Representing Objects in a Logic Programming Language with Scoping Constructs

Joshua S. Hodas and Dale Miller
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104 – 6839, USA
hodas@eniac.seas.upenn.edu dale@linc.cis.upenn.edu

Abstract

We present a logic programming language that uses implications and universal quantifiers in goals and the bodies of clauses to provide a simple scoping mechanism for program clauses and constants. Within this language it is possible to define a simple notion of parametric module and local constant. Given this ability to structure programs, we explore how object-oriented programming, where objects are viewed as abstractions with behaviors, state, and inheritance, might be accommodated. To capture the notion of mutable state, we depart from the pure logic setting by adding a declaration that certain local predicates are deterministic (they succeed at most once). This declaration, along with a goal-continuation passing style of programming is adequate to model the state of objects. We also examine a few aspects of how having objects embedded in logic programming can be used to enrich the notion of object: for examples, objects may be partial (that is, may contain free variables) and non-deterministic, and it is possible not only to search for objects with certain properties but also to do hypothetical reasoning with them.

1 Introduction

Many attempts have been made in recent years to extend logic programming with features found in object-oriented programming languages [2,3,5,6,12,9]. Much of this work has used Prolog and Horn clause as a foundation. In this paper, we start with an enrichment of Horn clause logic that contains a natural scoping mechanism and then show how aspects of object-oriented programming can be represented. Our account of state and state updates is the only place where we need to resort to a non-logical primitive. That primitive, however, is a familar one: certain predicates will be declared to be deterministic; that is, if they succeed, only their first solution is returned and the remaining ones are discarded.

2 A Logic for Scoping Clauses and Constants

Various extensions to the foundation of logic programming have been proposed to provide scoping constructs for program clauses and constants. We shall base our language on a logic similar to N-Prolog [8], the intuitionistic clausal system of [4,13,14], and the hereditary Harrop formulas of [15,16,17]. Since a simple modification of the latter logic is the logic we consider here, we refer to it simply as hH'. We briefly describe hH' from an operational point-of-view below. The reader interested in proof theoretic semantics should refer to the papers mentioned above.

Positive Horn clauses are universally quantified formulas of the form

$$A \texttt{ :- } B_1, \ldots, B_n \quad (n \geq 0),$$

where $A, B_1, \ldots, B_n$ are atoms. In order to provide for scoping of program clauses during execution, we shall need to introduce the notion of a *current program*, that is, a stack of program clauses, and a *current signature*, that is, a stack of constants. Implications in the body of clauses and in goals specify clauses that should be added to the current program. In particular, the atomic formulas B_i can be replaced with more complex formulas of the form

$$H_1 \wedge \ldots \wedge H_m \texttt{ => } B_i \quad (m \geq 0),$$

where $H_1, \ldots, H_m$ are Horn clauses and `=>` is the converse of `:-`. To prove such a goal the clauses $H_1, \ldots, H_m$ are first loaded into the current program and only then is the atom B_i is attempted. After B_i succeeds or fails, these clauses are discharged (removed from the current program). If we consider Horn clauses to be of *order* 0, then clauses with at least one such implicational formula in its body would be of order 1. Clauses of order 2 would then result from permitting the clauses $H_1, \ldots, H_m$ to be of order 0 or 1. The logic hH' contains clauses of all orders. We shall also include formulas where the consequence of the implication `=>` is a conjunction, in which case the clauses in the antecedent of `=>` are scoped over all conjuncts. The antecedent of `=>` may not be a disjunction.

Providing scope to individual, function, and predicate constants can be accommodated by using universal quantifiers over goal formulas. Universal quantifiers will be written as `all` x_1`,`$\ldots$`,`x_n`\`, for $n > 0$. In order to prove the universally quantified goal `all x\ G(x)`, some "new" constant (that is, a constant not in the current signature), say `c`, is added to the current signature and is used to instantiate this goal. The resulting goal, `G(c)`, is attempted. After this instance succeeds or fails, the constant `c` is discharged (removed from the current signature). Thus the interpretation of implications and universal quantification in goals is similar: the first provides scope to clauses and the second to constants.

