Object Oriented Programming with First Order Logic

John S. Conery

CIS-TR-87-09
August, 1987

Abstract

This paper introduces a scheme for object-oriented programming within the framework of the procedural interpretation of Horn clauses. Metalogical operations, such as assert and retract, or extensions to logical operations, such as unification with read-only variables, are not required. The effects of object-oriented programming are obtained through a control strategy that interprets some negative literals as procedure calls and others as instances of objects. A new type of logical formula, the *object clause*, is introduced to describe procedures that modify instances of objects. The paper concludes with a discussion of aspects of object oriented programming, such as class hierarchies, and how they can be realized in the new framework.

Department of Computer and Information Science
University of Oregon

1 Introduction

One of the advantages of programming with logic is that procedure calls in the body of a clause have the same meaning in every context. There are no hidden states that must be kept in mind when reading a program. A procedure call with identical parameters returns the same results each time; it does not use values in the state of the execution to return different results on different calls.

This form of referential transparency is a double-edged sword. The requirement that all of the data used by a procedure must be present in its arguments can present a heavy cognitive burden in large programs. The number of arguments for any given procedure can become quite large, and procedures at the higher levels of a program are unnecessarily complicated by adding parameters that serve only as place holders to pass data structures down to the lower level procedures that will actually use them. The situation is aggravated when a program is later modified. If it is decided that a lower level procedure needs more information, higher level procedures often have to be changed. Either another parameter has to be added to pass the new information, or the structure of an existing parameter modified. Both types of changes require maintenance on parts of the program that have little to do with the reasons for the change. In short, two respected principles of modern programming languages, referential transparency and information hiding, seem to be at odds in pure logic programming languages.

While global variables or other ways of representing state information are not part of the formalism of logic programming, most high level languages based on this formalism have such facilities. Programmers often use these mechanisms to "set down" a value so it doesn't have to be "carried around" on each procedure call, and the only procedures that need to be concerned with the structure and content of the data are the procedures that create and use it. The simplest schemes use built-in predicates to manipulate an internal database; an example is the evaluable predicate `record` from DEC-10 Prolog [11]. This technique allows Prolog systems to be examples of applicative state transition systems described by Backus: the more mathematical formal language can be used to define state transitions, but the actual maintenance of the state is accomplished through a call to a mechanism outside the formalism [1]. A similar scheme is to use `assert` and `retract` predicates that modify the program as it is executing. This leads to a type of nonmonotonic reasoning, as the axioms used to make inferences change as the reasoning progresses.

Another approach to information hiding is to merge concepts from the paradigm of object oriented programming with logic programming. Most such systems are based on Concurrent Prolog or other committed choice AND-parallel languages [3,12,13]. In this approach, a perpetual process is viewed as an object. One input parameter of the process is a stream of messages to the object. A process is suspended until the first element in the stream is instantiated to a term representing a message. The process then executes, up to the point where it makes a recursive call to itself, where it again blocks until the next message arrives. Other parameters to the process represent the current state of the object.

Two logic languages with more explicit notions of classes, objects, slot variables, and other concepts from object oriented programming are ESP [2] and Vulcan [9]. The former uses evaluable predicates to manipulate object instances, while the latter is a higher level language that compiles into Concurrent Prolog.

What the above mentioned techniques all have in common is that they implement information hiding as part of the higher level language that is built on top of the logic programming foundation. In some cases, such as adding `assert` and `retract`, the extensions are simple, and easy to implement, but they obscure the semantics of programs. In other cases, complex control strategies are used to decide when a unification can take place, and to postpone operations depending on the binding status of certain variables. As a result of merging control decisions with unification, the languages are harder to implement, and again have a significantly different semantics than pure logic programs.

In this paper, we introduce a technique for object oriented programming with pure first order logic. The goal is to incorporate notions of object oriented programming at the level of the formal foundations. If successful, the semantics of objects and procedures that manipulate them could be given the same, or similar, treatment as regular procedures. The techniques described here obtain the effects of mutable objects, or data structures that change over time as the program executes, without modifying the unification and resolution operations that are the logical foundations of logic programming languages, and without introducing metalogical operations to create and manipulate the objects. The control strategy is a simple sequential operation, not much more complicated than Prolog's depth-first control. It also allows for exploratory nondeterminism, of the variety usually implemented in Prolog through backtracking.

What is presented in this paper is a framework for object oriented programming with logic. Issues usually addressed by object oriented programming, such as inheritance hierarchies, are dealt with very briefly. A Prolog meta-interpreter that handles objects has been written, and used to implement the example programs presented here. Some of these other issues of object oriented programming are discussed in more detail in the paper that describes the meta-interpreter and its language [4].

2 Definitions

An object will be described by an atomic formula, or literal. The set of predicate symbols used as object names are disjoint from the predicate symbols used for procedure names. To distinguish the two, literals formed from object names will be called *object literals*, and literals with procedure names as predicate symbols will be called *procedure literals*. The arguments of an object literal are the "slot variables" of the object, and determine the current state of an instance of the object.

A clause is a collection of literals. If all of the literals are procedure literals, the clause has the usual declarative and procedural interpretations. We need to give new interpretations to a clause that contains an object literal, either as a positive literal in the head of the clause, or as a negative literal in the body.

