HOOPS
Users Manual

John S. Conery

CIS-TR-88-12
July 25, 1988

Abstract

HOOPS is an object-oriented Prolog. Procedures that manipulate objects are implemented by *object clauses*, which have two heads: one to match the pattern of a procedure call, and one to match the pattern of an object. HOOPS has a primitive abstraction facility that allows object definitions to be grouped into classes, but inheritance and other interactions among classes must be explicitly programmed.

This manual describes the syntax of class declarations and other HOOPS language constructs. It includes many example programs, and instructions for loading and executing programs.

Department of Computer and Information Science
University of Oregon

Contents

1 Introduction **1**

2 Object clauses **2**

3 HOOPS **5**

3.1 Class definitions 5
3.2 Instances of objects 6
3.3 ∄Class options 8
3.4 Object transformations 8
3.5 ∄Commit 9
3.6 Meta-classes 11
3.7 Object Oriented Programming 11

4 Using HOOPS **13**

4.1 Starting the system 14
4.2 Loading Programs 14
4.3 HOOPS Evaluable Predicates 15
4.4 Using the Debugger 15
4.5 Objects in the Internal Database 16
4.6 Restrictions 18

5 Examples **19**

5.1 Stacks and Expression Evaluator 19
5.2 Vectors 21
5.3 Queues 22
5.4 N Queens 25
5.5 Symbol Table 30
5.6 Class Methods 33

6 Implementation **35**

6.1 The HOOPS Translator 35
6.2 **$get** and **$put** 36

1 Introduction

HOOPS is a logic programming language with facilities for object oriented programming. Objects are modeled by *object literals*, and transformed by calls to *object clauses*, as described in [3]. HOOPS introduces the class concept to Prolog, allowing object clauses to be grouped together so that only procedures inside the class can directly access and modify objects of the class.

This paper is a programmer's manual, with instructions for writing and then executing programs. There is a complete description of the syntax and semantics of class definitions, a discussion of the implementation technique and the restrictions it introduces, and there are many example programs.

The main goal for the HOOPS implementation presented here is to allow experimentation with the object clause technique for modeling mutable objects in a logic programming language. This goal lead to the following decisions about the structure of the language and implementation:

- HOOPS is a proper superset of Prolog, implemented on top of SICStus Prolog [1]. Prolog programs can be consulted or compiled and then executed normally, with no overhead.

- Any new syntax is consistent with the Edinburgh Prolog style. Keywords are actually functors of terms, declared to be prefix operators so parentheses around subsequent arguments are optional.

- Objects are stored in the "internal database," using the built-in `record` procedure. This decision means there are some restrictions on the shape of objects and the level of nondeterminism in methods. The internal database route was taken because it is reasonably efficient, and was easiest for implementing this prototypical version of HOOPS.

These decisions lead to some awkward tradeoffs and inconsistencies in language design, but they are not too hard to live with. This implementation of HOOPS is a prototype, and the goal is to gain experience with this style of object-oriented programming before moving on to the next implementation. Most of the inconsistencies can be worked around without too much effort.

The next section of the manual is a review of object literals and object clauses, and defines the terms *object*, *method*, and *class* for this language. Section 3 is where the HOOPS language is introduced, with its syntax for class definitions and rules for object manipulation. This section also has a discussion of programming issues and the restrictions on objects based on the chosen implementation. Section 4 has instructions for loading and executing HOOPS programs. Six example programs are listed and described in section 5. Finally, details of how HOOPS is translated into Prolog are given in section 6.

First time readers can get a sense of the language by reading all but sections 4 and 6. Before writing a program, however, it is important to read both of those sections.

It is assumed that readers are competent Prolog programmers, but not necessarily familiar with SICStus Prolog. SICStus is very similar to Quintus Prolog, and uses the familiar Edinburgh syntax. Except for some of the sections on implementation, readers do not have to be familiar with constructs that are unique to SICStus; knowledge of Prolog in general is sufficient.

This manual describes a few things that have not yet been implemented. For the most part, these are grouped into sections which have titles that are marked with the symbol $\nexists$ (does not exist).