In implementations of this logic that use free (logic) variables and unification, these extensions provide some complications not found in implementations of Prolog. First, it is possible for the current program to contain free variables. This can happen, for example, when trying to prove the formula

$$H_1 \wedge \ldots \wedge H_m \;\texttt{=>}\; B_i,$$

if some H_i contains a variable free. (In this language, quantifiers must often be explicitly written. For instance, variables not explicitly quantified in the clauses H_i will be assumed to be bound around the outermost clause in which they are embedded.) An example of this will be seen in the latter part of this paper. To correctly enforce scoping of constants, unification must be modified so that a scoped constant does not escape its intended scope. This restriction amounts to requiring that when a new constant is introduced, all free variables in the current goal and program be marked so that no instantiation of them will be to terms that contain that constant. The paper [15] presents a proof procedure that is sound and (non-deterministically) complete for the intuitionistic theory of hH'.

On top of hH' we wish to add a syntax for modules similar to the one presented in [16]. Thus, certain names, called *module names*, will be used to denote (possibly parametric) collections of program clauses. For example, the module declaration

$$
\begin{array}{l}
\texttt{MODULE mod(}x_1, \ldots, x_n\texttt{).} \\
\texttt{LOCAL } y_1, \ldots, y_m\texttt{.} \\
H_1(x_1, \ldots, x_n, y_1, \ldots, y_m) \\
\qquad \vdots \qquad\quad \vdots \\
H_p(x_1, \ldots, x_n, y_1, \ldots, y_m)
\end{array}
$$

associates to `mod` the parameters $x_1 \ldots, x_n$, the local constants $y_1, \ldots, y_m$, and the clauses (of hH') $H_1, \ldots, H_p$, which may contain free occurrences of the variables $x_1, \ldots, x_n$ and constants $y_1, \ldots, y_m$. Modules are used within goal formulas using the syntax $\texttt{mod(}t_1, \ldots, t_n\texttt{)} \;\texttt{==>}\; B$. This syntax is considered only as short-hand for the formula

$$
\begin{array}{l}
\texttt{all } y_1, \ldots, y_m \backslash \\
\qquad [(H_1(t_1, \ldots, t_n, y_1, \ldots, y_m), \ldots, H_p(t_1, \ldots, t_n, y_1, \ldots, y_m)) \;\texttt{=>}\; B].
\end{array}
$$

Here, we overload the symbols $y_1, \ldots, y_m$ to be constants in the `LOCAL` declaration and bound variables in the displayed formula above. In general, this overloading should not cause problems. Also, in this example, it is assumed that the formula B and the terms $t_1, \ldots, t_n$ do not contain occurrences of $y_1, \ldots, y_m$. We may always assume this since the names of bound variables (and local constants) can be changed as needed.

An important programming style that we shall use in this paper can be referred to as *goal-continuation passing*. We use this term whenever a predicate takes as an argument a goal to be called after augmenting the current program or signature. Thus, the goal will be carried from one environment to another. For example, consider the clause

```
p(X,G) :- all y\ (assoc(y,X) => G).
```

where `assoc` is some binary predicate symbol.

To prove the goal `p(a,G)` requires adding a new constant, say `c`, and a new clause, `assoc(c,a)` and then calling the goal `G`. In a sense, `G` was carried from one context to this augmented context.

For the purposes of making examples easier to present, we shall assume that there is a special goal `top` that is the top-level of our interpreter; an attempt to prove `top` causes a prompt to appear, a goal to be read from the keyboard, that goal to be attempted, and answer substitutions to be printed. Those steps are repeated until the special goal `pop` is seen, in which case, the most recent call to `top` succeeds. If `mod0` is a module, then entering the goal `mod0 ==> top` at the top-level causes the next prompt to be issued from a top-level that has access to the clauses and local constants contained in `mod0`. Similarly, attempting to prove the goal `p(a,top)`, where `p` is defined as above, will result in invoking a new top-level in which a new constant, say `c`, and the new clause, `assoc(c,a)`, are available.