In the declarative reading, a positive object literal $s(a_1 \ldots a_n)$ is taken to be a declaration that $s(a_1 \ldots a_n)$ is an object. A negative object literal is a denial that the literal represents a valid object, or, equivalently, a request to prove that it is an object. Table 1 summarizes the declarative interpretations of several clauses containing object literals. In these examples, p, q, and r are procedure literals, and s is an object literal. As the examples show, object literals and procedure literals can be freely mixed in clauses.

A proof that a literal represents an object is accomplished the same way the truth of a procedure literal is established, through derivation of the empty clause via a series of resolution inferences. A negative object literal can be unified with an object literal in the head of a clause, and a new goal statement derived from the remaining literals in the goal plus the literals in the body of the clause.

The procedural interpretation of object literals is described in terms of the states of an object. A negative object literal in a goal statement is interpreted as an instance of the object. The procedural reading of a negative object literal s in the body of a clause is "create an object with

Clause	Declarative Reading
$p.$	p is true.
$s.$	s is an object.
$p \leftarrow q \wedge r.$	p is true if q and r are true.
$s_i \leftarrow s_o.$	s_i is an object if s_o is an object.
$p \leftarrow q \wedge s.$	p is true if q is true and s is an object.
$s_i \leftarrow q \wedge s_o.$	s_i is an object if q is true and s_o is an object.

Table 1: Declarative Interpretation of Object Literals

state s," since, when the clause is invoked, the literals in the body are added to the remaining parent goals to make the new goal statement. A positive object literal in the head of a clause is a precondition for execution of the goals in the body of the clause. In other words, in order to invoke the goals in the body of the clause, there must exist an object in the current goal statement with a state that matches the head of the clause. A rule with one object literal in the head and another object literal with the same name in the body can thus be interpreted as a rule for transforming the state of an object. The procedural interpretation of clauses containing object literals is summarized in Table 2.

As an example, the following clauses could be used to define of a stack of integers. The object literal `stack(L)` means the list `L` is a stack. Also, we use the Prolog syntax `[X|L]` to mean the list with head `X` and tail `L`.

```
stack([]).                  the empty list is a stack
stack([X|L]) ←              [X|L] is a stack if
      integer(X) ∧          X is an integer
      stack(L).             and L is a stack
push(X) ←                   the goal push(X) is solvable if
      stack([X|L]).         [X|L] is a stack
stack(L) ←                  the stack L can be transformed into
      stack([X|L]).         a new stack [X|L] (by pushing X)
```

The declarative readings of the last two clauses are: "`push(X)` is true if there exists an `L` such that `[X|L]` is a stack" and "`L` is a stack if there exists an `X` such that `[X|L]` is a stack." The procedural readings are "to solve a

Clause	Procedural Reading
$p.$	procedure p is solved.
$s.$	s is the current state of the object.
$\leftarrow p.$	call procedure p; show p is solvable.
$\leftarrow s.$	create an object with state s.
$p \leftarrow q \wedge r.$	to solve p, solve subgoals q and r.
$s_i \leftarrow s_o.$	given an object with state s_i, make an object with state s_o; transform s_i into s_o.
$p \leftarrow q \wedge s.$	to solve p, solve q and create object s.
$s_i \leftarrow q \wedge s_o.$	s_i can be transformed to s_o if q is solvable.

Table 2: Procedural Interpretation of Object Literals

goal **push(X)**, create a stack with X as top element and any stack L for the remaining elements" and "if there exists a stack L, it is possible to create a new stack of the form [X|L]."

This interpretation of object literals is not very useful. As the above examples show, binary resolution will allow us to call a procedure, such as **push**, to create a new object that has X on its top and an arbitrary tail, or it will allow us to transform a stack by pushing an arbitrary integer.

For the interpretation of objects to be practical, we want to combine the effects of procedure call and object transformation. In other words, we want to define a procedure that operates on an instance of an object to create a new instance of the object. We can do this by combining two Horn clauses, one with a head that is an object literal, and the other with a head that is a procedure literal, into a single logical formula. The "head" of this formula will be the conjunction of the procedure literal and the object literal. The new formula is no longer in clause form, since the "head" will be a conjunction and not a single literal. However, we will call this formula a clause, since, as will be shown below, it can be regarded as shorthand notation for two Horn clauses, and operations with this formula are shortcuts for successive operations with the corresponding Horn clauses.

Definition An *object clause* is a well-formed formula that is a disjunction of component wffs, where one component is a conjunction of a positive procedure literal and a positive object literal, and the other components are negative literals. The conjunction is called the *head* of the object clause, and the (zero or more) negative literals form the *body* of the clause. ∎

It is easy to show, by converting the implication into disjunction and distributing the conjunction over the body of the clause, that the object clause

$$p \wedge s_i \quad \leftarrow \quad q \wedge s_o \tag{1}$$

is equivalent to the two Horn clauses

$$\begin{aligned} p &\leftarrow q \wedge s_o \\ s_i &\leftarrow q \wedge s_o \end{aligned}$$

Note that the bodies of the two Horn clauses are identical.