2 Object clauses

Prolog programs consist of a collection of *Horn clauses*, of the form

$$Head \text{ :- } Goal_1, \ldots\ Goal_n.$$

where $Goal_1 \ldots Goal_n$ is the *body* of the clause. The body may be empty, in which case the clause is written without the implication symbol:

Head.

The head and each goal in the body of the clause are *literals*, which have a *functor* followed optionally by a set of arguments enclosed in parentheses:

foo(X,Y,a).

In Prolog, each literal in the body corresponds to a procedure call. The functor of the literal is the name of the procedure, and the arguments of the literal are the arguments passed to the procedure. The head of a clause is the entry point into a procedure. At runtime, in order to carry out a procedure call, the system finds a clause with a head that matches the call, and then executes the operations in the body of the selected clause.

In HOOPS, we will use literals to represent objects as well as procedures. Certain literals in the body of a clause will mean "create an object of the form *foo(1,2)*" instead of "make the procedure call *foo(1,2)*." The arguments of such a literal will represent the state of the object. Also, some literals in the head of a clause will describe the pattern of an object that must exist before the clause can be invoked in a procedure call. Literals that represent object states are called *object literals*, and those that represent procedure

There are several interesting things to note about these clauses:

- If there are no other clauses in the procedure for pop, a call could fail for one of two reasons: either there is no stack, or there is a stack but it is empty. In the latter case, the existing stack object would be represented by a literal `stack([])`, and the empty list parameter of this literal does not match the list parameter in the head of the clause. When a call fails, the existing stack is not modified.
- In the clause for `top`, the new object is the same as the old object. One might be tempted to omit the object literal in the body of this clause, since the new state is the same as the old state, but leaving out the new state in an object clause means something altogether different: if there is no new state, the procedure call destroys the object.
- In keeping with the logic programming paradigm, values are returned from procedure calls through unification. In both pop and `top`, if the parameter of the call is an unbound variable, the result of the procedure call is to unify that variable with the current top of the stack, in addition to creating the next stack state. Some implications of using unification to transmit information to and from objects will be discussed in greater detail later.

In Prolog, a procedure is a collection of clauses where the head literals all have the same functor (name) and the same number of arguments. A similar definition applies to a collection of object clauses. A set of clauses in which the object literals in the heads all have the same functor and arity is called a *class*. In the next section, the definition of a class will be extended a bit, to allow local procedures and other operations. The idea is to implement "abstraction barriers" in HOOPS, so that the only way to create and modify objects is by calling a procedure of the class that defines the object.

Within a class, the clauses that have procedure literals with the same name and arity are a *method*. A method is a procedure that is used to access and possibly modify an object. We will sometimes use terminology of object oriented programming, and say "send message M to object O" instead of "method M is called with object O."

A class can also be used as a template that describes the common features of a set of objects. Each element of such a set is an *instance* of the class. What this means is that there may be a number of different object literals with the same functor, but presumably each instance will have different arguments. For example, we may want to write a program that uses two or more stacks. The class for stacks would contain methods that describe how to push and pop items from stacks in general. At runtime, there will be different instances of object literals to represent the different instances of stacks that exist as the program executes. Any instance can be used in a method call, as long as it matches the form of the object literal in the head of the method. Techniques for differentiating among instances of the same class (so that a call pops an element from the correct stack, for example) will be discussed further, in section 3.2.

3 HOOPS

3.1 Class definitions

A class in HOOPS has the following basic structure.

```
class <name>.
    <options>*
new
    <procedure>
methods
    <method>*
private
    <method>*
    <procedure>*
end <name>.
```

Items marked with an asterisk are optional. If there are no lines in either the methods section or private section, then the corresponding keyword can be omitted.

`<name>` is an atom, and tells the system that literals constructed with this name will be object literals. Abstraction barriers are enforced in HOOPS by requiring that every method that accesses and/or modifies an object with this name be found somewhere within this class (*i.e.* before the `end <name>` line).

`<options>` are a set of assertions that describe attributes of the class, such as the number of instances that will be created, and so on; the available option declarations are listed in section 3.3.

