Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

AN OBJECT-ORIENTED LOGIC PROGRAMMING ENVIRONMENT FOR MODELING

Thomas Wingfield Page, Jr.

September 1989
CSD-890055

UNIVERSITY OF CALIFORNIA

Los Angeles

An Object-Oriented Logic Programming Environment for Modeling

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Thomas Wingfield Page, Jr.

1989

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank the members of my committee for their many helpful suggestions. I am especially grateful to my advisor and committee chair, Professor Richard Muntz, for the many patient hours spent meeting with me while this research was conceived and executed. It is now impossible to separate which ideas originated with him but his intellect pervades this work.

Special thanks to Professor David Martin for teaching me individually what I needed to know about formal semantics.

I am also grateful to Professors Gerald Popek, Stott Parker, and Richard Muntz, the principal investigators of the DARPA project, for providing me financial support throughout my tenure as a graduate student.

My research has benefited greatly from the dynamic collection of people in the Tangram group who have provided a sounding board for ideas. The large volume of code written as part of this work would not have been possible without the help of many people. I especially acknowledge William Cheng and Steve Berson who made the Tangram modeling environment happen; Robert Lindell who collaborated on the early work on modular and object-oriented Prolog; Cliff Leung who worked with me on the typed Log(F) compiler; Jon Edwards, Leana Golubchik and Gary Rozenblat who provided programming assistance; and Ted Kim who often sacrificed his own research to keep our computing environment in optimal shape.

Finally, I express my love and gratitude to my wife, Gayle, who gave me constant love and support even though she swore she would never marry a student.

ABSTRACT OF THE DISSERTATION

An Object-Oriented Logic Programming Environment for Modeling

by

Thomas Wingfield Page, Jr.

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor Richard R. Muntz, Chair

The human mind is primarily a machine for constructing and evaluating *models.* It is no surprise then that most of what we do with computers amounts to modeling real-world systems. Models allow us to ask questions like: What happened? What would happen if ...? Will it perform acceptably? We would like to be able to ask these types of questions about a real-world system before it is ever built, while it is in operation, or after it has failed.

While there is an ever expanding set of mathematical techniques for modeling, the actual practice of modeling remains the exclusive domain of a few experts. Each expert is typically highly proficient at only a small number of these techniques. The mathematical solvers associated with each technique are generally very inflexible, requiring input in a rigid and idiosyncratic format.

What is required is an advanced modeling environment; an environment for the construction, storage, maintenance, querying and solution of models. Such an

environment would harness the advanced modeling techniques for use by experts in the domain being modeled as opposed to experts in the solution techniques themselves. If realized, this meta-modeling system would truly be a tool for magnifying the power of the human brain.

The goal of this dissertation is to begin to put in place the technology out of which an advanced modeling environment could be constructed. We propose a programming language based on a combination of the logic and object-oriented programming paradigms. We argue that both the declarative knowledge representation via logic and the hierarchical organization and modularization of object-oriented structuring are critical to the flexible modeling environment problem. We identify a surprising weakness of all existing programming paradigms, the inability to bind a function name to an implementation for a given target object (model) flexibly. In response, we propose a new concept, *semantic binding*, to deal with the problem. Finally, we demonstrate the effectiveness of this new hybrid-paradigm language in combination with semantic binding by presenting a prototype modeling environment. The Tangram Object-Oriented Modeling Environment is operational and being used to model a variety of domains.

CHAPTER 1

Introduction

1.1 Introduction

Man is far from being the physically strongest species on earth, yet we dominate this planet. It is our ability to use tools, machines which magnify our muscles, which gives us dominance. When we look at the enormous changes in the way we live which have occurred in just the last several lifetimes, they are largely explained by the exponentially accelerating rate at which we are producing new machines.

As dramatic as the impact of these machines which magnify our muscles has been, imagine the effect of tools which magnify our minds. After all, what truly sets us apart from the other animals is the size of our brains. If we can amplify what is already our greatest strength, the impact should dwarf that of all the other machines mankind has produced. While applications to date have been relatively mundane, falling more into the category of replacing tedious work, computers have the potential to be this magnifier of the mind.

The human mind is primarily a machine for constructing and evaluating *models* [70]. It is not a surprise then that most of what we do with computers amounts to modeling real-world systems. Models allow us to ask questions like: What happened? What would happen if ... ? Will it perform acceptably? We would like to be able to ask these types of questions about a real-world system before it is ever built, while it is in operation, or after it has failed.

While there is an ever expanding set of mathematical techniques for modeling (queueing models, statistical models, Markov models, constraint-based models, simulation, to name but a few), the actual practice of modeling remains the exclusive domain of a few experts. Each expert is typically highly proficient at only a small number of these techniques. The mathematical solvers associated with each technique are generally very inflexible, requiring input in a rigid and idiosyncratic format.

What is required is an advanced modeling environment; an environment for the construction, storage, maintenance, querying and solution of models. Such an environment would harness the advanced modeling techniques for use by experts in the domain being modeled as opposed to experts in the solution techniques themselves. If realized, this meta-modeling system would truly be a tool for magnifying the power of the human brain.

1.1.1 Goal

The goal of this dissertation is to begin to put in place the technology out of which an advanced modeling environment could be constructed. We propose a programming language based on a combination of the logic and object-oriented programming paradigms. We argue that both the declarative knowledge representation via logic and the hierarchical organization and modularization of object-oriented structuring are critical to the flexible modeling environment problem. We identify a critical weakness of all existing programming paradigms, the inability to bind a function name to an implementation for a given target object (model) in a flexible manner. In response, we propose a new concept, *semantic binding*, to deal with the problem. Finally, we demonstrate the effectiveness of this new hybrid-paradigm language in combination with semantic binding by presenting a prototype modeling environment. The Tangram Object-Oriented Modeling Environment is operational and being used to model a variety of domains.

1.1.2 The Promise of Prolog

It has been observed that novice Prolog programmers are often lacking in moderation in extolling the virtues of their new-found panacea [50]. This is not hard to understand as logic programming seems to offer something for nothing. One writes a program, not by describing how to solve a problem, but by stating what is true of the application domain. The built-in inference engine then uses this information to search for a solution without further guidance from the programmer. Further, the program can be enhanced incrementally; additional rules can be added to the domain's knowledge base which are used automatically without impact on existing knowledge. Prolog's powerful yet simple pattern matching facility (unification), internal database with associative access, and simulated nondeterminism combine to free the programmer from tedious low-level concerns. Finally, logic is a formal way of encoding human language, a much more natural vehicle of expression for humans than conventional programming languages which provide an abstraction of the underlying computer hardware. Unfortunately, it soon becomes clear to programmers that Prolog delivers on these promises only for "toy" applications.

While it is extraordinarily successful for rapidly prototyping small programs, Prolog has been largely unused for writing large programs. Its selection as the basis of the Japanese Fifth Generation Project notwithstanding, most AI applications are still written in Lisp or conventional algorithmic languages like C. The problem is that even the most basic of support for accepted software engineering practices like modularization, data abstraction, scoping, and information hiding is entirely absent from Prolog [22]. Without these program structuring techniques, the size of programs that can be constructed in Prolog is severely limited.

1.1.3 Programming in the Large

What is needed is a facility for *programming in the large.* Programming in the large consists of designing a system out of sufficiently small components that conventional *programming in the small* techniques can be employed. While the application as a whole may be too large to be understood by any one person, each building block should be entirely comprehensible. These small components or *modules* must have a separate interface and implementation. Thus they can be used by other parts of the program, perhaps by other programmers without regard to how they are implemented. As the application evolves, the implementation may be called upon to change, say to improve performance; the constant interface shields the rest of the program from concern about the internals of modules. Further, the low level building blocks should be sufficiently general that they may be reused in many contexts, reducing the total amount of code that must be written. The task of programming in the large then becomes one of linking components.

1.1.4 Object-Oriented Programming

"'Object-oriented' is the latest *in* term, complementing or perhaps replacing 'structured' as the high-tech version of 'good'."[63] This buzzword is used to describe languages as different as Smalltalk-80 and Ada and is applied to databases, operating systems, knowledge representation schemes, and user interfaces [27]. Clearly it means many things to many people.

The Object-Oriented paradigm is an anthropomorphic style of programming in which the physical and conceptual entities of the problem domain are modeled as autonomous information processing agents called *objects*. Each object contains a piece of the domain knowledge and may be be thought of as an expert on that piece of the application domain. Objects consist of a hidden state and a public interface in the form of a set of messages to which the object will respond. In our view, object-oriented implies three essential concepts: encapsulation, inheritance, and late binding. A full introduction to the object-oriented paradigm is found in Chapter 2.

Objects inherit their behavior (the *methods* by which they respond to messages) from their class. Thus all instances of a class share their method implementations creating a first level of code reuse. The classes are arranged in a specialization-generalization hierarchy such that instances can inherit method implementations from their ancestor classes. Late (run-time) binding (as opposed to compile-time binding) insulates clients (object users) from changes made by class suppliers. Combined with inheritance, late binding creates an extremely flexible environment where great leverage is obtained from code reuse, reducing implementation time and enhancing software quality.

The primary appeal of the object-oriented paradigm is its capacity for structuring large systems. The design of a large facility begins with identifying the objects in the application domain. Tightly circumscribed modules of code which model the behavior of the domain entities can be designed, coded, debugged, and documented independent of the rest of the system. Moreover, the paradigm is a packaging technique whereby objects can be supplied by a programmer to a client and integrated into an application in a structured manner. Insulated by a well-defined interface, the implementation of an object may change without impact on the rest of the system. Thus the paradigm offers benefits, not only at system design and implementation time, but also over the life-time of the application as it inevitably evolves.

1.1.5 Combining Logic and Object-Oriented Programming

While a very valuable structuring tool, when it comes to actually programming the methods, current object-oriented languages are essentially procedural programming languages. Many types of knowledge are best expressed declaratively. If we can think, as stated above, of an object as an expert on its piece of the domain knowledge, we should be able to implement the object by stating what the object knows rather than how it answers messages. This goal leads us to the desire to create a combined object-oriented logic programming paradigm.

This dissertation proposes a hybrid programming paradigm which smoothly integrates object-oriented programming with Prolog. We have implemented the paradigm in a new language called Object-Oriented Prolog (O-OP). It is based on the "message-passing interpretation of logic" [91] which interprets a message M to an object O as a request to prove the goal corresponding to M in the context of the set of axioms defined by O. Because this is an interpretation of message passing within logic there is no "impedance mismatch" between the paradigms. [1] The sending of a message is a goal like any other in Prolog and may be freely

[1]The term "impedance mismatch" has been used in the past to describe the mixture of paradigms resulting from embedding a database query (set oriented) in a general programming languange program (often tuple at-a-time). Here, we refer to the potential for confusion due to possible mismatches in the computation model of logic programming and object-oriented progrogramming.

intermixed with other Prolog goals.

While the two paradigms can be mixed freely, it makes the most sense to use each paradigm for what it can do best; logic for programming in the small, and objects for programming in the large. The application as a whole is structured out of communicating objects. The individual methods for these objects are programmed declaratively; an object deduces the bindings for the message arguments both from its local knowledge base and by sending messages to consult other objects.

1.2 Significance of the Work

Today, many of us have more computing power on our desk than did the entire university campus merely one decade ago. This exponential growth in machine speeds has been accompanied by a revolution in the way hardware is designed and built. Each new device is not designed from scratch. Rather it is assembled from integrated circuits with well defined parameters and interfaces.

Many have observed that we are experiencing a software crisis. When a large project fails or experiences huge cost overruns, it is most often because of the software. The productivity of software producers as we have migrated slowly from assembly languages to Fortran, C and higher level languages has failed to keep up with that of hardware designers. It is estimated that software productivity has increased at a rate of 3-8 percent per year while processing capability has increased at 40-200 percent per year [69]. The advent of distributed systems promises continued dramatic increases in processing speed while vastly complicating the job of the programmer. The productivity crisis is accompanied by a similar crisis in software quality.

"Why isn't software more like hardware? Why must every new development start from scratch? There should be catalogs of software modules, as there are catalogs of VLSI devices: when we build a new system, we should be ordering components from these catalogs and combining them, rather than reinventing the wheel every time."[2] Cox, the inventor of Objective-C, an object-oriented extension to C, asserts that objects (which he calls Software-ICs) might do for software what the silicon chip did for hardware [27]. That is, applications could be assembled from existing pieces that have very well defined functions and interfaces. These component pieces could be very finely crafted due to the tremendous leverage obtained by reusing code.

The combination due to this dissertation of declarative programming with object-oriented structure is a leap forward in the object-oriented revolution.

The primary contribution of this dissertation is language which can take advantage of a declarative programming and object-oriented structuring. An object-oriented design coupled with a declarative statement of the behavior of each object becomes an *executable specification* in the Object-Oriented Prolog environment. Very complex programs may be very rapidly prototyped in this hybrid paradigm. Once the knowledge base of each object is debugged, critical

[2]From a 1968 speech by McIlroy later published in [60].

sections of the application may be reimplemented in an algorithmic language for greater efficiency; the paradigm naturally hides the internal implementation of objects so that such reimplementation may occur without impact on the rest of the system. The Object-Oriented Prolog language represents a leap forward in the object-oriented revolution in software crafting.

1.2.1 Significance of the Modeling Application

In order to demonstrate the viability of the Object-Oriented Prolog language as a tool for building a knowledge-based application which would not be possible by conventional means, we have designed and prototyped the Tangram Modeling Environment using O-OP. Tangram features a graphical interface for defining and querying models, an object-oriented organization of models and sub-models, a tool-kit of solution techniques, animated simulation, an extensible set of application domains with a customizable interface and query language, and declarative domain-expert knowledge bases for translating models and queries into a form that can be solved by one of the system's mathematical solvers.

The experience of the implementors is that the Tangram system could not have been built without the combined strengths of object-oriented structuring and Prolog's declarative knowledge representation. Customized application domains have been constructed in a very short time (2 to 4 man-weeks). This rapid response is made possible by the ability to create domains by specializing existing domains via object-oriented inheritance and by the facility of Prolog for rapid prototyping and ease of expression of expert knowledge. New solver packages have also been integrated quite easily due to the ability of the paradigm to treat them as abstract "black boxes". The Tangram Modeling Environment is already in use outside of the implementation group and the approach potentially represents a great leap forward in the practice of modeling.

1.2.2 Further Contributions

This dissertation also demonstrates the integration of the third major programming paradigm, functional programming, via the integration of the Log(F) language with Object-Oriented Prolog. A very important class of objects are stream valued; that is, they consist of an ordered sequence of values. One of the most common examples of object is temporal data. Often the most convenient way to process streams is via rewrite rules which apply a function to each element of the stream.

Previous work [73] has shown that functional programming can be smoothly accomplished in Prolog. We have shown how object-oriented programming can be realized cleanly in Prolog. We have the unique opportunity to combine the three major paradigms in one seamless language. As all three programming paradigms are interpreted in the context of Prolog there is no impedance mismatch as there are in hybrid languages which merely embed one type of language within another. The resulting multi-paradigm language can be compiled into modular Prolog and thus be implemented very efficiently.

An outgrowth of the design of the Tangram environment is a new concept, *semantic binding*, which goes beyond conventional object-oriented binding with multiple inheritance. In a system such as Tangram which must maintain an extensible set of tools, it is not sufficient simply to inherit a method based on a static typing of objects. An appropriate solver binding, in general, depends on the dynamic state of the object and the actual query posed (or the message parameters). No existing paradigm can handle this situation.

With semantic binding, user defined code or rules are used to either bind function names with implementations or to reclassify an object dynamically when the object receives a message. A reclassified object is moved to the most specific class for which it satisfies membership constraints and thereby inherits the appropriate method. The rules by which the semantic binding is achieved constitute a domain-specific knowledge base. Domains are structured hierarchically and domain knowledge expressed declaratively. Again, the combination of logic and object-oriented programming found in Object-Oriented Prolog is key.

1.3 Organization of the Dissertation

Chapter 2 presents necessary background material. The concept of logic programming and its realization in the Prolog language is introduced. An overview is given of the Warren Abstract Machine (WAM), the current state of the art in implementation techniques for Prolog. The level of detail presented is sufficient to understand the modifications to the WAM made as part of this research. The major concepts and terminology from the object-oriented paradigm are introduced. Finally, related research on the integration of logic and object-oriented programming is reviewed.

Chapter 3 discusses the design and implementation of the Object-Oriented Prolog language. We first present the overall philosophy of our approach to integrating the two paradigms. Then we detail the design of the underlying modules facility for the WAM upon which the object-oriented language is built. This is followed by a detailed description of the design of Object-Oriented Prolog.

Scholarship dictates that when we propose a new language, we define its formal semantics. In addition to rigorously specifying the meaning of a program, the formal semantics serves as a guide to and a test of the implementation of the language. Chapter 4 presents two equivalent formal semantics for Object-Oriented Prolog. First, conventional Prolog semantics are reviewed. We then detail an operational and denotational semantics for Prolog with modules. We show how with the addition of the send and inherit predicate to each module we can define the semantics of message passing and inheritance. The chapter concludes by showing the correspondence between the two semantic definitions.

Chapter 5 examines approaches to and benefits of integrating a third paradigm, stream processing, with logic and object-oriented programming. Two approaches are presented, both of which have been implemented. An example application using temporal data is used to illustrate the combined paradigms.

Chapter 6 presents the novel variation on object-oriented name binding we call *semantic binding*. We introduce the issue and present two classes of solutions.

If the proof is in the pudding, Chapter 7 is the pudding. It presents an overview of the Tangram Object-Oriented Modeling Environment which is written in and relies upon Object-Oriented Prolog. We introduce the problem of designing a modeling environment. We show how the problem of semantic binding occurs in practice and how it is solved in Tangram. A detailed example of an object-oriented model is presented.

Conclusions and a summary of the dissertation follow in Chapter 8.

CHAPTER 2

Background and Related Work

2.1 Introduction

This chapter provides background information necessary to the dissertation. Section 2 introduces Prolog and logic programming in general and may be skipped by readers already familiar with these areas. Section 3 introduces the basic concepts, motivations, and terminology of the object-oriented programming paradigm. This section may also be conveniently skipped by readers not requiring such an introduction. Section 4 summarizes other related research in the combination of logic and object-oriented programming.

2.2 Prolog and Logic Programming

Logic was developed as a means of expressing the exact meaning of sentences in natural language [56]. It has long been used in computer science as a specification language because of its precise declarative semantics [55]. In 1972 Kowalski [51,52] showed that the Horn clause subset of first order logic could have a *procedural interpretation* in addition to its traditional declarative semantics; this makes possible its use as a programming language. Prolog is an example (perhaps the first and most important) of a programming language based on this reading of Horn clause logic [25]. In the declarative interpretation, a Prolog program can be viewed as a hypothesis and a query as a theorem to be proved in using the hypothesis. A logic program, P, consists of a finite set of Horn clauses of the form

$$A \leftarrow B_1, B_2, \ldots, B_n$$

containing exactly one positive literal, A, called the *head* [55]. The conjunction of $n \geq 0$ negative literals $B_1, B_2, \ldots, B_n$ is called the *body*. A and each B_i have the form $R(a_1, a_2, \ldots, a_m)$, $m \geq 0$, where R is a relation (predicate) symbol and each a_i is a term. A term consists of either a variable, constant, or function application of the form $f(t_1, t_2, \ldots, t_p)$, $p \geq 0$, where f is a function symbol and each t_i is a term. Such a clause is interpreted as defining a procedure for A whose head defines the input and output parameters and whose body specifies a conjunction of procedure calls with the semantics, "For each assignment of each variable, if $B_1, B_2, \ldots, B_n$ is true then A is true." A *unit* clause has the form $A \leftarrow$ with the empty collection of sub-goals and is interpreted as "A is true".

A query on P is an existentially quantified conjunction of the form

$$\leftarrow C_1, C_2, \ldots, C_q, \quad q \geq 1.$$

If the query contains variables $v_1, v_2, \ldots, v_r$, $r \geq 0$, a substitution is a set of pairs of the form

$$\{< v_1/t_1 >, < v_2/t_2 >, \ldots, < v_r/t_r >\} \quad r \geq 0$$

where each v_i is one of the variables and t_i is a term, to which v_i is bound. The application of a substitution θ to an expression E, stated $E\theta$, is formed by simultaneously replacing each occurrence of v_i with the term t_i for each i. The proof procedure must find the most general substitution θ such that every ground instance of $(C_1, C_2, \ldots, C_q)\theta$ is a logical consequence of the set of clauses P. The proof proceeds by picking some clause C_j from the query and replacing the goal with

$$(C_1, C_2, \ldots, C_{j-1}, B_1, B_2, \ldots, B_n, C_{j+1}, \ldots, C_q)\theta_i$$

where

$$A \leftarrow B_1, B_2, \ldots, B_n$$

is a clause in P such that C_j *unifies*[1] with the head A with unifying substitution θ_i. The proof terminates successfully when the goal clause is replaced with the empty clause. The composition of the sequence of substitutions $\theta_1\theta_2\ldots\theta_N$ is one answer to the query. This proof procedure, known as *SLD-resolution*, is due to Kowalski and is a practical version of general resolution with unification discovered by Robinson in 1965 [81].

Prolog is an approximate implementation of logic programming. [2] The following is an example of a Prolog program to compute the `append` relation on lists. The first clause declares that the result of appending any list, say `X`, with

[1]Two clauses c_1 and c_2 unify if there exists a substitution θ such that $c_1\theta = c_2\theta$.

[2]The omission for efficiency of the *occurs check* and the addition for practical reasons of extra-logical features keep Prolog from being a true implementation of logic programming.

the empty list denoted by `[]`, is `X`. The second clause declares that the result of appending two lists, X and Y, can be obtained by consing the first element of X (its head) to the result of appending the rest of X (its tail) to Y. [3]

```
append([],X,X).
append([H|T],Y,[H|W]) :- append(T,Y,W).
```

The above is not an algorithm for appending two lists, but rather statements about what is true of the append relation. Thus, in addition to being able to query "What is the result of appending [a], and [b,c]?", we can also say, "Is there any X and Y such that appending X and Y results in [a,b]?" and get the three answers X=[], Y=[a,b]; X=[a], Y=[b]; and X=[a,b], Y=[]. Were the predicate for appending two lists expressed as in most languages as an algorithm rather than assertions of truth, using the predicate "backwards" to split a list non-deterministically would not be possible. This ability to use knowledge in a way which was perhaps not anticipated when the knowledge was codified is one of Prolog's great strengths[4].

In theory, a logic program is purely an expression of the knowledge about an application; the control is left to the system. In practice, the SLD resolution procedure is simple enough that programmers are aware of how the system will proceed to evaluate a query against their database. Programmers often take

[3] | is the cons function and the list notation `[H|T]`, read "H cons T", is short for the structure `.(H,T)` where `T` is a list. The notation `[a,b,c]` is short for `.(a,.(b,.(c,[])))` or equivalently, `[a|[b|[c]]]`. The `:-` symbol is used instead of $\leftarrow$ in the Edinburgh family of Prologs.

[4] It is unfortunately the case, however, that due to the use of the non first-order logic capabilities within Prolog (such as arithmetic expression evaluation) many predicates work only when a certain set of variables are bound on input. Note that constraint-oriented Prologs seek to repair precisely this problem.

advantage of this awareness to control the evaluation procedure through their use of explicit clause orderings, and extra-logical features such as the cut.

2.2.1 The Warren Abstract Machine

The current state of the art in high performance implementations of Prologs are based on the Warren Abstract Machine (WAM) [100] (for example: Quintus Prolog [1], Sicstus Prolog [20])[5]. While intended for logic programming and theorem provers in general, the WAM is much like any abstract machine for the execution of recursive, block-structured languages like Pascal or C [34]. A Prolog program is compiled into "WAMcode", the instruction set of the abstract machine, and is then interpreted by an emulator. We expect to see actual implementations on WAMs in silicon in the near future. We will not present here a complete tutorial on the WAM; for that the reader should see the excellent tutorial from the Argonne National Laboratory [34]. Rather, we will present only a high level overview of the architecture necessary to understand our modifications detailed in the next Chapter.

2.2.1.1 Data Structures

There are four main data areas in the WAM:

1. The *argument registers* contain the input/output arguments of a procedure call. There is no firm distinction between input and output arguments;

[5]Faster implementations may soon be available which compile Prolog directly to machine code.

a particular argument may sometimes be unbound on input being later bound in a procedure, thereby becoming an output argument; at other times the same argument may be bound at call time acting purely as an input variable.

2. The *local stack* (often just called "the stack") has two primary functions. First, the stack is used to store *choice points*. A choice point is a record used to store the information necessary to maintain alternative approaches to solving a goal. If one alternative fails and backtracks, information in the last choice point is used to reset the machine's state to try the next alternative.

   Second, the local stack is used as a scratch area to save critical information from registers which may be destroyed by a procedure. This information is saved in an *environment*. The most important information saved in an environment are the *continuation instruction*, the *current environment*, and the argument registers. The continuation instruction points to the instruction in the invoking procedure to execute next if the current procedure terminates successfully. The current environment is a pointer back into the stack at the environment in effect when the current procedure was called. The argument registers may need to be saved if the procedure is going to call other procedures internally with different arguments. The compilers are smart enough to save registers only when necessary.

3. The *heap* (sometimes called the global stack) is used to construct logical formulas to which procedure arguments are bound. A machine register contains a pointer to the current top of heap. Space on the heap is reclaimed by garbage collection.

4. The *code area* contains the actual compiled WAMcode instructions.

The registers may contain variables, constants (atoms, integers, floating point numbers, the nil list), lists, or structures. A register contains a *value cell*, a data item typed by the leftmost four bits (the tag bits). The remaining bits point to an object of the type specified. An unbound variable points to itself. An atom is represented as a pointer into the *atom table*. The atom table is a hash table where the string associated with the atom is actually stored. Integers and floating points employ similar hash tables.

The state of the WAM is summarized by the contents of the stack and the heap, the program counter, and by the following runtime registers:

Continuation Instruction: the address where control is to return upon successful termination of the current procedure.

Current Environment: a pointer to the calling environment stored in the stack.

Last Choice Point: a pointer to the latest choice point in the local stack.

Top of Stack: the current top of the stack.

Top of Heap: the current top of the heap.

Top of Trail: the current top of the *trail*, a data structure used to record variable bindings that must be undone on backtracking.

Argument Register: see above.

Next Clause: a pointer to the next alternative to be tried.

2.2.1.2 Backtracking

When multiple alternatives for a given goal are encountered, the WAM creates a choice point and saves the contents of the above registers in it. If the procedure backtracks, the state of the machine can be reset to try the next alternative by restoring the register values saved in the choice point. Variables that must be uninstantiated on backtracking are recorded on the trail. Upon backtracking, pointers to value cells are popped off the trail and reset to point to themselves until the value of the Top of Trail register reaches the value of Top of Trail saved in the choice point. Note that the effect of a cut is to remove choice points from the stack. Also note that good compilers implement *last call optimization* which removes the choice point from the stack when the last call in a determinate procedure is entered. This optimization (combined with garbage collection [94]) allows tail recursive programs to run in constant space rather growing with each recursive call.

2.2.1.3 Indexing

Alternative clauses making up procedures are linked in index structures to quickly eliminate clauses that could not possibly unify with the input arguments. The index key is the principal functor of the first argument. The **`switch_on_term`** instruction dispatches to one of four addresses depending on whether the first argument dereferences to a variable, constant, list, or structure. If the first argument is a constant or structure, the clauses will be further indexed using the **`switch_on_constant`** or **`switch_on_structure`** which make use of hash table access.

2.2.1.4 Sicstus WAM Implementation

The prototype implementation of architectural changes to the WAM proposed in the next chapter were done in the Swedish Institute of Computer Science version of the WAM. In Sicstus Prolog, predicates may be compiled or interpreted. There is a global functor table into which all principal functors of predicates are hashed. A bit in this table indicates whether a particular predicate is static (compiled) or dynamic (interpreted). Code for static predicates is stored in the code area as discussed above. Code for dynamic predicates is represented as structures on the heap and executed by an interpreter. The WAM emulator check the type of each predicate when it is called, setting the program counter to the first instruction of the procedure if it is compiled or to the first instruction of the interpreter if the procedure is interpreted.

2.3 Concepts from Object-Oriented Programming

As a means of knowledge structuring, the object-oriented programming paradigm is closely related to the concepts of frames and semantic networks in AI and to semantic models in knowledge based systems [35]. The ideas behind object-oriented programming originated in Simula 67 [28] and were first fully realized in Smalltalk-80 [38]. Entities in a model of a real world system are represented by computational objects each with an associated set of private local state variables and a public interface. The objects in a model "come alive" when they receive *messages*. Messages are used to query individual objects, to ask them to perform operations on their internal state or on other objects. In fact, messages are the only way to interact with an object. Sending a message to an object is analogous to calling a function with arguments in a conventional language or calling an exported procedure in a modular language. However, one cannot call a procedure to do something to an object. One can never "do something to" an object, only request an object to do something to itself by sending it a message which it is free to accept or reject.

Many people mean many different things by "object-oriented" programming. We take the view that there are three essential concepts:

1. Encapsulation: The details of an object's internal implementation are not visible.

2. Inheritance: Code is modularized according to class or type. Code copying

is avoided for similar objects by inheritance from ancestor classes.

3. Late binding: The code actually run depends on the type of the object, which is not known until runtime. Late binding may be relaxed (bindings may be compilted) for improved performance at the expense of flexibility.

Notice that messages are not viewed as an essential concept but rather as a metaphor for the "arms-length" relationship between objects. The message interface seems to impart intuition to users about encapsulation and late binding. The following sections examine these essential concepts and introduce the terminology of the object-oriented methodology.

2.3.1 Encapsulation

Objects combine the properties of procedures and data in a way analogous to abstract data types. They both perform computations and have local state. The messages to which an object responds define its interface. Receipt of an acceptable message causes an object to execute one of its *methods*. The message *selector* specifies which operation the object is being requested to perform. Thus, object-oriented programming implements the principle of *data abstraction*, i.e. that consumer programs should not know details about the implementation of an object. Consequently, the implementation can change without affecting consumers.

2.3.2 Late Binding

The recipient of a message decides what code to execute to implement a method. This allows for *polymorphism*, which in the object-oriented context corresponds to the ability for many types of objects to respond to exactly the same messages [93]. Thus a consumer need not know exactly what type of object is the recipient of a message, only that the object supports the message type. New types of objects can be created which respond to existing message *protocols* without affecting consumers. For example, a window manager program does not need to know about a new type of icon that is added in order to be able to display it, so long as that new object supports the basic set of messages that the window manager expects. Shifting the burden of binding from the consumer to the supplier of a service makes growth and change in a software system much more manageable.

From a different point of view, operators may be said to be *overloaded.* That is, the same operator (message selector) invokes different code, depending on the type of the object to which the operator is applied. Name conflicts are not a problem because names are resolved in the context of a single type. Different types of objects may have operators with the same names without interference. This is in contrast to Prolog which has a global name space.

2.3.3 Classes and Inheritance

Continuing the above analogy to abstract data type languages, in object-oriented programming, a *class* corresponds to a data type and an object to an instance of that type. A class may be thought of as a set of similar objects all of which support the same methods, but which will, in general, have different values in their instance variables. For example, "Unix file" is a class with methods such as open, close, read and write; "/etc/passwd" is an object which is an instance of a Unix file. "/etc/passwd", the file, can be opened and read. "Unix File", the class, cannot. "/etc/passwd" responds to the same methods as other Unix files.

2.3.3.1 Inheritance

Inheritance provides an economical way to define classes which are like some other class but have a few differences. For example, consider a system in which we need to model both cars and trucks. Many of the methods for these two classes will be identical. Rather than repeat the definition of the redundant code, we can define a vehicle class and in it represent those characteristics common to both cars and trucks. We can then define both cars and trucks as sub-classes which inherit these common methods from the vehicle class.

In general, if a class A is a *sub-class* of class B, it inherits all of the methods defined for B. B is said to be a *super-class* of A. Whenever an object of type A receives a message with a selector for a method not defined by class A, the system uses the definition found in class B. The definition for class B may itself be inherited from its super-class. Classes may have multiple sub-classes. If the definition of class A reimplements some its parent's methods, then the local method takes precedence. That is, the search for an implementation for a method starts in the class of the object which receives a message, and proceeds up the hierarchy.

2.3.3.2 Super and Self

It is often the case that the implementation of a specialized version of a method in a sub-class is very similar to the code used in the parent. Thus it is often convenient to write the method by doing some computation, calling the parent's version, and then doing some final computation. However, there is no simple way to name the parent's version of the method because the name binding always searches the class hierarchy bottom up. Thus, it is convenient to provide a mechanism to indicate an alternate name binding strategy for these explicit calls to the parent. This is accomplished with a *super* message. Similarly, it is often desirable that the implementation of one method make use of an existing method inherited by the same object. This is accomplished by a *self* message. Super messages send the message to the current object but begin trying to bind the selector to an implementation in the object's super-class. Self messages simply send a message to the current object.

2.3.3.3 Class and Instance Creation

Classes are generally implemented as objects themselves [27]. They are all instances of the *metaclass*, the class of all classes. Subclasses and instances are created by sending messages to a class. Some object-oriented languages make a distinction between objects and classes while others do not. Since objects generally greatly outnumber classes, there is the potential to optimize the implementations if a distinction is made. However, the optimization may be at the expense of conceptual purity.

2.3.3.4 Multiple Inheritance

The inheritance graph may be constrained to be a tree. That is, each class is a sub-class of only one super-class. The root of the tree is an object which implements the methods which all objects must support. Some languages allow *multiple inheritance* in which an object may be an instance of more than one class or a class can be a subclass of more than one superclass. This creates an inheritance structure which is a directed acyclic graph.[6] For example, a tape drive is both a node in the Unix filesystem and a piece of hardware. The set of methods inherited by instances of the tape drive class is the union of the locally defined methods and the inherited methods from both super classes in the absence of name conflicts among methods. There must be some precedence

[6]It is often called a lattice in the literature (see [93] for example) but multiple inheritance does not guarantee a unique least upper bound and greatest lower bound and so, strictly speaking, does not define a lattice.

rules for cases where both super-classes define methods with the same name.