The procedural interpretation of the object clause in equation (1) is: "to solve p, given object s_i, solve q and create a new object in state s_o." An object clause for a procedure that pushes an item X onto a stack with current state L is:

```
push(X) ∧ stack(L) ← stack([X|L]).
```

Combining the two Horn clauses into a single object clause means the two clauses now have a common set of variables, which is just what we want. Intuitively, by combining the two heads into a single term, we are implying that one is not reducible without the other. We do not want to call a procedure that modifies an object's state without having a clear statement of what that state is, and we do not want to spontaneously change the state of an object without parameters from a procedure call to define the values of the new state.

The following definitions expand the definition of a procedure in a Horn clause program to include our new notions of objects and the procedures that manipulate them.

Definition An occurrence of an object literal in a goal statement is an *instance* of an object. ∎

Definition A *method* is a collection of object clauses, where each procedure literal has the same predicate symbol and arity, and all object literals (in either a head or a body) have the same predicate symbol and arity. ∎

Descriptions of valid stacks

```
stack(ID,[]).
stack(ID,[X|L]) ← integer(X) ∧ stack(L).
```

A procedure to create a stack

```
new_stack(ID) ← stack(ID,[]).
```

Methods for the class `stack`*:*

```
empty(ID) ∧ stack(ID,[]) ← stack(ID,[]).

push(X,ID) ∧ stack(ID,S) ← integer(X) ∧ stack([X|S]).

pop(X,ID) ∧ stack(ID,[X|S]) ← stack(ID,S).

top(X,ID) ∧ stack(ID,[X|S]) ← stack(ID,[X|S]).
```

Figure 1: A Class for a Stack of Integers

Definition A *class* has three components:

1. A set of *object definitions*, Horn clauses where the head is an object literal, to define the structure of objects of the class.
2. A set of *creation procedures*, Horn clause procedures that contain object literals in the body, used to create instances of objects.
3. A set of methods, used to transform instances of objects. ■

An example of a class that defines a stack of integers is shown in Figure 1. Besides being defined by object clauses instead of Horn clauses, the new definition of a stack includes an extra parameter. The first parameter will be used as the ID of the stack object, and the second as its current state. More will be said about the ID parameter in later sections.

The three components of a class are used at different times in the computation. In general, the initial goal statement will contain the user's top level goal. Some of these goals will lead to calls to the object creation procedures, so that the set of goals for the system to solve will contain some object instances. Next, method calls will delete these instances, and add new instances, as the computation proceeds. Finally, after all procedure calls have been satisfied (the time at which a Prolog computation is finished, since all user goals have been reduced), the current goal statement will contain nothing but object instances. At this time, the system can use the object definition procedures to reduce the object states, until the current goal statement is the empty clause.

In principle, the object definition procedures could be applied at any time, and any valid object instance removed from the current goal statement. However, this would make any method calls in the goal statement unsatisfiable. The control strategy employed in HOOPS is to postpone solution of object literals until no procedure calls are left. Whenever an object literal is encountered in the current goal, it is set aside for later use; it is recalled when the user program makes a call to a method for the object's class.

3 An Inference Rule for Object Clauses

In this section we introduce a new inference rule to handle method calls. The rule allows us to pair up a procedure call with the current state of an object, and derive a new goal statement that consists of remaining goals from the input goal list and the body of the called object clause.

The proof of the soundness of the new inference rule relies on some standard definitions of substitutions and composition of substitutions, plus some existing inference rules. For further discussion of these, see references [10] and [14].

In this section, we will describe clauses as sets of literals, where it is implied that the elements of the set are items in a disjunction of literals. The implication $P \leftarrow Q \wedge R$ will be written $\{P, \overline{Q}, \overline{R}\}$. A literal such as $\overline{P}$ is a *negative* literal, and a literal without the overbar is a *positive* literal. Two literals have the same sign if both are positive or both are negative, and two literals are *complementary* if one is positive and one is negative.

A *substitution* is a set of *assignments* of the form $v = t$, where v is a variable and t is a term. The application of a substitution θ to a clause C, written $C\theta$, is an instance of C where every variable of C that occurs on the left hand side of an assignment in θ is replaced by the corresponding right hand side. $T\theta$, the application of θ to a term T, is defined similarly. The *composition* of two substitutions θ and σ, written $\theta\sigma$, is a new substitution obtained by applying σ to the terms on the right hand sides of the assignments of θ, and adding the assignments of σ that do not name the variables on the left hand sides of θ. A *variable pure* substitution is one where the terms on the right hand sides are all variables; such a substitution simply renames the variables of a clause it is applied to. Composition is associative, and $(C\theta)\sigma = C(\theta\sigma)$.

Factoring is an operation that can be applied to a clause, under certain conditions, to reduce the number of literals in the clause. The factoring rule states that the clause

$$\{P_1, P_2, Q, R, \ldots\}$$

can be reduced to

$$\{P, Q, R, \ldots\}\theta$$

if literals P_1 and P_2 have the same sign, and there is a substitution θ that unifies P_1 and P_2.

Binary resolution is an inference rule that can be applied to two clauses C and D, to yield a new clause R (called the *resolvent*) provided there is a literal L in C, a complementary literal M in D, and L and M are unifiable with substitution θ. The resolvent is formed by taking the union of C and D, minus the unified literals, and applying θ:

$$R = \{\{C\} - L \cup \{D\} - M\}\theta$$

In a logic program, resolution is used in a proof by contradiction. The program is a set of clauses. To execute the program, the user supplies a conjunction of literals to be proved. The system negates the conjunction to get an initial *goal statement*, which is a clause with all negative literals. It then tries to derive a contradiction, in the form of the empty clause $\square$, through a series of resolutions. When all the clauses are Horn clauses (at most one positive literal), binary resolution is complete, *i.e.* if the goal statement leads to a contradiction, binary resolution is the only inference rule that needs to be applied in order to derive the empty clause.