Many of the features hH' are available in the logic programming language λProlog language [19]. The examples in this paper have been developed and tested using the eLP implementation of λProlog [7].

3 Modules cum Objects

Several recent papers have suggested that, in the setting of positive Horn clauses, the proper view of objects is not as a pairing of state information with a set of behaviors, but rather simply as a set of behaviors. For example, in [12] objects are identified with parameterized modules of Prolog clauses and a goal that is used to "send a message" to an object is proved within the module corresponding to that object. hH' supports a similar notion of object-sans-state by identifying them as abstract data types as outlined in [15]. For example, a module for an object representing a locomotive (based on one in [12, pages 47-54]) could be given by:

```
MODULE locomotive.
LOCAL train.

make_train(train(S,Cl,Co),S,Cl,Co).
color(train(S,Cl,Co),Cl).
speed(train(S,Cl,Co),S).
country(train(S,Cl,Co),Co).
journey_time(Train,Distance,Time) :-
       speed(Train,S), Time is Distance div S.
```

A query about a particular train, such as how long one that travels 30 miles per hour will take to go 100 miles, would then be posed as:

```
?- locomotive ==> top.
?- make_train(Tr,30,blue,usa),journey_time(Tr,100,Time).
Tr == train1(30,blue,usa)
Time == 3.333333.
```

Here the function constant `train1` is just the new constant created for the local constructor: it will be different each time the module is loaded.

A point that should be made about this example is that a locomotive could just as well be built via its selectors. A French train that travels 125 miles per hour could be built, and a trip time computed, by the following query:

```
?- locomotive ==> top.
?- country(L,france),speed(L,125),journey_time(L,250,Time).
L == train2(125,Cl,france)
Time == 2.
```

This example also demonstrates the possibility of having partially described objects, where certain descriptors are left uninstantiated. This is a natural outgrowth of the Prolog proof procedure and will be useful in the dicussion of hypothetical object queries in Section 6.2.

The scoping rules of hH' are quite strict. Any query that would result in a locally defined constant being brought out of its scope will fail. By beginning these examples with `locomotive ==> top`, the subsequent query in each example is brought into the scope of the locally defined constructor. If the last example is instead posed as

```
?- locomotive ==> (country(L,france), journey_time(L,250,2),
                   speed(L,Speed)).
no.
```

it fails, since the scope of the imported module (and hence its local constants) ends at the end of the second line. For this goal to succeed it must be posed with an explicit existential quantifier.

```
?- locomotive ==> exists L\ (
                    country(L,france), journey_time(L,250,2),
                    speed(L,Speed)).
Speed == 125.
```

Thus, if we exclude the use of `top`, the data type is truly abstract, accessible only through its constructors and destructors.

4 Introducing State

The technique of goal-continuation passing can be used to introduce a notion of state. An obvious method for implementing mutable state in Prolog is to augment the database with predicates corresponding to the objects. An ad-hoc way of accomplishing this in a traditional Prolog is through the use of the extralogical `assert`, as in:

```
?- assert(object(state)).
?- goal.
```

In hH', the proof of an implicational goal is used with a continuation on the right hand side to achieve a similar result, as in:

```
?- object(state) => goal.
```

A very simple program for maintaining a database of switch values using this technique is given by:

```
MODULE switch(Name).
LOCAL register.

setting(Name,Setting) :- register(S), S = Setting.
set_on(Name,Goal) :- register(on) => Goal.
set_off(Name,Goal) :- register(off) => Goal.
```

Class definition modules are generally accompanied by a clause for a predicate used to create objects of the class, such as `make_switch(Name,Goal) :- switch(Name) ==> Goal`. When this module is loaded, as when proving `make_switch(sw1,top)`, a new constant is created for the `LOCAL` predicate `register`. The 3 clauses from the module are then loaded into the current program, instantiated with the name of the switch, `sw1`, and the new local constant. In this way a correspondence is set up between the object's name and the predicates used to represent its storage. That storage can be accessed only through the methods provided.