Following the keyword `new` is a Prolog procedure that users will call in order to create instances of this class. A recommended convention is to name this procedure `new_<name>`, for example `new_stack`. In order to actually create the object instance, the bodies of the clauses in this procedure should have an object literal with the desired initial state. So, if new stacks should be empty, the `new` procedure in a class for stacks would have the single clause:

```
new_stack :- stack([]).
```

The keyword `methods` starts a new section of the class. Following this line is where the methods – the procedures that access and transform objects – should be placed. It is also possible to have regular Prolog procedures in this section; not all the clauses have to

```
class stack.
new
    new_stack :- stack([]).
methods
    push(X) & stack(S)     :- stack([X|S]).
    pop(X)  & stack([X|S]) :- stack(S).
    top(X)  & stack([X|S]) :- stack([X|S]).
    empty   & stack([])    :- stack([]).
end stack.
```

Figure 1: One-of-a-kind Stack Object

be object clauses. One of the classes in section 5.3 has regular Prolog procedures in the methods section; the goal here is to export to users a collection of related procedures. This example is just using the class concept to implement a module package, with some procedures available to the outside world and others hidden in the private section.

After the last method, it is possible to have an optional section that begins with the keyword `private`. In this section the programmer can place any number of procedure and method definitions. As the section heading implies, these will be local to the class, and not callable from outside the class (how this is enforced by the HOOPS compiler is discussed in section 6.1).

Finally, the end of the class is marked with the line `end <name>`. A complete class definition for a stack is shown in Figure 1.

3.2 Instances of objects

When a class is a template for one or more instances of objects, users will presumably need to distinguish one instance from another. HOOPS does not provide any built-in assistance for this, in the form of attaching unique IDs to each instance. Instead, users must provide their own IDs for each object, and make sure there is an argument position for this ID in each object instance. Figure 2 shows another way to define a stack which is suitable for creating multiple instances of stacks. When the client program calls `new_stack` it provides an atom that will be used for the ID of the new instance, and this ID should also be passed in calls to push and pop and other methods to make sure the proper instance is updated.

At first this may seem like an unnecessary overhead and burden on the programmer, but there are several positive aspects. First, everything that defines an object – its ID, the value of its state variables, and the name of its class – is directly accessible to

```
class stack.
new
    new_stack(ID) :- stack(ID,[]).
methods
    push(X,ID) & stack(ID,S)     :- stack(ID,[X|S]).
    pop(X,ID)  & stack(ID,[X|S]) :- stack(ID,S).
    top(X,ID)  & stack(ID,[X|S]) :- stack(ID,[X|S]).
    empty(ID)  & stack(ID,[])    :- stack(ID,[]).
end stack.
```

Figure 2: Template for Multiple Stacks

programmers as they write methods and local procedures. All of these things are "first class citizens" that can be accessed and/or modified.

Second, not all calls to methods need to have the ID argument intantiated to an atomic term. If the ID is unbound, the call means "select an object from this class, unify its ID with this variable, and apply the transformation." This leads to a new and interesting type of nondeterminism in HOOPS programs, where messages can be sent out so that any instance of the class can respond. Suppose the following goal is executed in a context where there are a number of different stack objects:

```
:- top(Y,X), Y > 10, write(X), nl, fail.
```

To solve this goal statement, the system will select a stack and bind `X` to its ID and `Y` to the top of that stack. If `Y` is greater than 10 we will see the ID of that stack printed. If `Y` is not greater than 10, the call to `top` backtracks, the system selects another stack, and resumes forward execution with the ID and top of the new stack.[3] The net effect is that we will see on the terminal the names of each stack with a top element greater than 10.

Third, when the ID is a instance variable of the object, it allows us to implement a pattern-directed message system. In other words, a calling program can send a message, and the first object that matches the pattern of the parameters of the message is the one to respond. As the previous example showed, the ID field is not special in any way, and an object instance can be selected based on values of any fields the programmer

[3]Because of the way objects are stored in the system, it is not always possible to backtrack to a different object. It is possible to call a method with an unbound ID, but the first object that handles the message is usually the only object that will handle it. Why this irregularity occurs, and ways to work around it, are described in sections 4.5 and 6.2.

specifies. IDs do not have to be atomic, but can be any valid logical term, which gives another avenue for flexibility.

3.3 ∄ Class options

In the current implementation, the HOOPS translator ignores any class option declarations. It skips all clauses between the class header line and the first clause of the `new` section.