The technique of modifying a generic class of objects for a more specific use is called *specialization.* A tape drive is simultaneously a specialization both of the Device class, and the Hardware class. This practice provides great leverage in building large software systems as it provides a model for creating reusable code.

2.3.4 Composite Objects

The inheritance hierarchy is a partial ordering based on the isa relationship. We can also define hierarchies based on the is-a-part-of relationship. Hierarchies of this type create composite objects which allow sets of objects to be manipulated as a single entity. Computer aided design (CAD) is an example of an application area where is-a-part-of is a primary structuring relation among objects. In addition to the added semantic power of composite objects, they may be exploited to improve performance of object-oriented database systems. The object server may cluster the components of a composite object to take advantage of the locality of access that is likely to occur among objects in the same collection [49].

Most object-oriented languages provide some sort of primitive *collection class* which provides for sets of arbitrary numbers of members. Basic methods typically include adding and deleting elements, reporting the number of elements, looping through all members of the collection, etc. Specialized collections may be created which, for example, maintain ordering, use hashing, or perform type checking.

2.4 Related Work on Objects in Prolog

There are numerous object-oriented programming languages. Some, like Smalltalk [38], are completely object-oriented. Many others graft object-oriented constructs onto other languages, e.g. Loops [14] for Lisp, or Objective-C [27] and C++ [97]. Another recently exploding line of research concerns languages for querying databases of complex objects (see [8,6,4,13,21,30,47,49,56,74,99,103]) We concentrate here on related research concerning the combination of object-oriented concepts with logic programming languages, recognizing the potential of such a hybrid paradigm for database query languages[7].

2.4.1 Concurrent Logic Programming

An interesting approach, very different to that taken here, is based on Concurrent Prolog (CP) [91] and is being pursued by Shapiro [92], with syntactic enhancements in the Vulcan language (via a preprocessor) by Kahn et al. [46,98]. CP is based on the behavioral reading of guarded Horn clauses in which a goal corresponds to a process and a conjunction of goals corresponds to a set of processes. Shared variables between goals denote communication channels between processes. The clause

[7]Prolog subsumes the relational data model and is more expressive in many ways than conventional relational algebras and calculi. With Prolog, we can represent richer relationships between complex objects, supporting queries involving closures, metadata, non first-normal-form data, etc.

```
A :- G | B1, ..., Bn.    n > 0
```

is read as, "Process `A` can replace itself with a system of processes `B1` through `Bn` if the guard `G` is satisfiable". Normally a process terminates when it replaces itself with the empty process. A process survives to become a persistent object by calling itself recursively.

The simplest objects in CP have two arguments. The first argument is a shared variable or stream used to send messages to the object. The variable is annotated *read-only.* The processes implementing objects are realized in an And-parallel fashion and so, suspend until their read-only input variable is instantiated. Thus an object remains inactive until it receives a message on its input stream. Response to messages is via uninstantiated variables in the message selector.

The second argument is private to the object and implements its internal state. An object which changes its internal state does so by calling itself recursively with its new state. More sophisticated objects may have more than these two arguments. There may be additional stream channels for two way communication. The state may be represented in more than one argument. While CP maintains objects with mutable internal state, it does so in a very different way from that proposed in this dissertation. Objects do not have names or pointers; their identity is maintained by the message stream. Messages are sent by unifying the uninstantiated head of the stream with the message. Broadcasting messages is easy because multiple processes can share a read-only variable. Simultaneously sending different messages from two processes to a single object is problematic. One will succeed but the other will not generally be able to unify with the already instantiated head of the stream (unless the messages are the same). When an object is shared by multiple processes, it must have a *merge* process which merges the message streams from each of the clients into the one input stream for the shared object. The merge process must order the messages fairly so that all clients see the object in the same state.

Objects in CP are persistent only to the extent that processes are persistent; that is, there are no long-lived objects with disk based representations. CP implements object identity as changes to internal state which are visible to all branches which hold the stream variable for the object. However, CP's object identity is not meant to span long periods of time. Our goal, by contrast, is to implement objects which are persistent in the long term. We store object representations in the database, activating them when they are addressed with messages, and returning them to the disk when they are dormant.

Creation of an object B occurs in CP when an object A replaces itself with B via

```
A :- ...B...
```

The only name of the new object is the stream variable that it shares with its And-parallel siblings.

The following CP code defines a counter with methods **clear** (set to zero), **up**, and **show**.

```
counter([clear|S], State) :-
        counter(S?, 0).
counter([up|S], State) :-
        plus(State, 1, NewState), counter(S?, NewState).
counter([show(State)|S], State) :-
        counter(S?, State).
counter([], State). % for termination
```

Upon receiving a `clear` message on the head of the input stream, `counter` replaces itself with a new process having the same input stream but the new state of 0. The `up` message causes the object to reduce itself to a call to `plus` with the result of `plus` shared by the recursive call to `counter`. Receipt of a `show(X)` message causes `X` to be instantiated to the current state and the process replaced with another of the same state. Note that the correct operation of this `counter` object depends on the implementation of the `plus` predicate; `plus` must suspend until both of its first two arguments are available.

CP does not directly support inheritance though it can simulate class hierarchies by linking objects in filter chains. If an object receives a message which it does not support, it may forward that message to its parent, the next object in the chain. The hierarchical relationships among objects are simulated by the structure of the chain. Object creation is very expensive in CP with this kind of inheritance as an object (process) for each level of the hierarchy must be created.

Data encapsulation is achieved because there is no access to internal state other than via messages with uninstantiated variables. Component objects are also completely encapsulated since they are only nameable via their shared input variable which exists only in the protected state of the parent object. Even though they possess internal state, CP objects operate without side-effects. After processing each message, a process terminates replacing itself with a new process reflecting the changed state avoiding some of the problems associated with destructive assignment [3].

The object-oriented paradigm is particularly suited to CP since CP does actually pass messages between processes which act non-sequentially. CP and the Vulcan language are particularly interesting because they realize true concurrency in the sense of Hewitt's Actor model [24,43].

Programming in concurrent logic languages has been somewhat difficult; it is hoped that the object-oriented framework will alleviate this problem. Vulcan is an object-oriented language which compiles its more standard object-oriented syntax into CP. It retains the advantages of Shapiro's approach without the verbose and awkward syntax of CP [46].

2.4.2 Zaniolo

Zaniolo proposes a limited form of object-oriented programming which is implemented as a meta-interpreter in Prolog [102]. His work views an object as a parameterized theory. It does not support objects with state and does not hide implementations from clients. Zaniolo adds predicates `with`, `isa`, and `:` to denote method definition, inheritance, and message passing respectively. An object is defined via

```
object isa object2 with [method_list].
method1:        code()...
   .
   .
   .
methodN:        code()...
```

and a message sent to an object using the syntax

```
object : selector.
```

The infix operator : invokes the interpreter to locate code which implements the method indicated by the selector. For example, the declaration

```
reg_polygon(N,L) with [(perimeter(P) :- P is N*L)]
```

is implemented by asserting

```
perimeter(reg_polygon(N,L), P) :- P is N*L.
```

The interpreter first tries to unify a goal of the form `object : perimeter(P)` with `perimeter(object,P)`. If that fails, it tries with each of the ancestors of `object` as indicated by the `isa` hierarchy.

Zaniolo's system implements a flavor of specialization which is common in mathematical models but is seldom discussed in an object-oriented context. Consider a parallelogram which has state variables (base, height).[8] A rhombus also with (base, height) is a special case of a parallelogram. A square with (side) is a rhombus with (side, side); that is, we have rules

[8]Of course another piece of information such as another side or angle is required to fully specify a parallelogram. However, it is not necessary for purposes of calculating the area.

```
square(side) isa rhombus(side, side).
rhombus(base, height) isa parallelogram(base, height).
```

The parallelogram has a method "area" which computes area = base * height. Objects of type square can inherit this method, but only via the **isa** rule which tells how to translate the instance variables of the square into those of a parallelogram. Unlike more common examples, the specialized class, square, did not have additional variables, but rather fewer. However, it had a non-trivial **isa** rule which specified how to translate the instance variables of the sub-object into those of the super-class. Whereas in Concurrent Prolog, actual objects for the ancestors must be created, in Zaniolo's language, rules for creating the ancestors are declared but the objects do not actually exist.

This approach to object-oriented programming in Prolog amounts to providing a message sending predicate with the semantics, "Given an object O, what would be the response to a message M?" There are no object identifiers; objects exist only in the message and thus cannot be shared and cannot persist.

2.4.3 McCabe

McCabe's work [59] is similar to that of Zaniolo in that both view objects as parameterized theories and both propose a meta-interpreter or translator from their object-oriented language to a conventional Prolog. McCabe is primarily concerned with the semantics of the logic programming equivalents of object-oriented concepts. He examines a declarative first-order semantics, a higher order meta-linguistic semantics based on Bowen and Kowalski [15], and an operational semantics.

Objects are represented as *class bodies*, sets of clauses with a parameterized label. The translation to flat Prolog involves name extending the clauses with the label achieving a similar effect to our use of modules but without building on top of an underlying modules facility. An example of a class body is:

```
animal(Color,Transport) : {
        color(Color).
        transport(Transport).
        lives_on(land) :- transport(walking).
        lives_on(sea) :- transport(swimming).
        lives_on(sky) :- transport(flight).
        }
```

Inheritance relationships are expressed as *class rules.* For example, the rule

```
bird(Color) <= animal(Color, flight).
```

says, "A bird is an animal with attribute transport equal to flight." or "Whatever is true of flying animals is true of birds." An instance of a particular object is achieved by instantiating the parameter in a class label. For example,

```
toucan_Sam <= bird(multi_colored).
```

Structured objects can be created by using the labels of objects as parameters to instantiate new objects. There are no objects with mutable state, but the effect is simulated by allowing methods to return the label of a newly created object. This is not the same as true mutable state as other holders of the original label will not see the effect of the method which created a new object.

2.4.4 Spool

Spool is an object-oriented meta-interpreted extension of VM-Prolog done at IBM Japan [33]. Similar to Smalltalk-80, a class has four properties: superclasses, metaclass, instance variables, and methods. A Class definition takes the form

```
class Classname has Properties.
```

A method consists of a head and an optional body.

```
head :- body
```

where normal Prolog goal invocations are distinguished by prefixing a "#". Instance variables are accessed using the syntax

```
Name :: Value.
```

Sending a message is accomplished by

```
Receiver << Message
```

where the receiver may be bound or unbound thus allowing for anonymous message passing. Anonymous messages allow a sender to ask, "Is there anyone who can respond to this message?" Side effects are permitted; objects have mutable state. State changes are not undone by backtracking.

Spool does not attempt to unify object-oriented programming with logic programming as we do in this dissertation. It is simply an implementation of object-oriented programming on top of Prolog. It does take advantage of Prolog features to allow broadcast or anonymous messages as do CP and McCabe. Fukunaga reports experience with the Spool language in building a program annotator which works in both directions, producing annotations from a program or a program from annotations. These experiences support the value of these broadcast and anonymous message features of Prolog.

2.4.5 Bowen and Kowalski

Bowen and Kowalski propose a meta-level extension to Prolog in which theories and names of theories are first-class objects which may be the value of variables [15]. Later work by Bowen [16,17] shows how these extensions can represent frames, semantic nets, scripts and message passing. They view objects as theories and compile message passing into calls to their `demo` predicate which implements the proof theory. Thus,

```
send(<theory>, <message>, <response>)
```

is handled by the predicate

```
react(Theory, send(Destination_Theory, Message, Response),
      send(Destination_Theory,Message, Response, Trace), true)
  :-
      demo(Destination_Theory, receive(Message, Response), Trace).
```

where the Destination_Theory must contain a `receive` clause for the given message. While ascending to second-order logic, this approach still does not permit mutable state.

2.4.6 Biggertalk

Gullichsen [41,42] at MCC has proposed and prototyped an object-oriented Prolog front-end to the Gordion object server [31]. Though similar in aim to our work, Biggertalk is not built on top of a modules facility. Rather, it implements its own encapsulation.

Instance variables are stored just like methods only the name of the object is instantiated and the body of the axiom is simply *true*.

For each class, there is a clause of the form:

```
class(classname,[Super],[Sub],[Instances]).
```

which indicates the class's name, list of super-classes, list of sub-classes, and list of object instances. There is a clause of the form:

```
inst(objectname,[Classes]).
```

for each object instance which contains its name and list of classes (multiple inheritance is supported). Methods are stored in tuples of the form:

```
method(Tag,Name,Head,Body).
```

where the Tag is the unbound variable which will be dynamically instantiated to the name of the current object. Instance variables are stored just like methods only the name of the object is instantiated and the body of the axiom is simply *true*. Message passing is via an interpreter invoked by the **send** predicate.

Biggertalk takes advantage of the fact that Prolog has a printable external representation to store objects on disk and thus permit them to persist beyond a single session. The Gordion object server permits these disk-based object representations to be accessed concurrently by Biggertalk and Lisp users on remote workstations. The `store` message causes an object to write its external representation to the socket connected to the server. A `retrieve` message is forwarded to the server causing the external representation to be returned to Biggertalk, parsed by a definite clause grammar, and reconstructed in Biggertalk format. The system will automatically retrieve an object which exists in the server and not in Biggertalk if that object is sent a message. However, as in our work, updated objects are not automatically replaced in the database.

2.5 Summary

In this chapter we have introduced logic programming and its partial realization in the Prolog language. We have given a high level description of the Warren Abstract machine and essential details of the Sicstus Prolog implementation which forms the basis for our extensions to Prolog. We then defined the concepts of the object-oriented programming paradigm and reviewed other pieces of related work concerned with combining Prolog and objects.

The next chapter presents the design and implementation of our Object-Oriented Prolog language.

CHAPTER 3

Design and Implementation of Object-Oriented Prolog

3.1 Introduction

There are two fundamental ways to modularize programs. Programs may be divided into packages of related functions usually called libraries (e.g. a math library, a statistics library, a strings library). Alternatively, we may group functions according to the set of objects they may be applied to. The same function name may bind to different code when applied to different objects. The latter is the object-oriented approach.

This chapter presents the overall philosophy, design, and implementation of our Object-Oriented Prolog (O-OP) language. It will be seen that, in our view, object-oriented programming in Prolog amounts to a modules facility for encapsulation plus a set of name binding rules for inheritance and late binding.

3.2 Philosophy of Objects in Logic

Central to object-oriented programming is the equation $Module \equiv Class$ [62]. "Module" is an implementation level construct that governs scoping of names. Class is a programming abstraction that governs applicability of procedure implementations to objects. Our overall philosophy for combining object-oriented organization with logic programming is to use a low-level modules facility to encapsulate objects and classes. That is, modules will provide "fire walls" around code. An interpreter will provide structured binding of names across module boundaries. Both the concepts of modularity and typing are conspicuously absent from Prolog.

There are a myriad of design decisions implicit in this design. The previous chapter surveys previous "object-flavored" Prologs by other groups who have made different design decisions which, in our view, have yet to produce a useful programming language.

In making each decision, we are guided by a desire to create a more generally useful and usable programming language for implementing large systems, while retaining the most important advantages of Prolog. To that end, we elect to employ a strong concept of object identity, sacrificing logic programming's referential integrity. Referential integrity is of dubious value as it implies that predicates at high levels of a program structure are cluttered with arguments whose only function is to pass data between two low-level predicates. We base our language extensions on Prolog rather than one of its concurrent (committed choice) cousins in order to retain backtracking. We believe in the software engineering methodologies espoused by the object-oriented paradigm: structured design, data abstraction code reuse, rapid prototyping and iterative refinement, and specialization. To this end, we support a message passing interpretation of logic, encapsulation via modules, inheritance, multiple inheritance, default values, super and self messages, and libraries of classes.

3.2.1 Message $\equiv$ Goal

In Object-Oriented Prolog, the database of axioms is partitioned into modules. A module constitutes a named set of axioms or theory. The axioms within a theory comprise assertions about what is true for the model object[1] to which the theory refers. These assertions consist of both instance variables which contain the state of the object, and procedures that implement the methods.

Sending a message is interpreted as proving a goal. The recipient of a message identifies the set of axioms that are to be used to prove the goal. The goal itself is identified by the message selector. A message with selector g and arguments $(a_1, a_2, \ldots, a_n)$ to an object O is interpreted as a request to prove the goal $g(a_1, a_2, \ldots, a_n)$ using the axioms in the theory which defines object O. This is the "message passing interpretation of logic" [59].

3.2.2 Inheritance

As a large number of objects share many of their methods, differing only in their instance variables, the set of axioms which make up an object's theory are divided into two parts: the shared part and the private part. The shared part contains the method definitions and are stored in the class module. The private

[1]We will sometimes use the term "model" interchangeably with "object". Programming language level objects correspond to models of application domain entities.

part contains the instance variables and are kept in a module for each object instance. There is therefore a module for each class and for each instance of a class.

The recipient of a message identifies the set of axioms which are to be used to prove the goal indicated by the message. However, a module containing all of the axioms to be used is never actually assembled as that would require making a copy of all of the shared code for each object. Rather a set of inheritance rules with run-time binding achieve equivalent logical semantics.

Classes are linked in an *isa* hierarchy; the process of deciding which module in which to prove a goal involves searching the inheritance structure starting with the class of the object that received the message, continuing upward until a class defining code for the method is found or the root of the hierarchy is reached. Failure to find an ancestor class implementing a method or failure (in the Prolog sense) of the method itself results in the send goal failing and causes backtracking.

3.2.3 Programming in the Large

While the logic and object-oriented paradigms may be freely mixed in O-OP, it is natural to organize high levels of a program in an object-oriented (procedural) framework, while employing a more declarative style for the small, tightly circumscribed methods which implement object behavior. That is, a program is constructed first by identifying the conceptual entities in the problem domain.

These entities are grouped into classes and the operations on these classes identified. The classes are organized in an inheritance hierarchy to facilitate sharing of code among similar objects.

Next, the methods themselves must be programmed; that is, the code must be written that defines what each object type does in response to each type of message which it is capable of receiving. Methods consist of a set of Prolog clauses whose principle functor is the same as the message selector and whose body may involve sending messages to other objects or may simply query or modify the state of the recipient. The method clauses are treated non-deterministically (with backtracking), permitting multiple solutions to a send goal. In this hybrid paradigm each style may be used to its best advantage: Prolog for programming-in-the-small, and objects for programming-in-the-large.

In summary, an object names a set of axioms with which goals can be proved. The theory is partitioned into modules which correspond to the class hierarchy. So, the basic building blocks of O-OP are a modules facility and a run-time binder.

3.3 Design of Underlying Modules Facility

Each object and each class in O-OP has associated with it a module of Prolog code/data. Since standard Prolog has no notion of modules, we first present our design of a modules facility for Prolog.

The concept of reducing software complexity through modularization is well-known and essential. Conventional languages have employed procedures and abstract data typing techniques to achieve modularity. Program modules can be constructed independently and composed to form larger systems. Access to a module is permitted only via its published interface. Internal data structures and procedures are invisible outside the module. Correctness and reliability can be analyzed within modules in a way which is impossible in large, unconstrained systems. The software design and implementation process can be facilitated by transparently replacing initial, simple implementations of data structures or services with more sophisticated versions which maintain the same well-defined interface.

By contrast, relatively little work has been done on modularization in logic programming systems. In logic programming, all formulae are independent, any relationships being established at run-time by inferencing [19]. This degenerate case of modularity is a boon in prototyping small applications but renders Prolog very difficult to use for moderate to large problems.

3.3.1 Name Conflicts

Name conflicts present the major problem. A Prolog procedure is identified by its name and arity. When a large system is composed of several parts, possibly written by different programmers, it is likely that the same names have been used for different procedures. For example, one part of the program might implement a tree and employ a procedure called delete/2 to delete a node from the tree.

Another part of the system might use a predicate called delete/2 to remove an element from a queue. Depending on which part of the code was loaded first, one part of the system or the other will use the wrong version of the delete procedure. Even the one which uses the correct version might fail and backtrack into the other. The basic principle of data abstraction has been violated, i.e. a program cannot be composed out of building blocks without regard to how those pieces are implemented.

The solution to this problem is to partition the global database of a Prolog program into separate modules. Within a given module, the only visible predicates are those defined in that module, explicitly *imported* into the module, or perhaps inherited by that module. The *scope* of all the predicates defined in a module is limited to the module itself unless those predicates are explicitly *exported* (made visible outside). The ability to limit scoping allows the hiding of local objects, permitting the construction of modular software in which large systems are composed of independent components which may be integrated knowing only their interfaces.

Adding modules to Prolog is essentially a naming problem. When a goal is invoked, it names a procedure to be used to prove that goal. The naming mechanism of Prolog unifies the name of the called goal with the names of procedures in its internal database. The naming mechanism must be made to implement the scoping rules discussed above; that is, it must only unify with predicates that are currently visible in the calling environment or *context*.

3.3.2 Naming in Modules

Let us derive these naming rules, first for a simple modular Prolog, and then for object-oriented behavior. In standard Prolog, all procedures are visible to unification. We want to group procedures into modules such that by default, only those within the same module are visible. One way to look at this is to say that procedure "p" in module "M" has a unique global name "M:p". If all clauses are written using only the global names of procedures, we achieve the effect of normal Prolog without name conflicts.

We now add the notion of a context for resolving shorthand names so that we can use "p" or "q" instead of "M:p" or "M:q". The initial context consists of the *current module*, a state variable of the computation. So if the name "p" is used while the current module is M, the name refers to the global object "M:p".

3.3.3 Exports and Imports

If all names used within each module are "name extended" with that module's name, then no predicate is accessible outside of its defining module[2]. The information hiding in this scheme is absolute. We would like for modules to provide an interface; that is, a module should "export" a few entry points. These exported procedures can be named or called from other modules. The hidden, internal procedures can be called by the exported predicates but not directly by

[2]Within module M1 one might try to access the implementation of p in module M by calling M:p. However, when module M1 is name extended, this reference becomes M1:M:p which does not unify with M:p.

anyone outside of the module.

If a name is exported, we can think of it as being name extended with a special prefix, e.g. x. So, if module M exports predicate p then all occurrences of the name p within M are name extended to x:M:p. Another module, say M1 wishing to use the implementation of p exported by M must declare p to be "imported" from M. If M1 imports p from M, all occurrences of p in M1 are name extended to x:M:p such that they unify with the implementation in M. The use of the x name extension allows the system to verify that any imported predicate is actually exported by the corresponding module.

3.3.4 Implementation Via Name Translation

A name extension preprocessor system has been implemented by the author. The preprocessor for pure Prolog works just as described above, translating a modular program into ordinary Prolog. The advantage of such an implementation of modules is that it is easily portable to any Prolog environment. Most of the effort in building the modules system was devoted to correctly handling the extra-logical features of Prolog such as `call`, `clause`, `assert`, and `functor`. These require run-time interpretation which is implemented by replacing these calls with calls to a meta-interpreter. Similar independently developed name translation strategies are used in Quintus Prolog [1] and in a recent paper by Dietrich [29].

3.3.5 Further Name Binding

There is still one point in the above that causes problems. Consider the following pseudo code:

```
beginModule M1
        EXPORT g

        g(X,Y) :- f(X), ...
        f(X) :- ...
endModule

beginModule M2
        IMPORT g from M1

        h(...) :- g(W,Z), ...
        f(U) :- ...
endModule
```

Module M1 exports `g` which module M2 imports. Internally, `g` calls `f` for which a definition exists in both M1 and M2. To which version of `f` should the call bind?

In conventional languages it would be clear that the programmer of module M1 intended for the local definition in M1 to apply and the alternative definition in M2 is purely a coincidental name conflict. In object-oriented languages, on the other hand, (assuming `f` is a method and M2 inherits from M1) it is clear that the definition in M2 is meant to override that in M1. That is, M2 objects are like M1 objects except that a new definition of `f` is used. This illustrates the essential difference between name binding in modular systems and in object-oriented systems. It is to indicate which of these forms of name binding applies that object-oriented languages use a different syntax to apply methods (the message metaphor) from that used to call local procedures.

3.4 WAM Implementation of Modules

The Warren Abstract Machine provides no support for partitioning the Prolog name space. In this section, we present our design and implementation of an enhanced abstract machine supporting modules.

3.4.1 New WAM Architecture

The architecture of the Warren Abstract Machine presented in Chapter Two is modified to support modules. The WAM's global functor table is partitioned into many smaller tables, one for each module. A new register called the *Current Context* is added which points to the functor table of the *current module*, a new state variable of the abstract machine. Unless otherwise specified by a program, the new architecture treats the functor table pointed to by the Current Context register as the global database of predicates. That is, the only visible predicates are those in the functor table of the current module.

A given predicate may be simultaneously visible in more than one module as a result of its having been exported by one module and imported into others. While there is an entry for an exported predicate in the functor table of every module which imports it, there is only a single copy of the predicate's code; all functor tables point at the same code.

The module table provides access to information about modules. Entries are accessed by hashing on the module's name. An entry in the table consists of a pointer to a module structure, and a delete bit used for deallocating modules. A module structure stores the print name of the module, its property mask bits, and a pointer to its associated functor table (see Figure 3.1).

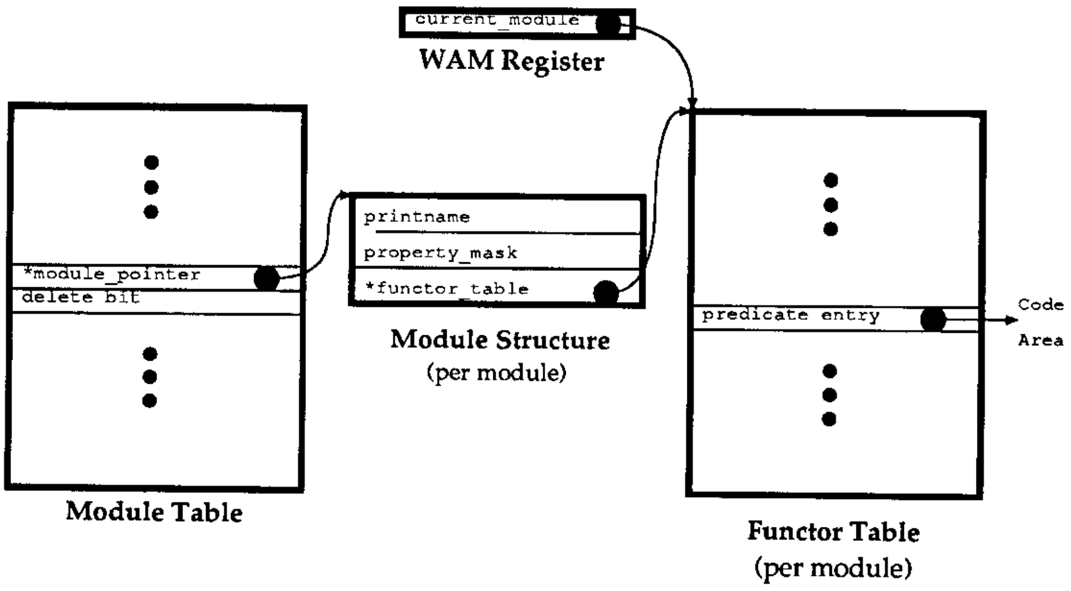


Figure 3.1: The Architecture of WAM Modules

3.4.2 Loading Modules

When the WAM boots, it first allocates all of the data areas (stacks, atom table, etc.). It then creates the "built-in context", its initial working module. All of the Prolog built-in predicates are loaded into the built-in context. For predicates that are written in C, an entry is created in the functor table for the built-in module with a pointer to the compiled C routine and a flag set indicating a C built-in. The compiled code for built-in predicates written in Prolog is consulted and an entry created in the functor table for each predicate. The built-in module is not visible to users but remains accessible to the system after booting is complete.

In the final stage of booting, the WAM creates the "user" context which is the initial module. All code that is not within a `begin_module/end_module` pair is loaded into the user module. If none of the modular Prolog features are used, the user module behaves just like the global name space of conventional Prolog.

The user module, and all subsequent modules, are initially created by copying the built-in module. In this way, modules may be created rapidly and are initialized with access to all of the Prolog built-in predicates. Measurements of the unoptimized prototype implementation show that a tight loop of Prolog code can create 100 modules per second on a Sun 3/60.

When consulting a file, a `:-begin_module(name, [imports], [exports])` may be encountered. There is no nesting of modules permitted; if the loading context is anything other than "user" when a `:-begin_module` directive is consulted, an error is generated. If the module does not already exist, it is created by allocating a slot in the module table, setting it to point to a new module structure, and setting that to point to a new functor table. If the module exists and the module does not have the "multi-file module" property set, an error is generated indicating a module name conflict.

An entry is created in the functor hash table for each predicate in the list of imports. If the module from which each predicate is imported already exists, the functor table entry is set to point to the predicate within that module. Thus the name is bound at load time. If the module has not yet been consulted, an initially empty module is created and a slot in its functor table allocated to the imported predicate. The initial instruction of the predicate is set to "undefined" until the code is eventually loaded. When code is eventually consulted for a predicate for which a dummy entry has already been created (as a result of its having been imported by another module), the loader checks that the predicate is in fact exported by the defining module. If it is not, an error is generated.

A flag indicating that the entry point is exported is set for every predicate in the export list. This flag is checked if the predicate is ever imported by another module.

All hash tables (module table, functor table, etc.) are expandable in size. Whenever they reach one-half full, their size is doubled. All pointers into tables are implemented as offsets from the top of the table to allow relocation. Thus, the only limit on the number of modules or the number of predicates in modules is the memory available on the machine and the number of bits used for offsets into the tables.

3.4.3 Procedure Calls and Backtracking

The contents of the Current Context register must be saved in the environment on the stack when a procedure is called. The register is then reloaded with the context of the called predicate. The new context may be stated explicitly via the infix colon operator (`Module:Predicate`). However, even when not explicitly stated, the context of a subgoal may differ from the calling environment if the called predicate is imported from another module. Imported predicates must be run in their defining environment so that names used internally will bind correctly. While a predicate may be pointed at by many functor tables, the name of the defining module is stored in the header information and can always be used to obtain the "home" module (available from Prolog via the new built-in `predicate_owner(Functor,Arity,HomeModule)`). The Current Context register is restored from the environment upon backtracking or subgoal termination.

3.4.4 User Interface to Modules Implementation

Figure 3.2 summarizes the new built-in predicates which provide the user interface to the modules system. The interface is a superset of that supported by Quintus Prolog.

Figure 3.3 summarizes the built-in predicates whose behavior is modified by the modules facility.

3.5 Design of Object-Oriented Prolog

We now present the design and implementation of the Object-Oriented Prolog language. It has been implemented on Sicstus Prolog's Warren Abstract Machine extended with modules as described above. It has also been ported to Quintus

:-begin_module(M,[I],[E]) Directive in consulted code. All of the code between this directive and the next :-end_module(M) is to be asserted into the module M.

:-end_module(M) Stop asserting clauses into module M.

:-import(M:P) All uses of the name P in the current module are to refer to the definition of P in module M.

:-export(P) The definitions in the active module of the predicates in the list P are available for import by other modules.

new_module(M) Create a new module named M. Module creation is a side effect not undone on backtracking.

module(M) Switch context to module M creating it if it does not exist. Context switch is backtracked, but creation is not.

use_module(M) Import into the active module all predicates exported by module M. The module M is loaded from the file of the same name in the directory named by the Unix environment variable LIBRARY if it is not a current module.

ensure_loaded(M) For Quintus compatibility. If module M is not already loaded, load the file of the same name from the library directory.

use_module(M,L) Import into the active module the definitions found in module M of the predicates in the list L.

active_module(M) Unify M with the name of the module from which the machine is currently executing. Succeeds once only.

current_module(M) M is the name of an existing module. If M is instantiated test if module M is in memory. If M is unbound, then backtrack through all modules currently in memory.

M:P P is implied by the axioms visible in module M. M and P must be non-variable.

predicate_owner(F,A,M) Module M defines a predicate with functor F and arity A.

Figure 3.2: New Built-in Predicates for Modules

current_predicate(Name,Head) Name is a predicate *in the current module* and Head is its most general form.

clause(Head,Body) Similar to current_predicate. Scope limited to current module.

listing Now lists only those clauses in the currently active module. **mlisting** is similar to listing except it does not list the bodies of imported predicates (whose implementation should not be visible). Instead it prints the clause head and the name of the source module.

assert(Clause) The clause is added to the set of axioms for the currently active module. If the clause is of the form m:p, p is instead asserted into module m.

retract(Clause) Similar to assert.

predicate_property(Head, Prop) Now has the additional properties `imported` and `exported`.

Figure 3.3: Built-in Predicates Modified by Modules Facility

Prolog.

3.5.1 Object Representation and Identity

An object is represented as a module of Prolog code. Objects do not export any predicates; thus unless the "backdoor" naming mechanism (the : operator) is used, there is no access to object internals. Use of : could be restricted to maintain absolute encapsulation, although in the current system, encapsulation is obtained by convention.

Objects are named by the corresponding module name. When an object is instantiated, the creator may assign it a meaningful name (object-id). Alternatively, a unique name may be assigned by the system. Unlike most of the other object-oriented Prolog's reviewed in the previous chapter, objects retain their identity despite changes in their state. Consequently, state changes are visible to all holders of an object's identifier.

Instance variables are represented as Prolog clauses in the object's module. The clause's functor is the name of the instance variable; the arguments store the value of the variable. As there may be more than one argument, an instance variable is, in general, a vector of values. By convention, the body of an instance variable clause is typically empty (`true`) but the system does not prevent non-empty bodies. There may be more than one clause with the same functor; the multiple values for instance variables are discovered via backtracking.