This module could be used as follows:

```
?- make_switch(sw1,top).
?- set_on(sw1,top).
?- set_off(sw1,top).
?- setting(sw1,S).
S == off.
yes.
```

There is a problem with this example that is best demonstrated by continuing the last query where it left off.

```
?- setting(sw1,S).
S == off;
S == on;
no
```

When an implication is to be proved, the term on the left of the implication is added to the current program, and an attempt is made to prove the term on the right. This means that as implicational goals are nested (as is the effect here), clauses are added to the program, but none are discharged. Thus in proving the goal `set_off(sw1,top)` a new state value for `sw1` was added, but the old state also remained. While this can be seen as a feature and not a bug — there have been proposals suggesting that objects with their full state history available might be useful in certain database applications [2] — for most applications one would expect an object to have only a single state. To this end we have introduced a new special declaration `STATE` which has the same basic meaning as `LOCAL` (that is, it can be used to create new scoped predicate constants), but these predicates will now be deterministic. When a predicate constant is created by a `STATE` declaration, any attempts to prove goals involving that predicate will succeed at most once. The difference between these two declarations can be seen in the following example:

```
MODULE example1.
LOCAL R.
set(Val,Goal) :- R(Val) => Goal.
get(Val) :- R(U), U = Val.

?- example1 ==> set(1,set(2,top)).
?- get(X).
X == 2;
X == 1;
no
?-
```

```
MODULE example2.
STATE R.
set(Val,Goal) :- R(Val) => Goal.
get(Val) :- R(U), U = Val.

?- example2 ==> set(1,set(2,top)).
?- get(X).
X == 2;
no
?-
```

Using `example2`, `get(t)` succeeds if and only if `t` unifies with the most recently `set` value; old values are not accessible within the call to top. The definition of `STATE` relies on several details of the actual proof procedure used. In particular, it assumes that the clauses added to the database in an implicational goal are added at the top of the database and that clause selection is top-down and depth-first. In this regard it is certainly extralogical, though it is related to the `deterministic` and `once` declarations in various logic programming languages. There is some evidence that this, or a similar construct, has a reasonable semantics in linear logic [3]. In any case, it is less problematic than general `assert/retract`. In particular, the nested goals are fully backtrackable, a feature that will be illustrated in Section 6.

5 Inheritance

An important feature of object oriented systems is inheritance, the ability to describe the features of a new class of objects in terms of a existing classes. The *subclass* being defined *inherits* all of the attributes (state information and methods) of the *superclasses* in terms of which it is being defined. It is easy to extend the style of programming demonstrated in the last section

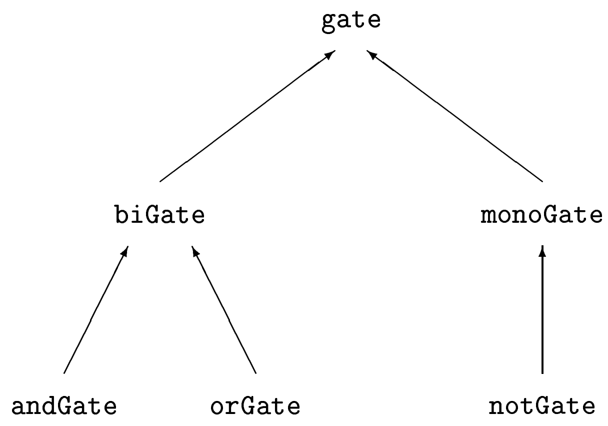

Figure 1: The class hierarchy for logic gates

to represent a class structure with multiple inheritance (though only single inheritance will be shown here).

To demonstrate this idea we will consider modelling simple digital logic circuits. In this system there are two sorts of objects: wires and gates. Wires are represented by just one class while six classes are used to represent the logic gates. The six gate classes are organized into a hierarchy as shown in Figure 1. This example is inspired by the circuit simulator used in [1, pages 219–230] but differs in that here wires and gates are objects while in [1] only wires are objects.