The new inference rule, which we will use to derive a new goal statement when the invoked procedure is defined by an object clause, is called *object clause resolution*, or *OC-resolution* for short. This rule states that from a parent goal statement and an object clause, it is possible to derive a new goal statement. If the parent goal statement G contains a negative procedure literal P_1 and a negative object literal S_1, and the object clause has a head $P_2 \wedge S_2$ and body B, we can infer a new goal statement R that contains the remaining literals from G plus the literals of B, provided there is a substitution θ that simultaneously unifies P_1 with P_2 and S_1 with S_2:

$$\begin{aligned} R &= \{\{G\} - P_1 - S_1 \cup \{B\}\}\theta \\ & \quad \text{if } P_1\theta = P_2\theta \text{ and } S_1\theta = S_2\theta \end{aligned}$$

The soundness of OC-resolution is the subject of the following theorem:

Theorem 1 (Soundness of OC-Resolution) *If S is an unsatisfiable set of well-formed formulae, where each formula in S is either a Horn clause or an object clause, then there is a deduction of $\square$ in which each step of the deduction is either binary resolution or OC-resolution.*

Proof. If S contains only Horn clauses, then, since binary resolution is refutation complete for Horn clauses, the theorem holds. We must prove that when we use OC-resolution in an inference step, the derived goal statement logically follows from the object clause and the parent goal statement. The proof is based on showing that an OC-resolvent is an instance of a clause derived via two binary resolutions plus factoring.

Assume the parent goal statement is

$$\{\overline{P}, \overline{Q}, \overline{F} \ldots \overline{G}\} \tag{2}$$

Assuming the object clause is

$$P \wedge Q \leftarrow R \ldots S \tag{3}$$

the desired resolvent is

$$\{\overline{R} \ldots \overline{S}, \overline{F} \ldots \overline{G}\}\theta \tag{4}$$

As described previously, the object clause is equivalent to the two Horn clauses

$$\{P, \overline{R} \ldots \overline{S}\} \tag{5}$$

$$\{Q, \overline{R} \ldots \overline{S}\} \tag{6}$$

which have identical bodies $\overline{R} \ldots \overline{S}$.

From (2) and (5), derive, with binary resolution, the clause

$$\{\overline{R} \ldots \overline{S}, \overline{Q}, \overline{F} \ldots \overline{G}\}\rho \tag{7}$$

and from (7) and (6) derive

$$\{\overline{R} \ldots \overline{S}, \overline{R} \ldots \overline{S}, \overline{F} \ldots \overline{G}\}\rho\sigma \tag{8}$$

where ρ and σ are the unifiers used in the two steps.

Each literal from the body of the object clause occurs twice in (8). Since these are identical literals (before ρ or σ is applied), each pair can be reduced to a single goal by factoring. The factored resolvent can be written as

$$\{\overline{R}\phi_1 \ldots \overline{S}\phi_n, \overline{F}\phi \ldots \overline{G}\phi\}\rho\sigma \tag{9}$$

where ϕ is the composition of the substitutions $\phi_1 \ldots \phi_n$ used in the factoring steps. The two resolvents in equations (4) and (9) have the same literals, and can be regarded as the same clause C, in the sense that if any instance of C can be used to derive the empty clause, the original goal statement is unsatisfiable. ∎

A few words are in order concerning the substitutions in equations (4) and (9). Since the literals used in the factoring steps were identical copies of one another, the substitutions ϕ_i are variable pure. The ϕ_i rename the variables of (9), with the effect that the bodies of (5) and (6) have variables in common, and an assignment to one of these variables occuring in θ will have an equivalent assignment in $\rho\sigma$.

If a variable occurs in both heads of the object clause, but does not occur in any body literal, θ is more restrictive than $\rho\sigma$. In other words, there can be more instances of the resolvent under substitutions derived through binary resolution and factoring than there are through OC-resolution. This does not affect the soundness of OC-resolution; by Herbrand's theorem, the original goal is unsatisfiable if we derive the empty clause through any consistent instantiation of the resolvent. In terms of the substitutions ϕ, a variable occuring in the head but not the body is not constrained by the factoring step to have the same assignment in both ρ and σ. The requirement that the two heads of the object clause be simultaneously unifiable restricts the possible assignments to this kind of variable, so that in general there are fewer OC-resolvents.

In terms of the intended uses of OC-resolution, for object oriented programming, the restriction is exactly what we want. In practice, variables that occur in both heads are used to identify different instances of the same class of object. An example is in the class for a stack given in Figure 1, where the variable named `ID` is a parameter in both heads of the method clauses. A program could use two stacks, by making two calls to the procedure `new_stack`, using a different term for the ID in each call. By requiring `ID` to have the same value in both heads, we ensure that a call to a method is matched with the proper instance of the object.