Options planned for the next implementation, which will compile object clauses directly to the instruction set of a slightly modified Prolog virtual machine, are pragmas that will tell the compiler how the objects will be created and accessed. The assertion `static(N)` where `N` is an integer will tell the system that exactly N instances of the class will be created. `dynamic(N)` will mean the objects will be created (and deallocated) dynamically, up to a maximum of N instances at any one time. Other options will have the system generate unique object IDs automatically, or use certain fields of the instances as keys for faster access.

3.4 Object transformations

An important deviation from the formal specification of inferences based on object clauses has to do with the times at which objects are produced and consumed by calls to methods. From the perspective of the formal inference rule, as soon as the call is made, all literals in the body of the clause are added to the system goal statement. According to the interpretation that object literals represent object states, the new objects denoted by object literals in the body are created as soon as the unification succeeds, at the time the body literals are added to the system goal statement.

However, in HOOPS, objects are consumed and produced at different times in the execution of an object clause, and programmers must be aware of the rules the system uses to create new objects. Instances of objects are consumed when they are used to match an object literal in the head of a method, but new objects are not created until the literal is "executed" as a goal. There is a period between the invocation of a method and the time the object literal is encountered in the body when no object exists [2].

The rules for consuming and producing instances are illustrated by the following clause from a method for `bar`:

```
foo(X) & bar(X,S) :- p(X,S,T), bar(X,T), q(X,T).
```

Object instances are consumed by a call to a method. When there is a procedure call, for example `foo(a)`, the system will look for an object that matches the object literal in the head of the method; say this object is `bar(a,10)`. The call consumes the existing object. After the pattern matching operations are complete, bar(a,10) no longer exists as an object literal.

This is where the abstract model and the HOOPS language diverge. In the model, the three literals on the right side are all added to the system goal statement, meaning there is now a new object with state bar(a,T). In HOOPS, the new object will not be created until after the call to p(a,10,T), when the system encounters bar(a,T) as the next literal in the goal statement. Since this literal is an object and not a procedure, the system creates the new object state instead of making the procedure call.

To summarize, the rules for producing and consuming objects in HOOPS are:

- When a method is called, the system tries to find an existing object to match against the object literal in the head; if the match succeeds, the object is consumed.
- Literals in the bodies of methods and procedures are processed sequentially, from left to right. If a literal is a procedure literal, the system calls the corresponding procedure, as in Prolog. If the literal is an object literal, the system creates a new instance of the object.

In most cases, programmers do not have to be aware of the differences between the abstract model and the HOOPS language with respect to how objects are created. However, there are two cases where it is important. One case is when execution of a method is recursive or leads to calls to other methods for this same object. In terms of the above example, if the call to p in the body of foo leads to a call to another method that requires an object of the form bar(a,_), there will be no existing object state for that method to match on, since the "call" to bar(a,T) in the body has not happened yet. If the solution of p assumes the new object instance has been created, the call to bar should be to the left of the call to p in the body of the object clause. The N Queens example in section 5.4 and the class methods in section 5.6 show situations where programmers must know when new objects are created. In the latter example, this "feature" is actually used to good advantage.

A second case where programmers must be careful of creating objects at the right time is a side effect of the way object instances are stored in Prolog's internal database. This is discussed in more detail in section 4.5.

3.5 $\nexists$ Commit

When new object instances are created, the old instances have to be trailed in case the system backtracks to a choice point that occurs before the new object was created. For some applications, this will result in a huge amount of space being used for old object states, and in many cases the application will never backtrack to a point where it needs those states. For example, it's hard to imagine a window manager that needs to remember the size and location of every window, recording every change since the system was started.

A *commit* operation, similar to cut in Prolog, can be used to tell the system that old object states are not needed.[4] In these cases, old objects are overwritten with the new object states. Commit is a goal with one argument; the syntax of the call to this goal is to use an exclamation mark as the functor, and the name of the class as the single argument. It must occur in the body of a method somewhere to the left of the object literal that creates the new state of the object. So, for example, we might have the following in a method for a window manager:

```
move(W,X,Y) & window(W,Loc,Size,Contents) :-
    new_loc(Loc,X,Y,NewLoc), !(window),
    window(W,NewLoc,Size,Contents).
move(W,X,Y) & window(W,Loc,Size,Contents) :-
    message(['Can''t move',W]),
    window(W,Loc,Size,Contents).
```

In the first clause, the commit operator tells the system to throw away the old window object before creating the new one.