The root class `gate` is defined by the parametric module below and represents the most abstract notion of a logic gate. It specifies only the features common to all logic gates.

```
MODULE gate(Gate,OutputWire).
STATE reg.

reg(OutputWire,off).

class(Gate,gate).
output(Gate,Wire) :- reg(W,_), Wire = W.
state(Gate,State) :- reg(_,S), State = S.
setOutput(Gate,Out,G) :-  output(Gate,OldW), state(Gate,State),
                          reg(Out,State) =>
                              setSignal(OldW,off,setSignal(Out,State,G)).
alert(Gate,G) :- computeState(Gate,New), output(Gate,Wire),
                 reg(Wire,New) => setSignal(Wire,New,G)).
```

In order to create a gate, we use the predicate `make_gate`, defined by the clause

```
make_gate(Gate,OutWire,G) :- gate(Gate,OutWire) ==> G.
```

Here, `Gate` is the name of the gate being created and `OutWire` is the name of the wire connected to the gate's output channel. All the `make`-clauses for a program would generally be gathered into a single module. See Figure 2 for an example.

The class definition for gates specifies two instance variables whose storage is represented by the predicate `reg`. The first position is used for the name of the `wire` object that is connected to the output of the gate. The second stores the current output value of the gate. Selectors are provided to check the value of both of these variables but only the name of the output wire can be changed directly. The gate can be sent an `alert` message that directs it to recompute its output value. The details of this computation are not, however, specified within the module. The `gate` class is, therefore, what Smalltalk-80 programmers refer to as an *abstract* superclass. Such a class is not intended to have instances created, since such instances would lack crucial functionality. Rather they are intended to represent the common aspects of two classes, neither of which properly contains the other [11].

The abstract class `biGate` given by:

```
MODULE biGate(Gate,InputA,InputB)
STATE reg.

reg(InputA,InputB).
class(Gate,biGate).
inputA(Gate,Wire) :- reg(W,_), Wire = W.
inputB(Gate,Wire) :- reg(_,W), Wire = W.
setInputA(Gate,InpA,G) :- inputA(Gate,OldA), inputB(Gate,InpB),
     reg(InpA,InpB) => addGate(InpA,Gate,remGate(OldA,Gate,alert(Gate,G))).
setInputB(Gate,InpB,G) :- inputA(Gate,InpA), inputB(Gate,OldB),
     reg(InpA,InpB) => addGate(InpB,Gate,remGate(OldB,Gate,alert(Gate,G))).
```

is used to describe those attributes common to all gates that have two input wires. Wire objects maintain a list of the names of gates to which they carry input. This list is maintained by the `addGate` and `remGate` methods defined for the wire class (see Figure 2). Therefore, when a `biGate` (or `monoGate`) is created with

```
make_biGate(Gate,InpA,InpB,Out,G) :-
   addGate(InpA,Gate, addGate(InpB,Gate,
        make_gate(Gate,Out,(biGate(Gate,InpA,InpB) ==> G)))).
```

(which specifies its name and the names of its input and output wires) it begins by informing the two input wires that they should add the gate being created to their respective lists. It then creates a set of methods for this gate that will include the methods from the generic `gate` description, as well as those for a `biGate`. This is accomplished by calling `make_gate`, and passing it, as a continuation, a goal that includes loading the `biGate` specification instantiated to the same gate name.

The `biGate` module specifies that a two input gate features a second binary storage predicate (though it uses the same local name, `reg`, it will be instantiated to a different constant than the one used to store the gate's output wire and state) which stores the names of the wires carrying input to the gate. To insure that the circuit remains consistent, the methods for changing the inputs of the gate send messages to the old and new input wires telling them to update their respective gate lists.

