(Block,nil):()->nil.
 map1(Block,c(H,T)):()->c( MSG(Block,{#value:},H),
                                        map1(Block,T))
´.


answer := rb solveEquations:
             ´(Z=map1( {[:x|x+1]} , c({0},c({1},c({2},nil)))))´
            instantiating: ´( Z )´.

^ answer .
```

```
"File: program.9a
 Set2 is a subclass of Set. The ´add:´ method returns the set
 receiving the message rather than the object added to the set as is
 the case with a ´Set´. This program shows the copying that is carried
 out when a logical variable is bound to an object. The ´single
 assignment´ rule is not compromized by state changing objects.
 Hence in order that state change may be possible, the state needs to
 be propagated with each method  returning ´self´ after a certain set
 of actions. Then we have the following restriction on methods: If a method
 causes a state change, it MUST return ´self´. If it returns something
 else, it must NOT change the state of the object!!! "


| rb answer |

rb := RuleBase new2.
rb addConstructor: ´c´.

rb addRule: ´ f(X):( MSG(X,{#add:},{1})=T1 ,
                     MSG(T1,{#add:},{2})=T2 ,
                     MSG(T2,{#add:},{3})=T3 )
               ->  MSG(T3,{#select:}, {[:e|(e=2) not]}) ´.


answer := rb solveEquations:  ´ (Z= f( {Set2 new} ) )´
              instantiating: ´( Z )´.

^ answer .
```

```
"File: program.10a
 This program tests the use of the ´exposure´ transformation in the
 operational  semantics of FLOOP"


| rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´  f(c(X)):()->a.
                g(X):()->a ´.


answer  := rb solveEquations: ´( X=c(f(X)), Y=c(g(Y)) )´
               instantiating: ´ (X,Y) ´.
^ answer
```

```
"File: program.11a
 This program tests nested messages. "


| rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´  id(X):()->X.
                f():()->{#add:}.
                g():()->{4}
             ´.


answer  := rb solveEquations:
                  ´( X={Set2 new},
                     Y1=MSG( X,id( MSG( f(),{#yourself}
                                      )
                                 ),
                           id( g() )
                         )
                   )´
              instantiating: ´(X,Y1)´.

^ answer
```

```
"File: program.12a
 Exploring database applications "


Tuple          subclass: #Person
         attributeNames: ´name age sex´
         attributeTypes: ´String Integer Character´.

 | rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´

  somebody(p1):() -> {(Person new) name:#zeki;
                                   age:27;
                                   sex:$M }.

  somebody(p2):()-> {(Person new) name:#george;
                                  age:128;
                                  sex:$M }.


  oldfolks():( somebody(ID)=P,
             MSG( MSG(P,{#age}),
                  {#>},
                  {125} ) = {true} )
                       -> P

´.

answer  := rb solveEquations:
                ´(  oldfolks() = Y )´
             instantiating: ´(Y)´.

^ answer
```

```
"File: program.13a
 Exploring database applications, getting real!!!
 This program does not use the function NOT.
 Does use sets! "


City removeFromSystem.
Cities removeFromSystem.

Tuple          subclass: #City
         attributeNames: ´name population´
         attributeTypes: ´Symbol Integer´.

MySet subclass: #Cities
      ofType: #City.

| rb answer |

rb := RuleBase new2.

rb addConstructor: ´c´.


rb addRules: ´
  city({#c1}):()->{ (City new)  name:#Miami;
                                  population: 1600 }.

  city({#c2}):()->{ (City new)  name:#Birmingham;
                                  population: 1000 }.

  city({#c3}):()->{ (City new)  name:#Washington;
                                  population: 2000 }.


  dist({#c1},{#c2}):()->{10}.
  dist({#c2},{#c3}):()->{20}.

  eq(X,Y):()->MSG(X,{#=},Y).

  member(X,c(X,Z)):()->{true}.
  member(X,c(Y,Z)):()->member(X,Z).

  comp(A,B):()->
        computeDistance(MSG({Cities new},{#add2:},city(A)), A, B).

 computeDistance(S,A,B):(dist(A,B)=D)->D.
```

```
computeDistance(S,A,B):
        ( dist(A,C)=D,
          MSG(S,{#includes:},city(C))={false},
          computeDistance( MSG(S,{#add2:},city(C)),C,B)=E )
         -> MSG(D,{#+},E).


 bigCity(X):( MSG( MSG(X,{#population}),
                           { #> },
                           { 1500 }
                         ) = {true}
                    )
                    -> MSG(X,{#name}) .


bigCities():(city(X)=Y)-> bigCity(Y)

´.

answer  := rb solveEquations:
                ´( W=comp({#c1},{#c3}))´
             instantiating: ´(W)´.