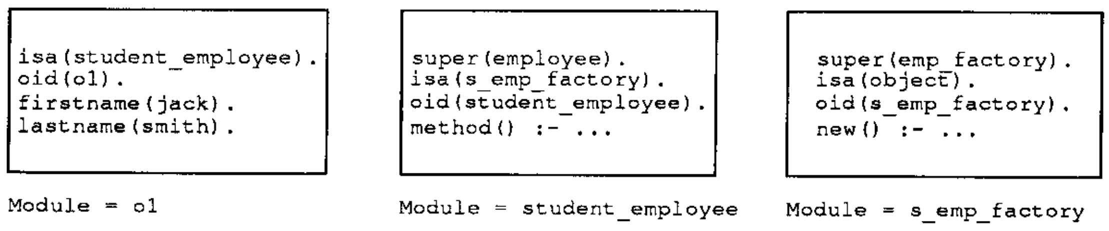
All objects have at least one distinguished instance variable, the *isa-pointer*. The isa-pointer contains the object-id of the instance's class.

3.5.2 Class Representation

Classes in O-OP are also represented as objects and hence have a corresponding Prolog module whose name is the same as the class name. The role of a class is to store the predicates that instances of the class should inherit. Class objects also play the role of *factories*[27] for creating instances of their class.

Each class is actually made up of two objects reflecting these two roles. The first acts as the repository of instance methods. The second is the repository for class methods, the methods for messages to which the class itself responds.

All class objects have two distinguished instance variables. Like all objects, class objects have an isa-pointer, the pointer to the module from which it should inherit method code when addressed with a message. When a class object receives a message (usually a request to create a new instance), it inherits its class methods from its corresponding factory object. Thus the isa-pointer for a class points to the class's factory object (see Figure 3.4) where the class methods are stored. All class objects also have a *super-pointer* which contains the object-id of the class's super class.



```
isa(student_employee).
oid(o1).
firstname(jack).
lastname(smith).
```

Module = o1

```
super(employee).
isa(s_emp_factory).
oid(student_employee).
method() :- ...
```

Module = student_employee

```
super(emp_factory).
isa(object).
oid(s_emp_factory).
new() :- ...
```

Module = s_emp_factory

Figure 3.4: Modules for Object Instance, Class, and Factory

3.5.3 Instantiation

Instances of objects are created at run-time by sending a **new** message to a class object. What actually happens when a class receives a **new** message is entirely up to the programmer of the class. Even the form of the message, its functor name and number of arguments depends of the implementation of the factory methods. However, a base instance creation method is provided by the built-in Object-object and may be inherited by all classes.

This basic method takes two arguments:

```
new(ObjectID, [InstanceVariables]).
```

The method creates a new module, generating a unique name and binding `ObjectID` to it if `ObjectID` is uninstantiated, or using its value for the module name if it is bound. The distinguished instances variable **`isa(Self)`** is first asserted into the new module followed by each element of the list of initial instance variables[3].

3.5.4 Creating Classes

In addition to creating new instances, a class is also responsible for creating new subclasses. A new subclass is created by sending an existing class a **`new_class`** message. If the class specializes the method, code for it is inherited from the class's factory object. Otherwise, as is normally the case, the method is inherited from above, often from the built-in Object-object.

The built-in method takes four arguments:

```
new_class(classname,
        [InstanceMethods],
        [ClassMethods],
        [InstanceVariables]).
```

The method creates a two new modules; the first is called by the name supplied in the first argument, the second by an internally generated unique identifier. The first module is the class object for the new class. Its isa-pointer contains

[3]`Self` is the object-id of the original recipient of the message; in this case the class of the new object instance.

the name (object-id) of the other new module and its super-pointer contains the object-id of the class which received the `new_class` message. The value of the super instance variable for the new factory is the factory object of the message recipient and the value of its isa instance variable is the object-id of the root object.

There is also a five argument version of the `new_class` method provided by the system's root object. The arguments are the same as above except that the super class of the new class is inserted as the second argument. This argument may be list valued to create classes with multiple inheritance (see section 3.5.6).

The second argument of the `new_class` message contains a list of the methods which instances of the new class are to inherit. These clauses are preprocessed to add an extra argument (see section 3.5.8) and asserted into the new class module. The third argument contains the class methods for the new class. These are similarly preprocessed and asserted into the new factory module. The final argument contains class variables (see section 3.5.9) and local predicates used by the methods. These are not preprocessed but asserted directly into the class module.

Having described how objects and classes are represented, we are now in a position to describe how the interpreter handles inheritance.

3.5.5 Sending Messages

Object-Oriented Prolog is a superset of standard Prolog; existing programs not using the object-oriented features still run and may be embedded within new object-oriented programs. Since all one can do to an object is address it with messages, the `send/2` predicate is the only user-visible addition to standard Prolog.

The infix predicate `send/2` may be freely embedded in a program. Informally, the semantics of a goal `send(Object, Message(Args))`, are "prove the goal `Message` with arguments `Args` in the context associated with `Object`." For each object instance, there is a module of Prolog code which contains the instance variables of that object, including the object identifier of its class (the "isa" pointer). For each class, there is also a module containing both instance variables (in particular a pointer to its super class) and method code.

When Prolog tries to prove the goal `send(Object, Message(Args))`, it first locates the module associated with `Object` and uses its "isa" pointer to locate the module associated with that object's class. If an implementation for the `Message(Args)` occurs in that class, it is used to attempt to prove the goal and may succeed (and thus bind some of the arguments) or fail causing backtracking. If none is found, the interpreter looks for the method in the super class. The search continues up the hierarchy until an implementation is found or the root object is reached, causing the `send` goal to fail, prompting backtracking.

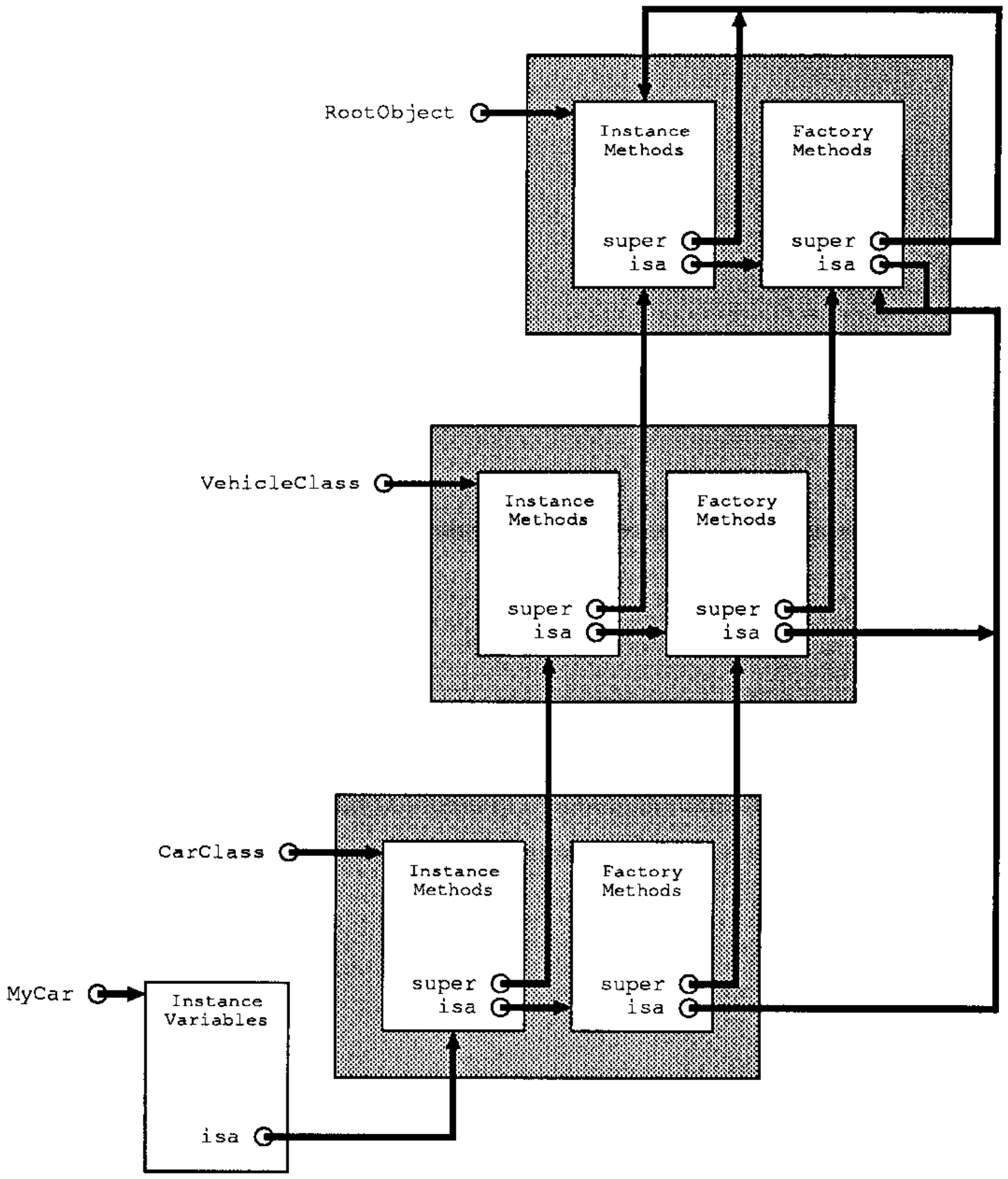


Figure 3.5: Structure of Object System

```
%
%       Send a message to an object.  Increase the arity of the
%       message by 1.  This adds the identity the recipient to the
%       message for future reference.
%
send(Object,Message) :-
        create_message(Message,Object,Newmessage),
        Object isa Class,
        inherit(Class,Newmessage).
```

Figure 3.6: Implementation of Send

Figure 3.6 shows part of the implementation of the built-in **send** predicate. **Create_message** binds **Newmessage** to a transformed version of the input **Message** which differs only by the addition of an extra argument containing the object-id of the object which received the message (see section 3.5.8 concerning the implementation of self messages). The **isa/2** predicate instantiates **Class** to the object-id of the class of the instance receiving the message. Failure of the **isa** goal is discussed in section 3.5.11. **Inherit** then tries to inherit **Newmessage** from that class (see figure 3.7).

The first clause for inherit checks whether the module for the specified class actually contains method code whose head unifies with the message. If there is such a method, a cut commits to the implementation of the method in the specified class, causing any implementations in ancestor classes to be overridden. Then **Class:Message** tries to prove the goal specified by the message using the implementation found in the class module.

```
%
%       Try to inherit the method from the superclasses.
%       We give up when the superclass of an object is itself.
%
inherit(Class,Message) :-
        Class:current_predicate(_, Message),
        !,
        Class:Message.

inherit(Class,Message) :-
        Class super Super,
        Class \== Super,
        inherit(Super,Message).
```

Figure 3.7: Implementation of Inheritance

If `Class:current_predicate(M,Message)` fails, no implementation of the method exists in this class and one must be inherited from an ancestor. The first clause fails and the second clause is tried after backtracking. `Class super Super` binds `Super` with the object-id of the super-class of `Class`. `Class \== Super` makes the check that we have not reached the root of the inheritance hierarchy (see figure 3.5). If the root is reached, the message goal fails, causing backtracking. Otherwise, via the final clause, we try to inherit an implementation from the super class.

In summary, the interpreter always follows the isa-pointer to find the message recipient's class. If the class does not implement the method it continues inheriting by following the super-pointer. Referring to figure 3.5, when the instance `MyCar` receives a message, the interpreter looks for the method in the module indicated by the `isa` instance variable (`CarClass` in this example). If the method is not found there, the interpreter continues looking up the hierarchy formed by the `super` instance variables until the method is found. The search fails when a class's `super` variable names itself, namely at the Object object.

Note that users are not intended to send messages to factory objects, only to inherit from them. Thus the `isa` variable for all factories contains the object-id of the root factory. That is, factories inherit only those methods which all objects have in common such as house-keeping methods like `delete`.

3.5.6 Multiple Inheritance

Multiple inheritance is supported via backtracking. There may be more than one "isa" or "super" instance variable in an object. Via the multiple solutions to the `isa` predicate in figure 3.6 and the `super` predicate in figure 3.7 the interpreter backtracks through the multiple inheritance paths if more than one exists. Multiple inheritance is normally used to allow an object to inherit the methods of two different classes. Usually, there is no overlap in the methods provided by the multiple ancestors. However, if a name conflict does occur, the ordering of `isa` and `super` clauses determines the order in which results are found. The cut inside the `inherit/2` predicate (figure 3.7) ensures that if more than one ancestor implements a method with the same name, only the first method found is used.

3.5.7 Multiple Solutions

While we do not permit multiple solutions via multiple inheritance[4], we do allow non-determinism within methods. `Send` goals may backtrack, leading to the discovery of multiple bindings for message arguments. There may be several clauses for a given instance variable providing multiple bindings for a single message.

3.5.8 Self Messages

The programs which implement methods are written in the context of the class of all objects to which they apply; the programmer does not know the identity of the object to which the method is applied. In fact, it will be applied to many different instances. Method execution is triggered by the receipt of a message by an instance object. Methods must have a way to refer to *Self*, the object on whose behalf they are running. The most common reason to refer to Self is to implement one method in terms of others. For example a bank account class might have a method which deposits interest in an account. This method is implemented by sending two messages to Self, first to query the balance and then credit the account.

The object-id of the instance on whose behalf a method is triggered is always added as a hidden additional argument to method predicates by the system.

[4]This is a somewhat arbitrary decision at this point and could be changed by the removal of a single cut. To date, little use has been made of multiple inheritance so an informed choice has yet to be made.

Thus if the programmer has written a method `pay_interest(Rate)`, what is actually stored in the class module is a predicate `pay_interest(Rate, Self)`. Inside the method, a variable is bound to the object-id of Self using the clause `sender(Self)`. When the method is loaded, the extra argument is added to the head, `Self` is bound to the extra argument, and the clause `sender(Self)` replaced with true. When a message is sent, it is similarly transparently extended with the hidden argument containing the identity of the recipient, thus allowing it to unify with the method.

3.5.9 Inheriting Instance Variables

When method code stored in a class is running, it is doing so on behalf of an object instance which received a message. All references to the instance variables within the method must bind to values in the context of the message recipient. Default values for instance variables may also be inherited from an object's class. Thus instance variable binding is much like message binding with inheritance only the search starts with the instance itself and not with the instance's class.

Access to instance variables is via the built-in infix predicate `inst/2` which acts very much like `send`. The variable `Self` is instantiate to the object-id of the message recipient just as with self messages above. The value of an instance variable named, for example, `instname` is then unified with the variable `Value` via `Self inst instname(Value)`.

The implementation of `Inst` (figure 3.8) also greatly resembles that of `send`.

```
%       Inherit an instance variable.  Different from send in that
%       we check for the instance variable in the object instance
%       first, and then try to inherit from ancestor classes.
%
inst(Object,VariableName) :-
        Object:current_predicate(_, VariableName),
        !,
        Object:VariableName.

inst(Object,VariableName) :-
        Object isa Class,
        inherit(Class,VariableName).
```

Figure 3.8: Implementation of Instance Variable Access

The first clause checks whether the instance variable is stored in the instance's module. If it is (`current_predicate` succeeds) the code cuts to avoid inheriting any further and unifies its second argument with the stored instance variable. If no clause for the instance variable was found in the instance's module (`current_predicate` fails), the second clause instantiates `Class` with the class of the object and tries to inherit the instance variable from above, using the same `inherit/2` predicate used by send (figure 3.7).

3.5.10 Super Messages

When a method in an ancestor class is specialized by creating another of the same name in a lower level class, the lower level version overrides the higher. However, it is sometimes convenient to implement a specialized method in terms of the version it is specializing. However, it is impossible just using **send** to call the higher level method. The interpreter would repeatedly bind to the specialized version creating a loop.

In order to call an over-ridden method, some object-oriented languages provide *super messages*. In O-OP we provide a version of send called **sendsuper**. It behaves just like send only it begins trying to inherit the method from the level above the class in which it is used. **sendsuper** only works for messages to Self[5].

3.5.11 Persistent Objects

The implementation of **send** shown in figure 3.6 treated only the case of active, in memory, objects.

As presented thus far, objects are represented as modules of Prolog code within the WAM. With each new session, all of the objects making up an application program have to be recreated by sending messages to the builtin classes. At the end of the session, the states of the objects are lost and the are recreated in their initial state by the next session.

Object-Oriented Prolog has a facility whereby objects may be made persistent. That is, their identity and state may survive the end of the session in which they are created. When an object receives a **save** message the method inherited from the Object-object causes a summary of the current state to be recorded

[5]In the current implementation, **sendsuper** requires three arguments: the object-id of the super class, the message, and the object-id of Self. In principle, however, only the message is required.

on disk. As usual, intervening classes may further specialize the **save** method. Subsequent changes to the object's state are not reflected in the disk resident version until another **save** message is received.

Objects for which a main memory representation (module) exists are called *active*. An active object may become *dormant* by saving its state and then sending it a `drop` message. The `drop` method throws away the in memory representation of the object.

Dormant objects are reactivated via a `revive` message which locates the object's image in the system's database directory and creates the main-memory representation[6]. Alternatively, dormant objects may be addressed just as active objects using their object identifier. If a dormant object is sent a message, the system transparently reactivates it and delivers the message to the object. Thus the environment incorporates a very primitive object-oriented database.

3.5.12 Mutable State

Changes in object state in Object-Oriented Prolog are represented by altering the value of instance variables. As destructive update has no meaning in pure Prolog this must be accomplished by retracting the old value and asserting a new value. In most Prolog implementations (including this one), assert and retract are relatively slow operations. Applications like transaction processing with frequent update characteristics may be impractical. However, most appli-

[6]The `revive` method is currently implemented as a builtin addition to the **send** predicate and thus cannot be specialized.

cations (our modeling environment in particular) tend not to update frequently, but rather perform complex queries on mostly static objects. The alternative of having the representation of objects returned to clients as in the majority of the related projects reviewed in Chapter 2 sacrifices object identity [48] and is therefore judged inferior.

3.5.13 Collection Classes

A very useful programming technique is the ability to group a set of objects into a unit that can be manipulated as a whole. Such a capability allows the functioning of an object-oriented language as a data model for databases. For this purpose, the system provides a Collection Class. Collections are a convenient way to implement many complex (or composite) objects allowing us to model such things as file folders, part bins, networks of queues, ledgers, symbol tables, relations and dictionaries.

An instance of a Collection Class is an object whose only instance variable is a list of other objects. The class provides basic methods to enumerate (backtrack through) the elements of a collection, insert a new object into a collection, remove an existing object from a collection, broadcast a message to all elements of a collection, etc. Thus a collection class provides a rudimentary object-oriented database querying facility.

3.5.14 Compiling Away Search

We stated that *late binding* was fundamental to the object-oriented paradigm. That is, while the programmer names a function to be applied to an object, the binding to the code that performs that function is not made until run-time. This maintains great flexibility to make localized changes in a software system without modifying or recompiling other parts of the system which might be effected. However, this flexibility comes at the expense of performing a search of the inheritance hierarchy at run-time.

This search can be avoided with minimal impact on flexibility by compiling the inheritance. If we are willing to give up the ability to change the structure of the inheritance hierarch in a particular part of the graph, we can *partially evaluate* the program with respect to the current graph. That is, all calls to **send** can be replaced by calls to the appropriate predicate in the module from which the method should be inherited. This partial evaluation strategy can be used to improve the performance of Object-Oriented Prolog running on top of any modular Prolog (e.g. Quintus).

An even higher performance approach may employed in our WAM-based Prolog engine. During the compilation phase, we can add to the module which contains an object's instance variables entries for all of its methods. These entries point to the implementation in the module from which they should be inherited. Thus the search is performed at compile time. A single copy of method code, stored in the class module, is still shared by all instances; multiple copies are not made; rather, the method is imported into the instance module creating multiple pointers to the same code. This optimization has not yet been implemented as the performance improvements for current applications (see Chapter 6) are considered small compared to other known potential improvements in the applications themselves.

3.5.15 Mixing Modules and Objects

We have said that there were two fundamental ways to modularize programs:

1. by grouping related functions (eg. a math library), or

2. by grouping functions by the type of object to which they may be applied.

While the second is the fundamental organizing paradigm of object-oriented programming, we do not have to rule out using the first as well. It is likely to be the case that the developer of an O-OP program would like to access predicates in existing libraries (e.g. the Quintus library [1]). There is no difficulty with importing predicates exported by libraries into class modules.

3.6 Example of Object-Oriented Prolog Code

The following example is due to Robert Lindell. Figure 3.9 shows O-OP code to define the class of all antennas. A new class is created by sending a `new_class` message to the super class which the newly defined class is specializing. In this case, antenna has no super class other than the system defined root object. The

```
:- object send new_class(
%
%       Class name (new class ID)
%
antenna,
%
%       Instance Methods
%
[(gain(Wave,Gain) :-
        sender(Self),
        Self send capture_area(Wave,Area),
        Gain is 4 * 3.1415 * Area / ( Wave * Wave))],
%
%       Class Methods
%
[],
%
%       Instance variables
%
[]).
```

Figure 3.9: The definition of the Antenna class.

antenna class defines only the `gain` method which unifies `Gain` with the gain (ratio of output power to input power) for a given wavelength (`Wave`). The gain method is defined in terms of the capture_area method which depends on the specific geometry of the antenna.

Figure 3.10 shows the implementation of the class of parabolic antennas, a subclass of antennas. The subclass defines the `capture_area` method as a function of the instance variables `diameter/1` and `efficiency/1`.

When the system consults the definitions of the antenna and parabolic an-

```
:- antenna send new_class(
%
%       Class name (new class ID)
%
par_antenna,

%
%       Instance Methods
%
[(capture_area(Wave,Area) :-
        sender(Self),
        Self inst diameter(Diam),
        Self inst efficiency(Eta),
        Area is 3.1415 * Diam * Diam * Eta / 4)],
%
%       Class Methods
%
[],
%
%       Instance variables (diameter/1, efficiency/1)
%
[]).
```

Figure 3.10: The definition of the Parabolic Antenna class.

tenna classes, it creates the corresponding class and factory objects. Now, to create an instance of a parabolic antenna, we send a message to the parabolic antenna class object:

```
| ?- par_antenna send new_object(Oid,
        [efficiency(0.6),diameter(5)]).
```

Since neither the parabolic antenna nor the antenna classes specialize the `new_-object` method, the built in implementation is inherited from the root object. The method creates a new module, instantiating `Oid` with the newly generated unique object-id of the new object. In our example the new object-id is `o0`. It then asserts the two instance variables into the new module.

We can now exercise the new object by sending it a message:

```
| ?- o0 send gain(0.075,G).
```

Figure 3.11 shows a partial trace of the system sending the above message (some lines have been deleted for compactness). It first uses **`create_message`** to add the object-id of the recipient object, o0, to the message (line 2 in figure 3.11). Next, it finds the class of o0 is par_antenna (16) and tries to inherit the message from there (18). Failing to find an implementation of the gain method in the par_antenna module (19), it obtains the super class, antenna and tries to prove inherit there (22). The interpreter finds the gain method in the antenna class and calls it (26).

The gain method, in turn, sends the `capture_area` message to `Self` (27). The method is inherited from the parabolic antenna class (44). It first retrieves the

```
| ?- o0 send gain(0.075,G).
1 1 Call: o0 send gain(0.075,_95) ?
2 2 Call: create_message(gain(0.075,_95),o0,_231) ? s
2 2 Exit: create_message(gain(0.075,_95),o0,gain(0.075,_95,o0)) ?
16 2 Call: o0 isa _237 ? s
16 2 Exit: o0 isa par_antenna ?
18 2 Call: inherit(par_antenna,gain(0.075,_95,o0)) ?
19 3 Call: par_antenna : current_predicate(_1191,gain(0.075,_95,o0)) ? s
19 3 Fail: par_antenna : current_predicate(_1191,gain(0.075,_95,o0)) ?
19 3 Call: par_antenna super _1189 ? s
19 3 Exit: par_antenna super antenna ?
21 3 Call: par_antenna \== antenna ?
21 3 Exit: par_antenna \== antenna ?
22 3 Call: inherit(antenna,gain(0.075,_95,o0)) ?
23 4 Call: antenna : current_predicate(_1571,gain(0.075,_95,o0)) ? s
23 4 Exit: antenna : current_predicate(gain,gain(0.075,_95,o0)) ?
25 4 Call: antenna : gain(0.075,_95,o0) ?
26 5 Call: gain(0.075,_95,o0) ?
27 6 Call: o0 send capture_area(0.075,_1911) ?
28 7 Call: create_message(capture_area(0.075,_1911),o0,_2009) ? s
28 7 Exit: create_message(capture_area(0.075,_1911),o0,capture_area(0.075,_1911,o0)) ?
42 7 Call: o0 isa _2015 ? s
42 7 Exit: o0 isa par_antenna ?
44 7 Call: inherit(par_antenna,capture_area(0.075,_1911,o0)) ?
45 8 Call: par_antenna : current_predicate(_2969,capture_area(0.075,_1911,o0)) ? s
45 8 Exit: par_antenna : current_predicate(capture_area,capture_area(0.075,_1911,o0)) ?
47 8 Call: par_antenna : capture_area(0.075,_1911,o0) ?
48 9 Call: capture_area(0.075,_1911,o0) ?
49 10 Call: o0 inst diameter(_3308) ?
50 11 Call: o0 : current_predicate(_3414,diameter(_3308)) ? s
50 11 Exit: o0 : current_predicate(diameter,diameter(_3308)) ?
52 11 Call: o0 : diameter(_3308) ?
53 12 Call: diameter(_3308) ?
53 12 Exit: diameter(5) ?
52 11 Exit: o0 : diameter(5) ?
49 10 Exit: o0 inst diameter(5) ?
54 10 Call: o0 inst efficiency(_3316) ? s
54 10 Exit: o0 inst efficiency(0.6) ?
59 10 Call: _1911 is 3.1415 * 5 * 5 * 0.6 / 4 ?
59 10 Exit: 11.780625 is 3.1415 * 5 * 5 * 0.6 / 4 ?
48 9 Exit: capture_area(0.075,11.780625,o0) ?
47 8 Exit: par_antenna : capture_area(0.075,11.780625,o0) ?
44 7 Exit: inherit(par_antenna,capture_area(0.075,11.780625,o0)) ?
27 6 Exit: o0 send capture_area(0.075,11.780625) ?
60 6 Call: _95 is 4 * 3.1415 * 11.780625 / (0.075 * 0.075) ?
60 6 Exit: 26317.39266666667 is 4 * 3.1415 * 11.780625 / (0.075 * 0.075) ?
26 5 Exit: gain(0.075,26317.39266666667,o0) ?
25 4 Exit: antenna : gain(0.075,26317.39266666667,o0) ?
22 3 Exit: inherit(antenna,gain(0.075,26317.39266666667,o0)) ?
18 2 Exit: inherit(par_antenna,gain(0.075,26317.39266666667,o0)) ?
1 1 Exit: o0 send gain(0.075,26317.39266666667) ?
G = 26317.39266666667 ?
```

Figure 3.11: Trace of message inheritance.

values of the **diameter** and **efficiency** instance variables (49 and 54). These are found in the object module itself. The capture area is then calculated (59) and returned; the **capture_area** message succeeds. The value of capture area is then used to calculate the gain (60). The original **gain** message succeeds instantiating the unbound variable G in the message to the result (1).

3.7 Summary

This chapter has presented our strategy for exercising the best features of both the object-oriented and logic programming paradigms in a single language. We designed and built a modified version of the Warren Abstract Machine to support a high performance modules facility for Prolog. We then implemented the Object-Oriented Prolog language using the modules facility for encapsulation and an interpreter for run-time binding and inheritance.

The O-OP language supports most common object-oriented concepts: super messages, self messages, mutable state, multiple inheritance, default class variables, object persistence. In addition, several features unique to the combination with Prolog are supported: multiple solutions, backtracking across messages, enumerating members of a collection via backtracking. An example showed the definition of a class and its super class and presented a Prolog trace of the interpreter's execution as an instance object was sent a message.

The next chapter presents the formal abstract semantics of Object-Oriented Prolog.

CHAPTER 4

Formal Semantics of Modular and Object-Oriented Prolog

4.1 Conventional Prolog Semantics

Horn clause logic program semantics are usually expressed in terms of the model theory or fixpoint theory of first-order logic. Briefly, an *interpretation* of a program P is a subset of the Herbrand Base of P (the set of all goals formed from the predicate symbols, constants, and function symbols in P). An interpretation, I, is a model for P if for each ground instance of a clause $A \leftarrow B_1, \ldots, B_n$ in P, $A \in I$ if $B_1, \ldots, B_n$ are in I. The intersection of two models is also a model. Most importantly, the intersection of all models is a model, called the *minimal model.* The minimal model, $M(P)$, is the *declarative* meaning of P [7,32,55].

In fixpoint semantics, the meaning of a program is viewed as the input-output relation as expressed by the least fixpoint of the transformation associated with the program [95]. Given a program P, there is a mapping on interpretations denoted $T_P : I \rightarrow I$ defined:

$T_P(I) = \{A \textit{ in } B(P)$ such that $A \leftarrow B_1, \ldots, B_n$ $n \geq 0$ is a ground instance of a clause in P and $B_1, \ldots, B_n$ are in $I\}$.

$T_P(I)$ simply models a single deduction step. We can define the exponentiation of T to apply the function to an initial interpretation multiple times. As T_P is continuous and hence monotonic ($I_1 \sqsubseteq I_2 \Rightarrow T_P(I_1) \sqsubseteq T_P(I_2)$), by the Tarski Fixed Point Theorem, $T_P^{\omega} = fix(T_P) = M(P)$ (where ω is the least infinite ordinal). This important result relating the model theoretic (declarative) semantics with the fixpoint (denotational) semantics is due to Van Emden and Kowalski [32].

Unfortunately, these "logical" semantics fail to account for the way Prolog is actually implemented and used in practice (cuts, depth first search, var check, database access). An operational semantics can come much closer to modeling practical logic programming languages.

Operational semantics also define the meaning of a program as its input-output relation, but as computed by the program causing state transitions on an abstract machine. The machine may or may not model a practical implementation of the language. Operational semantics for a logic program P are traditionally expressed as the set of goals G (a subset of the Herbrand Base) such that $\forall g \in G,\ P \perp g$ where $X \perp Y$ means there exists a derivation of Y from X. When the program is interpreted by SLD resolution [81], $X \perp Y$ means there exists a refutation of $X \wedge \neg Y$ and it can be shown that the set G is equal to the minimal model $M(P)$ [32].

Once again, however, the traditional semantic basis for logic programming is inadequate for modeling Prolog as it is most often used. In particular, there is little point in basing our semantics for object-oriented and modular Prolog on the original logical semantics as we make valuable use of cuts, goal ordering, and access to the database. Further, in practice, we are concerned not simply with the "success set" of a program, but with the set of substitutions or variable bindings which result in the refutation. We will therefore define our operational semantics using a more sophisticated abstract machine and our denotational semantics using the Strachey-Stoy style more commonly used for specifying conventional languages.

4.2 Relation to Other Work

There are several proposals for extending the semantics of logic programming to accommodate modularity. O'keefe proposes an algebra for combining blocks of code with associated signature morphism (corresponding to our import- export declarations) [75]. However, this semantics is for a fictional logic language, not Prolog as it is used. Sannella proposes a similar calculus for modular logic [84]. [68,17,2,64] define interesting higher order or meta-level semantics for modular logic programming. These proposals take the point of view of constructing a modules facility out of a mathematically interesting semantic extension to logic which has the potential to produce a very clean semantics.

We have taken a different tack; we have admitted that Prolog as a programming language is only an approximation of first order Horn clause logic. We have extended that programming language with modules in such a way as to improve its usability for programming. At the same time, we have moved it further away from its pure logic semantics. Rather than trying to produce a programming language based on a mathematically beautiful semantics, we have designed a programming language and used the available mathematical tools to specify its semantics. While we are encouraged that the "high road" approach of the above mentioned related work may one day yield a usable language, our "low road" approach is in use today.

4.3 Operational Semantics

4.3.1 Background

This operational semantics is loosely based on the definition of standard Prolog (without modules) due to Jones and Mycroft [45]. The semantics of a Prolog program in this model are defined as the (possibly infinite) sequence of substitutions representing the alternative bindings for variables in the query produced by resolution via backtracking. Thus, we define an abstract machine whose input is a set of modules of Prolog code along with a query, and whose output is this sequence of substitutions.

4.3.2 Syntax

We assume the existence of four disjoint sets of symbols: *Pred*, *Functor*, *Var*, and *ModName*, ranged over by the syntactic variables P, F, V and M respectively.

Notation

The notation α^* stands for the set of finite sequences of elements of the domain α.

$\alpha^* = \{nil\} + \alpha \times \alpha^*$

Here $+$ is the disjoint union operator. We use ::, a right-associative infix cons operator to construct sequences. The notation [] will sometimes be used for *nil* while $[\alpha_1, \alpha_2, ..., \alpha_n]$ will represent $\alpha_1 :: [\alpha_2, ..., \alpha_n]$ where convenient.

4.3.2.1 Domains and Syntactic Equations

$$
\begin{array}{rcl}
t & : & Term = V + F(t^*) \\
a & : & Atom = P(t^*) \\
b & : & Body = (a + \{\text{``!''}\} + X)^* + true \\
c & : & Clause = a :- b. \\
Z & : & Sentence = c^* \\
X & : & ModExp = M : a \\
E & : & ExportList = [P|P^*] + [] \\
I & : & ImportList = [[P, M]|I^*] + [] \\
W & : & Module = :-\texttt{begin\_module}(M, E, I).\ c^* \ :-\texttt{end\_module}(M). \\
Q & : & Program = W^*;\ :-b.
\end{array}
$$

4.3.2.2 Functions

Substitution

$\theta, \phi : \; Subst = Var \rightarrow Term$

Substitutions are applied in a prefix form allowing standard functional composition notation (e.g. $\theta \circ \phi$). The identity substitution will be written *id*.