Finally, the concrete class `andGate` is given by:

```
MODULE andGate(Gate)

class(Gate,andGate).
computeState(Gate,on) :- inputA(Gate,WireA), signal(WireA,on),
                         inputB(Gate,WireB), signal(WireB,on).
computeState(Gate,off) :- inputA(Gate,WireA), signal(WireA,off).
computeState(Gate,off) :- inputB(Gate,WireB), signal(WireB,off).
```

A member of this class is built using the predicate defined with the clause:

```
make_andGate(Gate,InpA,InpB,Out,G) :-
     make_biGate(Gate,InpA,InpB,Out,(andGate(Gate) ==> alert(Gate,G))).
```

which specifies that an and-gate is just a two input gate together with the knowledge of how to compute its output value. When an `andGate` is created it is immediately told to compute its initial state based on the signals being carried on its input wires. The same sort initialization is used in the creation of an `orGate` or a `notGate`. The remaining class definitions for the circuit simulator example are given in Figure 2. One predicate worth examining is the `setSignal` method for the class `wire`. The `notify` predicate is entirely local to `setSignal`. Since, `G` (the eventual continuation) is free in it, `notify` can be written with only a single argument.

6 Logic Programming as an Enhancement to Objects

A motivating factor behind our research has been a desire to not only enhance logic programming by adding object-oriented extensions but also enhance objects with logic programming features. We describe two such enhancements to object-oriented programming that arise from this approach to embedding it into logic programming.

6.1 Searching Over Objects

In traditional object-oriented languages it is possible to query an individual object about its current state, but there is generally no way to search through the entire space of objects for those objects which satisfy a given constraint. (Such a facility could be programmed in Smalltalk-80 as a method for the system dictionary object, but this sort of direct global manipulation of the object space is considered bad style.) It is possible, in this system, to use

```
MODULE orGate(Gate)
class(Gate,orGate).
computeState(Gate,on) :- InputA(Gate,WireA), signal(WireA,on).
computeState(Gate,on) :- InputB(Gate,WireB), signal(WireB,on).
computeState(Gate,off) :- InputA(Gate,WireA), signal(WireA,off),
                          InputB(Gate,WireB), signal(WireB,off).


MODULE uniGate(Gate,InputWire)
STATE reg.
reg(InputWire).
class(Gate,uniGate).
input(Gate,Wire) :-  reg(W), Wire = W.
setInput(Gate,Inp,G) :- input(Gate,OldW),
      reg(Input) => addGate(Inp,Gate,remGate(OldW,Gate,alert(Gate,G))).


MODULE notGate(Gate)
class(Gate,notGate).
computeState(Gate,off) :- Input(Gate,Wire), signal(WireA,on).
computeState(Gate,on).


MODULE wire(Wire).
STATE reg.
reg(off,[]).
class(Wire,wire).
signal(Wire,Signal) :- reg(S,_), S = Signal.
setSignal(Wire,Sig,G) :-
   all notify\  (notify([]) :- G,
       all H,T\ (notify([H|T]) :- alert(H,notify(T))))
                     => (Reg(_,Outs),reg(Sig,Outs) => notify(Outs))).
addGate(Wire,Gate,G) :- reg(Sig,Outs), reg(Sig,[Gate|Outs]) => G.
remGate(Wire,Gate,G) :- reg(Sig,Outs), delete(Gate,Outs,NewOuts),
                        reg(Sig,NewOuts) => G.


MODULE circuit_simulator.
make_wire(Wire,G) :- wire(Wire) ==> G.
make_gate(Gate,OutWire,G) :- gate(Gate,OutWire) ==> G.
make_biGate(Gate,InpA,InpB,Out,G) :-
      addGate(InpA,Gate, addGate(InpB,Gate,
                  make_gate(Gate,Out,(biGate(Gate,InpA,InpB) ==> G)))).
make_andGate(Gate,InpA,InpB,Out,G) :-
      make_biGate(Gate,InpA,InpB,Out,(andGate(Gate) ==> alert(Gate,G))).
make_orGate(Gate,InpA,InpB,Out,G) :-
      make_biGate(Gate,InpA,InpB,Out,(orGate(Gate) ==> alert(Gate,G))).
make_uniGate(Gate,Inp,Out,G) :-
      make_gate(Gate,Out,(uniGate(Gate,Inp) ==> G)).
make_notGate(Gate,Inp,Out,G) :-
      make_uniGate(Gate,Inp,Out,(notGate(Gate) ==> G)).
```