^ answer
```

```
"File: program.14a
 Testing negation as finite failure!!!
 Also solving  shortest path problem. "

Tuple        subclass: #City
       attributeNames: ´name population´
       attributeTypes: ´Symbol Integer´.

| rb answer1 answer2 aStream smallestDistance aNumber |

rb := RuleBase new2.

rb addConstructor: ´c´.

rb addRules: ´

 city(a):()->{ (City new)  name:#Atlanta;
                   population: 1600 }.

 city(b):()->{ (City new)  name:#Birmingham;
                    population: 1000 }.

 city(c):()->{ (City new)  name:#Cambridge;
                    population: 3000 }.

 city(d):()->{ (City new)  name:#Denver;
                     population: 200 }.

 dist(a,b):()->{10}.
 dist(a,d):()->{50}.
 dist(b,c):()->{20}.
 dist(b,d):()->{30}.
 dist(c,d):()->{5}.

 twoWayDist(X,Y):()->dist(X,Y).
 twoWayDist(X,Y):()->dist(Y,X).

 eq(X,Y):()->MSG(X,{#=},Y).

 member(X,nil):()-> false.
 member(X,c(X,Z)):()->true.
 member(X,c(Y,Z)):( NOT(X=Y)=true )->member(X,Z).

 comp(Name1, Name2):( city(A)=C1,
                     MSG(C1,{#name}) = Name1,
                     city(B)=C2,
```

```
                      MSG(C2,{#name}) = Name2
                    ) ->  computeDistance( c(A,nil), A, B).

 computeDistance(L,A,B):()->twoWayDist(A,B).
 computeDistance(L,A,B):
               (
                 twoWayDist(A,C)=D,
                 NOT(C=B)=true,
                 member(C,L)=false,
                 computeDistance(c(C,L),C,B)=E
               )
               -> MSG(D,{#+},E).

 bigCity(X):( MSG( MSG(X,{#population}),
                   { #> },
                   { 1500 }
                 ) = {true}
                        )
                        -> MSG(X,{#name}).

 bigCities():(city(X)=Y)-> bigCity(Y)

´.

answer1  := rb solveEquations:
                         ´( W=comp({#Denver},{#Atlanta}))´
              instantiating: ´(W )´.

answer2 := rb solveEquations:´( Y = bigCities() )´
               instantiating:´ ( Y )´.


answer1 do:
     [ :anOrderedCollection |
              aStream := ReadStream on: anOrderedCollection.
              aNumber := (aStream peek) object.
              (smallestDistance isNil)
                    ifTrue: [ smallestDistance := aNumber ]
                    ifFalse: [ (smallestDistance > aNumber)
                                 ifTrue: [ smallestDistance := aNumber ]
                             ]
     ].

^ smallestDistance
```

```
" File: program.Ob
  This example demonstrates that describing external relationships
  between objects internally through sets is very cumbersome.
"

| t |
"clean up previous classes"
#( Student  Teacher Person   Course IntSet StudentSet TeacherSet
   PersonSet  CourseSet ) do:
  [ :aSymbol | t:= ( Smalltalk at: aSymbol ifAbsent: [nil] ).
               (t isNil) ifFalse: [ t removeFromSystem ] ].


Tuple          subclass: #Person
        attributeNames: ´name   age      sex´
        attributeTypes: ´Symbol Integer  Character´
                   key: ´name´.

Tuple          subclass: #Course
        attributeNames: ´name     timeAt´
        attributeTypes: ´Symbol   Integer´
                   key: ´name´.

Person         subclass: #Teacher
        attributeNames: ´teaches    salary´
        attributeTypes: ´CourseSet  Integer´
                   key: ´´.

Person         subclass: #Student
        attributeNames: ´takes      gpa´
        attributeTypes: ´CourseSet  Float´
                   key: ´´.



MySet subclass: #CourseSet
       ofType: #Course.