In the stack example, and in most methods, the common variable also occurs in the body, in the declaration of the new state of the object. In this case, resolvents (4) and (9) will be the same, since the factoring steps make sure the substitutions for ID are consistent. But consider a method such as the following, that could be used to destroy an instance of a stack:

```
abolish(ID) ∧ stack(ID,S) ←.
```

Since ID does not occur in the body, it will not occur in any substitution ϕ_i, and the factoring steps applied after binary resolution of the equivalent pair of Horn clauses do not guarantee the substitutions for ID are consistent. If there are a number of instances of stacks in the parent goal statement, this is a case where there are more resolvents than OC-resolvents, and any of the instances could be abolished by a call to this method. OC-resolution, by requiring that the substitution for ID be the same in each head, is more restrictive, but still correct.

4 Semantics

The operational semantics of a logic programming language is defined by the proof procedure used in the derivation steps. For Horn clause programs, the meaning of an n-ary predicate p is:

$$\mathbf{D}(p) = \{(t_1, \ldots t_n) : \mathbf{A} \vdash p(t_1, \ldots t_n)\}$$

(van Emden and Kowalski [6]). $\mathbf{D}(p)$, the denotation of predicate p, is the set of n-tuples of terms $t_1 \ldots t_n$ such that $p(t_1 \ldots t_n)$ is provable from program $\mathbf{A}$. In a computation based on resolution, the meaning of $\mathbf{A} \vdash P$ is that there is a derivation of $\Box$ from $\mathbf{A} \wedge \{\overline{P}\}$.

The semantics of predicates used to define object clauses can be defined similarly. The meaning of a procedure predicate m in a method for a class s should reflect its role as a procedure that transforms instances of objects into new instances. $\mathbf{D}(m)$ should include the states of the objects that m is applied to, and not just the tuples of terms that make instances of m provable. One way to do this is to define $\mathbf{D}(m)$ as the set of all tuples of terms $t_1, \ldots t_i$ and $t_{i+1}, \ldots t_n$ such that $m(t_1, \ldots t_i)$ is provable with respect to objects with states $s(t_{i+1}, \ldots t_n)$:

$$\mathbf{D}(m) = \{(t_1, \ldots t_n) : \mathbf{A} \vdash m(t_1, \ldots t_i) \wedge s(t_{i+1}, \ldots t_n)\}$$

In this definition, A $\vdash P \wedge Q$ means there is a derivation of $\Box$ from A$\wedge\{\overline{P},\overline{Q}\}$ using resolution and OC-resolution.

This definition does not quite capture the notion of a method as a procedure that transforms objects. The denotation of a predicate in a Horn clause program is complete, in some sense, since inputs and outputs of functions are contained within a tuple. For example, given suitable definitions of integers and arithmetic operations, the denotation of a predicate *sum* would be a set of 3-tuples $< i, j, k >$ such that $k = i + j$; the two input values and the corresponding output are all represented within a tuple. For object clauses, the denotation relates the arguments of a method call and the input states of the objects that the method can be applied to, but it does not tell us explicitly what the corresponding output states of the object are. The output states are defined, but they are hidden in the answer substitution developed during the proof, and not explicitly part of the denotation.

We can make output states part of the denotation of a method through the following transformation. Replace the vector of arguments $x_1 \ldots x_i$ in the procedure literal of the method with a single structure $A = f(x_1 \ldots x_i)$. Similarly, combine the arguments of object literals in the head and body of an object clause into single arguments S_i and S_o, respectively. Finally, replace every occurrence of $m(A)$ in the program with $m(A, S_i, S_o)$. Then the meaning of a method is a 3-tuple $< A, S_i, S_o >$ defined as follows, depending on whether there are object literals in the head, body, or both:

$$\begin{aligned} \mathbf{D}(m) &= \;< A, S_i, S_o > && \text{for } m \wedge s \leftarrow b \wedge s \\ &= \;< A, S_i, \perp > && \text{for } m \wedge s \leftarrow b \\ &= \;< A, \perp, S_o > && \text{for } m \leftarrow b \wedge s \end{aligned}$$

In the first case, the meaning of m is a 3-tuple relating the vector of arguments of m, the input states of the object s that m is applied to, and the resulting output state. The second case is for object clauses that do not have an object literal in the body; this corresponds to a method that consumes an object instance, but does not create a new instance, effectively destroying the object. In this case, the output state of the object is undefined. In the third case, we have an alternative semantics for a Horn clause that creates objects. In this view, the procedure defines a tuple that maps no objects into a new object S_o.

The semantics of an object literal can be defined in two ways. One is to use the semantics of the Horn clauses in the class that define the structure of the instances of the class. Recall that every class has a set of Horn clauses where the head is an object literal. The usual semantics of these clauses define the meaning of the object predicate. In the stack example, the denotation of the predicate *stack* is the set of all lists of integers, including the empty list.

A more restricted set of tuples of terms might be derived if we define the tuples in the denotation as the union of all objects that can be created by the methods of the class. Informally, initialize the set with the instances in the bodies of the procedures of the class that create instances of objects. Then add any instances that can be created through method calls, where input states are members of the denotation.

5 Examples

This section contains some example programs written in pure logic with object clauses. The examples illustrate the power of combining logic programming, with its clear programs based on pattern matching and nondeterministic search of a problem space, and the object oriented style, where data structures and operations on them are encapsulated into a single coherent structure.