Figure 2: The remaining code for the circuit simulator

Figure 3: A sample half-adder circuit

Prolog's built-in search facility to ask questions about the database of objects. The circuit simulator program will be used for an example. Figure 3 shows a half-adder circuit. This circuit can be built with the following interaction, which also turns on the wire `a`. All the other wires are off by default:

```
?- circuit_simulator ==> top.
?- make_wire(a, make_wire(b, make_wire(c,
      make_wire(d, make_wire(e, make_wire(s, top)))))).
?- make_orGate(or1,a,b,d,top).
?- make_andGate(and1,a,b,c,top).
?- make_notGate(not1,c,e,top).
?- make_andGate(and2,d,e,s,top).
?- set_signal(a,on,top).
```

It is then possible to ask which wires are currently carrying positive signals.

```
?- signal(Wire,on).
Wire == s;
Wire == e;
Wire == d;
Wire == a;
no
```

Note that the search facility is quite powerful. In this example it will find any object which responds (succeeds) to a `signal/2` method, even if it is not in the class of `wire` objects. A technique that we have adopted for constraining the search is to include a class-name predicate in the methods for an object. In the circuit simulator example we have used the predicat `class/2` for this purpose, though the name, of course, is arbitrary. This predicate can be used to limit the scope of search to a specific class of objects and its subclasses (since an object will have a class axiom for each class above it in the hierarchy). A constrained version of the above query would be posed as:

```
?- class(Wire,wire),signal(Wire,on).
```

The ability to use such straightforward generate-and-test techniques opens up new possibilities for object-oriented systems, and in particular for object-oriented database systems. It is surprising how few papers investigating objects in logic have discussed this possibility; see [9,12] for some discussion.

6.2 Hypothetical Queries

In Section 4 it was stated, without justification, that the pairing of implicational goals with the `STATE` declaration has distinct advantages over `assert/retract` as a method of implementing mutable state. A major advantage is the ability to pose hypothetical queries about the object space. While this paper has relied heavily on a continuation-passing style of programming, thus far the continuations passed have been open-ended, stretching out to the end of program execution. Hypothetical queries use a closed-ended continuation to ask "what-if" questions. For instance, continuing from the end of the last example, we can ask whether turning on the wire `b` will cause the half-adder's carry-out wire (`c`) to be enabled:

```
?- set_signal(b, on, signal(c,on)).
yes.
```

Here, the `set_signal` message to `b` sets up a lengthy computation which propagates the state change through the network. When the network settles, the one remaining task is to execute the continuation, which has been maintained by being passed from object to object as the computation progressed. In this case, that continuation is just the query `signal(c,on)`, which succeeds in the new circuit state. The nested implicational goals then succeed one by one, eventually causing the original query to succeed. If the query had failed, the system would backtrack looking for alternative solutions; in this example no backtrack points are available, and the entire goal would fail. In either case, the implication discharges its assumption and the circuit is left in its original state. This type of query can also be combined with a generate-and-test query of the type demonstrated in the last section. It is thereby possible to ask queries of the form:

```
?- set_signal(b, State, signal(c,off)).
State == off;
no
```

which will drive the network through a series of possibilities based on different signal values for the wire `b`, backtracking as necessary, until a settled state is found in which `c` is enabled. The trace of this proof is interesting. The proof of `set_signal` leads to a new state for the wire `b` being added to the current program. This state, however, has an uninstantiated logic variable in place of the signal value. The proof then sends an `alert` message to `and1` (as part of the proof of the local procedure `notify`) which attempts to compute its new state based on the signal values of its input wires. It picks the first clause for `computeState` and this forces the uninstantiated signal value stored for the signal on `b` to instantiate to `on`. Eventually the computation propagates through the circuit, but the continuation `signal(c,off)` fails. This causes a backtrack to the point at which the signal value was committed, and the proof is reattempted, picking the third clause of `computeState` (since the second clause cannot be used given the current signal on wire `a`). The signal on `b` is instantiated to `off` and this value is propagated. This time the continuation succeeds. If there were other values for `b` (or other proofs paths with the same value) which would allow the entire proof to succeed, these would be presented as well.