Renaming

Rename functions are used to guarantee that an atom and clause selected for resolution share no common variables. The method (suggested by [45]) for performing renaming is as follows: Partition the set *Var* into a countable number of disjoint isomorphic sets $\{Var_0, Var_1, \ldots\}$ where Var_0 contains the set of valid variables that may be used in programs. Then employ a sequence of bijective renaming functions:

$\psi_n : \; Var_0 \rightarrow Var_n \;\; (n > 0)$

where ψ_0 is the identity function.

Unification

$MGU : \; Atom \rightarrow Atom \rightarrow (Subst + \{fail\})$

Robinson first provided an algorithm for determining the most general unifier of two atoms [81]. In the interpreter which follows, a renamed body is represented by the tuple $(body, n)$ where n represents the renaming ψ_n. By deferring application of substitutions and renamings, *structure sharing* interpreters can avoid creating new atoms and terms. In our semantics, we will assume the existence of MGU_{ss}, a version of MGU for structure sharing that is very close to that used by most modern Prolog implementations. [18] and [80] detail the algorithm which realizes this function without actually performing the substitutions.

$MGU_{ss} :\ Subst \rightarrow Atom \rightarrow Num \rightarrow Atom \rightarrow (Subst + \{fail\})$

$MGU_{ss}(\phi, a_1, n, a_2) = MGU(\phi(a_1), \psi_n(a_2))$

The Prolog Database

$D :\ Database = ModName \rightarrow Clause^*$

The abstract machine accepts programs of the form $Module^* \wedge \neg q^*$. We assume that the system can translate $Module^*$ into the initial database function D_0. For a more formal treatment, we could include the database function in the state starting with an initial database $D_0 = \lambda ModName.nil$. We would then define a built in "consult" predicate which loads modules, changing the Database function. This method could be used similarly to model other predicates having non-logical side-effects modify the database (such as `assert` and `retract`).

We have also assumed the ability to freely access another function:

$home: \ atom \rightarrow ModName \rightarrow ModName$

This function, given a predicate and a module name, returns the module in which the predicate is defined. Recall that a predicate may be imported from another module. This function returns the name of the "source" module and may be thought of as a different view of the database. Again, for a more formal treatment, we would add a variable ranging over functions of this type to the state and provide the requisite transition rules to modify the function as modules are consulted. In keeping with the greater formality customary for denotational semantics, we will define the *database* and *home* functions in a later section detailing the denotational meaning of modular Prolog programs.

4.3.3 The Abstract Machine

$\sigma: \ State = (GoalStack \times Subst \times Clause^{*} \times Num)^{*}$

$s: \ GoalStack = (Body \times Num \times Dump \times ModName)^{*}$

$d: \ Dump = State$

$Start\ State = ((q, 0, nil, user) :: nil, id, D_0(user), 0) :: nil$

$Stop\ State = nil$

The *State* of the machine is represented as a runtime stack storing a sequence of choice points. A choice point is a four-tuple (s, ϕ, c^*, n) consisting of a *GoalStack*, an accumulated answer substitution, a sequence of clauses in the current module remaining to be tried, and the current renaming index. A *GoalStack* is itself a sequence of four-tuples consisting of a goal body, a renaming index for that goal, the tail of the choice point stack (for cuts), and the name of the module in which the current goal is to be proved. The *Dump* portion of the goal stack saves the state at the time the current goal was called. If a cut is encountered, the state saved in the *Dump* is restored thereby removing all choice points created since the call.

As Jones and Mycroft point out, the sequence of clauses in *State* as well as the goals in *Stack* can be implemented as pointers into the original program to avoid copying. Similarly, as the *Dump* is always equal to the terminal segment of the *State*, it may be implemented as a pointer.

Transition Rules

The transition rules define a relation $\rightarrow$ on $State \times State$.

TR1 Apply clause saving σ as dump for cuts.

$$((a \wedge b, m, d, M) :: s, \phi, (a' :- b') :: c^*, n) :: \sigma$$

$$\rightarrow ((b', n+1, \sigma, M') :: (b, m, d, M) :: s,$$

$$\theta \circ \phi, D(M'), n+1) :: \sigma' \textbf{ where } M' = home(a', M)$$

$$\textbf{if } \theta = MGU_{ss}(\phi \circ \psi_m, a, n+1, a') \textbf{ exists}$$

$\rightarrow\ \sigma'$ **otherwise**

where $\sigma' = ((a \wedge b, m, d, M) :: s, \phi, c^*, n) :: \sigma$

Try to unify the current goal, a, with a' where $a' :- b'$ is the first clause in the sequence of untried clauses. If a most general unifier θ exists, compose it with the existing accumulated substitution ϕ and save it in the *State* along with the incremented renaming index. Add a tuple to the top of the *Stack* with current goal b' renamed using index $n + 1$; save the previous *State* σ in the *Dump*; and run the new goal in its home module M'. The sequence of clauses in the *State* (to be utilized in refuting b') is also initialized to $D(M')$, the complete set of clauses in module M'. If no such unifier exists, simply remove the top clause from of the sequence of untried clauses and try the next one.

TR2 Goal not further satisfiable, backtrack.

$((a \wedge b, m, d, M) :: s, \phi, nil, n) :: \sigma \rightarrow\ \sigma$

If the sequence of untried clauses is exhausted, the interpreter backtracks resetting the *State* to the last choice point.

TR3 Cut operator.

$((\text{“!”} \wedge b, m, d, M) :: s, \phi, c^*, n) :: \sigma$

$\rightarrow\ ((b, m, d, M) :: s, \phi, c^*, n) :: d$

Replace the tail of the choice point stack with the stack saved in the *Dump*. This effectively eliminates all choice points created after the *Dump* was saved thereby committing to the current bindings.

TR4 Satisfied goal. Continue with siblings.

$$((true, m, d, M) :: s, \phi, c^*, n) :: \sigma$$

$$\rightarrow \ (nil, \phi, c^*, n) \ \textbf{if} \ s = nil$$

$$\rightarrow \ (s, \phi, D(M'), n) :: \sigma \ \textbf{otherwise}$$

$$\textbf{where} \ M' = ModuleName(Head(s))$$

When the current goal is reduced to *true*, pop off the top of the *GoalStack*, resetting the current module and sequence of clauses to try, so long as the *GoalStack* is not now empty. *ModuleName* and *Head* are auxiliary functions, extracting the module name from a *GoalStack* frame and returning the top frame from a sequence respectively.

TR5 Satisfied main goal. Backtrack.

$$(nil, \phi, c^*, n) :: \sigma \rightarrow \sigma$$

Output answer substitution ϕ.

When the *GoalStack* is empty, output the current answer substitution and backtrack to produce alternative solutions.

If the machine halts in the stop state never having taken this transition and thus producing no outputs, the query is said to fail. There is no guarantee the machine will ever halt. It may proceed infinitely never producing output; produce an infinite sequence of outputs; or produce a finite number of outputs but never halt. In practice, real Prolog implementations provide a way to extract the partial output of the machine even though the interpreter may never halt.

TR6 Goal in another module.

$$((M' : a \wedge b, m, d, M) :: s, \phi, c^*, n) :: \sigma$$
$$\rightarrow ((a, m, d, M') :: (b, m, d, M) :: s, \phi, D(M'), n) :: \sigma$$

Colon is a binary operator which says, "Run the goal in the second argument in the module given in the first argument." Here, the goal $M' : a \wedge b$ is replaced with two goals which first run a in module M', and then run b in the original module M. Note that : binds more tightly than does $\wedge$.

4.4 Semantics of Object-Oriented "Send"

The `send` predicate is the heart of our object-oriented extension to Prolog. Informally, `Object send Message` means, "If a predicate unifying with `Message` exists in the module associated with the class of `Object`, then attempt to satisfy `Message` in that module. Otherwise, find the super class and try to inherit from there, failing if the root object is reached without finding a definition."

We can define the semantics of `send` by assuming the following two predicates are defined in every module.

```
send(Object, Message) :-
     Object isa Class,
     inherit(Class, Message).

inherit(Class, Message) :-
     Class:current_predicate(_, Message),
     !,
     Class:Message.
inherit(Class, Message) :-
     Class super Super,
     Class \== Super,
     inherit(Super, Message).
```

Adding these predicates to every module is equivalent to adding the following additional transition rules to the abstract machine.

TR7. `Send` predicate.

$$((\mathtt{send}(M',a) \wedge b, m, d, M) :: s, \phi, c^*, n) :: \sigma$$

$$\rightarrow \; ((\mathtt{isa}(M',M'') \wedge \mathtt{inherit}(M'',a), n+1, \sigma, M')$$

$$:: (b, m, d, M) :: s, \phi, D(M'), n+1) :: \sigma$$

TR8. `Inherit` predicate.

$$((\mathtt{inherit}(M',a), m, d, M) :: s, \phi, c^*, n) :: \sigma$$

$$\rightarrow \; (M' : a, m, d, M) :: s, \phi, c^*, n) :: \sigma$$

$$\textbf{if } a \text{ defined in } M'$$

$$\rightarrow \; ((M' : \mathtt{super}(M',M'') \wedge \mathtt{inherit}(M'',a), n+1, \sigma, M)$$

$$:: s, \phi, D(M'), n+1) :: \sigma$$

$$\textbf{if } M' \neq M''$$

$$\rightarrow \sigma \textbf{ otherwise}$$

Transitions 7 and 8 add no power to the language; equivalent operations could be obtained by plugging the Prolog definitions above into the first six transitions. Note that for clarity of presentation, the notation used in transitions 7 and 8 is not exactly that obtained by pushing the Prolog definitions through the production rules. In particular, the built in predicate `current_predicate/2` is handled informally, and the "cut" in `inherit` which commits to an implementation of a method if one exists is modeled via the if-else construct in rule 8.

4.5 Denotational Semantics

4.5.1 Background

Denotational semantics defines meaning as a function from input to output, skipping over all of the intermediate states. The paradigm is to define the meaning of the program as a function which is itself defined in terms of other functions which define the meaning of individual components of the program. The meaning of each component is further defined by the meaning functions of its sub-components until a primative layer is reached. The primatives have some generally agreed to meaning. For example, the meaning of $[\![3]\!]$ is the number 3. Following the usual notation, sub-components of the original program are surrounded by double brackets. This approach to programming language semantics has a firm mathematical basis due primarily to the work of Strachey [96] and Scott [89,87,88] and is written in Church's λ-calculus notation with data-types [23]. The background assumed for reading this is to be found in [5] or [39].

Our denotational semantics is loosely based on the definition of standard Prolog (without modules) due to Jones and Mycroft [45]. Again, the semantics of a program in this model are defined as the (possibly infinite) sequence of substitutions representing the alternative bindings for variables in the query.

4.5.2 Syntax

Same as operational semantics.

Further Notation

Using notation from domain theory, we define the sequence of substitutions output by the denotational semantics as

$$Subst^{+} = \{nil\}_{\perp} + Subst \times (Subst^{+})_{\perp}$$

where	$+$	is the coalesced sum
	$\times$	is the smash product
and	$S_{\perp}$ is the domain S augmented with a new least element $\perp$.	

$\hat{}$ is a domain constructor such that $\alpha\hat{}$ defines a finite sequence of elements from the domain α terminated by *nil* or *cut* or an infinite sequence of elements of α.

$$\alpha\hat{} = \{nil, cut\}_\perp + \alpha \times (\alpha\hat{})_\perp$$

The standard append function @ on sequences is extended as follows:

$$@ : \ \alpha\hat{} \rightarrow \alpha\hat{} \rightarrow \alpha\hat{}$$

$$(a :: k)@m = a :: (k@m)$$

$$nil@m = m$$

$$cut@m = cut$$

$$\perp@m = \perp$$

Thus, @ acts like the standard append function on *nil*-terminated lists but ignores the second parameter if the first is terminated by *cut* or $\perp$. This definition on finite elements of $\alpha\hat{}$ is lifted to infinite sequences by defining:

$$l@m = \bigsqcup_{\substack{l' \sqsubseteq l \\ l' \ finite}} l'@m$$

What is meant by appending infinite objects? If l is infinite, it can be written $[l_0, l_1, \ldots]$ and is the least upper bound of the increasing chain of finite approximations

$$\perp \sqsubseteq (l_0 :: \perp) \sqsubseteq (l_0 :: (l_1 :: \perp)) \sqsubseteq \cdots$$

The meaning of $l@m$ for infinite l is the $limit_{l' \rightarrow l} l'@m$.

While this limit is a correct mathematical definition for appending infinite sequences, for all practical purposes, we can define $l@m = l$ when ever l is infinite. This can be seen by recalling that the finite approximations of l are all terminated by $\perp$ and the definition of $\perp@m = \perp$. In the denotational semantics which follows, this corresponds to the case where depth first search of the left hand side of the SLD-tree produces an infinite sequence of bindings (l). The right hand side may produce additional bindings (m) but the the interpreter would require 2ω steps to explore the right branch. The fixpoint semantics, like current implementations, produces only those bindings discovered in ω steps. Thus this definition of @ conforms with the intuitive intent of the semantics.

4.5.3 Semantic Domains

The semantic function **V** maps the collection of input modules W^* and a negated query q to the sequence of answer substitutions which define the program's semantics.

$\mathbf{V} : Program \rightarrow Subst^+$

Environments are used to associate meaning with predicate symbols. An environment maps an *Atom* with appropriate renaming information and input substitution to a sequence of substitutions.

$\rho : Env = Atom \rightarrow Num \rightarrow Subst \rightarrow Num \rightarrow ModName \rightarrow$
$\quad (Subst \times Num)^+$

Next, the meaning of a program is defined in terms of the meaning of the components of the program. The function **D** defines the meaning of a sequence of modules as a function over environments. The function also requires the *Export* relation which maps a $Pred \times ModName$ pair to a boolean allowing the semantics to check whether the given predicate is actually exported by the it is imported from.

$\mathbf{D} : \; Module^* \rightarrow Export \rightarrow Env \rightarrow Env$

The meaning of a sequence of clauses is given by **E**.

$\mathbf{E} : \; Sentence \rightarrow Env \rightarrow Env$

The semantics of importing clauses into a module is defined by the semantic function **F**.

$\mathbf{F} : \; ImportList \rightarrow Export \rightarrow Env \rightarrow Env$

Finally, the meaning of a goal with respect to an environment, renaming, and input substitution is defined by **B**.

$\mathbf{B} : \mathit{Body} \to \mathit{Num} \to \mathit{Env} \to \mathit{Subst} \to \mathit{Num} \to \mathit{ModName} \to$

$(\mathit{Subst} \times \mathit{Num})^{\hat{}}$

4.5.3.1 Definition of Semantic Functions

$$\mathbf{V}\,[\![W^*; :-q]\!] = \mathit{First}(\mathbf{B}\,[\![q]\!]\; 0\; \rho\; \mathit{id}\; 0\; \texttt{user})$$

$$\textbf{where } \rho = \mathit{fix}_{\mathit{Env}} \mathbf{D}\,[\![W^*]\!]\;\eta$$

$$\eta = \xi\,[\![W^*]\!]\eta_0$$

$$\eta_0 = \emptyset$$

The operation of $\mathbf{V}$ may be understood as follows: First, $\rho : \mathit{Env}$ is created by taking the fixpoint of the function $\mathbf{D}\,[\![W^*]\!]\eta$. The Env is the initial database function built out of the input modules which computes the sequence of substitutions refuting a simple query; it may be thought of as a "lookup" function. The relation $\eta : \mathit{Export}$ on $\mathit{Pred} \times \mathit{ModName}$ which allows us to check that a given predicate is actually exported by the module we try to import it from is built by passing the initial empty relation through each module accumulating all pairs (P, M) such that P is exported by M. Since $\mathbf{D}\,[\![W^*]\!]\eta$ is a continuous, monotone function on $\mathit{Env} \to \mathit{Env}$, its least fixpoint is guaranteed to exist. $\mathit{fix}_{\mathit{Env}}\mathbf{D}\,[\![W^*]\!]\eta$ is equal to the function $\rho \in \mathit{Env}$ such that $\mathbf{D}\,[\![W^*]\!]\eta\rho = \rho$.

Then, the query q is run by the semantic function **B**. $\mathbf{B}\,[\![q]\!]$ is passed the initial renaming index 0, the initial environment function ρ, the identity substitution function id, and the initial module "`user`". Finally, the utility function, $First$, extracts the sequence of substitutions from the sequence of $(Subst, Num)$ pairs.

Together, **D**, **E** and **F** process the input sequence of modules W^* to create the environment $(\rho : \; Env)$. Conceptually, the idea behind **D** is that given a summary of what other modules mean as characterized by the input environment, we can create a "better defined" environment by adding the meanings of the remaining modules. Trivially, the empty sequence of modules adds no information:

$$\mathbf{D}\,[\![[]]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M = []$$

For a non-empty sequence of modules, **D** passes the clauses in the first module through **E**, import definitions in the header of the first module through **F**, and calls itself recursively on the remaining modules. That is, **D** steps through the sequence of modules; it produces an environment ρ which, when passed a in module M', peals off modules from the input until the definition of module M' is found. It then tries to refute a using the clauses in M' (via **E**) or via predicates imported into M' in the header (using **F**).

$$\mathbf{D}\,[\![:-\mathtt{begin\_module}(M', E, I)\ c^*\ :-\mathtt{end\_module}(M')\ W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M$$

$= \mathbf{E}\,[\![c^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M\ @\ F\,[\![I]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M$ **if** $M' = M$

$= \mathbf{D}\,[\![W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M$ **otherwise**

The following permits an initial sequence of program clauses to default to the **user** module.

$\mathbf{D}\,[\![c^* W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M$

$= \mathbf{E}\,[\![c^*]\!]\ \rho\ a\ m\ \phi\ n\ M$ **if** $M =$**user**

$= \mathbf{D}\,[\![W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M$ **otherwise**

E captures the meaning of a sequence of clauses within a model again as a function on environments. The empty sequence adds nothing to the definition.

$\mathbf{E}\,[\![[]]\!]\ \rho\ a\ m\ \phi\ n\ M = []$

The meaning of a non-empty sequence of clauses $a' : -b' :: c^*$ is a function which, given an initial environment ρ produces a more defined environment. The resulting environment, given an *Atom* a, creates a sequence of substitutions by appending the sequence of bindings (σ') produced using only clauses c^* to the bindings produced using the first clause (σ). σ is equal to *nil* unless the head of the first clause unifies with a under the appropriate renaming and substitution. If the unifier exists, **B** is used to refute the body b'. σ' is created by calling **E** recursively with the tail of the clause sequence (c^*).

$\mathbf{E}\,[\![a' : -b' :: c^*]\!]\ \rho\ a\ m\ \phi\ n\ M = uncut(\sigma @ \sigma')$

$\quad$ **where** $\sigma = \mathbf{B}\,[\![b']\!]\ (n+1)\ \rho\ (\theta \circ \phi)\ (n+1)\ M$

$\quad\quad\quad$ **if** $\theta = MGU_{ss}(\phi \circ \psi_m, a, n+1, a')$ **exists**

$\quad\quad$ $=$ [] **otherwise**

$\quad$ **and** $\quad \sigma' = \mathbf{E}\,[\![c^*]\!]\ \rho\ a\ m\ \phi\ n\ M$

F is similar to **E** except it steps through the list of imports rather than the sequence of clause definitions. Internally, **F** ensures that an imported clause is actually exported by the specified module. This is accomplished by checking that if P is being imported from module M, the pair (P, M) is in the relation η. If it is not, the predicate is simply not used, causing the goal to fail. The semantics might have been defined to generate an error message in this instance.

$\mathbf{F}\,[\![[]]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M = []$

$\mathbf{F}\,[\![[[P, M']|I]]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M = uncut(\sigma @ \sigma')$

$\quad$ **where** $\sigma = \mathbf{B}\,[\![M' : a]\!]\ m\ \rho\ h\ \phi\ n\ M$

$\quad\quad\quad$ **if** $(P, M') \in \eta$ **and** $P = functor(a)$

$\quad\quad$ $=$ [] **otherwise**

$\quad$ **and** $\quad \sigma' = \mathbf{F}\,[\![I]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M$

The semantic function **B** uses its input environment ρ to map a goal body to its sequence of substitution functions. When applied to the trivial body *true* representing a successful computation, **B** returns its current accumulated substitution function.

$\mathbf{B}\,[\![true]\!]\ m\ \rho\ \phi\ n\ M = [(\phi, n)]$

When applied to a conjunction of the form $(a \wedge b)$ running in module M, the first atom a is simply replaced with $M : a$ and $\mathbf{B}$ is run on the renamed body.

$\mathbf{B}\,[\![a \wedge b]\!]\ m\ \rho\ \phi\ n\ M = \mathbf{B}\,[\![M : a \wedge b]\!]\ m\ \rho\ \phi\ n\ M$

$\mathbf{B}$ handles cut ("!") as the first atom in the body by appending the *cut* symbol to the sequence of substitutions produced by calling $\mathbf{B}$ recursively on the rest of the body. Recall at @ behaves like a normal concatination operator if the first parameter is terminated with *nil* but ignores the second parameter if the first ends in *cut*. Thus, the $(\sigma @ \sigma')$ in the definition of $\mathbf{E}$ models cuts by ignoring the alternative bindings σ' if σ ends in *cut*. The *uncut* function applied to $(\sigma @ \sigma')$ limits the scope of the cut to the current body.

$\mathbf{B}\,[\![\text{"!"} \wedge b]\!]\ m\ \rho\ \phi\ n\ M = (\mathbf{B}\,[\![b]\!]\ m\ \rho\ \phi\ n\ M)\ @\ cut$

Finally, the meaning of $M' : a \wedge b$ in an environment ρ with current module M is the concatination of the sequences of substitutions produced by recursively calling $\mathbf{B}$ on the tail of the body for each set of bindings produced by running the head of the goal conjunct a in module M'. That is, the environment ρ is used to obtain the sequence of bindings which refute a in module M'. Then, the rest of the body b is repeatedly refuted in module M, once for each of the bindings produced by refuting a in M'.

$\mathbf{B}\,[\![M' : a \wedge b]\!]\ m\ \rho\ \phi\ n\ M = Bindings(\rho\,[\![a]\!]\ m\ \phi\ n\ M')$

where $Bindings(X) = nil$ **if** $X = nil$

$= \mathbf{B}\,[\![b]\!]\ m\ \rho\ (\theta \circ \phi)\ n'\ M$

$@Bindings(tail(X))$ **otherwise**

where $(\theta, n') = head(X)$

4.5.3.2 Handling Imports and Exports

$\eta : Export = 2^{Pred \times ModName}$

$\xi : Exp = Module^* \rightarrow Export \rightarrow Export$

$\xi\,[\![[]]\!]\eta = \eta$

$\xi\,[\![WW^*]\!]\eta = \xi\,[\![W^*]\!](\xi\,[\![W]\!]\eta)$

$\xi\,[\![:-\texttt{begin\_module}(M, E, I).\ c^*\ \ :-\texttt{end\_module}(M).]\!]\eta = \pi\,[\![E]\!]\eta M$

$\pi : ExportList \rightarrow Export \rightarrow ModName \rightarrow Export$

$\pi\,[\![[]]\!]\eta M = \eta$

$\pi\,[\![[P_1|P_2^*]]\!]\eta M = \pi\,[\![P_2^*]\!]\eta_2 M$

where $\eta_2 = \eta \cup \{(P_1, M)\}$

In order to be imported by one module, a predicate must be exported by another. The function ξ builds up the *Export* relation η by pushing the cur-

$$
\begin{aligned}
&\mathbf{V}[\![W^*; :-q]\!] = First(\mathbf{B}[\![q]\!]\ 0\ \rho\ id\ 0\ \texttt{user})\\
&\qquad\quad \textbf{where } \rho = fix_{Env}\mathbf{D}[\![W^*]\!]\ \eta\\
&\qquad\qquad\qquad \eta = \xi[\![W^*]\!]\eta_0\\
&\qquad\qquad\qquad \eta_0 = \emptyset
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{D}[\![[]]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M = []\\
&\mathbf{D}[\![:-\texttt{begin\_module}(M', E, I)\ c^*\ :-\texttt{end\_module}(M')\ W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M\\
&\qquad\quad = \mathbf{E}[\![c^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M\ @\ F[\![I]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M \quad \textbf{if } M' = M\\
&\qquad\quad = \mathbf{D}[\![W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M \quad \textbf{otherwise}\\
&\mathbf{D}[\![c^*W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M\\
&\qquad\quad = \mathbf{E}[\![c^*]\!]\ \rho\ a\ m\ \phi\ n\ M \quad \textbf{if } M = \texttt{user}\\
&\qquad\quad = \mathbf{D}[\![W^*]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M \quad \textbf{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{E}[\![[]]\!]\ \rho\ a\ m\ \phi\ n\ M = []\\
&\mathbf{E}[\![a' :-b' :: c^*]\!]\ \rho\ a\ m\ \phi\ n\ M = uncut(\sigma @ \sigma')\\
&\qquad \textbf{where } \sigma = \mathbf{B}[\![b']\!]\ (n+1)\ \rho\ (\theta \circ \phi)\ (n+1)\ M\\
&\qquad\qquad\qquad\quad \textbf{if } \theta = MGU_{ss}(\phi \circ \psi_m, a, n+1, a')\ \textbf{exists}\\
&\qquad\qquad\qquad = []\ \textbf{otherwise}\\
&\qquad \textbf{and} \quad \sigma' = \mathbf{E}[\![c^*]\!]\ \rho\ a\ m\ \phi\ n\ M
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{F}[\![[]]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M = []\\
&\mathbf{F}[\![[[P, M']|I]]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M = uncut(\sigma @ \sigma')\\
&\qquad \textbf{where } \sigma = \mathbf{B}[\![M' : a]\!]\ m\ \rho\ h\ \phi\ n\ M\\
&\qquad\qquad\qquad\quad \textbf{if } (P, M') \in \eta \textbf{ and } P = functor(a)\\
&\qquad\qquad\qquad = []\ \textbf{otherwise}\\
&\qquad \textbf{and} \quad \sigma' = \mathbf{F}[\![I]\!]\ \eta\ \rho\ a\ m\ \phi\ n\ M
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{B}[\![true]\!]\ m\ \rho\ \phi\ n\ M = [(\phi, n)]\\
&\mathbf{B}[\![a \wedge b]\!]\ m\ \rho\ \phi\ n\ M = \mathbf{B}[\![M : a \wedge b]\!]\ m\ \rho\ \phi\ n\ M\\
&\mathbf{B}[\![\text{``!''} \wedge b]\!]\ m\ \rho\ \phi\ n\ M = (\mathbf{B}[\![b]\!]\ m\ \rho\ \phi\ n\ M)\ @\ cut\\
&\mathbf{B}[\![M' : a \wedge b]\!]\ m\ \rho\ \phi\ n\ M = Bindings(\rho[\![a]\!]\ m\ \phi\ n\ M')\\
&\qquad \textbf{where } Bindings(X) = nil \quad \textbf{if } X = nil\\
&\qquad\qquad\qquad\qquad\qquad = \mathbf{B}[\![b]\!]\ m\ \rho\ (\theta \circ \phi)\ n'\ M\\
&\qquad\qquad\qquad\qquad\qquad\quad @Bindings(tail(X)) \quad \textbf{otherwise}\\
&\qquad\qquad\qquad\qquad\qquad \textbf{where } (\theta, n') = head(X)
\end{aligned}
$$

Figure 4.1: Denotational Semantics of Modular Prolog

rent value of η through each module using the function π to update η with the information contained in each module's *ExportList*.

With this definition of the export function, we are now in a position to formally define the *Home* function used in the operational semantics.

$h : Home = Pred \rightarrow ModName \rightarrow ModName$

$h = H\,[\![W^*]\!]\ \eta\ h_0$

$h_0 = \lambda P.\lambda M.M$

$H : Module^* \rightarrow Export \rightarrow Home \rightarrow Home$

$H\,[\![[]]\!]\eta h = h$

$H\,[\![WW^*]\!]\eta h = H\,[\![W^*]\!]\eta H\,[\![W]\!]\eta h$

$H\,[\![\ \texttt{:-begin\_module(}M, E, I\texttt{).}\ c^*\ \ \texttt{:-end\_module(}M\texttt{).}]\!]\eta h =$

$$\delta\,[\![I]\!]M\eta h$$

$\delta : ImportList \rightarrow ModName \rightarrow Export \rightarrow Home \rightarrow Home$

$\delta\,[\![[]]\!]M\eta h = h$

$\delta\,[\![[[P_1, M_1]|I]]\!]M\eta h = \delta\,[\![I]\!]M\eta h'$

$$\textbf{where } h'P'M' = M_1 \quad \textbf{if } P' = P_1 \textbf{ and } (P_1, M_1) \in \eta$$
$$= hP'M' \quad \textbf{otherwise}$$

The syntax "`begin_module`(M, E, I)" where I is a list of pairs of the form $[P, M']$ allows module M to import predicate P from another module M'. In the operational semantics, an imported predicate is added to the sequence of clauses produced by the *Database* function for each module. However, the names of predicates used in the body of the imported predicate must be bound in the context of the module in which the predicate is defined. The function $h : Home$ maps the functor of the head of a predicate and the name of the current module to the predicate's "home" module. Thus the body can be interpreted in the defining context.

Starting with an initial function h_0, H builds a new h by pushing the initial value through each module using δ to update h with the information in each module's *ImportList*. Internally, δ checks that each predicate that is imported into a module from another module is actually exported by that module. This is accomplished by checking that the imported predicate/module pair is in the *Export* relation.

4.5.3.3 Auxiliary Functions

$First : (Subst \times Num)^{+} \rightarrow Subst^{+}$

$First(x) = x$ **if** $x \in \{\perp, []\}$

$First\ ((\phi, n) :: L) = \phi :: First(L)$

The utility function, *First*, extracts the sequence of first components from a sequence of pairs.

$functor : atom \rightarrow Pred$

The *functor* function simply maps an atom (of the form $P(t^*)$ to its principal functor P. Most Prolog implementations distinguish predicates, not only by the name of their principal functor, but also by their arity. Strictly speaking, therefore, *functor* should map $P(t^*)$ to $P/arity$, where $arity = length(t^*)$.

$Uncut : \alpha\hat{} \rightarrow \alpha^+$

$Uncut(cut) = Uncut(nil) = nil$

$Uncut(\alpha_1 :: \alpha_2) = \alpha_1 :: Uncut(\alpha_2)$

Uncut is used to limit the scope of a cut to immediate subgoal. It replaces any terminal *cut* in its parameter with *nil*.
This definition can be extended to infinite sequences as with the definition of the @ function by taking the least upper bound of the ascending chain of finite prefixes of an infinite input parameter.

4.6 Informal Correspondence of Operational and Denotational Semantics

Formal equivalence proofs of operational and denotational semantics are typically extremely complicated, lengthy, and unreadable; consequently no such proof is attempted here. Instead we point out how each transition rule in the abstract machine is modeled in the denotational version.

The operational semantics processes the input database into a function, D : *Database* which takes a module name and returns the sequence of clauses in that module. The machine is initialized with the sequence of clauses for the **user** module in the slot in the state which holds the sequence of untried clauses. In the denotational model, the input database is "compiled" into the the function ρ which proves simple queries. ρ is defined as the fixpoint of the recursively defined $\mathbf{D}$. Each unwinding of the recursion corresponds to the application of one or more transitions in the operational model.

The role of the first transition rule (TR1) is to consume the next untried clause in the current module. Either the head of that clause unifies with the current goal in which case the body is added to the goal stack, and the remaining clauses saved as the next choice point, or unification fails and the remaining clauses in the module are tried immediately. An analogous role is played by the definition of σ in the functions $\mathbf{E}$ and $\mathbf{F}$. Each time the recursive function is "unwound", if the clause heads unify, the definition of σ tries to prove the body. If they do not unify, no bindings are produced ($\sigma = []$). In either case, the bindings σ' produced by the rest of the clauses in the module are appended to σ (corresponding to the choice point in the operational semantics). Note that the same unification function (MGU_{ss}) is used in both versions.

TR2, which backtracks to try alternatives if a goal is not further satisfiable (the sequence of clauses is empty), corresponds to the definition of $\mathbf{E}\,[\![[]]\!]$ and $\mathbf{F}\,[\![[]]\!]$ which produce no bindings, terminating the recursive definitions of $\mathbf{E}$ and $\mathbf{F}$.

The function of TR3, the cut operator, is handled explicitly in the definition of $\mathbf{B}$ which appends a *cut* to the sequence of bindings it produces. This *cut* causes the @ function to ignore any bindings produced by alternative bindings. This has the same effect as removing the choice points (which prevents the alternative bindings from ever being generated) in the operational semantics.

TR4, which fires when the current goal is *true*, corresponds to the definition of *Bindings* within $\mathbf{B}$. When the current goal is satisfied producing a substitution θ and renaming index n', the remaining goals in the conjunct are attempted using the bindings generated therein composed with θ to build the answer substitution.

TR5 which fires when the top level goal is satisfied is again modeled by the definition of *Bindings* within $\mathbf{B}$. Backtracking to pick up multiple solutions in the operational semantics is modeled in the definition of $\mathbf{B}\,[\![M' : a \wedge b]\!]m\rho\phi nM$ by iterating through the list of alternative substitutions produced by $\rho\,[\![a]\!]m\phi nM'$ (which "looks up" a in ρ)), resolving $\mathbf{B}\,[\![b]\!]m\rho(\theta \circ \phi)n'M$ for each (θ, n') produced.

Goals in modules other than the current module (TR6) are handled similarly in both semantics. Given the conjunct $(M : a \wedge b)$, the semantics "looks up" a in module M and then proves b in the current module (again see the last case in the definition of **B**).

Imports and exports are modeled quite differently, though equivalently in the two semantics. In the operational semantics, imported clauses are viewed as being added to the sequence of clauses for a module. This is handled by the preprocessor which creates the *Database* function. This preprocessing phase also creates the *home* function which returns the name of the module in which any imported predicate is defined. This function is then used in TR1 to bind names in the body of an imported predicate in the definition module. In the denotational semantics, the insertion of imported clauses into the client module is done by appending the bindings produced by the actual clauses in a module $(\mathbf{E}\,[\![c^*]\!])$ to those produced by using the imported clauses $(\mathbf{F}\,[\![I]\!])$. The definition of σ in **F** takes care to resolve names in the body of an imported clause in its home module.