It is obvious that commit implies a cut. If the system ever backtracks to the place where the new object was created, it could retract that object, as usual. However, it cannot continue the normal backtracking sequence and restore the old object, since the old object was thrown away by the commit. In this example, the system cannot continue backtracking into the next clause for `move`, since the state of the old window object is not around any more, and therefore cannot be used to invoke any other clauses in this method. The user will not see the message, since the second clause in this method can only be invoked if a window with ID equal to `W` exists, and that object was thrown away by the commit operator. Users will see the message only if `new_loc` fails, since in this case the commit operator is never executed.

What is not so obvious is that the commit operator does an ancestor cut, throwing away choice points deep in the execution stack.[5] When an object is overwritten instead of being saved in a place where it can be restored, every choice point that would resume a computation that occurs after the old object was created must also be cut. Logically, it would be incorrect to fail back to one of these choice points, since they correspond to inferences that depend on the existence of the old object.

[4]Alas, this operation is not yet implemented in the current version of HOOPS. The system prints a warning when the user program tries to do a commit.

[5]The syntax of commit in HOOPS is identical to the syntax of ancestor cut in some Prologs [4].

To continue the window manager example, suppose the context of the call to `move` is:

```
    :- foo, new_window(w1), bar(w1), move(w1,100,200), baz.
```

After the window with ID `w1` is created, there is a call to a nondeterministic procedure named `bar`. Next, the window is moved, and the system calls `baz`, but `baz` fails. If there is a commit operator in `move`, it throws away all the choice points since the window was created, so after the failure of `baz` we cannot backtrack to one of the alternatives in `bar`. Instead, we have to go back to a choice point in either `new_window` or `foo`. It would be impractical, if not incorrect, to resume execution of `bar(w1)` for two reasons: one, `bar` might contain a call to a method of `w1`, and the old state of `w1` no longer exists; and, two, even if the system could get through `bar` without referring to the old state of `w1`, it would need that state in order to make the call to `move` when it moves forward again. Thus, when the commit operator throws away the old version of the object, it is effectively doing an ancestor cut, and removing all choice points built since the previous state of the object was created.

3.6 Meta-classes

At times it is desirable to have objects that do not lose their state on backtracking. For example, suppose we want to have a counter to keep track of the number of times the system backtracks into a method named `foo`. It is a simple matter to define the counter, and put a call to a method that increments it at the start of each clause of `foo` after the first. When the system backtracks to the second clause for `foo`, the counter will be incremented, and so on for each additional clause in `foo`. The problem is, as `foo` is backtracked, the calls to `increment` are also backtracked, and the counter reverts to the value it had before the clause was tried.

A *meta-class* is used in HOOPS to describe an object that does not return to its previous state on backtracking. If the counter is declared to be a meta-object, then we will get the behavior we are looking for, and at the end of the program it will tell us how many times the alternate clauses for `foo` were selected.

Defining a meta-class is simply a matter of putting the keyword `meta` at the front of the first line of the class definition. Figure 3 is a definition of a simple counter; instances of this class do not revert to their previous values when the procedures or methods that use them backtrack. Counters are used in the N Queens example in section 5.4.

3.7 Object Oriented Programming

HOOPS is not an object oriented programming language, in the strictest definition of that term. Our definition of "object" is consistent with other systems' use of the term

```
meta class counter.
new
    new_counter(ID) :- counter(ID,0).
methods
    inc(ID)   & counter(ID,N) :- N2 is N+1, counter(ID,N2).
    dec(ID)   & counter(ID,N) :- N2 is N-1, counter(ID,N2).
    set(ID,N) & counter(ID,_) :- counter(ID,N).
    val(ID,N) & counter(ID,N) :- counter(ID,N).
    zap(ID)   & counter(ID,N).
end counter.
```

Figure 3: Example of a meta-class

– HOOPS objects are self-contained structures, they have states, the state can change over time, and the only way to change a state is to let the object do it itself. It is the word "oriented" that is questionable. HOOPS, like C++ and Ada, is probably more correctly identified as a procedure oriented language that supports objects.