5.1 Expression Evaluator

The first program is a simple expression evaluator, as an illustration of how to use the class `stack` defined previously. The program, shown in Figure 2, consists of a procedure that takes in a postfix expression represented by a list of symbols, evaluates it, and prints the result. Since there is only one stack, this program uses a definition of `stack` that does not have the variable named `ID` in either the object states or the method calls.

The program is started when the user gives an initial call to `evaluate`, such as `evaluate([2,3,+,4,*])`. The procedure for `evaluate` creates an initially empty stack object, and then calls `eval` to carry out the arithmetic operations. Note that after the last procedure call has been solved, the final goal statement will not be the empty clause, but will have a negative object literal representing the final state of the stack. To terminate the execution, the system uses the object definition procedures in class `stack` to show that the final stack is in fact a valid stack, and reduce the goal statement to the empty clause.

A Prolog version of the expression evaluator with roughly the same structure as the object oriented program is also shown in the figure. There are two things to notice about the object oriented version, when comparing it

Object oriented version:

```
evaluate(Str) ← new_stack ∧ eval(Str).

eval([]) ← top(X) ∧ write(X).
eval([X1|Xn]) ← integer(X1) ∧ push(X1) ∧ eval(Xn).
eval([X1|Xn]) ← operator(X1) ∧ apply(X1) ∧ eval(Xn).

apply(+) ← pop(N1) ∧ pop(N2) ∧ N is N1 + N2 ∧ push(N).
apply(-) ← pop(N1) ∧ pop(N2) ∧ N is N2 - N1 ∧ push(N).
```
etc.

Prolog version:

```
evaluate(Str) :- eval(Str,[]).

eval([],[N]) :- write(N), nl.
eval([X1|Xn],S) :- integer(X1), eval(Xn,[X1|S]).
eval([X1|Xn],Si) :- operator(X1), apply(X1,Si,So), eval(Xn,So).

apply(+,[N1,N2|Rem],[N|Rem]) :- N is N1+N2.
apply(-,[N2,N1|Rem],[N|Rem]) :- N is N1-N2.
```
etc.

Figure 2: Expression Evaluator, Version 1

to the Prolog program. One is that the stack is not passed as a parameter on calls to `eval` and `apply`, but rather exists as a separate literal in the goal statement. In a Prolog version, extra parameters are required, to hold the current state of the stack.

The second point is that the object oriented version is sensitive to the order in which methods are called. The two calls to pop in the body of `eval` return different numbers, and, in the case of noncommutative operations like subtraction, it is important which number is popped first from the stack. In the Prolog version, the numbers are named by their location in the stack, and it is apparent from the structure of the argument in the head of the clause which is which. Perhaps a better way to define this class, when the order of operations is important, is to write a method called pop2 that returns the top two elements on the stack.

A second evaluator, defined for infix expressions and using two stacks, is given in Figure 3. This example shows how a program can make two object

```
evaluate(Str) ←
      new_stack(ops) ∧ new_stack(vals) ∧
      push(@,ops) ∧ eval(Str).

eval([]) ← check(@) ∧ top(X,vals) ∧ write(X).
eval([X1|Xn]) ← integer(X1) ∧ push(X1,vals) ∧ eval(Xn).
eval([X1|Xn]) ← operator(X1) ∧ check(X1) ∧ eval(Xn).

check(Op) ← top(X,ops) ∧ maybe_apply(Op,X).

maybe_apply(Op,X) ←
      gtr_precedence(Op,X) ∧ push(Op,ops).
maybe_apply(Op,X) ←
      leq_precedence(Op,X) ∧ pop(X,ops) ∧
      apply(X) ∧ check(Op).

apply(+) ← pop(N1,vals) ∧ pop(N2,vals) ∧
      N is N2 + N1 ∧ push(N,vals).
apply(-) ← pop(N1,vals) ∧ pop(N2,vals) ∧
      N is N2 - N1 ∧ push(N,vals).
```
etc.

Figure 3: Expression Evaluator, Version 2

instances from one class definition, and uses the version of the class **stack** that was given in Figure 1.

5.2 N Queens

The second example is a solution to the *N Queens* problem. The solution presented here is derived from a constraint based program described by van Hentenryck and Dincbas [7]. In this approach, the basic idea is to assign each of n queens to different columns at the start of the program. Van Hentenryck and Dincbas use the notion of a finite *domain* to represent the possible rows a queen can be placed in. The domain of each queen is initially the set $1..n$. When a queen is assigned to a specific row, that row must be in the queen's domain, or the assignment fails. When a queen is assigned a row, the domains of other queens can be reduced; for example, when queen i is assigned to row j, j can be removed from the domain of every other queen, since no two queens can occupy the same row. A similar restriction can reduce the domains of the other queens so they will not occupy a square on the diagonal from (i, j). What makes this solution interesting is that if the placement of a queen reduces the domain for another queen to a single row, the other queen can immediately be assigned to that row, and the constraints implied by this placement can be propagated. In the 5 queens problem, only two nondeterministic choices are required; constraint propagation leads to a deterministic placement of the remaining three queens, and the problem is solved without backtracking.

In our object oriented solution, we will use a class named `queen` to describe each object (Figure 4). The state variables of a queen will be the ID of the queen (which is the same as the column the queen is assigned to) and the queen's current domain. If the domain is a list of integers, the list represents the possible rows for the queen. If the domain is a single integer i, it means the queen has been assigned to row i.