In both semantics, name scoping is obtained by always maintaining a "current module". The definition of **D** (in the denotational case) and the reinitialization of the sequences of clauses from the current module in transition rules 1, 4, and 6 ensures that only clauses currently visible (or explicitly named via :) are used to resolve a goal.

CHAPTER 5

Combining Object-Oriented Prolog and Stream Processing

5.1 Introduction

Logical or Relational programming is based on set theory; all relations are viewed as sets of tuples. This concept has been generalized to ordered sets both for efficient processing and for presentation of semantic content to users. We call this concept of ordered sets *streams* [78].

Relational database query processing is potentially an important application of stream processing. Our experience with both tuple-at-a-time and whole query interfaces to a relational database from Prolog has led us to believe that a stream interface represents a useful middle ground. A stream interface uses iterative (tail recursive) processing and avoids backtracking through a database. The minor extensions to Prolog discussed in [77] are sufficient to provide efficient stream processing.

Narain has shown via the Log(F) language [73] the benefits of extending Prolog to handle stream processing. However, the combined logical-functional paradigm still suffers from several of the shortcomings of Prolog, namely the difficulty of writing, understanding, debugging, verifying, and maintaining large applications. These are precisely the strengths of the object-oriented paradigm. So, we can motivate the integration of object-oriented features with Log(F) from the point of view of adding a stronger program structuring methodology to Log(F).

We can also approach it from the point of view of adding stream processing to Object-Oriented Prolog. An important class of objects are stream structured; that is, the object is an ordered collection of other objects. For example, the employee object might have a sub-object called "salary history" as one of its attributes. The salary history can be viewed as a stream of event records ordered in the time domain, each one of which corresponds to a change in the employee's salary. We would like to be able to write methods in the employee class which access the salary history instance variable using an appropriate stream processing language like Log(F).

Chapters 2, 3, and 4 show how the complementary strengths of logic programming and object-oriented programming can be incorporated in a single language. Since both Log(F) and O-OP are implemented as modest additions to Prolog, we have the unique opportunity to harness the advantages of each in one environment. This chapter now examines integrating elements of the functional paradigm. We first present as background a review of the Log(F) language.

5.2 Background: Functional Programming in Prolog

Functional programming is a form of declarative programming based on equational logic, the substitution of equals for equals. While deduction performed by the unconstrained substitution of equals for equals can be very inefficient (due to the infinite branches introduced by the symmetry property of the equality relation), directed substitution or rewrite rule systems with appropriate restrictions offer the potential for high performance [37]. A rewrite rule system simulates a functional programming system in which the "reduces to" operator is interpreted as equality.

5.2.1 Log(F)

Stream processing in Tangram is accomplished using Log(F), an integration of Prolog with a functional language called F*, developed by Sanjai Narain at UCLA [73,71,72]. Log(F) rests on subsuming within Prolog the concepts of *lazy evaluation* and *rewrite rules.*

A statement in F*, called a *reduction rule* has the form

$$LHS \Rightarrow RHS$$

where both LHS and RHS are terms as previously defined but with some restrictions[1]. The statement is read "LHS rewrites to RHS." A term T is said to

[1]Rewrite rules in F* must satisfy the following restrictions [71]:

1. LHS is not a variable.
2. LHS is not of the form $c(t_1, \ldots, t_n)$ where c is an n-ary constructor symbol, $n \geq 0$. Constructor symbols are special functors (constants) which are not reducible.
3. If LHS is $f(t_1, \ldots tn)$, $n \geq 0$, each t_i is a variable or a term of the form $c(X_1, \ldots, X_m)$ where c is an m-ary constructor symbol, $m \geq 0$, and each X_i is a variable.
4. There is at most one occurrence of any variable in LHS.
5. All variables of RHS appear in LHS.

reduce to T' if there exists a subterm U in T, a rule $A \Rightarrow B$, and a substitution θ such that $U = A\theta$ and T' is the result of replacing U in T by $B\theta$.

Consider a program to append two streams. A program in F* is a set of reduction rules. A stream is represented syntactically in Log(F) as a Prolog list.

```
append([],W) => W.
append([U|V],W) => [U|append(V,W)].
```

Append of the empty stream with another stream rewrites to that other stream. Append of two arbitrary streams rewrites to a stream whose head is the head of the first stream and whose tail is the result of appending the tail of the first stream and the second stream.

An important property of F* is its capacity for *lazy evaluation.* Notice that in the application of the rewrite rules for append above, a single reduction step produces only the head of the output stream `U`, plus a *continuation* `append(V,W)`, which tells how to compute additional elements of the output stream. The continuation can be further reduced when additional elements of the output stream are required. Thus the execution is demand-driven; computation is only performed when its results are needed. This gives the language the ability to define computations on infinite streams which would never terminate if the stream was materialized *eagerly*, but may succeed in finite time if the stream is materialized lazily. Similarly, computations on large streams such as database relations are

These rules are sufficient for proving soundness and completeness for F* [73]. These restrictions also permit high performance as F* programs can use pattern-matching rather than unification. Determinacy is easily detected syntactically, avoiding backtracking and hence expensive choice points.

possible without requiring large buffers for intermediate results. Programmers may also specify that some computations be performed eagerly.

Log(F) consists of Prolog plus the F* compiler. In Log(F), F* rules are compiled into Prolog clauses. For example, the rewrite rules for **append** above are compiled into

```
reduce(append(A,B), C) :- reduce(A,[]),
                          reduce(B,C).
reduce(append(A,B), C) :- reduce(A,[D|E]),
                          reduce([D|append(E,B)], C).

reduce([],[]).
reduce([H|T], [H|T]).
```

which is semantically equivalent to and almost as concise as the Prolog definition of append given in the previous section. The last two reduce rules simply declare that the empty list, [], and the cons function, |, are *constructor symbols* and cannot be further reduced.

5.2.2 Transducers

The basic unit of computation in the Tangram Stream Processing system (TSP) is the *transducer*. Informally, a transducer is a mapping from one or more input streams to one or more output streams. A simple transducer maps each element on its input stream to a corresponding output value. For example, the transducer

```
incr([]) => [].
incr([H|T]) => [H+1|incr(T)].
```

increments each integer on its input stream[2].

Another kind of simple transducer is a filter which rewrites some of its input to its output. The following transducer, for example, rewrites only those values which are greater than its second argument.

```
gt([],X) => [].
gt([H|T],X) => if (H > X, [H|gt(T,X)], gt(T,X)).
```

`if` is a transducer which rewrites to its second argument if its first argument reduces to true and otherwise rewrites to its third argument. > is an eager arithmetic operator which reduces to true or false.

Some transducers are *generators*. That is, they produce output streams but do not consume an input stream. `intsfrom` is an example of a generator which produces an infinite stream of integers counting up from the input integer.

```
intsfrom(N) => [N|intsfrom(N+1)].
```

Database relations are an important form of generator transducer which will be examined shortly.

Accumulator transducers compute aggregate functions on streams. The `sum` transducer produces the sum of the integers on its input stream.

```
sum(S) => sum(S,0).
sum([],N) => [N].
sum([H|T],N) => sum(T,H+N).
```

Transducers can also be hybrids of these types. For example, the following transducer, which outputs a running total of the values read so far, is both an accumulator and a mapping transducer.

[2]Note that +, like other arithmetic functions, behaves eagerly.

```
runningSum(S) => runningSum2(S,0).
runningSum2([],X) => [X].
runningSum2([H|T], Sum) => [Sum+H|runningSum2(T,Sum+H)].
```

Programs or queries are written by composing transducers. For example, a database relation may be retrieved using the transducer `tuples(DatabaseName, RelationName)`. Say the database contains a relation "book" whose first attribute is the name of the author. The transducer `tuples(dbase,book)` returns a lazy stream of tuples of the form `book(Author,Title,Publisher,Date)`. The generating transducer, `books`, defined by

```
books => tuples(dbase, book).
```

produces all of the tuples from the book relation. We can print all of the tuples for books by "smith" by composing several transducers:

```
print_list(select(books,book(smith,_,_,_),true)).
```

Here, `print_list` is the stream equivalent of the Prolog `findall` which reduces its argument completely; `select(Stream, Template, Condition)` lazily selects from its input `Stream` all items unifying with `Template`, and satisfying `Condition`. The condition is trivially satisfied by `true` in this case.

Log(F) is an appropriate language for stream processing and thus for database query processing. It combines the declarative nature and power of logic programming with rewrite rules yielding a logic language with a functional flavor. Stream operators are conveniently expressed as recursive functional programs. Further, streams and transducers are a reasonable paradigm for expressing parallel computation; the design and implementation of a distributed Log(F) is the subject of on-going research [53,54].

5.3 Combining Paradigms

We have pursued two approaches to integrating object-oriented concepts with Log(F). In our first approach, we view streams as statically typed objects and transducers as overloaded operators. We will show how a compiler can preprocess an object-oriented Log(F) program into a standard Log(F) program. This approach corresponds to adding object-oriented notions to Log(F). Our second approach takes advantage of the fact that both O-OP and Log(F) reduce to Prolog. It permits the free inter-mixing of Log(F) reduction rules with message sending and ordinary Prolog in method definitions.

5.3.1 Operator Overloading for Transducers

We have seen that it is often convenient to view a relation as a stream. Queries on a database can then be constructed as networks of transducers which are fed streams from the database. Operator overloading can allow us to implement layers of abstraction on streams and hence an extensible database as well as a degree of modularity in the query language.

Relational databases typically have a two level hierarchical structure; an underlying storage layer with a large amount of uninterpreted data, and a smaller layer consisting of schema or meta-data information which tells how the stored data is to be interpreted in the relational model [44]. From the point of view of higher levels of the system, however, the relational database is an underlying storage facility with "uninterpreted data". The knowledge to interpet that data is built into the application programs. This state of affairs is equivalent to giving a query language access only to the access methods and requiring knowledge of the schema to be built into the queries. The same data cannot be used by multiple programs without repeating its interpretation information. Further, the small, fixed set of types typically supported by relational databases cannot be extended as applications require.

This leads us to the desire to place higher level schemas on top of the database. We can have schemas for temporal databases, graphics databases, statistical databases. The main purpose of the higher level schema is to allow us to view the stream as an object with a given semantics without regard to its underlying implementation. Alternatively, it can allow us to specify different semantics for streams with the same information. Both of these views amounts to operator overloading. That is, given the name of an operation (transducer) to be applied to a stream, the correct implementation of that transducer must be automatically invoked based on the semantics (type) of the stream specified by the schema. We will consider for example the definition of a temporal database.

5.3.2 Temporal Data

Much of the data generated in the Tangram modeling system is ordered in the time domain. Unlike traditional business oriented data which is primarily concerned with the *current value*, in scientific, statistical, and simulation databases, we are interested in the history of a value over time. In this discussion we will follow the temporal data model of [90] and show how time series semantics can be captured using an extended database schema and Log(F) with operator overloading.

A temporal data value is a triple $< s, t, a >$ where s is a surrogate (or key) for an object, t is a time, and a is the value of an attribute. All of the temporal data values for a given object are totally ordered in time. This ordered set is called a *time sequence* or TS. The collection of TSs for all objects of the same class is called a *time sequence collection* or TSC. For example, the salary history for an individual employee is a TS. The salary history for all employees is a TSC.

It is easy to see how a TS or TSC can be stored in a relational database. We can use relation with three attributes, eg. `<Employee#, Date, Salary>`. This could be viewed in Log(F) retaining this first-normal-form representation or factoring out the surrogate to obtain a stream of `<Date, Salary>` pairs for each Employee.

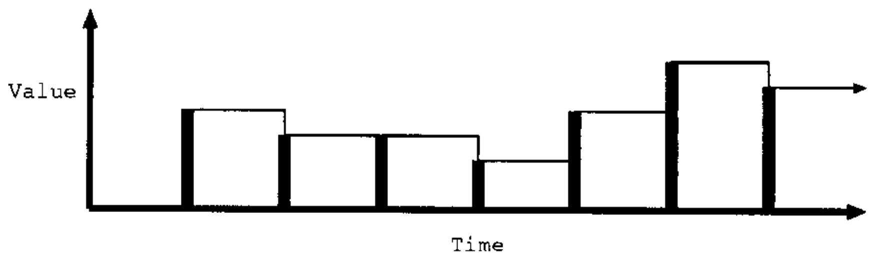


Figure 5.1: Account balance; step-wise constant

5.3.2.1 Temporal Types

We can have many different semantics for a TS. A *step-wise constant* TS has the semantics that each time a value appears in the sequence, that value is in effect until the time of the next value in the sequence (see figure 5.1). An example of this type of semantics is the balance of a bank account. Each time the balance changes, a temporal data value appears in the sequence. If we were to query the account balance TS about the balance at a time t' which occurs between times t_i and t_j, the nearest times on either side of t' for which we have temporal tuples, then the value that should be returned for t' is the value associated with t_i. That is, interpolation is done by using the nearest preceding value.

In a *discrete* TS, the sequence has a value only at those times for which there is a temporal data value (see figure 5.2). A time series which gives for each day, the total amount withdrawn from the account that day is an example of a discrete TS. The temporal value applies only to the time specified; there is no

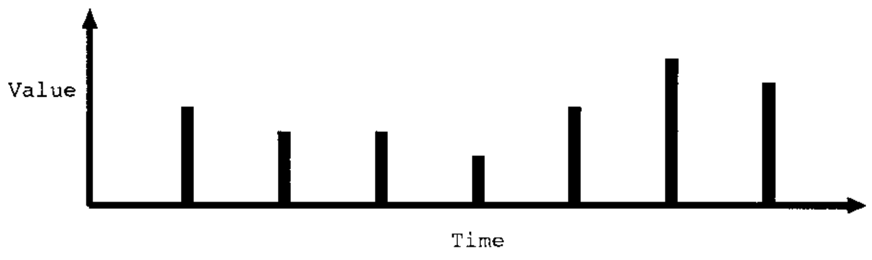


Figure 5.2: Daily account withdrawal summary; discrete

interpolation as the value is undefined between the times for which values are supplied. That is, we can ask, How much money was withdrawn on the 25th? but it is meaningless to ask, How much was withdrawn at 2:47PM on the 25th? given these semantics.

A *continuous* TS has sample values of a continuous function taken at regular intervals (see figure 5.3. A sequence of values can be interpreted as samplings from a continuous wave form. Interpolations between successive values can be performed if needed. An example might be digitally encoded music on a compact disk. Figures 5.1, 5.2, and 5.3 depict three different interpretations for an identical sequence of temporal data values. There may be many more reasonable semantics.

5.3.2.2 Operations on TSs

An operator on a TS is a transducer which takes as input one or more TSs and produces another TS, some other type of stream, or a value. For example,

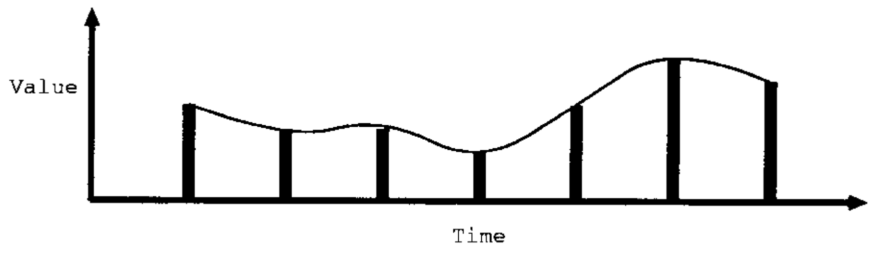


Figure 5.3: Digitized music; continuous

consider a "lookup" operator which takes as input a TS, T, and a stream of dates, D, and produces as output a discrete TS, T', containing the temporal data values found in T for each date in D. As we have seen, the "lookup" operator must be different depending on the type of T. The operator behaves the same for each type whenever the date in D corresponds exactly to a date in T. However, if the date in D falls between two entries in T, interpolation occurs for constant time sequences, the preceding value is used for stepwise cases, and no value is defined when T is discrete.

The conventional way of dealing with this situation is to include a "switch" on the type of the argument in the user's code which calls the correct flavor of lookup. However, this requires that users know about all of the sub-types of TS. Consider the addition of another flavor of TS with slightly different semantics and hence its own implementation of lookup. All existing application programs (clients) which manipulate time sequences must be tracked down, the additional case added to the switch, and the application recompiled.

The object-oriented methodology gives us an elegant way to handle this situation. We can view the lookup operator as a method for time sequences. Each of the sub-types (discrete, stepwise, and continuous) of TS have an implementation of lookup defined. The name, *lookup* is an overloaded operator. Users can use the lookup operator on any TS without even knowing which sub-type of TS to which it is actually being applied and hence without knowing which implementation of the function to use. The system automatically binds the name of the method in the context of the object to which the operator is applied. If a new sub-type of TS is later added, its supplier simply adds it to the type hierarchy and its methods are dynamically bound just like any other; no change to client application programs is required.

5.3.3 Typed Log(F) Compiler

We have prototyped a preprocesser for Log(F) which handles operator overloading statically[3]. Given a Log(F) program, the preprocessor converts it to a graph representation in which each transducer is a node and each stream is an arc. The input arcs representing the arguments to the inner most transducers (leaf nodes) are labeled with the type of the stream obtained from the temporal database schema. The overloaded operator name for a leaf node is then replaced with the low level name of the correct implementation of the transducer inherited

[3]Implementation by Cliff Leung and the author.

by the types on its input.[4] The output arcs are then labeled with their types obtained from the schema for the transducer. This is repeated for each node and arc until the whole graph has been labeled. The compiler then uses the input and output types of each node to look up the correct low-level implementation of each transducer. The overloaded transducer names in the network are replaced with the low-level names. The graph is then converted back into Log(F) and compiled into Prolog by the standard Log(F) compiler.

The compiler is operational and has been demonstrated using time sequences and temporal schemas stored in an Ingres database and queried using Log(F).

5.3.4 Example

We will demonstrate how our system would handle an example query from Segev and Shoshani paper on temporal data modeling [90]. Complete code for the compiler and the schemas for this example are in Appendix 1. The example utilizes the following two time sequences:

bookSales (type = discrete) - contains the daily sales of books. A tuple in the

`bookSales` relation has the form:

[4]For unary operators (operators with only one typed stream input and possibly several untyped parameters), the implementation is simply inherited by the type of the input. For higher arity operations (e.g. stream multiplication), the implementation of the operation is a function of the tuple of types. We can view higher arity operations as operators on composite objects and treat them just like unary operators. In general, for each arity n, there is a function of arity $n+1$ which maps an n-tuple containing the operator name and the types of each of the typed input arguments to a low level implementation name. The role of the compiler is to apply this function to each transducer. The function may be represented as an $n+1$ array as in the case of a database schema or distributed among the object definitions using inheritance to avoid redundancy.

```
(book\#,date,qty_sold)
```

The `book#` is the *surrogate* or object identifier field and may be factored out. The meaning of a tuple `(34,3,12)` is that on date 3, 34 copies of book number 12 were sold. If for a given book number, we project out just the date and quantity sold, we get a time sequence (time/value pair). This TS has discrete TS semantics (see figure 5.2 as it represents a sequence of data values for which there is no value defined in between.

bookPrice (type = stepwise constant) - contains the daily prices of books. A tuple in the `bookPrice` relation is of the form

```
tuple(book#,date,price)
```

A tuple `(34,3,12.0)` means the price of book 34 was changed to 12 dollars on date 3. This is a stepwise semantics (corresponding to figure 5.1 as each price value is in effect until it is superceded by the next change.

In this example query, we want to obtain a time sequence which tells, for each date, the volume of sales in dollars for book number 34. We need to "multiply" the corresponding entries in the two streams. In typed Log(F) the query is:

```
execute(
    multiply(projectList([2,3],
             select(bookSales,tuple(A,B,C), A=34)),
             projectList([2,3],
             select(bookPrice,tuple(D,E,F), D=34)))).
```

The `execute` predicate invokes the typed Log(F) compiler on the query given in its argument.

Both `bookSales` and `bookPrice` are generating transducers. In our system, they may be stored as a list of tuples in a file, or be extracted from an Ingres database. `projectList` and `select` are database function transducers provided in libraries by the stream processing system [78]. The `select` here unifies `tuple(A,B,C)` with each tuple in the input stream, outputting the tuple if the criteria A=34 is met. The `projectList` takes a list of attribute numbers and a stream and outputs a stream of tuples with only the listed attributes.

The `multiply` transducer is an overloaded operator which takes as input two time sequences and outputs another; the specific type of the output stream as well as the implementation of the stream multiply function actually run depend on the types of the input time sequences. Thus we must have a schema from which the compiler can look up both the types of the input streams, and the behavior of the multiply transducer. Referring to figure 5.4, the `transducerName/5` predicate declares that the high level overloaded name (first argument) can be substituted for by the low-level name (given in the second argument) with the number of input and output arguments found in the third and forth arguments and the definition of the low-level predicate is found in the file named in the fifth argument. Thus the first three clauses declare that the compiler may replace the high-level name `multiply` with `multdd`, `multds`, or `multss` found in the file `library/multiply` depending on the types of the input arguments.

```
%
% transducerName(HighName/Arity, LowName/Arity, NoInputArg,
% NoOutputArg, FileLocation)
%
transducerName(multiply/2,multdd/2,2,1,'library/multiply').
transducerName(multiply/2,multds/2,2,1,'library/multiply').
transducerName(multiply/2,multss/2,2,1,'library/multiply').

transducerArg(multiply/2,[pipe,pipe]).

% transducerVersion(LowName/Arity,I_O,ArgNo,Type).
%
transducerVersion(multdd/2,i,1,dts).
transducerVersion(multdd/2,i,2,dts).
transducerVersion(multdd/2,o,1,dts).

transducerVersion(multds/2,i,1,dts).
transducerVersion(multds/2,i,2,sts).
transducerVersion(multds/2,o,1,dts).

transducerVersion(multss/2,i,1,sts).
transducerVersion(multss/2,i,2,sts).
transducerVersion(multss/2,o,1,sts).
```

Figure 5.4: Schema for `multiply` transducer

The transducerVersion clauses declare the types of the arguments (input an output for each of the low-level transducer version. They will be used by the inference engine in the compiler to pick an appropriate low level version to replace high-level, overloaded transducers. This defines the schema of the multiply transducer.

Next, the schema for the input data streams must be defined.

```
bookSales => file_terms('data/bookSales').
bookPrice => file_terms('data/bookPrice').
```

This defines the streams named bookSales and bookPrice to be the streams of terms from the files 'bookSales' and 'bookPrice' respectively in the 'data' directory.

```
%
% timesequence(StreamName,RecordStructure,Type,
%             LifeSpan,Regularity,TimeGranularity,Ordering).
%
timeSequence(bookSales/0,
             tuple(bno:integer, time:time, value:integer),
             dts, (0,20), reg, 1, asc).
timeSequence(bookPrice/0,
             tuple(bno:integer, time:time, value:integer),
             sts, (0,20), irreg, 1, asc).
```

The above declares that each element on the bookSales stream has the functor 'tuple' and three attributes of type integer, time, and integer respectively. The system incorporates an 'isa' type hierarchy which further defines these types (see Appendix). The third argument gives the type of the TS (dts = discrete, sts = stepwise). Next is the lifespan (range of dates that the TS covers. The following field contains either `reg` (regular) or `irreg` (irregular). A regular TS contains a value for each time point within the range (e.g. a number of books sold for each day). An irregular TS contains temporal data values for only a subset of points within the range (e.g. a price change entry only when a change occurs). The 'TimeGranularity' field declares the granularity of time values. The time can be either ordinal (1,2,3,...) or calendar (year:month:day:...). In our prototype, only ordinal times are provided and the granularity of both TSs are 1 unit. The final 'Ordering' attribute tells whether the TS is in ascending (asc) or descending (dsc) order.

The compiler uses the schema to replace the overloaded transducer name `multiply` with the low-level name of the correct implementation of multiply. In this case, `multds` is used to multiply a discrete TS with a stepwise TS. The appendix shows a representation the type-labeled graph of the query. The compiler translates the graph back into Log(F) yielding

```
print_list(multds(projectList([2,3],select(bookSales,
    tuple(_6949,_6969,_6989),_6 949 = 34)),
    projectList([2,3],
    select(bookPrice,tuple(_7145,_7165,_7185),_7145 = 34))))
```

which is called and executed.

5.4 Adding Log(F) to Object-Oriented Prolog

Our second approach to integrating the functional, object-oriented, and logic programming paradigms is to encorporate Log(F) into O-OP. In our hybrid object-oriented paradigm, at some level, the instance variables of an object are represented as Prolog terms. If we wish to model objects with stream-valued instance variables (for example the account balance time sequence), we can represent them as a Prolog list of terms. In fact, the instance variable can actually be a generating transducer which retrieves tuples from an Ingres relation. Log(F) can then be used to write the methods which query the stream-valued attribute.

This capability has been added to O-OP. All methods which contain re-write rules are preprocessed by the Log(F) compiler before being asserted into their class module. Within Log(F) rules, messages may be sent to other objects allowing a free and natural intermixing of the three paradigms. Because both the object-oriented and functional paradigms are given their meaning here in terms of equivalent Prolog, there is no impedance mismatch as the styles are mingled.

5.4.1 Streams of Objects

We have seen the power of viewing streams as objects and of allowing stream valued attributes for objects. The full power of combining streams and object-oriented programming is obtained when we permit streams to be sequences of typed objects.

For example, imagine a graphics display system in which a complex diagram is composed of many sub-objects. The object corresponding to the whole diagram has a stream attribute with a method for displaying the diagram represented by the stream. This transducer rewrites each object on the stream to a sequence of drawing commands to a graphics processor. There is operator overloading occurring here at multiple levels. First, the display transducer on the stream is overloaded, selecting the correct version for that type of stream. Then, within display, there is a re-write rule for each element. This will involve invoking methods on each of the objects on the stream which select the correct implementation for their type. These objects may be complex objects themselves, invoking methods on their sub-objects. This example illustrates the power and economy of combining stream-based and object-oriented paradigms.

A stream of objects may be implemented as a stream of Oids; this is precisely what a collection class in O-OP is (see section 3.5.13). When the objects on the stream are themselves streams, we can represent this as streams of generating transducers. These transducers may generate the elements of a collection eagerly by sending a `list(L)` message. Alternatively, the members of a collection may be generated lazily via the `element(E)` message which backtracks through the sequence of elements of the collection.

This chapter proposes taking advantage of the fact that both object-oriented and functional programming have convenient interpretations within logic programming. By interpreting both in Prolog, we obtain a seamless combination of the three major paradigms in which each can be used to its best advantage, without suffering the usual problems with mixed paradigms. The next chapter exposes a new form of name binding: *semantic binding.*

CHAPTER 6

Semantic Binding and Dynamic Classification

6.1 Introduction

In the conventional object-oriented paradigm, the class is an invariant of an object. That is, an object is created in a particular class and all methods which transform that object are guaranteed to transform valid instances of that class into other valid instances. It is also central to the paradigm that an object's class determines what code is run when the object receives a particular message. In fact, the primary purpose of classes is to group together objects which can share identical implementations of methods.

While there are many benefits inherent in the object-oriented approach to software design and construction, we will show that dynamic or evolving software environments pose problems which the object-oriented paradigm (and in fact any other existing paradigm) fails to address. That this short-coming has been not been discussed previously is particularly surprising given that the object-oriented paradigm is strongly touted [63,27] as the answer to the demands of long term maintainability and evolvability of large software systems.

6.1.1 The Problem

Above, we stated two characteristics of the object-oriented programming paradigm: 1) class membership alone determines the binding of an object to the method code to run in response to a message, and 2) objects do not change classes. However, as the next two sections demonstrate, under a very reasonable and expected set of circumstances, 1 and 2 cannot both be true.

6.1.1.1 Multiple Alternative Method Implementations

When an object receives a message, it is requested to perform a function, identified by the message's name, on itself. For example, a matrix may be sent the message "invert". However, there may be more than one implementation (solver) for the same logical function, e.g. one invert procedure for small matrices and another for large, sparse matrices. The version for sparse matrices may not work for smaller dense matrices or it may exhibit much worse performance than the more appropriate solver. The choice of which version of the method to employ depends on the state of the object.

In traditional object-oriented programming, given an object's class, a function name, and an inheritance lattice, the system performs the binding to an implementation of a method. To accommodate more than one implementation for a method within the object-oriented paradigm, the class (matrices in the case of this example) may be further specialized into subclasses with each subclass providing a different implementation of the function. However, this approach, while satisfying 1 and 2, is unsatisfactory in a multiple implementations situation because it is generally not knowable when the object is created which version of the method is appropriate. If more than one instance of the method is applicable, the best choice is often a non-trivial function of the object's dynamic state, not simply its class. Furthermore, there are typically many different messages to which an object can respond; each corresponding method may have multiple implementations. This situation necessitates further subdividing the classes into as many as one for each element of the set of cross-products of the sets of alternative solvers for each method. Thus adding a new solver for one method requires the creation of many additional subclasses, one for each of the other combinations of solvers, and the static reclassification of all existing instances of model objects into one of the subclasses. This approach is clearly untenable.

An alternative to specializing the class with artificial subclasses for each different implementation of the method is to combine all the versions of the solution into a single method with "switch" logic prepended to choose the most appropriate branch. This approach is superior in that the switching logic takes into account the dynamic state of the object instance in making the choice of a solver. However, it sacrifices the flexibility of adding new implementations of the function without changing existing methods.

6.1.1.2 Extensible Tool Kits

A large scale programming environment must be able to maintain an extensible tool kit of solvers. It must manage solvers as objects themselves providing a convenient facility to add new solution techniques to a system, and cause them to be used where appropriate. The choice of the most appropriate solver is generally a function of the queried object's structure or instance variables and values specified in the query. The knowledge about how to select an implementation for a method must be sufficiently modular that a new solver can be added to the environment without modifying existing solvers. Further, the new solver should be used to answer queries on existing objects where appropriate without modifying those objects. Given an object and the name of a function, the system must dynamically bind to an implementation of the function.

If, as in 2, the class is an invarient of an object, class membership alone cannot determine method bindings as state changes may alter which method version is appropriate. Thus 1 cannot hold.

If as in 1, the class determines the mapping to implementations, objects must be permitted to change class when their state changes in order to permit them to bind to the most appropriate method version given their new state. Thus 2 cannot hold.

6.2 Two Solution Approaches

We present two approaches within an object-oriented framework to dealing with the problem of connecting a version of a method to an object in response to a given message. The first, *semantic binding*, corresponds to relaxing 1 while the second, *dynamic classification*, is obtained by relaxing 2. The two approaches will be seen to achieve the same effect.

6.2.1 Semantic Binding

In general, a program must run to choose the correct binding of a function name to an implementation. We term this process *semantic binding.* Connecting a solver to a model for a given query involves running a domain specific expert system which queries the model and solvers and chooses the most appropriate binding. We call this binding program an expert system firstly because the choice of a solution technique for a particular model is an expert decision; a naive user, for example, need not be concerned with whether or not his queueing network is product-form. Secondly, the expert knowledge used to select a solver is most often best expressed in the form of rules. Such declarative knowledge representation is highly extensible; adding a new solver involves adding rules for that solver, not modifying rules. This is the programming style of an expert system.

The challenge is to devise a way to organize the knowledge required to perform the binding. The organization must be sufficiently modular to allow easy addition of new solution methods. As the binding process depends on characteristics of the candidate solvers and specific attributes of the target model, all model objects and solver objects must provide a standardized interface to query this type of information. This standardization is achieved by organizing classes of models into *domains*.

A domain is an encapsulated portion of of the class hierarchy and serves as the granularity for specification of expert knowledge about semantic binding. The domain is named by the class which forms the root of the subtree. This is the most general class of models within the domain. For each domain there is a set of candidate solvers for models. All models and solvers within a particular domain must satisfy the standard interface so that the semantic binding logic can query their characteristics. See Sections 7.2.2 and 7.4.2 for a description of how semantic binding and domains are implemented in the Tangram Modeling Environment.

6.2.2 Dynamic Classification

Another way to view the process of binding a solver to an object is to reclassify, automatically and dynamically, an object when it receives a message. We first introduce the concept of dynamic classification, and then show how it is used to implement semantic binding.

In the standard object-oriented paradigm, it is assumed that an object's class is fixed when it is created. The creation of an object is generally accomplished by sending a message to the most specific class of which the object is a member instructing that class to generate an instance of itself. Any methods defined by more general classes are applied via inheritance. Any method which changes the state of an object is guaranteed to transform an object in a valid state for a particular class into another valid state for that class. Methods cannot cause objects to change class.

This is, in general, too restrictive. For example, suppose we have an financial investment modeling system. Within the class Investments, we have a subclass Bonds. Bonds are further specialized into High Grade Bonds and Junk Bonds based on their rating which is maintained as an instance variable. Some methods for Bonds may behave differently for the two subclasses; e.g. they may be listed in separate parts of a balance sheet. It is conceivable that the rating of a particular instance of an investment grade bond be downgraded say when the company files for bankruptcy protection. This necessitates the moving of the object instance from one class to a sibling class. This can only be accomplished in current systems by deleting the instance and recreating it in the other class, resulting in an interruption in the object's identity. Such an interruption is intolerable as many other objects may contain pointers to the original instance which will be left dangling.

The above is an example of movement of an object instance from one sibling class to another. Movement up and down in the hierarchy is also possible. Consider the class Polygon with subclass Rhombus which in turn has a subclass

Square. If a shape is represented by instance variables containing the coordinates of all corners, it is clearly possible for a message to a Rhombus to move two coordinates such that the object becomes a Square. Similarly a message to a Rhombus could move a single coordinate rendering it no longer a Rhombus, but still a Polygon. Thus an instance can become more or less generally classified.