A common example that illustrates the difference between procedure oriented and object oriented languages is to consider a program that wants to transform each object in a list by sending them the message, such as `insert(X)`. In Smalltalk, an object is pulled from the list, and a message is generated and sent to it. Only when the object receives the message does the system figure out which procedure to apply. Each object has a list of methods, and the procedure that is applied is determined according to this list. Depending on the type of the object, different pieces of code are invoked. If the list is heterogeneous, different pieces of code are invoked, since queues, symbol tables, and text strings insert elements in different ways. By contrast, in a procedure oriented language, the procedure for insert is called, it determines what type of object it has as a parameter, and applies the correct piece of code within its body. In some languages, the decision about which code to invoke can be made at compile time.

It is possible to overload method names in HOOPS, and implement this same kind of program. It is possible to use the same method name in different classes; the symbol table and queue examples in section 5 both have methods named `insert`. Like Smalltalk, each method can be written separately, without concern that any other method may have the same name.[6] When a HOOPS program executes the procedure call `insert(X)`

[6]There is one concern, however: if the classes come from different files, programs will have to use the SICStus Prolog `multifile` declaration to let the system load clauses from the same procedure from more than one file.

the system tries the clauses for `insert` in order. If the clauses are object clauses, the system tries to find object literals to match with the other head of each clause. If `X` is the ID of a queue object, and no other object has ID `X`, then only the clauses for `insert` in the class `queue` can be applied. Note that it is the programmer's responsibilty to ensure that no two objects of different types have the same ID. If they do, and if they are instances of classes that have methods with the same name and arity, it is not clear which method will be applied, and which object will be transformed.

Another important aspect of object oriented programming is *class inheritance*. A class S can be defined to be a subclass of class T. Whenever a new instance of an S is created, it automatically has instance variables defined in the superclass T, along with the corresponding methods. The HOOPS class syntax does not provide any inheritance mechanism, but programmers can define such class hierarchies explicitly as part of their programs. An example of programmed inheritance is the double-ended queue in section 5.3. The top of the hierarchy is a vector. The next level is a set of procedures that increment and decrement indices into the vectors, wrapping around at either end. The bottom level has double and single ended queues built out of operations from the higher two levels.

A suggested technique for programming inheritance is that if S is to be a subclass of T, the object creation procedure of S should call the object creation procedure of T. Note that this is a case where we probably do want different object literals to have the same ID. When the object creation procedure in the class for a queue is called and asked to make a queue with ID `X`, it does so, and also calls on its superclasses to make objects with ID `X`. It makes sense to use the same ID in this case, since the set of object literals together implement pieces of the same abstract object.

A benefit of this way of building inheritance hierarchies is that programmers can build any form of inheritance structure, including multiple inheritance, and not simply a tree. On the other hand, it will take a bit of planning to arrange the order of classes within a program to make sure the right methods are applied. For example, suppose the superclass T has a method named `foo`, but we want instances of S to handle calls to `foo` in a special way. If the clauses in S are defined before the clauses in T, the underlying Prolog system will try them first whenever `foo` is called. If the clauses for `foo` in S fail, the clauses in T will be invoked via backtracking; this is the HOOPS way of passing a message up to the superclass.

4 Using HOOPS

The HOOPS runtime environment is an extension to the SICStus Prolog environment. HOOPS programs are executed by translating them into Prolog programs which are then compiled or consulted and run as usual. The HOOPS runtime system also has a few new built-in predicates that can be called from user programs.

4.1 Starting the system

The HOOPS system has been implemented in two different Prolog files: a runtime system (`hrts.pl`) and the translator (`hcomp.pl`). There is an executable binary file named `hoops` that was created by starting Prolog, compiling and loading `hrts` and `hcomp`, and then saving the Prolog state. Thus all one has to do to start HOOPS is type the name `hoops` in a Unix shell.

When the system comes up, it displays the SICStus Prolog banner plus a line that shows the current version of the HOOPS runtime and translator. Also, the prompt is changed to `HOOPS>` to remind users that the HOOPS runtime is installed and working.