The methods of the class are `choose` and `restrict`. The call `choose(Q)` directs queen q to nondeterministically choose a row from among the possibilities remaining in the domain. The call `restrict(Q,R)` tells queen q to stay out of row r. If q has already been placed in r, the call will fail. Whenever a queen is placed in a row (either directly, via a call to `select`, or indirectly, when `restrict` leaves it with only one place to go), it then calls `restrict` for the other queens. In the parlance of object oriented programming, it sends a message to the other queens, telling them to stay out of certain rows.

A version of this program was implemented in HOOPS, a Prolog meta-interpreter that can handle object states and method calls [4]. The top level structure of the program (not shown in the figure) is a simple tail-recursive loop that makes the n queens with calls to `new_queen`, and then calls `choose` for each of the queens. In the first few iterations, `choose` succeeds by nondeterministically choosing a row. Later calls will find the queens already placed, via restrictions propagated by placement of earlier queens. The program performed comparably to the program of van Hentenryck and Dincbas, in terms of the number of nondeterministic choices that later had to be backtracked.

```
queen([R1|Rn]).
queen(R) ← integer(R).

new_queen(N) ← init_domain(D) ∧ queen(N,D).
```

Method for `choose(Q)`*: if queen* `Q` *is already in a row, ignore call; otherwise select a row from the domain and broadcast constraints to others.*

```
choose(Q) ∧ queen(Q,N) ←
      integer(N) ∧ queen(Q,N).
choose(Q) ∧ queen(Q,D) ←
      list(D) ∧ member(Row,D) ∧
      propagate(Q,Row) ∧ queen(Q,Row).
```

Method for `restrict(Q,R)`*: if* `Q` *is already in row* `R`*, fail; if* `Q` *is in another row, ignore the call; otherwise remove* `R` *from domain. If removing* `R` *leaves* `Q` *with only one place to go, take that place and broadcast constraints.*

```
restrict(Q,R) ∧ queen(Q,X) ←
      integer(X) ∧ neq(X,R) ∧ queen(Q,X).
restrict(Q,R) ∧ queen(Q,D) ←
      list(D) ∧ select(R,D,Dr) ∧
      next_domain(Q,Dr,D2) ∧ queen(Q,D2).
restrict(Q,R) ∧ queen(Q,D) ←
      list(D) ∧ nonmember(R,D) ∧ queen(Q,D).
```

Local procedures

```
next_domain(Q,[R],R) ← propagate(Q,R).
next_domain(Q,[R1,R2|Rn],[R1,R2|Rn]).

propagate(Q,R) ←
      check(Q,R,1,-1) ∧ check(Q,R,1,0) ∧ check(Q,R,1,1) ∧
      check(Q,R,-1,1) ∧ check(Q,R,-1,0) ∧ check(Q,R,-1,-1).

check(1,_,-1,_).
check(_,1,_,-1).
check(N,_,1,_) ← nq(N).
check(_,N,_,1) ← nq(N).
check(Q,R,H,V) ←
      sum(Q,H,Q2) ∧ sum(R,V,R2) ∧
      restrict(Q2,R2) ∧ check(Q2,R2,H,V).
```

Figure 4: Class queen for N-Queens Problem

6 Discussion

The approach to object oriented programming presented in this paper is based on a new procedural reading of a goal statement, which is a set of negative literals. In the standard reading, all literals are interpreted as procedure calls. In the new reading, some literals are taken to be instances of objects. A method is a procedure that can only be called when it can be paired up with an instance of an object. In a call to a method, two negative literals, one for the procedure call and the other for the current state of the object, are removed from the current goal statement, and the body of the selected object clause is added to the remaining goals to make the new goal statement. When one of the body goals is an object literal, we interpret this as the new state of the object.

The notion that an object is consumed by the method call, and a new object created by successful execution of the body, is similar to view of complex structures in dataflow languages. Conceptually, a structure is carried as the value of a token. Each time a function is applied, it consumes its input tokens, and generates new tokens for the results. This view may lead to some awkward situations when we want the method to make a recursive call to itself, or when we want it to call another method for the same object. If the object is consumed by the call to a method, and the new instance created when the method succeeds, how do we handle recursive calls or messages to "self"?

From the logical point of view, the new instance of the object is created as soon as the unification with the heads of the object clause is complete. The new instance is defined by a negative literal in the body of the method, and this literal is immediately part of the inferred goal statement. Thus there is an instance of the object available for recursive calls. All that is required is that we be careful to balance the number of method calls with the number of new object creations. In practice, however, the body of the method is expected to operate on existing values of state variables in order to create the values of the new instance. What is found in the new instance are the names of the new values, which are unbound variables until procedure calls in the body bind them. What we have to be careful for is that recursive calls do not consume objects that are not yet "filled in" with values to be computed by other calls. The HOOPS language used to implement the examples in this paper has rules for defining the relative order in which objects are created and procedures in the body are called.