7 Conclusion

We have shown how a logic programming language with scoping constructions for constants and program clauses can be used to model several aspects of object oriented programming. To capture state and state changes, a non-logical feature was required, that of declaring a local predicate to be deterministic. Given this mild departure from logic, mutable state can adequately be modeled. Via an example, we showed how search and hypothetical reasoning, integral parts of logic programming, can be easily performed with objects.

There have been proposals to add scoping to Horn clauses that are in some senses less dynamic and open than the one based on the intuitionistic implication used in this paper. The examples in Section 5 illustrate why this more dynamic notion of scoping may be needed. In particular, abstract object classes refer to predicates that are defined only later when members of concrete object classes are created. This style of programming would not be possible using the proposal presented in [10,18].

We hope in the future to extend the analysis of this programming style in several directions. The question of how a class definition might redefine an inherited method will be investigated, as will changes to the current syntax to make continuation-passing implicit. It is expected that this may lead to a syntax similar to the "dynamic predicates" of [5]. Another area of interest is how paramaterized modules might be implemented in a more efficient manner, avoiding the need to load multiple copies of a segment of code.

Acknowledgements

Both authors are supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, DARPA N00014-85-K-0018, and ARO DAAL03-89-C-0031PRI. They would like to thank Amy Felty, John Hannan, Remo Pareschi, Lynn Stein, Ehud Shapiro, David S. Warren, and the ICLP90 reviewers for their helpful comments.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press / McGraw-Hill, 1985.

[2] R. Agrawal and N. H. Gehani. ODE (object database and environment): The language and the data model. *SIGMOD*, pages 36–45, 1989.

[3] Jean-Marc Andreoli and Remo Pareschi. Linear objects: logical processes with built-in inheritance. In *Logic Programming: Proceedings of the Seventh International Conference*, 1990.

[4] Anthony J. Bonner, L. Thorne McCarty, and Kumar Vadaparty. Expressing database queries with intuitionistic logic. In *Logic Programming: Proceedings of the North American Conference*, pages 831–850, 1989.

[5] Weidong Chen and David Scott Warren. Objects as intensions. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 404–419. MIT Press, 1988.

[6] John S. Conery. Logical objects. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 420–434. MIT Press, 1988.

[7] C. Elliot and F. Pfennig. eLP, a common lisp implementation of λProlog. January 1989.

[8] D. M. Gabbay and U. Reyle. N-prolog: An extension of prolog with hypothetical implications. i. *The Journal of Logic Programming*, 1:319–355, 1984.

[9] H. Gallaire. Merging objects and logic programming: Relational semantics. In *AAAI*, pages 754–758, 1986.

[10] L. Giordano, A. Martelli, and G. Rossi. Local definitions with static scope rules in logic programming. In *Proceedings of the FGCS International Conference*, pages 389–396, 1988.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison Wessley, 1989.

[12] F. G. McCabe. *Logic and Objects: Language, application, and implementation.* PhD thesis, Imperial College of Science and Technology, 1989.

[13] L. Thorne McCarty. Clausal intuitionistic logic I. fixed point semantics. *The Journal of Logic Programming*, 5:1–31, 1988.

[14] L. Thorne McCarty. Clausal intuitionistic logic II. tableau proof procedure. *The Journal of Logic Programming*, 5:93–132, 1988.

[15] Dale Miller. Lexical scoping as universal quantification. In *Logic Programming: Proceedings of the Sixth International Conference*, pages 268–283, 1989.

[16] Dale Miller. A logical analysis of modules in logic programming. *The Journal of Logic Programming*, pages 79–108, 1989.

[17] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proof systems as a foundation for logic programming. *To appear in the Annals of Pure and Applied Logic.*

[18] L. Monteiro and A. Porto. Contextual logic programming. In *Logic Programming: Proceedings of the Sixth International Conference*, 1989.

[19] Gopalan Nadathur and Dale Miller. An overview of λProlog. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proceedings of the 5th International Conference and Symposium*, pages 810–827. MIT Press, 1988.