MySet subclass: #StudentSet
       ofType: #Student.

MySet subclass: #TeacherSet
       ofType: #Teacher.
```

```
 | teachers students courses aTeacher aStudent aCourse
  answer1 answer2 answer3 answer4 temp1 temp2 |
" first need to add the objects themselves, then fill in the attributes
  that determine relationships "


  teachers := TeacherSet new.
  students := StudentSet new.
  courses  := CourseSet new.


  students add: ( (Student new)
                    name: #s1;
                    age:20;
                    sex:$M;
                    takes:(CourseSet new);
                    gpa: (3.5) ).

  students add: ( (Student new)
                    name: #s2;
                    age:16;
                    sex:$F;
                    takes:(CourseSet new);
                    gpa: (4.0) ).

  students add: ( (Student new)
                    name: #s3;
                    age:22;
                    sex:$F;
                    takes:(CourseSet new);
                    gpa: (3.8) ).


  students add: ( (Student new)
                    name: #s4;
                    age:25;
                    sex:$M;
                    takes:(CourseSet new);
                    gpa: (3.6) ).

  teachers add: ( (Teacher new)
                    name: #t1;
                    age:37;
                    sex:$F;
                    teaches: (CourseSet new);
                    salary: 10000 ).
```

```
teachers add: ( (Teacher new)
                  name: #t2;
                  age: 30;
                  sex:$M;
                  teaches: (CourseSet new);
                  salary: 12000 ).

teachers add: ( (Teacher new)
                  name: #t3;
                  age: 26;
                  sex:$F;
                  teaches: (CourseSet new);
                  salary: 32000 ).

courses add: ( (Course new)
                  name: #c1;
                  timeAt:10 ).

courses add: ( (Course new)
                  name: #c2;
                  timeAt:12 ).

courses add: ( (Course new)
                  name: #c3;
                  timeAt:11 ).

courses add: ( (Course new)
                  name: #c4;
                  timeAt:10 ).

" object creation completed, now the relationships "

" s1 takes c1 etc.."
((students name: #s1) takes)  add: (courses name:#c1).
((students name: #s1) takes)  add: (courses name:#c4).

((students name: #s2) takes)  add: (courses name:#c2).

((students name: #s3) takes)  add: (courses name:#c2).
((students name: #s3) takes)  add: (courses name:#c3).

((students name: #s4) takes)  add: (courses name:#c3).


((teachers name: #t1) teaches) add: (courses name:#c1).
((teachers name: #t1) teaches) add: (courses name:#c3).
```

```
((teachers name: #t2) teaches) add: (courses name:#c2).
((teachers name: #t2) teaches) add: (courses name:#c4).

((teachers name: #t3) teaches) add: (courses name:#c4).


"  what courses does #s1 take that are taught by #t2 ? "
answer1 := CourseSet new.
((teachers name: #t2)  teaches)
       do: [ :course1 |
                ((students name: #s1) takes)
                    do: [ :course2 | (course1=course2)
                                        ifTrue:
                                           [ answer1 add: course1 ]

                        ]
           ].




"whether #t1  teaches a course in the same time slot as #t2 ?"

answer2 := false.
((teachers name: #t1) teaches)  do:
  [:aClass1 |
      ((teachers name: #t2) teaches) do:
         [ :aClass2 |
              (  (aClass1 timeAt) =  (aClass2 timeAt))
                   ifTrue: [ answer2 := true ]  ]] .




" all students of #t1 "
answer3 := StudentSet new.

students
  do:[ :aStudent |
         ((teachers name: #t1) teaches)
             do: [ :aCourse1 |
                    (aStudent takes)
                       do: [ :aCourse2 |
                                (aCourse1 =  aCourse2)
                                    ifTrue: [ answer3 add:  aStudent
                                            ]]]].
```

```
" what other teachers teach a course at the same time that
  #t1 teaches a course ? "

  answer4 := TeacherSet new.

 teachers do:
    [ :aTeacher|
        (aTeacher teaches) do:
           [ :aCourse1|
                ((teachers name:#t1) teaches) do:
                   [ :aCourse2|
                        ((aCourse1 timeAt)=(aCourse2 timeAt)
                             and: [(((aTeacher name)=#t1) not)
                                  ])
                         ifTrue:[answer4 add: aTeacher]]]].