One of the aspects of using this scheme for objects is that object states created during execution of nondeterministic methods are erased on backtracking, and alternative states are created in their place. For example, if a call to `choose` in the eight queens example results in a queen being placed in a row that cannot lead to a solution, the instance of the object that shows the queen in the wrong row is destroyed on backtracking, and the instance created by the new choice takes its place. There are times when it would be useful to have objects keep their values through backtracking. For example, suppose we wish to instrument the N queens program, and count the number of times a queen has to be moved. If we simply add a counter object to the program, and call a method to increment it every time we place a queen, the calls that increment the counter would be backtracked whenever the queen backtracks. What is needed is a metalogical object, outside of the proof that there exists a configuration of queens that satisfies the constraints, that is not affected by operations in the proof. From the programmer's standpoint, we either need to have a notation that tells the system not to backtrack certain objects, or a metacall facility that allows us to create metaobjects such as the counter, and then call parts of the proof of eight queens as a subroutine.

A major aspect of object oriented programming that has not been dealt with in this paper is the notion of class hierarchies, where a class can be declared to be a subclass of another class. All objects that are instances of the subclass automatically inherit the slot variables and methods of the superclass. For example, a queen might be declared to be a subclass of the class of chesspieces; if there is a method for moving or placing a chesspiece, we would not have to explicitly write such a method for the queen; all we have to do is write the rules that are specific for the queen and distinguish it from other chesspieces.

The formalism presented here does not make provisions for this type of inheritance, but neither does it rule it out. Inheritance can be programmed, such that when an instance of a subclass is created, new instances of each of its superclasses are also created. To continue the example, the procedure to make a queen would be written in such a way that it would place an instance of a queen literal in the inferred goal statement, and also call the procedure to make a chesspiece. The slot variables of the queen would consist of the arguments of the queen literal, plus the arguments of the new chesspiece literal. If we can keep the object IDs separate, so that no two objects of any class have the same IDs, calls to superclass methods can be handled as efficiently as any method calls. A call such as `move(Q,R,C)`, where `Q` is the ID of the queen, will immediately be matched with the literal that defines the queen as a chesspiece. As is the case for other object oriented languages, rules will have to be developed for deciding which method, or combination of methods, to apply when there are name conflicts.

Finally, the opportunities for parallelism in this scheme are, in the abstract, the same as for Horn clause programs without objects [5]. *OR parallelism* is parallelism in exploring alternative paths in the sequence of resolutions. In this model, it would lead to "parallel universes" of objects, where objects in one universe would not interact with objects in another universe. When the system reaches a choice point, either in a procedure call or within a nondeterministic method, each new alternative path would have its own independent copy of all objects that exist at the time.

AND parallelism is obtained when multiple goals along a single path are reduced simultaneously. It is this form of parallelism that would correspond to the usual notion of parallelism in an object oriented model (for example, in the Actors model [8]). Objects are viewed as independent threads of computation, and there is no reason why messages being sent to different objects cannot be processed simultaneously by the objects. One of the difficulties in implementing AND parallelism in logic programs is deciding which goals can be solved in parallel, and which have interactions that force some to be solved before others. An interesting research project would be to see if the coordination implied by method calls could be the basis for scheduling AND-parallel tasks.

References

[1] Backus, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.

[2] Chikayama, T. ESP – Extended Self-contained Prolog – as a preliminary kernel language of fifth generation computers. *New Generation Computing 1*, (1983), 11–24.

[3] Clark, K.L. and Gregory, S. Notes on system programming in Parlog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan, Nov. 6–9), 1984, pp. 299–306.

[4] Conery, J.S. *HOOPS: An Object Oriented Prolog*. Tech. Rep., Univ. of Oregon, 1987. In preparation.

[5] Conery, J.S. and Kibler, D.F. Parallel interpretation of logic programs. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, (Wentworth-by-the-Sea, NH, Oct. 18–22), ACM, 1981, pp. 163–170.

[6] van Emden, M.H. and Kowalski, R.A. The semantics of predicate logic as a programming language. *J. ACM 23*, 4 (Oct. 1976), 773–742.

[7] van Hentenryck, P. and Dincbas, M. Forward checking in logic programming. In *Proceedings of the Fourth International Conference on Logic Programming*, (Melbourne, Australia, May 25–29), 1987, pp. 229–256.

[8] Hewitt, C.E., Attardi, G., and Lieberman, H. Specifying and proving properties of guardians for distributed systems. In Kahn, G., editor, *Semantics of Concurrent Computation*, Springer-Verlag, New York, NY, 1979, pp. 316–336.

[9] Kahn, K., Tribble, E.D., Miller, M.S., and Bobrow, D.G. Objects in concurrent logic programming languages. In *OOPSLA '86 Proceedings*, (Portland, OR, Sep.), 1986, pp. 242–256.

[10] Lloyd, J.W. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[11] Pereira, L.M., Pereira, F.C.N., and Warren, D.H.D. *Users Guide to DECsystem-10 Prolog.* Tech. Rep., Dept. of Artificial Intelligence, Univ. of Edinburgh, Sep. 1978.

[12] Shapiro, E. and Takeuchi, A. Object oriented programming in Concurrent Prolog. *New Generation Computing 1*, (1983), 25–48.

[13] Ueda, K. *Guarded Horn Clauses.* Tech. Rep. TR-103, Institute for New Generation Computer Technology, Tokyo, Japan, June 1985.

[14] Wos, L., Overbeek, R., Lusk, E., and Boyle, J. *Automated Reasoning.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.