How is this related to binding solvers, queries and objects? The above has argued that it is useful to permit changes of object's states which affect the binding to methods. This is only possible if objects can be re-classified. This approach can also be used to bind to (select) an appropriate solver based on the values of instance variables.

In our initial statement of the problem, we rejected the idea of specializing classes into subclasses, one for each solution method. This was based on the grounds that the choice of the correct solver is a function of the object's dynamic state and could not be made a priori. Further, the number of such subclasses would become unmanageable as we would require $\prod_{i=1}^{n} N_i$ subclasses where n is the number of methods and N_i is the number of alternative implementations of the i^{th} method. However, if the reclassification is done dynamically and automatically, there is no need ever to actually enumerate all of the possible classes.

6.2.2.1 Classification Via Constraints

Our proposal for dynamic reclassification is as follows: An object is created within a domain as with semantic binding. The domain fixes the most general type (class) that the model object can ever take on. Each sub-class within the domain must supply a set of membership constraints. The constraints are arranged hierarchically such that each sub-class need only define the additional requirements that an object must meet for membership given that it is already a member of its parent class.

When an object within the domain receives a message, it is "pushed" down the hierarchy to the most specific class for which it satisfies the membership constraints. The constraints can name the selector and arguments of the message received by the target object such that the class selection is a function, not only of the object's state, but of the message.

The binding to solvers in this approach is "hard-wired" in the class (as with conventional object-oriented name binding). The selection of the most specific class for an object with respect to a given message determines the binding to a solution implementation. Optimizations are clearly possible (e.g. cache previous bindings which are only invalidated by changes in the object's state).

The two approaches, semantic binding and dynamic classification offer equivalent functionality. They differ only in the style in which the expert knowledge which determines the name binding is expressed. Both will incur significant but as yet unknown run-time overhead for dynamic binding but may benefit from clever implementation.

6.3 Related Work

We are aware of only a single piece of related work, the "law-based approach to object-oriented programming" by Minsky and Rozenshtein [65] utilized in the Darwin programming environment [82]. Their premise is that "inheritance, hierarchical or otherwise, is simply too narrow a concept to serve as a unifying principle for object-oriented programming." They propose the concept of a *law governed system* in which the exchange of messages is subject to the *law* of the system.

A message of the form $< s, m, t >$ (sender object, message text, target object) is presented to the system via a send goal: $send(s, m, t)$. This goal is evaluated by the Prolog interpreter with respect to the set of rules which comprise the law. There are three possible outcomes:

1. The message is *delivered* to the target object via the system primitive $deliver(m, t)$
2. The message may be modified and/or rerouted. That is $send(s, m, t)$ may result in $deliver(m', t')$.
3. The message may be rejected.

Thus the law acts as a filter on the exchange of messages (see Figure 6.1). The figure illustrates a message (1) from object a to object b which is delivered unmodified (2) by the law filter. Another message (3) sent by a is blocked by

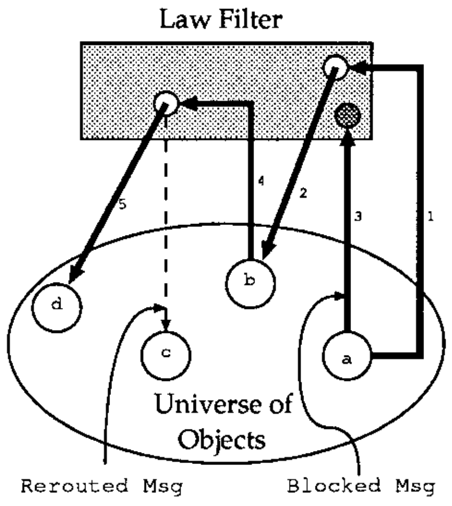


Figure 6.1: A Law Governed System

the filter. The message (4) sent from b addressed to c is intercepted by the law and rerouted (5) to object d.

Users may specify their own message exchange discipline by asserting clauses for the *send* predicate. Using this very primative base, the authors show how many of the features found in object-oriented languages including multiple inheritance, differential inheritance, active values, etc., may be built using laws.

While they do not address the problem of semantic binding, this work is related in that it offers a potential framework for a solution. Minsky and Rozenshtein are the first to raise the possibility of a user supplied program imposed between message sending and receiving which can alter the default binding procedure. By themselves, however, completely unconstrained user defined laws offer too little structure to use as the basis for representing the knowledge required to bind an object to a solver for a particular query.

CHAPTER 7

Application: The Tangram Object-Oriented Modeling Environment

7.1 Introduction

Will an increase in the discount rate drive this country into a recession? Is the weather going to be good next year for growing avocados? Is the satellite likely to remain operational for the entire mission time? Getting answers to this type of question can be of critical importance. Unfortunately, it is generally infeasible to conduct an experiment to answer these questions without committing to the very course of action whose outcome is in doubt. Instead, we try to capture the behavior of these types of systems in a *model.* Querying this model gives predictions of the future. These predictions can be anywhere from exact to wildly inaccurate depending on how well the model represents the essential behavior of the real world system. Experience with the model as to how well it explains previous behavior can give confidence in its predictions of the future.

While there have been tremendous advances in the mathematical techniques, the practice of modeling has advanced more slowly. Modeling typically requires specialized skills; knowledge of both the problem domain and a solution package. It is very labor intensive. The process involves an expert abstracting the application domain, selecting a solution method or solver, translating the abstraction into the input format of the solver, and then interpreting the numerical results of the solution. The solution tools typically have a very primitive user interface requiring input in a rigid format. An expert is often only highly proficient in one tool; "If you only have a hammer, everything looks like a nail." In addition, there is a shortage of experts and using the ones there are is costly.

The answer to these problems is a whole new generation of modeling tools. Driven by the personal computer revolution and by wider acceptance of primitive modeling tools such as spreadsheets and databases, the opportunity and the need for an advanced environment which can harness powerful modeling tools for non experts has never been greater. This chapter proposes an architecture for such an advanced modeling environment.

We have implemented Tangram [70,76], a prototype of such a system, using Object-Oriented Prolog. This modeling environment provides a clear example of an application for which the effective combination of both logic programming and object-oriented organization is essential for success.

7.1.1 Mathematical Modeling

Mathematical modeling spans a large variety of techniques; queueing theory, mathematical programming, Markov chains, semi-Markov models, etc. In the same sense that this body of knowledge is organized into areas and subareas, one of our goals in Tangram is to organize modeling knowledge into modules that we call *domains.* A domain encompasses a class of models, a set of solution methods (solvers) and an interpreter for queries. The idea is that models are created "in a domain" and any such model must be a member of the class of model associated with that domain. The choice of what class of models a domain encompasses is a design decision. Basically, one would like to (a) incorporate a useful, general class of models and (b) subsume within the abstraction provided by the domain a significant amount of detail. As a simple example, suppose we construct a domain that deals with finite Markov chains. This domain may have a number of different solution techniques available; e.g. some may be appropriate for nearly decomposable chains, others for sparse chains, etc. The idea of the domain abstraction is that a client should not have to be concerned about details such as how to choose the best solution methods, but rather just ask for a solution and have the system choose the appropriate solver.

When the domain supports what is a classical mathematical abstraction (e.g. Markov chain, linear programming, queueing network, etc.) then it is perhaps more appropriate to call this a *modeling domain.* A specialization-generalization hierarchy of these domains occurs quite naturally. Further, as we shall demonstrate later in the paper, it is also convenient to form domains for particular applications, which we call *application domains.*

In addition to the central concept of domains, the following constitute the goals of the Tangram system which we believe are essential.

Extensibility The system must be able to cope with a wide variety of application areas. In our view, an application domain will be customized by an expert for a non-expert to use. New applications may be created by specializing existing ones. New solution techniques must be incorporated in the system without modifying existing models or knowledge bases and be employed when appropriate.

User-friendliness The user interface should make extensive use of graphics for defining models, queries, and expressing results. The form of the graphical communications should be customizable for each application. Applications should provide their own palette of objects customary to their domain.

Flexibility It must handle both top-down and bottom-up modeling. In the top-down view, models are successively refined into more detailed sub-models. The bottom-up approach abstracts detailed low-level models into simpler representations.

Meta-modeling The environment must function as a model-base management system for creating, storing, retrieving, updating, sharing, and querying models.

The following simple example is used to illustrate some of these features as they currently exist in Tangram.

7.1.2 Example Modeling Application

The Tangram system contains several base modeling domains (e.g. queueing networks, Markov chains) from which specialized problem-oriented environments may be created. We consider a trivial example of an environment for creating models of car-washes, similar to those in [38] and [66], chosen to be easily understandable withoug a background in computer system perforamance modeling. The car-wash domain is constructed (by an expert) as a specialization of the system's basic queueing network domain. It provides a set of objects from which to build car-wash models, consistency constraints, a query language, and rules for translating a car-wash model into one the system can solve.

First, the user selects the application domain (from a pull-down menu). Graphical objects are then instantiated by the user from a palette of icons representing object classes provided by the creator of the domain. Figure 7.1 shows a simple model constructed in the car-wash domain using Tangram's graphical interface. Model objects include the attendants, car washing facilities (two shown), and a hand waxing station. In this particular example the user specifies that 10 percent of customers want their cars waxed (modeled by the branching probabilities at the exits of the car washing facilities), and that customers arrive at the attendant at a rate of 20 per hour. In this case, the very primitive query language consists of a generic query object (icon at upper left corner) with the attached key word avg_wait= requesting the average customer waiting time.

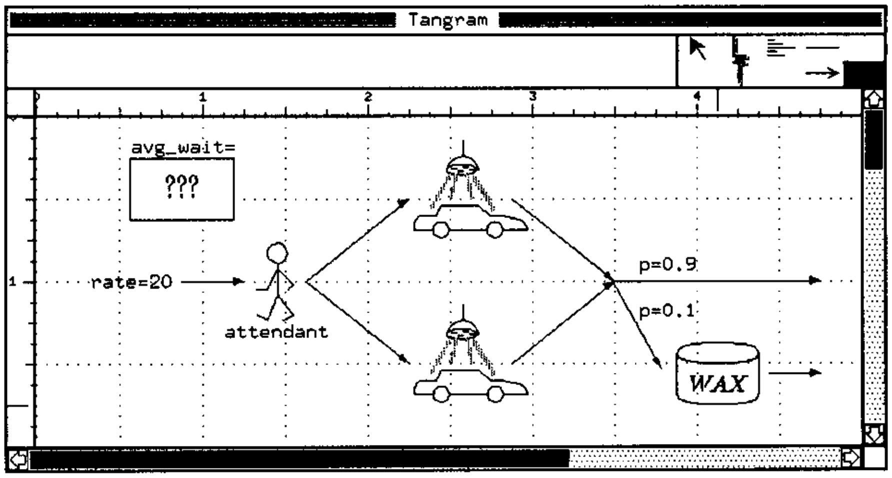


Figure 7.1: A Model in the Car-Wash Domain

When the user selects solve from the pull-down menu (not shown), the model is sent the solve message. The rules provided by the domain expert can be viewed as an expert system which examines the model and the query and generates appropriate sub-models. In this case, the system simply derives an equivalent queueing network model as shown in Figure 7.2. [1]

In this queueing model, the customer arrival is modeled as a "source" with Poisson arrival with a rate of 0.3 per minute (20 per hour). The attendant and car washing facilities are aggregated and modeled as a multi-server (whose name is wash) with mean service time of 1 minute. The hand waxing facility is modeled as a first-in-first-out server with mean service time 15. The query to the queueing model becomes a function of the average waiting time at the 'wash' queue and the 'wax' queue.

[1] Tangram does not currently generate the graphical representation of the derived queueing model; it exists only internally. Such a representation could be generated and would be useful for model debugging and explanation.

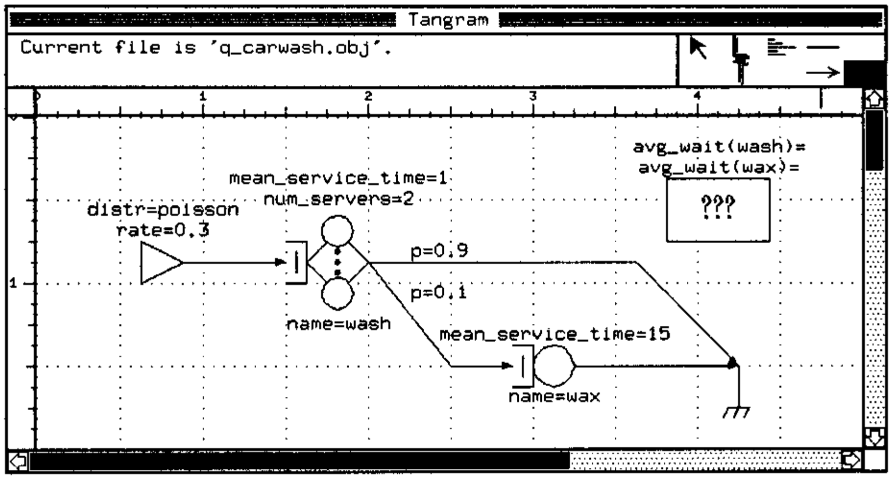


Figure 7.2: Queueing Model of the Car-Wash

The basic queueing network domain is able to solve the derived queueing model. The queueing theory expert knowledge base embedded in the queueing domain is exploited in the car-wash domain by transforming the original problem into a queueing problem. The answer to the original query is computed from the answers to the sub-model queries; in this case,

$$avg\_wait \;=\; avg\_wait(wash) \;+\; (0.1)(avg\_wait(wax)).$$

To illustrate the flexibility required of the modeling system, suppose we are now interested in the availability of the car-wash; car washing machinery may break down and the car waxer may call in sick. In this domain's simple query language, the user attaches availability= to the query object and re-solves the model. The domain expert system reinterprets the car-wash model in the reliability domain generating an instance of a reliability model (graphical representation omitted due to space constraints). The reliability domain, already provided by the base system, solves the model with Markov solution techniques.

Further, we can imagine extending the car-wash domain to handle profit and loss queries. We could model the effect of throughput on reliability (the waxing person becomes sick easier when he has to work above some threshold) or add a backup waxer who works slower, but is healthier. Solving complex models generally implies approximations and coordinated use of mathematical tools in problem dependent ways. An advanced modeling environment encourages experimentation with approximate analytic techniques such as decomposing the model and iteratively solving the sub-models (relaxation). In general, these sub-models are solved in different domains utilizing their own solution techniques, possibly using further decomposition. When the analysis procedure is validated through comparison with measurement data or simulation, it can be incorporated in the domain knowledge base.

7.1.3 Related Work

Currently, most modeling packages are designed either around one application (e.g. communication networks) or around one solver (e.g. simulation). Tools designed for analyzing the performance and reliability of computer and communication networks are most similar to Tangram's. While many packages exist for modeling reliability (e.g. SAVE [40] from IBM, HARP [10] and SHARPE [83] from Duke University, ARIES [57] from UCLA and SURF [26] from CNRS), all have the same problem. They provide a convenient interface for models that fit into the anticipated mode but no others. Tools for modeling queueing networks have similar problems (e.g. PAW [61], QNA [101] and PANACEA [79] from Bell Labs, RESQ [85,86] from IBM and PAWS [11] from University of Texas). RESQ offers both exact solution for a certain class of models and simulation for others, but the structure of the models is different depending on the analysis method. PANACEA and PAW go a step further and use the same language for several different analysis methods, but have no facilities for easily adding new modeling primitives or new analysis methods. At another extreme are tools such as petri nets [67,58] that provide generality by using only the most primitive constructs. This places too much burden on the modeler. More recent efforts are starting to provide better tools. Structured Modeling [36] is attempting to provide a general high level interface for linear programming and other operations research tools. ANALYTICOL [9] attempts to provide a high level interface for statistical analysis. Finally, [12] describes an environment for generating and analyzing Markov chains using an object-oriented approach.

7.1.4 About this Chapter

This introduction has motivated the need for an advanced environment for modeling and has given an example using the prototype Tangram system. The following section presents our approach of encapsulating models and solvers in domains using Object-Oriented Prolog and shows how the problems of semantic binding presented in Chapter 6 are handled in Tangram. Section three discusses the use of Object-Oriented Prolog in the Tangram system and the relevance of this hybrid paradigm language for modeling. Section four describes the architecture and current state of our prototype and conclusions follow in section five.

7.2 Smart Models

The key to realizing the myriad design goals for a multi-domain modeling environment is the concept of *smart models*. A smart model must be able to respond to messages by performing high-level operations on itself: solve yourself, display yourself, suggest a solution technique, etc. We call the models "smart" because much of the burden of solving models is shifted off of the user onto the system. In order to create a model in an existing application domain, a user need only know what base objects the domain provides, how to connect them, and the query language for this type of model. The model has the intelligence (inherited from its domain) to select an appropriate solver, translate itself into the solver's input format, interpret the output of the solver, and answer the query.

The concept of smart models is realized in Tangram via the novel combination of several powerful ideas. The use of object-oriented modeling organized in domains, declarative programming, non-determinism, and a new variation on multiple inheritance called *semantic binding* combine to make possible a very flexible modeling environment.

7.2.1 Object-Oriented Modeling

Foremost for the success of the system is the object-oriented structure of the modeling environment. It is natural to represent entities in an application domain as objects which respond to a well defined set of messages. For example, in a flexible manufacturing system model, domain objects might be tools, parts, and bins. New types of objects may be created by specializing existing ones. Complex subsystems can be modeled with composite objects (also called sub-models) and can be used in other models. A model as a whole is itself a composite object which responds to a set of messages.

Solvers in Tangram are also represented as objects. This is as opposed to representing solvers as methods for classes of models. Solvers are a resource that must be managed by the system. We must be able to maintain many different solvers which perhaps perform the same function only with somewhat different characteristics and would consequently occupy the same method slot (name) if implemented as methods. Further, the collection of solvers must be extensible, the addition of a solver being transparent to existing models.

A solver must know what types of models it is capable of solving and be able to estimate its complexity and accuracy. Solvers must adhere to well defined interface protocols so that alternative implementations may be substituted. The object-oriented approach to both models and solvers immediately obviates the monolithic nature of most modeling tools, naturally distributing model specifications and integrating multiple solvers.

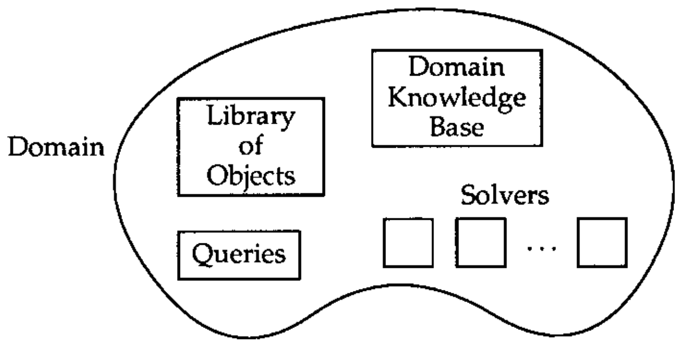


Figure 7.3: Conceptual Diagram of a Domain

Models are declared within *domains*. A domain is an environment which encapsulates the object definitions, query language, solvers, and heuristic knowledge base which combine to make up a customized package for models in a particular application. The knowledge base contains rules for checking the consistency (well-formedness) of a model, selecting a solver for a given model and query, and selecting an appropriate representation for results. Domains may be customized by expert users to create problem-specific domains for use by novices. For example, an environment for modeling the reliability characteristics of computer systems with repairable components has been created in Tangram as a specialization of a general Markov chain domain. Specializing a domain involves declaring rules for transforming a model and a query in the application domain into a sub-model and query in a more general domain.

Domain knowledge bases in Tangram are not implemented as separate software components. Rather, the knowledge is distributed among the objects in each domain. For example, solvers must "know" their own complexity; model objects must "know" how to display themselves. These objects must provide methods which query their portion of the domain knowledge. Details of how this is implemented are discussed in section 4.

7.2.2 Semantic Binding

The modeling environment application provides an excellent example of the semantic binding problem presented in Chapter 6. When a query is posed on a model, the model must first be solved, and then the answer to the query extracted from the solution. However, the solve method cannot be simply inherited from the class of the model. There are, in general, multiple solvers which could perform the task. Selecting the best one depends on the current state of the model. It may also be the case that the choice of the solver is a function, not only of the model, but of the query.

There must be an expert user supplied piece of code that runs which performs the semantic binding of model and query to solver. No other existing paradigm provides any guidance about how this binding program should be managed.

In the Tangram system, we view this binding program as part of the domain level expertise. The program which performs query interpretation and solver selection is implemented as a method within the class of models in a given domain and is inherited by all such model instances. This method must capture the domain expert's knowledge to select a solver. In writing the binding method, we take great advantage of Object-Oriented Prolog's ability to program declaratively. The expert's knowledge can be captured in the form of Prolog rules which allows the system to deduce an appropriate binding.

In order to maintain extensibility, the data used by the binding method is kept with the objects to which it pertains. In order to be used by a particular domain, a solver must provide an interface supporting the set of messages with which the binding program can query it to determine its characteristics. Similarly, all models in a given domain support the messages required by the domain to determine the characteristics of the model.

For example, a model builder in the queueing network domain should not have to be concerned about whether or not the resulting model has a product form solution. When a user queries the model, the semantic binding system must inquire of the state of the model whether any of the centers specifies a scheduling discipline which renders it non-product form. It must find out if the model uses state dependent routing (e.g. for load balancing), and whether the model uses blocking. Depending on the answers to these questions, the system may select an exact product form solver, a Markov chain solver, or an approximation technique using decomposition or simulation. Section 7.4.2 gives an example of how semantic binding is currently implemented in Tangram.

7.3 Role of Object-Oriented Prolog

It is crucial to the success of the modeling environment that we harness the advantages of both the logic and object-oriented paradigms. Declarative programming is essential to rapidly prototyping the behavior of model objects. Such rapid prototyping is necessary, as one of the primary characteristics of a modeling environment is extreme flexibility. Further, the domain expertise is generally most conveniently and flexibly expressed as rules. For these reasons, the modeling envrionment application motivated the development of Object-Oriented Prolog and was the driving force behind this whole dissertation.

Prolog's powerful knowledge representation and knowledge base querying capabilities are used in Tangram by specifying object behaviors via Prolog rules. The built-in *unification* pattern matching in Prolog allows very general rules to be expressed quite simply. Solution packages written in other languages may be encapsulated within Prolog procedures using the foreign function interface allowing the right language to be used for each task. By combining rule-based specification with object-oriented structuring and inheritance, O-OP is an ideal language for building a smart modeling environment. The next section describes the architecture of Tangram and the current status of the system.

7.4 The Tangram Modeling System

A prototype of the Tangram object-oriented modeling system is operational on SUN 3/60s. The user can construct models, define new objects, and query models graphically. The system's immediate applications are in computer systems performance modeling and the first domains were chosen to support this area's queueing network models and Markov chain analysis. In the queueing network domain, we have incorporated several exact and approximate analytic solvers. In addition, animated simulation is available. The Markov chain domain **uses Prolog's backtracking to generate a set of reachable states of the model and** the state transition rate matrix. Several numeric solvers for Markov chains are present. A specialized reliability analysis domain for modeling repairable computer systems is operational on top of the Markov chain domain. Each domain was built in a very short time frame (2 to 3 man-weeks).

7.4.1 The Architecture of Tangram

Figure 7.4 shows the basic components of the Tangram modeling system. Rectangles represent software components and ovals depict classes and object instances inside the Object-Oriented Prolog language.

In the *Graphical Front-End*, models are represented by a collection of icons (graphical representations of objects) and lines (relationships among icons) with attributes (graphical instance variables) attached to them as in Figure 7.1. Mod-

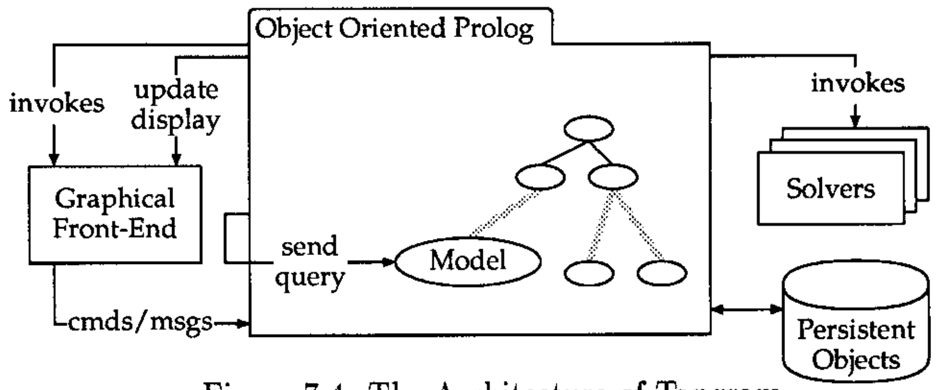


Figure 7.4: The Architecture of Tangram

els are constructed with a MacDraw[2]-like user interface (all figures in this dissertation are generated with this front-end tool), with object orientation extensions, entirely implemented in C running in the X Window System[3]. The graphics interface also supports model hierarchies; sub-models may be designed and then represented by icons as primatives used in higher level models. The user may highlight an icon in a composite model and then *push* inside it to view the underlying model.

In our prototype implementation, after the graphical representation of the model is specified, commands in O-OP to create the objects are generated by a translator and batched together to be sent to the object system. In the future, the front-end will also be implemented in O-OP; instantiation of an object at the front-end will cause immediate instantiation of the corresponding object module in the O-OP language and database.

[2]MacDraw is a trademark of Apple Computer Inc.
[3]X Window System is a trademark of the Massachusetts Institute of Technology.

7.4.2 Sample Interaction

We again use the car-wash example to illustrate the features of the Tangram prototype. When the user selects `solve` from the pull-down menu after composing the model shown in Figure 7.1, the translator generates the corresponding model and associated objects (Figure 7.6) in the object system. The model is sent the `avg_wait` query.

The class of all models in the car-wash domain contains a query method which is inherited by this model instance. The query method invokes the car-wash domain expert system. From the constituents of the model and the query posed, the domain expert system generates O-OP code as shown in Figure 7.5 to create a queueing model and sends it a list of queries. Figure 7.6 shows part of the object hierarchy after the queueing model is created. Ovals represent classes and boxes are object instances with instance variables. The small tabs on top of the objects depict object IDs; internally generated IDs are shown with single quotes.

Each `new_object` message in Figure 7.5 causes an instance of the specified class to be created. The first argument of `new_object` is bound to the object ID of the newly created instance and the second argument specifies a list of initial instance variables. The `add_center` message registers a list of queue objects with the model object, and `add_routing` specifies a list of routes between queues with associated branching probabilities. Finally, the `query(Queries,Results)`

```
   queueing_model send new_object(M1, []).
   source send new_object(SrcObj, [distr(poisson), rate(0.3)]).
   ms send new_object(MsObj,
           [mean_service_time(1),num_servers(2)]).
   fifo send new_object(FifoObj, [mean_service_time(15)]).
   sink send new_object(SinkObj, []).
   M1 send add_center([SrcObj,MsObj,FifoObj,SinkObj]).
   M1 send add_routing([
           route(SrcObj,MsObj,1.0),
           route(MsObj,FifoObj,0.1),
           route(MsObj,SinkObj,0.9),
           route(FifoObj,SinkObj,1.0)]).
   M1 send query([avg_wait(MsObj),avg_wait(FifoObj)], Results).

Results = [1.023, 27.273]
```

Figure 7.5: Sample O-OP Code

message causes `M1` to solve itself, answering the queries specified in the first argument.

`M1` inherits the `query` method from the class of all queueing models in the queueing network domain. The method invokes the queueing domain expert system which deduces that this model can be solved directly using numerical solvers. Each solver implements a `query(Model,Queries,Results)` method which binds `Results` to the list of numerical values of the answers to `Queries` for `Model`. The domain expert system tries to semantically bind the `query` message sent to the `M1` with the `query` methods of the numerical solvers in the domain.

To accomplish this, the domain's semantic binding system asks each element of the domain's list of candidate solvers (implemented as the collection object

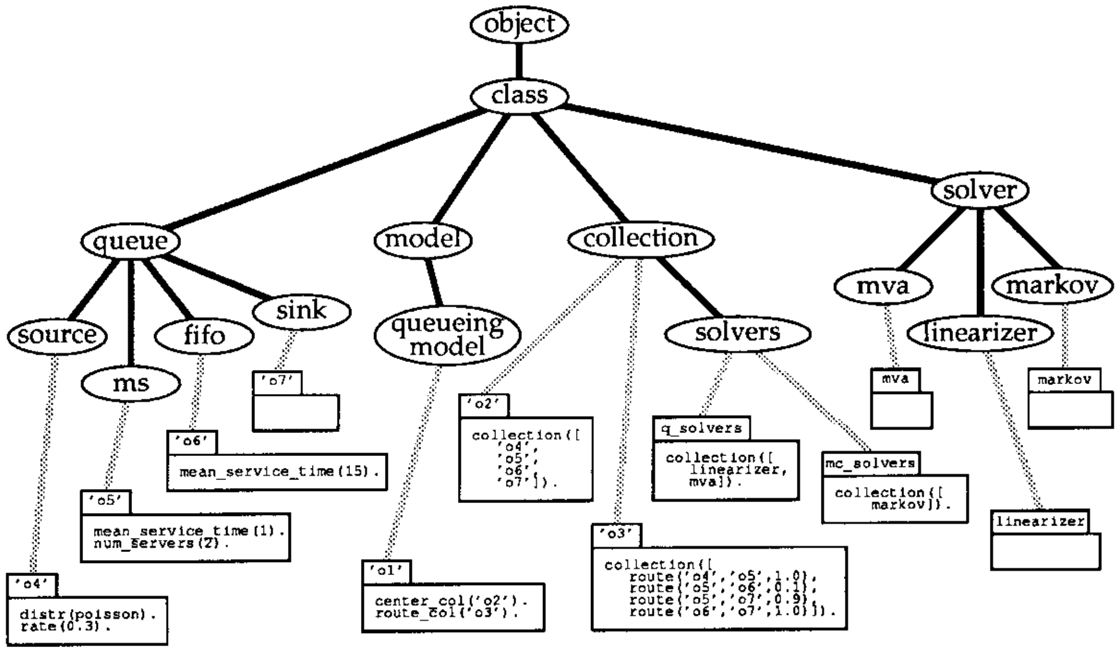


Figure 7.6: Part of Object Hierarchy Containing A Queueing Model

"q_solvers" in Figure 7.6) to estimate the complexity of answering the query. The solver with the smallest complexity measure is selected and its query method is *semantically bound* to the query message sent to M1. The selected solver is sent the message query(M1,Queries,Results) which will bind Results to a list containing the average waiting time for the car wash and the car wax facilities.

If the queueing model is more complicated and can not be solved directly with numerical solvers, more sophisticated techniques such as decomposition can be invoked to solve the model. Once the queueing model has solved itself, the car-wash domain expert system uses the results to compute the answer to the original query and updates the display in the front-end.

7.5 Conclusions

We began with the goal of creating a modeling system which could accommodate a variety of analytic and simulation modeling techniques and be easily extensible with respect to both integrating new solution techniques and tailoring the system to specialized applications. In support of these goals we developed a design philosophy that combines features from both the object-oriented and the logic programming paradigms. We introduced the notion of "smart models" which allows us to think of models which are not merely passive but rather can respond to high level queries to solve themselves, suggest solution methods, etc. This is accomplished by creating models in a "modeling domain" from which a model instance inherits knowledge of how to solve itself, etc.

A prototype of the system exists and is being used. It currently features modeling domains for queueing networks and Markov chains. Several specialized domains (e.g. reliability models) have been built on the basic system. About 35 classes of objects have been defined in the current system from which models can be created. Most models used to date have been quite small and used for demonstration purposes only. Models containing over 100 objects have been created in the system but the performance for these and larger models is currently limited by the batched translation interface between the graphical front-end and the object-oriented system. Once translated into Object-Oriented Prolog, the system's performance for large models is limited primarily by the speed of the underlying solution techniques and not by the object management system.

Due to the object-oriented structure, we have found it very easy to add new solution modules to existing domains, create new domains, or specializing existing domains. In the near future we expect the system to expand quickly. We will be adding domains for analysis of distributed algorithms, load balancing, etc. The set of users from outside the implementation group is expanding rapidly both within the department and in industry. The expanding user base and range of applications will test our goals of providing a sufficiently flexible and powerful system satisfying diverse needs. While this remains to be verified, our experience thus far and the reaction of the user community has been quite positive.

CHAPTER 8

Summary and Continuing Research

8.1 Summary

The goal of this dissertation is a language to support the development of an advanced object-oriented modeling environment. We utilize an object-oriented structure for the modeling environment. Such a structure allows models to be constructed top-down (by specializing high level models) or bottom-up (by composing detailed sub-models). It allows for the customizing of model components and interfaces to individual problem domains by specializing general concepts. It provides a paradigm for encapsulating "off the shelf" solvers for use by the system to answer queries on models. As a modeling environment is not a static system but rather intended to evolve constantly, the software engineering practices the paradigm encourages and supports are even more important than in conventional applications. The object-oriented approach is essential to the success of such a general and flexible modeling environment.

The other key component to the success of the modeling environment is the ability to specify knowledge declaratively in a logic language. When knowledge is represented declaratively instead of algorithmically, it may be used in contexts that were not anticipated when the knowledge was encoded. Human expert knowledge is often much more conveniently expressed declaratively. Freed from the burden of stipulating flow of control, programs can be written in logic much more rapidly than in conventional procedural languages.

8.1.1 Object-Oriented Prolog

In this dissertation, we have designed and implemented a new language, Object-Oriented Prolog, which delivers the best of both paradigms. The methods which define an object's behavior are specified using Prolog while higher level structure and flow of control are organized using communicating objects. O-OP delivers a "seamless" integration of the two paradigms; message sending is interpreted as goal invocation from the point of view of Prolog allowing messages to be freely embedded within Prolog clauses. Similarly, within the method code that runs when an object receives a message, conventional Prolog may be mixed with messages to other objects.