^ answer4 printString
```

```
" File: program.1b
  This program demonstrates that  describing external relationships
  between objects declaratively using conditional rewrite rules is very
  convenient, compared to using sets of pointers to represent the same
  relationships
"

Tuple          subclass: #Person
        attributeNames: 'name   age      sex'
        attributeTypes: 'Symbol Integer  Character'.

Tuple          subclass: #Course
        attributeNames: 'name     timeAt'
        attributeTypes: 'Symbol   Integer'.

Person         subclass: #Teacher
        attributeNames: 'salary'
        attributeTypes: 'Integer'.

Person         subclass: #Student
        attributeNames: 'gpa'
        attributeTypes: 'Float'.

  |    rb  answer1 answer2 answer3 answer4 |

  rb := RuleBase new2.

  rb addRules: '
   students({#s1}):() ->
            {  (Student new)
                    name: #s1;
                    age:20;
                    sex:$M;
                    gpa: (3.5) }.

   students({#s2}):() ->
           {  (Student new)
                    name: #s2;
                    age:16;
                    sex:$F;
                    gpa: (4.0) }.
```

```
 students({#s3}):() ->
         {  (Student new)
                  name: #s3;
                  age:22;
                  sex:$F;
                  gpa: (3.8) }.

students({#s4}):() ->
         {  (Student new)
                  name: #s4;
                  age:25;
                  sex:$M;
                  gpa: (3.6)  }.

teachers({#t1}):() ->
         {  (Teacher new)
                  name: #t1;
                  age:37;
                  sex:$F;
                  salary: 10000 }.

teachers({#t2}):() ->
         {  (Teacher new)
                  name: #t2;
                  age: 30;
                  sex:$M;
                  salary: 12000  }.

teachers({#t3}):() ->
         {  (Teacher new)
                  name: #t3;
                  age: 26;
                  sex:$F;
                  salary: 32000  }.

courses({#c1}):() ->
         {  (Course new)
                  name: #c1;
                  timeAt:10  }.

courses({#c2}):() ->
         {  (Course new)
                  name: #c2;
                  timeAt:12  }.
```

```
courses({#c3}):() ->
          {  (Course new)
                  name: #c3;
                  timeAt:11 }.

courses({#c4}):() ->
           {  (Course new)
                  name: #c4;
                  timeAt:10 }.

takes({#s1}):()->{#c1}.
takes({#s1}):()->{#c4}.

takes({#s2}):()->{#c2}.

takes({#s3}):()->{#c2}.
takes({#s3}):()->{#c3}.

takes({#s4}):()->{#c3}.

teaches({#t1}):()->{#c1}.
teaches({#t1}):()->{#c3}.

teaches({#t2}):()->{#c2}.
teaches({#t2}):()->{#c4}.

teaches({#t3}):()->{#c2}

´.

 "  what courses does #s1 take that are taught by #t2 ? "
    answer1 := rb solveEquations:´ ( takes({#s1}) = C,
                                     teaches({#t2}) = C ,
                                     courses(C)=C2 ) ´
                 instantiating:´(C2)´.
```

```
"whether #t1  teaches a course in the same time slot as #t2 ?
 in fact this query tells us even more! it gives us the actual courses"
 answer2 := rb solveEquations:´ ( teaches({#t1}) = C1,
                                  courses(C1)=C2,
                                  teaches({#t2})=C3,
                                  courses(C3)=C4,
                                  MSG(C2,{#timeAt}) =
                                     MSG(C4,{#timeAt}) )   ´
                instantiating: ´ (C2,C4) ´.


 "all students of #t1 "
 answer3 := rb solveEquations: ´( teaches({#t1})=C1,
                                  takes(S)=C1,
                                  students(S)=S2 )´

               instantiating: ´( S2 )´.


"what other teachers teach a course at the same time that
 #t1 teaches a course ? "
 answer4 := rb solveEquations:´ ( teaches({#t1}) = C1,
                                    courses(C1)=C2,
                                    teaches(T)=C3,
                                    NOT(T={#t1})=true,
                                    courses(C3)=C4,
                                    MSG(C2,{#timeAt}) =
                                        MSG(C4,{#timeAt}),
                                    teachers(T)=T2
                                  ) ´
                  instantiating: ´(T2)´.
```