Object-Oriented Prolog supports the modularization of code into an hierarchy of classes. Object instances inherit their behavior from their class. Any methods not overridden in the object's class may be inherited from the (or one of the) super class(es). The system provides a basic set of utility methods which are inherited from the root object in the class hierarchy. Thus the O-OP language eliminates the primary impediment to the use of Prolog for large applications by providing for well structured modularization of programs.

8.1.2 Design and Implementation

The Object-Oriented Prolog language is implemented as an interpreter running on top of a Warrent Abstract Machine extended to support modules. We have added a modules facility to the Swedish Institute of Technology Warren Abstract Machine. This modules system partitions the name space of Prolog into arbitrarily many naming contexts to create the effect of many cooperating abstract machines. The system supports static or dynamic binding of names used across context boundaries. The O-OP interpreter has also been ported to run on Quintus Prolog using their modules implementation.

8.1.3 Formal Semantics

In Chapter 3, we specify the formal semantics of Object-Oriented Prolog. We first give an operational semantics in which the abstract machine maintains separate code databases for each module. The machine also has a state variable which holds the current module for the goal being tested (the top of the goal stack). Transition rules which change the current module are provided. The context to return to upon backtracking is saved on the choice point stack. The semantics of object-oriented message passing are then defined in terms of the behavior of the modular Prolog program into which the message sending is interpreted.

Next, we give a Strachey-Stoy style denotational semantics for the language. The denotation of a program is modeled as a function from inputs to outputs. The meaning function is recursively defined in terms of functions which define the meaning of individual components of the program. Each of these functions are, in turn, defined by the meaning functions of sub-components of the program until a primative layer is reached. This method has a firm mathematical basis which ensures the well-formedness (existence of fix points) of the recursive definitions.

8.1.4 Semantic Binding

One of the most exciting aspects of this research is the discovery that a very important and general class of name binding is not supported by any existing programming paradigm. In conventional languages, when a function is applied to an object, the name uniquely identifies the address of the code to run to realize the function. Later languages allow for some overloading of the function name such that the code applied to the object depends not only on the name, but on the type of the object. Object-oriented languages allow for a hierarchical structuring of types and inheritance. Surprisingly however, this is still far from being a general enough name binding mechanism for an environment as dynamic as the Tangram Modeling system.

In general, there may be many alternative implementations for a single logical function that must be simultaneously maintained by a system. When a programmer calls the function by its logical name, the system must bind to the correct, or one of the correct, implementations or solvers. The object-oriented paradigm allows for this binding to be made on the basis of the type of the target object. This is, in general, insufficient as the choice of the best binding often depends on the current state of the dynamic object (type is generally a static property). It is often the case that expert knowledge must be brought to bear to determine which solver is correct or most efficient.

We propose two approaches to providing this type of binding. The first we call *semantic binding*. In this paradigm, a user supplied program is run to select a solver to apply to the target object for a given query. This program is specified in Object-Oriented Prolog allowing for the declarative knowledge representation that is generally most convenient for expressing expert knowledge. The knowledge that this binding program must access is partitioned among the solver objects and the target model objects. These objects must obey a standard interface protocol so that they respond to messages from the binding program with information about their current state or capabilities. Hence, new solvers may be added to the system and may be used by the system to solve queries where appropriate without modification to any other part of the environment.

The second approach is based on relaxing the static typing traditional in object-oriented systems. Rather than fixing the type of an object at creation time, we can specify only its most general type. When it receives a message, the system must classify the object to the most specific type for which it satisfies the constraints. The name binding is then fixed by the dynamic type. There may, however, be more than one possible type assignment so an expert decision must be made. Adding a new solver necessitates creating a new subtype and specifying its membership constraints. These two approaches are logically equivalent. The

Tangram Modeling Environment employs the first.

8.1.5 Stream Processing

This dissertation also proposes that Prolog be the integrating vehicle for all three major programming paradigms. We show how object-oriented and logic programming can be cleanly integrated. Narrain's dissertation shows via the Log(F) language that functional programming can be realized within Prolog. In Chapter 5, we propose to combine all three, taking advantage of the fact that both Log(F) and O-OP compile to Prolog. This allows each paradigm to be used to its best advantage: object-oriented for structuring, logic for rapidly specifying methods and expressing complex queries, and re-write rules for stream processing. Many important data types within objects are inherently stream valued and are most conveniently queried with Log(F). In Object-Oriented Prolog, the Log(F) super-set of Prolog can be used to write methods and message sending (like any Prolog goals) can be embedded within re-write rules.

8.1.6 The Tangram Modeling Environment

Its use in implementing the Tangram Modeling Environment is the ultimate test of the utility of Object-Oriented Prolog. The modeling system is entirely written in O-OP with the exception of certain solvers and the graphical front-end (written in C). The combination of object-oriented structuring and software engineering practices with Prolog's declarative style, backtracking, and unification facilities were essential to the success of the Tangram system. The O-OP language will continue to be used beyond this dissertation, as the implementation language of the modeling environment.

8.2 Avenues for Continued Research

8.2.1 Streams of States

Our desire to allow objects to have mutable state prevents a pure, first-order logic interpretation of object-oriented programming. The issue of mutable state of objects arises because we wish to model objects of the real world whose state changes over time. The concept of temporal series presented above suggests a different approach to modeling objects with mutable state. Instead of implementing mutable state with destructive assignment, we can implement an evolving object as a stream of states [3]. A changing quantity such as the balance of a bank account is modeled as a time sequence history of balances. When the value of the balance changes, the uninstantiated tail of the stream is instantiated to the new value. From a logical point of view, state is not changing; more state is being discovered.

This approach is not entirely without problems. First, it is not on completely sound logical ground as the meta-logical **var** predicate must be used to find the "current state", the last value before the uninstantiated tail [46]. Second, it is difficult to model concurrent or asynchronous access to objects [3]. As with Concurrent Prolog, this approach requires a *merge* process to combine independent streams of messages into a single stream of messages in a "fair" way. This is difficult to do without introducing the same kind of time dependencies which the avoidance of destructive assignment prevents. Further, the purely functional, side-effect free programming style is inherently oriented towards modeling objects with rigid inputs and outputs. It is not clear how to represent constraints such as $X + Y = Z$ which have no inputs and outputs, but rather are relations.

8.2.2 Compiling Inheritance

One of the primary criticisms of object-oriented languages is the performance penalty due to late binding. We have the opportunity to trade off the added flexibility that late binding achieves for higher performance if we are willing to commit to part of the inheritance hierarchy. Our implementation modules on the Sicstus Warren Abstract Machine allows us to add physical pointers in the functor table of an object module so that inherited predicates appear to be in the object module. Thus, barring any change in the hierarchy, all run-time interpretation can be avoided. Such a compiler should be investigated and added to the Object-Oriented Prolog environment.

8.2.3 Formal Semantics of the Three Paradigm Language

While we have defined a formal semantics for the combined object-oriented logic programming language, no such semantics has been attempted for Log(F) or its combination with O-OP. Log(F) with its functional re-writing paradigm appears particularly amenable to modeling with a continuation style denotational semantics.

8.2.4 The Modeling Environment

Further research and development of the Tangram Object-Oriented Modeling Environment is continuing on several fronts. In the short term, many more solvers will be added. Another near term task is to redesign the interactive graphical front-end to integrate its object model with that of O-OP. In such a design, activity on the screen would be immediately reflected in the underlying object-oriented model (as opposed to the batched interaction that is currently used. Performance of the user interface will dramatically improve.

More support and a less ad-hoc structure is needed for the expression of the domain expert's knowledge. The system must move more towards a design in which domain experts can customize the environment for a particular modeling application without assistance from the implementors.

8.2.5 Distributed Object-Oriented Prolog

The O-OP appears to be an excellent candidate for parallel programming. The arms length interaction between communicating objects allows objects to be located on separate machines transparently to the program. The addition of asynchronous messages would allow programmers to express some of the potential for concurrency in their programs. There is a considerable amount of work in progress on concurrent Prologs and concurrent object-oriented programming environments. Parallel versions of Log(F) have already been prototyped [54]. A distributed computation model for O-OP appears to be a promising direction to pursue.

8.3 Final Summary In a Nutshell

We have successfully designed and implemented an hybrid object-oriented logic programming language. We defined its formal semantics and demonstrated its utility by using it as the implementation language for a large scale application program, the Tangram Modeling Environment. The resulting modeling system could not have been achieved in so short a time and with such flexibility without the Object-Oriented Prolog language. We proposed extensions to the language to include stream processing via embedding Log(F) in O-OP programs. We identified an important general type of name binding which is not supported by current paradigms and showed how it could be provided in O-OP.

References

[1] *Quintus Prolog Development Environment.* Quintus Computer Systems, Inc., Mountain View, California, August, 1987.

[2] M A Nait Abdallah. Procedures in horn-clause programming. In *Proceedings The Third International Conference in Logic Programming*, pages 433–447, July 1986.

[3] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[4] Hamideh Afsarmanesh, Dennis McLeod, David Knapp, and Alice Parker. An extensible object-oriented approach to databases for vlsi/cad. In *Proceedings VLDB 85*, pages 13–24, Stockholm, Sweden, 1985.

[5] Lloyd Allison. *A Practical Introduction to Denotational Semantics.* Cambridge Computer Science Texts 23, Cambridge, UK, 1986.

[6] Timothy Andrews and Caraig Harris. Combining language and database advances in an object-oriented development environment. In *Proceedings OOPSLA '87*, pages 430–440, October 1987.

[7] K.R. Apt and M.H. Van Emden. Contributions to the theory of logic programming. *JACM*, 29(3):841–862, July 1982.

[8] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[9] AT&T. Analyticol - an analytical computing environment. *AT&T Technical Journal*, 64(9), November 1985.

[10] S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothmann, and W. E. Smith. Analysis of typical fault-tolerant architectures using harp. *IEEE Transactions on Reliability*, 36(1):176–185, June 1987.

[11] R. Berry, K. M. Chandy, J. Misra, and D. M. Neuse. *Paws 2.0: Performance Analyst's Workbench Modeling Methodology and User's Manual.* Information Research Associates, Austin, Texas, 1982.

[12] S. Berson, E. de Souza e Silva, and R.R. Muntz. *An Object Oriented Methodology for the Specification of Markov Models.* Technical Report CSD-870030, UCLA Computer Science Department, Los Angeles, CA 90024-1596, 1987.

[13] Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proceedings OOPSLA '87*, pages 441–451, Orlando, Florida, October 1987.

[14] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: merging common lisp and object-oriented programming. In *Proceedings ACM Conf on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, September 1986.

[15] K.A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In Clark and Tarnlund, editors, *Logic Programming*, pages 153–172, Academic Press, 1982.

[16] Kenneth A. Bowen. *Meta-Level Programming and Knowledge Representation.* OHMSHA, LTD and Springer-Verlag, 5 August 1985.

[17] Kenneth A. Bowen and Tobias Weinberg. A meta-level extension of prolog. *IEEE Symposium on Logic Programming*, 48–53, 1985.

[18] R.S. Boyer and J.S. Moore. The sharing of structure in theorem-proving programs. In B. Meltzer D. Michie, editor, *Machine Intelligence 7*, Edinburgh University Press, 1972.

[19] Michael L. Brodie and Matthias Jarke. On integrating logic programming and data bases. In *Proceedings 1st International Conference on Expert Data Base Systems*, pages 40–62, Kiowah, SC, October, 1984.

[20] Mats Carlsson and Johan Widen. *SICStus Prolog User's Manual.* Technical Report SICS R88007, Swedish Institute of Computer Science, February 20, 1988.

[21] Michael Caruso and Edward Sciore. Meta-functions and contexts in an object-oriented database language. In *Proceedings ACM*, pages 56–65, 1988.

[22] Jan Chomicki and Naftaly H. Minsky. Towards a programming environment for large prolog programs. In *Proceedings Symposium on Logic Programming*, pages 230–241, 1985.

[23] A. Church. The calculi of λ-conversion. In *Annals of Mathematical Studies 6*, Princeton University Press, 1951.

[24] W. Clinger. *Foundations of Actor Semantics.* Technical Report AI-TR-633, MIT, Cambridge, MA, May 1981.

[25] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. *Un Systeme de Communication Homme-Machine en Francais.* Group de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.

[26] A. Costes, J. E. Doucet, C. Landrault, and J. C. Laprie. Surf: a program for dependability evaluation of complex fault-tolerant computing systems. In *Proceedings FTCS-11*, pages 72–78, June 1981.

[27] Brad J. Cox. *Object-oriented Programming: An Evolutionary Approach.* Addison Wesley, 1986.

[28] O.J. Dahl, B. Myhrhang, and K. Nygaard. *Simula67 Common Base Language.* Technical Report S-22, Norwegian Computing Center, 1970.

[29] Roland Dietrich. A preprocessor based module system for prolog. In *Proceedings TAPSOFT'89 International Joint Conference on Theory and Practice of Software Development*, pages 126–139, Barcelona, Spain, March 13-17, 1989.

[30] Klaus R. Dittrich. Object-oriented database systems: the notion and the issues. In Association for Computing Machinery, editor, *Proceedings 1986 International Workshop on Object Oriented Database Systems*, pages 2 – 91, Pacific Grove, California, September 1986.

[31] A. Ege, C.E. Ellis, and A. Wexelblat. *Gordion Functional Specification.* MCC-STP, Austin, TX, February 1986.

[32] M.H. Van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–742, October 1976.

[33] Koichi Fukunaga and Shin-ichi Hirose. An experience with a prolog-based object-oriented language. In *Proceedings OOPSLA '86*, pages 224–231, Portland, Oregon, September 29 -October 2, 1986.

[34] John Gabriel, Tim Lindholm, E.L. Lusk, and R.A. Overbeek. *A Tutorial on the Warren Abstract Machine for Computational Logic.* Technical Report ANL-84-84, Argonne National Laboratory, Argonne, Illinois, June 1985.

[35] H. Gallaire. Logic programming: further developments. In *Proceedings Symposium on Logic Programming 1985*, pages 88–96, Boston, Masschusetts, July 1985.

[36] A. M. Geoffrion. An introduction to structured modeling. *Management Science*, 34(5):547–588, May 1987.

[37] Joseph A. Goguen and Jose Meseguer. *Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism, and Partial Operations.* Technical Report draft, SRI International, Menlo Park CA 94025, Center for the Study of Language and Information, Stanford University 94305, May 17 1988.

[38] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[39] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[40] A. Goyal, W. C. Carter, E. de Souza e Silva, S. S. Lavenberg, and K. S. Trivedi. The system availability estimator. In *Proceedings FTCS-16*, pages 84–89, July 1986.

[41] E. Gullichsen. *BiggerTalk: Object-Oriented Prolog.* Technical Report STP-125-85, MCC-STP, Austin, TX, November 1985.

[42] Eric Gullichsen. *BiggerTalk* = BiggerTalk + Gordion.* Technical Report STP-053-86, MCC, Austin, Texas, February 1986.

[43] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–363, 1977.

[44] Matthias Jarke. Control of search and knowledge acquisition in large-scale kbms. In Michael L. Brodie John Mylopoulos, editor, *On Knowledge Base Management Systems*, pages 507–522, Springer-Verlag, New York, NY, 1986.

[45] N.D. Jones and A. Mycroft. Stepwise development of operational and denotational semantics for prolog. In *Proceedings 1st International Conference on Logic Programming*, pages 281–288, 1984.

[46] Kenneth Kahn, E. Tribble, M. Miller, and D. Bobrow. Vulcan: logical concurrent objects. In *Proceedings ACM Object-Oriented Programming, Systems Languages and Applications Conference*, pages 580–618, Oregon, September 1986.

[47] Martin L. Kersten and Frans H. Schippers. Towards an object-oriented database language. In *Proceedings 1986 International Workshop on Object-Oriented Database Systems*, pages 104–112, Pacific Grove, California, September 25-26, 1986.

[48] Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proceedings OOPSLA '86*, pages 406–416, September 1986.

[49] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Composite object support in an object-oriented database system. In *Proceedings OOPSLA '87*, pages 118–125, Orlando, Florida, October 1987.

[50] F. Kluzniak and S. Szpakowicz. Prolog - a panacea? In J. A. Campbell, editor, *Implementations of Prolog*, pages 71–84, Ellis Horwood Limited, Chichester, UK, 1984.

[51] R.A. Kowalski. *Logic for Problem Solving.* Elsevier North Holland, New York, 1979.

[52] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP 74*, pages 569–574, 1974.

[53] Brian K. Livezey. *The ASPEN Distributed Stream Processing Environment.* Technical Report CSD-880102 (Master's Thesis), UCLA Computer Science Departement, Los Angeles, CA 90025-1596, July 1988.

[54] Brian K. Livezey and Richard R. Muntz. Aspen: a stream processing environment. In *Proceedings PARLE'89*, Amsterdam, Netherlands, June 1989. (Also UCLA Technical Report CSD-880080).

[55] J. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 1984.

[56] David Maier. A logic for objects. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming: Preprints of Workshop*, pages 6–26, Washington, DC, August 18-22,1986.

[57] S. V. Makam and A. Avizienis. Aries 81: a reliability and life-cycle evaluation tool for fault tolerant systems. In *Proceedings FTCS-12*, pages 276–274, June 1982.

[58] A. M. Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*, 92–122, May 1984.

[59] F. G. McCabe. *Logic and Objects.* Technical Report DOC 86/9, Imperial College, London, England, 14 May 1987.

[60] M. D. McIlroy. Mass-produced software components. In J. M. Buxton P. Naur B. Randell, editor, *Software Engineering Concepts and Techniques*, pages 88–98, 1976.

[61] B. Melamed and R. J. T. Morris. Visual simulation: the performance analysis workstation. *IEEE Computer*, 87–94, August 1985.

[62] Bertrand Meyer. Genericity versus inheritance. In *Proceedings OOPSLA '86*, pages 391–405, September 1986.

[63] Betrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, 1988.

[64] Dale Miller. A theory of modules for logic programming. In *Proceedings Symposium on Logic Programming*, pages 106–114, Salt Lake City, Utah, September 22-25,1986.

[65] Naftaly H. Minsky and David Rozenshtein. A law-based approach to object-oriented programming. In *Proceedings OOPSLA '87*, pages 482–493, Orlando, Florida, October 1987.

[66] J. Misra. Distributed descrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[67] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, 31(9):913–917, September 1982.

[68] Luis Monteiro and Antonio Porto. *Contextual Logic Programming.* Technical Report UNL DI-50/88, Universidade Nova De Lisboa, Portugal, November 1988.

[69] J. H. Morrissey and L. S. Wu. Software engineering- an economic perspective. In *Proceedings Fourth Conference on Software Engineering*, pages 412–422, New York, NY, 1979.

[70] R.R. Muntz and D.S. Parker. *Tangram: Project Overview.* Technical Report Technical Report, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, March 1988.

[71] S. Narain. *LOG(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation.* Technical Report CSD-870027, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1987.

[72] S. Narain. A technique for doing lazy evaluation in logic. *J. Logic Programming*, 3(3):259–276, October 1986.

[73] Sanjai Narain. *LOG(F): An Optimal Combination of Logic Programming, Rewriting, and Lazy Evaluation.* Technical Report Ph.D. Dissertation also CSD-880040, UCLA Computer Science Department, Los Angeles, CA 90024-1596, 1988.

[74] Patrick O'Brien, Bruce Bullis, and Craig Schaffert. Persistent and shared objects in trellis/owl. In *Proceedings 1986 International Workshop on Object-Oriented Database Systems*, pages 113–123, Pacific Grove, California, September 25-26, 1986.

[75] Richard A. O'Keefe. Towards an algebra for constructing logic programs. In *Proceedings Symposium on Logic Programming*, pages 152–160, Boston, Massachusetts, July 15-18, 1985.

[76] Thomas W. Jr. Page, Steven Berson, William Cheng, and Richard R. Muntz. An object-oriented modeling enviroment. In *Proceedings Object-Oriented Programming Systems, Languages and Applications'89*, New Orleans, LA, October 2-6, 1989.

[77] D. Stott Parker, Thomas W. Page Jr., and Richard R. Muntz. *Improving Clause Access in Prolog.* UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1988.

[78] D. Stott Parker, Richard R. Muntz, and Lewis Chau. *The Tangram Stream Query Processing System.* Technical Report preprint, Department of Computer Science, University of California, Los Angeles, Los Angeles, CA 90024-1596, 1988.

[79] K. G. Ramakrishnan and D. Mitra. An overview of panacea, a software package for analyzing queueing networks. *Bell System Technical Journal*, 10(, December 1982.

[80] J.A. Robinson. Computational logic: the unification computation. In B. Meltzer D. Michie, editor, *Machine Intelligence 6*, Edinburgh University Press, 1971.

[81] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.

[82] D. Rozenshtein and N. H. Minsky. *The Darwin Software Evolution Environment.* Department of Computer Science, Rutgers University, New Brunswick, NJ, 1987.

[83] R. A. Sahner and K. S. Trivedi. Reliability modeling using sharpe. *IEEE Transactions on Reliability*, 36(2):186–193, June 1987.

[84] D. T. Sannella and L. A. Wallen. A calculus for the construction of modular prolog programs. In *Proceedings SLP '87*, pages 368–378, September 1987.

[85] C. H. Sauer, E. A. MacNair, and J. F. Kurose. *Computer Communication System Modeling with the Research Queueing Package Version 2.* Technical Report RA-128, IBM, November 1981.

[86] C. H. Sauer, E. A. MacNair, and J. F. Kurose. Queueing network simulations of computer communication. *IEEE Journal on Selected Areas in Communications*, 2(1):203–220, January 1984.

[87] D. Scott. *Continuous Lattices.* Technical Report PRG-7, Oxford University Programming Research Group, Oxford, UK, 1971.

[88] D. Scott. Data types as lattices. *SIAM Journal of Computing*, 5(3):522–587, September 1976.

[89] D. Scott and C. Strachey. *Towards a Mathematical Semantics for Computer Languages.* Technical Report PRG-6, Oxford University Programming Research Group, Oxford, UK, 1971.

[90] Arie Segev and Arie Shoshani. Logical modeling of temporal data. In *Proceedings ACM SIGMOD*, pages 1–13, 1987.

[91] Ehud Shapiro. *A Subset of Concurrent Prolog and Its Interpreter.* Technical Report TR-003, ICOT, 1983.

[92] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in concurrent prolog. In *New Generation Computing*, pages 25–48, Ohmsha Ltd. and Springer-Verlag, 1983.

[93] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: themes and variations. *The AI Magazine*, 6(4):40–62, 1986.

[94] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, MA, 1986.

[95] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[96] C. Strachey. Towards a formal semantics. In T. B. Steel, editor, *Formal Language Description Languages*, pages 198–220, North-Holland, London, UK, 1966.

[97] Bjarne Stroustrup. *The C++ Reference Manual.* Addison-Wesley, 1986.

[98] Eric Dean Tribble, Mark S. Miller, Kenneth Kahn, Daniel G. Bobrow, Curtis Abbott, and Ehud Shapiro. Channels: a generalization of streams. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers, vol. 1*, pages 446–463, MIT Press, 1987.

[99] O. De Troyer, J. Keustermans, and R. Meersman. How helpful is an object-oriented language for an object-oriented database model? In *Proceedings 1986 International Workshop on Object-Oriented Database Systems*, pages 124–132, Pacific Grove, California, September 25-26, 1986.

[100] David H.D. Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, SRI International, Menlo Park, CA 94025, October 1983.

[101] W. Whitt. The queueing network analyzer. *Bell System Technical Journal*, 62(9):2779–2815, November 1983.

[102] Carlo Zaniolo. Object-oriented programming in prolog. In *Proceedings Int. Logic Programming Symposium*, pages 265–270, 1984.

[103] Carlo Zaniolo. The representation and deductive retrieval of complex objects. In *Proceedings Very Large Data Bases 85*, pages 458–469, Stockholm, Sweden, 1985.

APPENDIX A

Prolog Code for Object Interpreter System

```
%       The object_system module implements the Object-
%       Oriented Prolog interpreter.
%
:- begin_module(object_system,
    [isa/2,super/2,send/2,inst/2, make_object/5,
    remake_object/5, save_object/2, load_instance/2,
    sendsuper/3]).

:- use_module(basics,[append/3]).

:- prolog_flag(unknown,_,fail).
:- op(875,xfx,isa).
:- op(875,xfx,super).
:- op(875,xfx,send).
:- op(875,xfx,inst).

%
%       Lookup 'isa' and 'super' relations.
%
isa(Oid,Object) :-
        isa_relation(Oid,Object).

super(Oid,Object) :-
        super_relation(Oid,Object).


%       Inherit an instance variable.  Different from send in
%       that we check for the instance variable in the object
%       instance first, and then try to inherit.
%
inst(Object,Message) :-
        Object:current_predicate(_, Message),!,
        call(Object:Message).
```

```
inst(Object,Message) :-
        Object isa Class,
        inherit(Class,Message).

%
%        Send a message to an object.  Increase the arity of the
%        message by 1.  This adds who was the sender to the
%        message for future reference.
%
send(Object,Message) :-
        create_message(Message,Object,Newmessage),
        Object isa Class,
        inherit(Class,Newmessage).

send(Object,revive) :-
        current_module(Object), !,
        format("Error: cannot revive already active object
                (~w)~n",Object),
        fail.

send(Object,revive) :-
        active_module(Active),
        check_exists(Object),
        module(Object),
         use_module(object_system),
        load_instance(Object,Isa),
        module(Active),
        assert(Isa).

send(Object,Message) :-
        \+(Message=revive),
        \+(current_module(Object)),
        Object send revive,
        create_message(Message,Object,Newmessage),
        Object isa Class,
        inherit(Class,Newmessage).


%
%        sendsuper
%
sendsuper(Super, Message,Sender) :-
        create_message(Message,Sender,Newmessage),
        inherit(Super,Newmessage).
%
%        Try to inherit the method from the superclasses.
%        We give up when the superclass of an object is itself.
%
inherit(Class,Message) :-
        Class:current_predicate(M, Message),
        !,
        call(Class:Message).
inherit(Class,Message) :-
        Class super Super,
        Class \== Super,
        inherit(Super,Message).

%
%        Make a new object.  Create a Prolog module with the
%        object's name and assert the following items.
%        1) oid/1 - this is the object id for this object.
%        2) assert the isa/2 and super/2 relations in the system
%               module for this object.
%        3) assert method and instance clauses for this object.
%
make_object(Oid,Isa,Super,Methods,Instances) :-
        active_module(X),
        module(Oid),
         use_module(object_system),
        assert(oid(Oid)),
        module(X),
        assert_relationships(Oid,Isa,Super),
        assert_methods(Methods,Oid),
        assert_instances(Instances,Oid).

remake_object(Oid,Isa,Super,Methods,Instances) :-
        drop(Oid),
        active_module(X),
        module(Oid),
         use_module(object_system),
        assert(oid(Oid)),
        module(X),
        retract_relationships(Oid),
        assert_relationships(Oid,Isa,Super),
        assert_methods(Methods,Oid),
        assert_instances(Instances,Oid).
```

```
%
%        Place instance variables in the new object.
%        Assert the list of clauses into the module
%        for that object.
%
assert_instances([],_).
assert_instances([(:- Directive) | Tail],Oid) :-
        call(Directive),
        assert_instances(Tail,Oid).
assert_instances([Head|Tail],Oid) :-
        assert(Oid:Head),
        assert_instances(Tail,Oid).

%
%        Place methods in an object.  Increase the arity of a
%        clause by 1 to include the message sender's oid.
%        Substitute sender/1 in the clause body to be unified
%        with the sender of the message.
%
assert_methods([],Object) :-
        !,
        translate_logf,
        do(storeNewRules(Object)).
assert_methods([(X=>Y)|Tail],Object) :-
        !,
        recordz('$tmp',(X=>Y),_),
        assert_methods(Tail,Object).
assert_methods([],_).
assert_methods([(Head :- Body)|Tail],Object) :-
        !,
        create_message(Head,Sender,Newhead),
        create_body(Body,Sender,Newbody),
        assertz(Object:(Newhead :- Newbody)),
        assert_methods(Tail,Object).
assert_methods([Term|Tail],Object) :-
        assert_methods([(Term :- true)],Object),
        assert_methods(Tail,Object).

%
%        Parse a Prolog clause body to change a sender/1 term to
%        be the send argument of the method message.  Replace
%        send/1 with true/0.  Doesn't handle every Prolog
%        clause, but this is a prototype.
%
create_body((Body1 , Body2),Sender,(Newbody1 , Newbody2)) :-
        !,
        create_body(Body1,Sender,Newbody1),
        create_body(Body2,Sender,Newbody2).
create_body((Body1 ; Body2),Sender,(Newbody1 ; Newbody2)) :-
        !,
        create_body(Body1,Sender,Newbody1),
        create_body(Body2,Sender,Newbody2).
create_body((Body1 -> Body2),Sender,(Newbody1 -> Newbody2)) :-
        !,
        create_body(Body1,Sender,Newbody1),
        create_body(Body2,Sender,Newbody2).
create_body(call(Term),Sender,call(Newterm)) :-
        !,
        create_body(Term,Sender,Newterm).
create_body((\+ Term),Sender,(\+ Newterm)) :-
        !,
        create_body(Term,Sender,Newterm).
create_body(sender(Sender),Sender,true) :- !.
create_body(Body,_,Body).

%
%        Add a 'isa' and 'super' relation.
%
assert_relationships(Oid,Isa,Super) :-
        assert_isa(Oid,Isa),
        assert_super(Oid,Super).

assert_isa(Oid,Isa) :-
        nonvar(Isa),
        !,
        assert(isa_relation(Oid,Isa)).
assert_isa(_,_).

assert_super(Oid,Super) :-
        nonvar(Super),
        !,
        assert(super_relation(Oid,Super)).
assert_super(_,_).
```

```
retract_relationships(Oid) :-
        retract(isa_relation(Oid,Isa)),
        retract(super_relation(Oid,Super)).
%
%        Create the real form of message by increasing the arity
%        of the message by 1 and adding the sender as the last
%        argument.
%
create_message(reduce(X,Y),Sender,reduce(X,Y)) :- !.
create_message(Message,Sender,Newmessage) :-
        functor(Message,Name,Arity),
        Arity1 is Arity + 1,
        functor(Newmessage,Name,Arity1),
        copy_arguments(0,Arity,Message,Newmessage),
        arg(Arity1,Newmessage,Sender).

%
%        Copy arguments of of one term on another term.
%
copy_arguments(Arity,Arity,_,_) :- !.
copy_arguments(N,Arity,Term1,Term2) :-
        N1 is N + 1,
        arg(N1,Term1,Arg),
        arg(N1,Term2,Arg),
        copy_arguments(N1,Arity,Term1,Term2).

do(G) :- G, !.
do(_).

storeNewRules(Object) :-
        recorded('$tmp',R,Ref),
        erase(Ref),
        assertz((Object:R)),
        fail.
%
%        Create a filename by appending Dir and Oid.  Write
%        necessary info to that file to reconstruct the object.
%

save_object(Oid,Dir) :-
        name(Dir,DirL),
        name(Oid,FileL),
        append(DirL,FileL,PathL),
        name(Path,PathL),
        object_system:isa_relation(Oid,Class),
        telling(Current),
        tell(Path),
        portray_clause(isa_relation(Oid,Class)),
        write_ivs(Oid),
        told,
        tell(Current).

write_ivs(Module) :-
        module(Module),
        current_predicate(_, Pred),
        functor(Pred,F,A),
        predicate_owner(F,A,M),
        M==Module,
        clause(Pred,Body),
        portray_clause((Pred :- Body)),
        fail.
write_ivs(_):- module(object_system).

%
%        check if a saved object exists in the database
%

check_exists(Oid) :-
        name('/u/w6/page/objinterp/obase/',DirL),
        name(Oid,FileL),
        append(DirL,FileL,PathL),
        name(Path,PathL),
        prolog_flag(fileerrors,Oldvalue,off),
        see(Path),
        prolog_flag(fileerrors,_,Oldvalue).

%
%        Load the instance variables from the saved file.
%        Return the isa_relation.
%

load_instance(Oid,isa_relation(Oid,Class)) :-
        read(isa_relation(Oid,Class)),
        read_rest(Oid),
        seen.
```

```
read_rest(Oid) :-
        read(Term),
        handle(Term,Oid).

handle('end_of_file',_).
handle(Term,Oid) :-
        assert((Oid:Term)),
        read(NewTerm),
        handle(NewTerm,Oid).

:- end_module(object_system).
```

APPENDIX B

Root (Object) Object Definition

```
%
%       The object object is the root of the inheritance
%       hierarchy.
%
:-  make_object(

%
%       Object Id
%
object,

%
%       The isa relation
%
object,

%
%       The super relation
%
object,

%
%       Methods of the object
%
[
(       list:- sender(Sender), Sender:listing(oid)),

%
%       Create a new object by assigning it a unique ID
%       and placing it in the class of the caller.
%
       (new_object(Oid,Super,Methods,Instances) :-
              var(Oid),
              !,
                generate_oid(Oid),
                sender(Sender),
                make_object(Oid,Sender,Super,Methods,Instances)),
        (new_object(Oid,Super,Methods,Instances) :-
                Oid isa _,
%               format("Recreate ~w? (y or n)",[Oid]),
%               get(Char), name(C,[Char]),
%               handle(C,Oid),
                format("Recreating ~w~n",[Oid]),
                sender(Sender),
                remake_object(Oid,Sender,Super,Methods,Instances)),

        (new_object(Oid,Super,Methods,Instances) :-
                sender(Sender),
                make_object(Oid,Sender,Super,Methods,Instances)),
        (savestate :-
                sender(Sender),
                save_object(Sender,'/u/tangram/u/obase/')),
        (set_inst(Inst) :-
                sender(Sender),
                functor(Inst,F,A),
                functor(OldInst,F,A),
                ((retract(Sender:OldInst)); (true)),
                assert(Sender:Inst))
],

%
%        The Instance variables of the object
%
[

%
%        Object ID's start at zero and increment by 1.
%
(generate_oid(Name) :-
        retract(current_oid(N)),
        !,
        checkNconvert(N,NewN,Name),
        N1 is NewN + 1,
        assert(current_oid(N1))),
(generate_oid('o0') :-
        \+(current_module('o0')),
        assert(current_oid(1))),
(:- object:use_module(basics,[append/3])),

(checkNconvert(N,N,Name) :-
        number_chars(N,List),
        append("o",List,List1),
        atom_chars(Name,List1),
        \+(current_module(Name)),!),
(checkNconvert(N,New,Name) :-
        N1 is N+1,
        checkNconvert(N1,New,Name)),

(handle('y',Oid)),
(handle('n',Oid):-
        format("Not creating duplicate object (~w)~n",Oid),
        !,fail),
(handle(Other,Oid) :-
        Other \== 'y', Other \== 'n',
        format("Recreate ~w?  Must answer y or n ",Oid),
        get(Char), name(C,[Char]),
        handle(C,Oid)),

(cleanup(Oid):-
        format("cleanup ~w.   Do nothing for now.~n",[Oid]))
]).
```

APPENDIX C

Class Definition

```
%
%       The current version of the system does not use
%       factory objects.  Instead, factory methods are
%       asserted along with instance methods.  When that
%       is fixed, this object will be obsolete.  Until then,
%       new classes are created by sending a new_object
%       message to the class object.
%
:- object send new_object(
%
%        Object ID
%
class,

%
%       superclass
%
object,

%
%       Methods
%
[

%
%       Create a new instance of a class.
%
        (new_object(Oid) :-
                sender(Sender),
                Sender send new_object(Oid,[])),

        (new_object(Oid,Instances) :-
                sender(Sender),
                Sender send new_object(Oid,_,[],Instances),
                   Oid send initialize),

          (initialize)
],

%
%         Instance variables
%
[]).
```

APPENDIX D

Prolog Code for Typed Log(F) Compiler

```
%
% Filename: ./execute.pl
% Authors:  Tom Page & Cliff Leung
% Remarks:  The compiler selects the appropriate implementation
%   version of transducers.  It first converts the input Log(F)
%   query to a network structure and then does some transformation
%   which selects the correct version of transducers.  Finally it
%   converts the (modified) network structure back to a Log(F)
%   expression.
%
% Nodes correspond to transducers; pipes correspond to the links
% which connect transducers.
%

% execute a Log(F) query
execute(LogfQuery) :-
    compile(LogfQuery,Nodes,Pipes),   % generate network structure
    translate(Nodes,Pipes,NewNodes),  % transform network
    codeGenQuery(NewNodes,Pipes,Goal),% convert back to Log(F)

    writeQuery(NewNodes,Pipes),       % debugging
    nl, nl, writeq(Goal), nl, nl,     % debugging

    call(Goal),                       % execute the query
        !.

compile(LogfQuery,Nodes,Pipes):-
    LogfQuery =.. [_|Args],
    functor(LogfQuery,Name,N),
    graphGen(Name/N,Args,Nodes,Pipes,output).

graphGen(Name,Args,[node(ID,Name,[],[OPID],[])],
           [pipe(OPID,_,ID,ParentNodeID)],ParentNodeID) :-
    stream(Name),
    Args=[],
    !,
    genID(ID),
    genID(OPID).

graphGen(Name,Args,[node(ID,Name,InPipes,[OPID],ListArgs)|Nodes],
        [pipe(OPID,_,ID,ParentNodeID)|Pipes],ParentNodeID) :-
    transducers(Name),
    genID(ID), genID(OPID),
    transducerArg(Name,ArgTypes),
    procArgs(ID,Args,ArgTypes,Nodes,Pipes,ListArgs),
    findInPipes(ID,Pipes,InPipes).


procArgs(_,[],[],[],[],[]).
procArgs(ID,[Arg1|Args],[reg|ArgTypes],Nodes,Pipes,
    [reg(Arg1)|ListArgs]) :-
        procArgs(ID,Args,ArgTypes,Nodes,Pipes,ListArgs).
procArgs(ID,[Arg1|Args],[pipe|ArgTypes],Nodes,Pipes,
    [pipe(PID)|ListArgs]) :-
        Arg1=..[_|Arg1s],
        functor(Arg1,Name,N),
        graphGen(Name/N,Arg1s,Nodes1,Pipes1,ID),
        [pipe(PID,_,_,_)|RestPipes] = Pipes1,
        procArgs(ID,Args,ArgTypes,Nodes2,Pipes2,ListArgs),
        append(Pipes1,Pipes2,Pipes),
        append(Nodes1,Nodes2,Nodes).


%
%
codeGenQuery(Nodes,Pipes,print_list(NewGoal)) :-
    findOutputTransducer(Pipes,Nodes,OutputTransducer,RestNodes),
    codeGenNode(OutputTransducer,RestNodes,Pipes,Goal),
    makeVariable(Goal,NewGoal).

codeGenNode(node(ID,Name/N,ListInput,ListOutput,ListArgs),Nodes,
    Pipes,Goal) :-
      codeGenArg(ListArgs,Nodes,Pipes,GoalList),
      Goal =.. [Name|GoalList].

codeGenArg([],_,_,[]).
codeGenArg([pipe(X)|Args],Nodes,Pipes,[Goal|Goals]) :-
```

```
    pickPrecedingNode(X,Nodes,Pipes,Node,RestNodes),
    codeGenNode(Node,RestNodes,Pipes,Goal),
    codeGenArg(Args,RestNodes,Pipes,Goals).
codeGenArg([reg(X)|Args],Nodes,Pipes,[X|Goals]) :-
    codeGenArg(Args,Nodes,Pipes,Goals).

pickPrecedingNode(X,Nodes,Pipes,node(Y,N,I,O,Info),RestNodes) :-
    member(pipe(X,_,Y,_),Pipes),     % pick the preceding node
    separate(node(Y,N,I,O,Info),Nodes,RestNodes).

findOutputTransducer(Pipes,Nodes,node(X,N,I,O,ArgList),
      RestNodes) :-
    member(pipe(_,_,X,output),Pipes),
    separate(node(X,N,I,O,ArgList),Nodes,RestNodes).

% this has to be fixed...
makeVariable(Goal,NewGoal) :-
    telling(OldOutput),
    tell('/tmp/query'),
    write(Goal), write('.'), nl,
    told,
    tell(OldOutput),
    seeing(OldInput),
    see('/tmp/query'),
    read(NewGoal),
    seen,
    see(OldInput).

% find all stream types and then translate all transducer nodes
% and find the type of remaining pipes.
%
translate(Nodes,Pipes,NewNodes) :-
    findDBStreamTypes(Nodes,Pipes,StreamNodes,TransducerNodes),
    translateNodes(Pipes,TransducerNodes,NewTransducerNodes),
    append(StreamNodes,NewTransducerNodes,NewNodes).

% ****************************************************************
% the following code figures out the types of stream from the DB
%
findDBStreamTypes(Nodes,Pipes,SNodes,TNodes) :-
    findDBStreamTypes(Nodes,Pipes,[],SNodes,[],TNodes).

findDBStreamTypes([],Pipes,SNodes,SNodes,TNodes,TNodes).
findDBStreamTypes([Node|Nodes],Pipes,InSNodes,OutSNodes,InTNodes,
  OutTNodes) :-
    Node = node(_,Name,_,[OutpipeNo],_),     % only one output!
    ( streamInfo(Name,Type)             % DB stream retrieval??
      -> setPipe(Type,OutpipeNo,Pipes),
         findDBStreamTypes(Nodes,Pipes,InSNodes,TmpNodes,
                       InTNodes,OutTNodes),
         OutSNodes = [Node|TmpNodes]
      ;  findDBStreamTypes(Nodes,Pipes,InSNodes,OutSNodes,
                       InTNodes,TmpNodes),
         OutTNodes = [Node|TmpNodes]
    ).

% ***************************************************************



% set the output pipes to their corresponding type
%   -- can't use forall/2.
% setPipes(Types,OutPipeNos,Pipes).
%
setPipes([],[],_) :- !.
setPipes([Type|Types],[P|Ps],Pipes) :-
        setPipe(Type,P,Pipes),
        setPipes(Types,Ps,Pipes).

setPipe(Type,P,Pipes) :-        % via unification!
        member(pipe(P,Type,_,_),Pipes),
    !.

% *************************************************************
% translate transducer nodes -- pick appropriate implementation
% version.
% translateNodes(Pipes,TransducerNodes,NewTransducerNodes),
%
translateNodes(_,[],[]) :- !.
translateNodes(Pipes,Nodes,[NewNode|NewNodes]) :-
    pickOneNode(Pipes,Nodes,Node,RestNodes),
    translateNode(Pipes,Node,NewNode),
    translateNodes(Pipes,RestNodes,NewNodes).

% Node is transducer being picked with all input streams defined.
% pickOneNode(Pipes,Nodes,Node,RestNodes).
```

```
%
pickOneNode(_,[],_,_) :-
    !,
    nl, nl,            % should execute this clause
    write('Error:pickOneNode/4 -- cannot execute the query!'),
    nl, nl.
pickOneNode(Pipes,[node(N,A,InPipes,B,C)|Nodes],
    node(N,A,InPipes,B,C),Nodes) :-
      forall(member(PipeNo,InPipes),
           (member(pipe(PipeNo,Type,_,_),Pipes),nonvar(Type))),
      !.
pickOneNode(Pipes,[Node|Nodes],NodePicked,[Node|NewNodes]) :-
      pickOneNode(Pipes,Nodes,NodePicked,NewNodes).

% translateNode finds the correct implementation version of the
% transducer and figures out the correct output type(s).
%
translateNode(Pipes,node(Tno,Name,Input,Output,Information),
           node(Tno,NewName,NewInput,Output,Information)) :-
    matchHighTransducer(Name,Input,Output),
    selectLowTransducer(Name,Pipes,Input,NewInput,NewName),
    findOutputTypes(NewName,Information,Pipes,NewInput,
       OutputTypes),
    setPipes(OutputTypes,Output,Pipes).    % set the output pipes.
translateNode(_,Node,_) :-
    nl, nl,                 % should execute this clause
    write('*** Error:translateNode/3'),
    write(' -- cannot translate this transducer!'),
    nl, nl,
    write(' ----- '), write(Node),
        nl, nl.

findOutputTypes(Name,Information,Pipes,Input,OutputTypes) :-
    getInputTypes(Pipes,Input,InputTypes),
    defaultOutputTypes(Name,Information,InputTypes,OutputTypes).

getInputTypes(_,[],[]).
getInputTypes(Pipes,[Pno|Pnos],[Info|Output]) :-
    member(pipe(Pno,Info,_,_),Pipes),
    getInputTypes(Pipes,Pnos,Output).


% does the high level transducer name (HighLevelName) exist??
% matchHighTransducer(+HighLevelName,+Input,+Output) :-
%
matchHighTransducer(HighLevelName,Input,Output) :-
    count(Input,NumInput),
    count(Output,NumOutput),
    transducerName(HighLevelName,_,NumInput,NumOutput,_).

% select a correct version of transducer given
% 1) Name of the transducer,
% 2) pipes connecting transducers, and
% 3) a list of (input) pipe numbers.
% selectLowTransducer(+Name,+Pipes,+Input,-NewINput,-NewName)
%
selectLowTransducer(Name,Pipes,Input,NewInput,VersionName) :-
    getInputStreamTypes(Pipes,Input,InputTypes),
    transducerName(Name,VersionName,N,_,_),
    getTransducerInputTypes(VersionName,N,Trans_Types),
    matchInputTypes(InputTypes,Trans_Types,NewInput).

getTransducerInputTypes(VersionName,N,Types) :-
    getTransducerInputTypes(VersionName,1,N,Types).

getTransducerInputTypes(VersionName,N,N,[Type]) :-
    transducerVersion(VersionName,i,N,Type),
    !.                  % to avoid backtracking... speed.
getTransducerInputTypes(VersionName,I,N,[Type|Types]) :-
    I < N,
    transducerVersion(VersionName,i,I,Type),
    Next is I + 1,
    getTransducerInputTypes(VersionName,Next,N,Types).


matchInputTypes([],[],[]).
matchInputTypes(L1,[Type|L2],[PipeNo|L3]) :-
    match(L1,Type,PipeNo,NewL1),
    matchInputTypes(NewL1,L2,L3).

% this is done non-deterministically
%
match(List,Type,PipeNo,RestList) :-
    member((PipeNo,Subtype),List),
    once(( Subtype = Type ; isa(Subtype,Type) )),
    once(separate((PipeNo,Subtype),List,RestList)).
```

```
% From a list of input pipe numbers (Input) and the pipes (Pipes),
% find the corresponding stream types.
%
getInputStreamTypes(_,[],[]).
getInputStreamTypes(Pipes,[Pno|Pnos],[(Pno,Type)|Output]) :-
    member(pipe(Pno,Info,_,_),Pipes),
    streamType(Info,Type),
    getInputStreamTypes(Pipes,Pnos,Output).

% separate the one qualified from the Input and return the rest.
% This assumes that the qualified one is unique in the Input.
% separate(X,Input,RestOfInput).
% e.g. separate(t(1,R),[t(3,a),t(4,b),t(1,a),t(2,b)],Rest) yields
% R = a and Rest = [t(3, a),t(4, b),t(2, b)].
%
separate(X,[],RestOfInput) :- !, fail.
separate(X,[X|Rest],Rest) :- !.
separate(X,[Y|Rest],[Y|NewRest]) :-
    separate(X,Rest,NewRest).


%
%    Generate a unique ID  (Probably should be more sophisticated)
%
genID(Y) :- clause(currentID(X),true)->
            (retract(currentID(X)),
            Y is X+1,
            asserta(currentID(Y)));
            (Y is 0,
            asserta(currentID(0))).

findInPipes(_,[],[]).
findInPipes(ID,[pipe(Pid,_,_,ID)|Pipes],[Pid|InPipes]) :- !,
    findInPipes(ID,Pipes,InPipes).
findInPipes(ID,[P|Pipes],InPipes) :-
    findInPipes(ID,Pipes,InPipes).

count([],0).
count([_|L],N) :- count(L,NL), N is NL + 1.

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

```
once(G) :- call(G), !.

writeQuery(Nodes,Pipes) :-
    nl, nl, write('['), nl,
        forall(member(Node,Nodes),(writeq(Node),nl)),
    write(']'), nl, nl, write('['), nl,
        forall(member(Pipe,Pipes),(writeq(Pipe),nl)),
        write(']'), nl, nl.

forall(Condition,Goal) :-
        \+ (Condition, \+ Goal).


% ***************************************************************

% Facts about the library of transducers.

% (high-level) transducers that are available to programmers.
transducers(Hname/A) :- transducerName(Hname/A,_,_,_,_).

transducerNames(HighName/A,LowName/B) :-
    transducerName(HighName/A,LowName/B,_,_,_).

% transducerName(HighName/Arity,LowName/Arity,NoInputArg,
%          NoOutputArg, FileLocation)

% SELECT

transducerName(select/3,select/3,1,1,
    '/u/s4/tangram/epigram/stream_ops.logf').
transducerArg(select/3,[pipe,reg,reg]).

% projectList
transducerName(projectList/2,projectList/2,1,1,
    '/u/s4/tangram/epigram/stream_ops.logf').
transducerArg(projectList/2,[reg,pipe]).

% MULTIPLY
transducerName(multiply/2,multdd/2,2,1,
    '/u/s4/tangram/epigram/schema/multiply').
transducerName(multiply/2,multds/2,2,1,
    '/u/s4/tangram/epigram/schema/multiply').
```

```
transducerName(multiply/2,multss/2,2,1,
    '/u/s4/tangram/epigram/schema/multiply').

transducerArg(multiply/2,[pipe,pipe]).

% transducerVersion(LowName/Arity,I_O,ArgNo,Type).
%
transducerVersion(select/3,i,1,stream).
transducerVersion(select/3,o,1,stream).

transducerVersion(projectList/2,i,1,stream).
transducerVersion(projectList/2,o,1,stream).

% multiply discrete times discrete

transducerVersion(multdd/2,i,1,dts).
transducerVersion(multdd/2,i,2,dts).
transducerVersion(multdd/2,o,1,dts).

% multiply discrete times stepwise

transducerVersion(multds/2,i,1,dts).
transducerVersion(multds/2,i,2,sts).
transducerVersion(multds/2,o,1,dts).

% multiply stepwise times stepwise

transducerVersion(multss/2,i,1,sts).
transducerVersion(multss/2,i,2,sts).
transducerVersion(multss/2,o,1,sts).

% *************************************************************
%
% This is the inference engine!!!
%
% defaultOutputTypes(LowName,Info,InputTypes,OutTypes).
%
% projectArgs/3 is defined in the file stream_ops.logf.
%
defaultOutputTypes(Name,_,[X],[X]) :-
    transducerNames(select/_,Name),
    !.
defaultOutputTypes(Name,[reg(ListOfProjectFields)|_],[X],[Y]) :-
```

```
    transducerNames(projectList/_,Name),
    !,
    streamStructure(X,Structure),
    projectArgs(ListOfProjectFields,Structure,NewStructure),
    setStreamStructure(X,NewStructure,Y).
defaultOutputTypes(Name,ListOfProjectFields,[X1,X2],[Y]) :-
    transducerNames(multiply/_,Name),
    !,
    streamType(X1,X1Type),
    streamType(X2,X2Type),
    multiplyOutputType(X1Type,X2Type,OutputType),
    setStreamType(X1,OutputType,Y).        % can do a lot more...


multiplyOutputType(dts,dts,dts).
multiplyOutputType(dts,sts,dts).
multiplyOutputType(dts,cts,dts).
multiplyOutputType(sts,dts,dts).
multiplyOutputType(sts,sts,sts).
multiplyOutputType(sts,cts,sts).
multiplyOutputType(cts,dts,dts).
multiplyOutputType(cts,sts,sts).
multiplyOutputType(cts,cts,cts).



% problems remained to be solved
% 1) properties of transducers e.g. commutativity, sorting
% 2) type hierarchy
% 3) relation types...
% 4) intermediate transducer network structure
%    node = (NodeID,HighName,ListInputPipes,ListOutputPipes,
%    Information)
%    i.e. node = (NodeID,HighName,[PipeID|_],[PipeID|_])
%    pipe = (PipeID,TypeOfData,InputNodeID,OutputNodeID)
% 5) several implementations for the same operations e.g. join,
%    and the cost model associated with them.

:- op( 999, xfy, ':').
% *************************************************************
%
% SCHEMA for streams
%
```

```
stream(StreamName) :- timeSequence(StreamName).
stream(StreamName) :- nonTimeSequence(StreamName).

setStreamStructure(Info,Structure,NewInfo) :-
    Info =.. [Functor,_|Tail],
    NewInfo =.. [Functor,Structure|Tail].

setStreamType(Info,Type,NewInfo) :-
    Info =.. [Functor,Structure,_|Tail],
    NewInfo =.. [Functor,Structure,Type|Tail].

streamStructure(Info,Type) :-
    arg(1,Info,Type).

streamType(Info,Type) :-
    arg(2,Info,Type).

streamInfo(Name,type(Structure,Type,LS,Reg,Gran,SortOrder)) :-
        timeSequence(Name,Structure,Type,LS,Reg,Gran,SortOrder).
streamInfo(Name,type(Structure,Type,SortOrder)) :-
        nonTimeSequence(Name,Structure,Type,SortOrder).

% TS stream type in the DBMS
%
dbStreamType(Name,Struct,Type) :-
         timeSequence(Name,Struct,Type,_,_,_,_).
dbStreamType(Name,Struct,Type) :-
         nonTimeSequence(Name,Struct,Type,_,_,_,_).

% non-TS streams in the DBMS (StreamName names a relation)
%
nonTimeSequence(StreamName) :- nonTimeSequence(StreamName,_,_,_).

% TS streams in the DBMS (StreamName refers to a relation name)
%
timeSequence(StreamName) :- timeSequence(StreamName,_,_,_,_,_,_).

% timeSequence(StreamName,RecordStructure,Type,LifeSpan,
%           Regularity,TimeGranuarity,Ordering).
%
% bookSales DISCRETE
%
timeSequence(bookSales/0,
    tuple(bno:integer,time:time,value:integer),
    dts,(0,20),reg,1,asc).
bookSales => file_terms('/u/s4/tangram/epigram/schema/bookSales').
bookSales(Book) => projectList([2,3],select(book_sales,
        tuple(Book,D,V), true)).
%
% bookPRICE STEPWISE
%
timeSequence(bookPrice/0,
    tuple(bno:integer,time:time,value:integer),
    sts,(0,20),irreg,1,asc).
bookPrice => file_terms('/u/s4/tangram/epigram/schema/bookPrice').
bookPrice(Book) => projectList([2,3],select(book_price,
        tuple(Book,D,V), true)).

%
% book STREAM
%
nonTimeSequence(book/0,
        tuple(bno:integer,subject:string,author:string,
            count:integer), nonTSstream, asc).
book => file_terms('/u/s4/tangram/epigram/schema/book').

%
% ibookSales DISCRETE (from ingres database)
%
timeSequence(ibookSales/0,tuple(bno:integer,time:time,
    value:integer),dts, (0,20),reg,1,asc).
%
%ibookPrice STEPWISE (from ingres database)
%
timeSequence(ibookPrice/0,
    tuple(bno:integer,time:time,value:integer),sts,
    (0,20),irreg,1,asc).
%
% ibook STREAM (from ingres database)
%
nonTimeSequence(ibook/0,
    tuple(bno:integer,subject:string,author:string,count:integer),
    nonTSstream,
    asc).
```

```
%
% definition of ingres db based streams.
%
ibook => newfunctor(tuples(times,book)).
ibookSales => newfunctor(tuples(times,book_sales)).
ibookPrice => newfunctor(tuples(times,book_price)).

ibookSales(Book) => projectList([2,3],select(ibook_sales,
        tuple(Book,D,V), true)).

ibookPrice(Book) => projectList([2,3],select(ibook_price,
        tuple(Book,D,V), true)).

newfunctor(X) => newfunctor1(reduce(X)).
newfunctor1([]) => [].
newfunctor1([S|Ss]) => if(success((S=..[F|Xs], A=..[tuple|Xs])),
    [A|newfunctor(Ss)],[]).


:- multifile user_reduce/2.

% type (isa) hierarchy -- isa(A,B) means A is a subtype of B.
%
% term is the root of the type hierarchy.
%
isa(X,Y) :- isSubtype(X,Y).
isa(X,Y) :-
    isSubtype(X,Z),
    isa(Z,Y).

isSubtype(stream,term).
isSubtype(time,term).
isSubtype(integer,term).
isSubtype(float,term).
isSubtype(string,term).

isSubtype(tsStream,stream).
isSubtype(nonTSstream,stream).
isSubtype(dts,tsStream).
isSubtype(sts,tsStream).
isSubtype(cts,tsStream).

isSubtype(ordinal,time).
isSubtype(integer,ordinal).
% isSubtype(date,time).


%
%    multss: multiply two stepwise.
%
multss(S1,S2) => multr(t2r(S1),t2r(S2)).

multr(R1,R2) => multr1(reduce(R1),reduce(R2)).

multr1([],_) => [].
multr1(_,[]) => [].
multr1([tuple(T1,T2,T3)|Ts],[tuple(D1,D2,D3)|Ds]) =>
    if(T1@<D1, multr(Ts,[tuple(D1,D2,D3)|Ds]),
    if(success((T1>=D1,T2==end,D2\==end,Vnew is T3*D3)),
       [tuple(T1,Vnew)|
       multr(Ds,[tuple(T1,T2,T3)|Ts])],
    if(success((T1>=D1,T2==end,D2\==end,Vnew is T3*D3)),
        [tuple(T1,Vnew)|
        multr(Ds,[tuple(T1,T2,T3)|Ts])],
    if(success((T1>=D1,T2==end,D2==end,Vnew is T3*D3)),
        [tuple(T1,Vnew)],
    if(success((T1>=D1,T2\==end,D2==end,Vnew is T3*D3)),
        [tuple(T1,Vnew)|
        multr(Ts,[tuple(D1,D2,D3)])],
    if(success((T1>=D1,T2<D2,Vnew is T3*D3)),[tuple(T1,Vnew)|
        multr(Ts,[tuple(D1,D2,D3)|Ds])],
    if(success((T1>=D1,T2>D2,Vnew is T3*D3)),[tuple(T1,Vnew)|
        multr(Ds,[tuple(T1,T2,T3)|Ts])],
        multr(Ds,Ts))))))).
%
%    multdd: multiply two discretes.
%
multdd(D1,D2) => multdd1(reduce(D1),reduce(D2)).
multdd1([],_) => [].
multdd1(_,[]) => [].
multdd1([tuple(D1,V1)|D1s],[tuple(D2,V2)|D2s]) =>
    if(success((D1==D2, Vnew is V1*V2)),
        [tuple(D1,Vnew)|multdd(D1s,D2s)],
    if(D1@<D2, multdd(D1s,[tuple(D2,V2)|D2s]),
        multdd([tuple(D1,V1)|D1s], D2s))).
%
```

```
%    multds: multiply a stream of discretes by a stream of type
%    stepwise to produce a stream of type discrete.
%
multds(D,S) => multds1(reduce(D),reduce(S)).

multds1([],_) => [].
multds1(_,[]) => [].
multds1(D,S) => concatDV(project(1,common(T,D)),
    mult(lookupst(common(T,D),S),project(2,common(T,D)))).

%
%    t2r(A) given a stepwise time sequence A, produce a sequence
%    of tuples (t1,t2,v1) which means v1 is the value in effect
%    from time t1 until time t2.
%

t2r(A) => t2r2(reduce(A,2)).

t2r2([tuple(T1,V1)|[]])=> [tuple(T1,end,V1)].
t2r2([tuple(T1,V1),tuple(T2,V2)|Ts]) =>
    [tuple(T1,T2,V1)|t2r([tuple(T2,V2)|Ts])].
t2r2([]) => [].

%    lookup(R,D) =>... given a time sequence R of type range,
%    and a stream of dates D, produce the value for each date.
%
lookup(R,D) => lo(reduce(R),reduce(D)).
lo([],D) => [].
lo(R,[]) => [].
lo([tuple(T1,T2,V1)|Rs],[D|Ds]) =>
    if(D @< T1,[undefined|lookup([tuple(T1,T2,V1)|Rs],Ds)],
    if(success((D @>= T1, (D @< T2; T2=end))),
        [V1|lookup([tuple(T1,T2,V1)|Rs],Ds)],
    if(D @>= T2,lookup(Rs,[D|Ds]),[]))).
%
%    lookupst: Given a discrete and a stepwise,  produce the
%    stream of values for the stepwise for the dates in the
%    discrete.
%
lookupst(D,S) => lookupst1(reduce(D),reduce(S)).
lookupst1([],_) => [].
lookupst1(_,[]) => [].
lookupst1(D,S) => lookup(t2r(S),project(1,D)).
```

```
%
%    concatDV: take a date stream and a value stream and produce a
%    stream of structures in the form of a time series.
%
concatDV(D,V) => concatDV1(reduce(D),reduce(V)).
concatDV1(_,[]) => [].
concatDV1([],_) => [].
concatDV1([D|Ds],[V|Vs]) => [tuple(D,V)|concatDV(Ds,Vs)].

%
%    multiply two streams of numbers assuming each stream has the
%    same number of elements.
%
mult(D,S) => mult1(reduce(D),reduce(S)).

mult1(_,[]) => [].
mult1([],_) => [].
mult1([D|Ds],[S|Ss]) =>  [D * S|mult(Ds,Ss)].

user_reduce(common(T,G), H) :-
    nonvar(T),
    !,
    H = T.
user_reduce(common(T,G), T) :-
    reduce(G,R),
    common(R,T).
common([A|B], [A|common(L,B)]).
common([],[]).


%
% ***********************************************************

% reduceTwice (Author: Cliff Leung) -  reduce the first two
% elements in the stream.
% It handles when the stream is nil or has only one element.
%
:- eager reduceTwice/1.
reduceTwice(X,[]) :- reduce(X,[]).
reduceTwice(X,[A1|L]) :- reduce(X,[A1|Xs]), reduce(Xs,L).
```

```
% constructor symbol.
%
tuple(A,B,C) => tuple(A,B,C).

% *****************************************************************
%
stepLookup(X,State,L) =>
             stepLookupReducedTwice(X,State,reduceTwice(L)).
stepLookupReducedTwice(tuple(A,T1,_),tuple(A,_,V),[]) =>
             [tuple(A1,T1,V)].
stepLookupReducedTwice
    (tuple(A1,T1,_),tuple(A,T,V),[tuple(A1,T2,V2)]) =>
             if( T1 @>= T2, tuple(A1,T1,V2), tuple(A1,T1,V) ).
stepLookupReducedTwice(tuple(A1,T1,_),tuple(A,T,V),
    [tuple(A1,T2,V2),tuple(A3,T3,V3)|L]) =>
             if( A1 == A3,
               if( T1 @< T2,
                 tuple(A1,T1,V),
               if( T1 @< T3, tuple(A1,T1,V2),
                 stepLookup(tuple(A1,T1,_),[tuple(A3,T3,V3)|L]) )
                 ),
               if( T1 @>= T2, tuple(A1,T1,V2), tuple(A1,T1,V) )
             ).


:- eager t1LTt2/2.
t1LTt2(tuple(A1,T1),State,L,V) :-
    L = [tuple(A1,T2,V2)|L2],
    T1 < T2,
    reduce(stepLookup(tuple(A1,T1,_),State,L),tuple(A1,T1,V)).

:- eager t2LTt1/2.
t2LTt1(L,State,tuple(A1,T2),V) :-
    L = [tuple(A1,T1,V1)|L1],
    T1 > T2,
    reduce(stepLookup(tuple(A1,T2,_),State,L),tuple(A1,T2,V)).


multiplyss(S1,S2) =>
             multiplyssReduced(reduceTwice(S1),tuple(0,0,0),
             reduceTwice(S2),tuple(0,0,0)).
multiplyss(S1,State1,S2,State2) =>
```

```
             multiplyssReduced(reduceTwice(S1),State1,
                                          reduceTwice(S2),State2).
multiplyssReduced([],_,[],_) => [].
multiplyssReduced([],tuple(SA1,ST1,SV1),[tuple(A2,T2,V2)|NewL2],_)
      =>
             if( A2==SA1,
                 [tuple(A2,T2,V2*SV1) |
                 multiplyss([],tuple(SA1,ST1,SV1),NewL2,_)],
             []).
multiplyssReduced([tuple(A1,T1,V1)|NewL1],_,[],tuple(SA2,ST2,SV2))
      =>
             if( A1==SA2,
                 [tuple(A1,T1,V1*SV2) |
                 multiplyss(NewL1,_,[],tuple(SA2,ST2,SV2))],
             []).
multiplyssReduced([tuple(A1,T1,V1)|NewL1],tuple(SA1,ST1,SV1),
          [tuple(A2,T2,V2)|NewL2],tuple(SA2,ST2,SV2)) =>
  if( A1==A2,
      if( success(t1LTt2(tuple(A1,T1),tuple(SA2,ST2,SV2),
             [tuple(A2,T2,V2)|NewL2],V)),
      [tuple(A1,T1,V1*V)|multiplyss(NewL1,tuple(A1,T1,V1),
                        [tuple(A2,T2,V2)|NewL2],
                     tuple(SA2,ST2,SV2))],
      if( success(t2LTt1([tuple(A1,T1,V1)|NewL1],
                 tuple(SA1,ST1,SV1), tuple(A2,T2),V)),
          [tuple(A2,T2,V2*V)|multiplyss([tuple(A1,T1,V1)|NewL1],
                        tuple(SA1,ST1,SV1),
                        NewL2,tuple(A2,T2,V2))],
          [tuple(A1,T1,V1*V2)|multiplyss(NewL1,tuple(A1,T1,V1),
                               NewL2,tuple(A2,T2,V2))]
          )
      ),
      if( SA1==A1,
      [tuple(A1,T1,V1*SV2)|multiplyss(NewL1,tuple(A1,T1,V1),
                       [tuple(A2,T2,V2)|NewL2],
                       tuple(SA2,ST2,SV2))],
      [tuple(A2,T2,V2*SV1)|multiplyss([tuple(A1,T1,V1)|NewL1],
                       tuple(SA1,ST1,SV1),
                       NewL2,tuple(A2,T2,V2))]
      )
  ).

% ************************************************************
% Stepwise interpolation of an sts-stream -- it converts an
% sts-stream into a regular dts-stream.  For example,
% [tuple(1,0,W),tuple(1,2,X),tuple(2,2,Y),tuple(2,3,Z)]
% is interpolated as
% [tuple(1,0,W), tuple(1,1,W), tuple(1,1,X), tuple(2,2,Y),
% tuple(2,3,Z)]
%
% It assumes each element in the stream is in the form
%     tuple(A,B,C),
% where A is the surrogate (of integer), B is the time (of ordinal
% value) and C is the value.  Also, the input stream is clustered
% on A (essentially a group_by operation & not necessarily sorted)
% and then sorted on B in ascending order.
%
stepInt(L) => stepIntReduced(reduce(L)).
stepIntReduced([]) => [].
stepIntReduced([X|L]) => reduce(stepInterpolate(X,L)).

stepInterpolate(X,L) => stepInterpolateReduced(X,reduce(L)).
stepInterpolateReduced(X,[]) => [X].
stepInterpolateReduced(tuple(A1,T1,V1),[tuple(A2,T2,V2)|L]) =>
   if( A1==A2,
       if( success((NewTime is T1+1,T2==NewTime)),
           [tuple(A1,T1,V1)|stepInterpolate(tuple(A2,T2,V2),L)],
           if( success(NewTime is T1+1),
             [tuple(A1,T1,V1)|stepInterpolate(tuple(A1,NewTime,V1),
                 [tuple(A2,T2,V2)|L])],
             []
           )
       ),
       [tuple(A1,T1,V1)|stepInterpolate(tuple(A2,T2,V2),L)]
   ).
```