From this point on, interaction with HOOPS is almost identical to interaction with Prolog. Users type goal statements terminated by periods, and the system executes the goals. If the statement succeeds, the values of variables are printed and the system waits for user interaction. If the user type a semicolon, it will try to find another solution; if the user just types a carriage return, the system quits the current goal and returns to the top-level prompt.

In the current version, no object states are carried along from one top level goal to the next. Each user query must call procedures that create the initial objects, and all objects are removed before the next query starts. An extra option is available when the system prints out the values of variables in top level queries. When the system is waiting for a semicolon or carriage return, users can type the letter 'o' followed by a return, and the system will print out the current state of every object, using the procedure `print_objects` (see section 4.3).

All of the procedures that are "private" to `hrts` and `hcomp`, *i.e.* the Prolog procedures used to implement the runtime and translator, have names that begin with a dollar sign. As long as HOOPS programs do not have procedures and methods with names that begin with a dollar sign, there will be no conflict between user clauses and system clauses.

4.2 Loading Programs

Whenever the user issues a top-level goal that asks the system to consult or compile a file, the HOOPS-to-Prolog translator is invoked automatically if the specified file is a HOOPS source file. The system invokes the translator, which creates a Prolog file as output, and then consults or compiles the Prolog file. If the input file is a Prolog file, it is consulted or compiled directly, without passing through the HOOPS translator.

Just as the Prolog system tries to be friendly, and let users type a simple name like "foo" when they want to read in the clauses in the file "foo.pl", the HOOPS runtime system tries to find a HOOPS file (a file with a name that ends in ".oop") when it sees a simple file name. Unfortunately, this leads to complicated rules for deciding whether or not the user means a HOOPS file or Prolog file, *i.e.* whether "foo" means "foo.oop" or "foo.pl". The system uses the following rules to look up a file and decide, based on its name, whether the user wants a HOOPS file or Prolog file:

1. A file is a HOOPS file only if the name ends in ".oop".
2. When the user asks for a file with name F, first see if it is a HOOPS file: if F ends in ".oop", look for it; if F does not end with ".oop", append ".oop" and look for that. If the file is found, invoke the translator, and load the translated program.
3. If the previous steps fail, F is not the name of a HOOPS file. Look for the file named F, and if it does not exist append ".pl" and look again. If the file is found in this step, it is assumed to be a Prolog file, and loaded without passing through the HOOPS translator.

When an input file is a HOOPS file, with a name of the form "foo.oop", the translated program is written to a file with the name "foo.pl" in addition to being loaded into the system.

4.3 HOOPS Evaluable Predicates

Three new evaluable predicates have been added to support HOOPS operations. These can be called from within a program or at the top level. They are:

`print_objects` Print out the entire internal database; this will print the current state of each object known to the system, plus any terms the user program stores there on its own.

`print_objects(K)` Same as `print_objects`, but restricted to terms with functor `K`. Example: `print_objects(stack)` will print all object literals with principal functor `stack`.

`zap_objects` Delete all terms from the internal database.

4.4 Using the Debugger

It is possible to use the regular SICStus Prolog debugger to debug HOOPS programs, but things get a little awkward at times. One must be aware of how the HOOPS

In spite of the problems listed above, logical variables can still be quite useful. The symbol table in section 5.5 is an example of an object that makes effective use of logical variables in object states.

Another way the internal database affects programs is the cost of storing and retrieving object states. Every time an object is stored, the system has to make a copy of it, and every time an object is retrieved, the system makes a new instance on its internal heap. To cut down on the amount of copying, the HOOPS translator tries to figure out when an object is not updated in a call to a method. If an object does not change, then there is no need to generate a call to `recorda` to write an identical copy back to the database, and no need to post a goal to retract this copy in case of backtracking to the method.

The compiler determines that a method does not change an object in those cases where there is just one object literal in the body, and it is identical to the object literal in the head. The method for `top` in the stack is a good example. Unfortunately, there are two loopholes that allow programs to change objects even though the head literal and body literal are identical. If the object retrieved from the database has an unbound variable, a goal within the method can bind that variable, and the object literal in the body describes a different state than the object literal in the head. The symbol table in section 5.5 has an example. Second, SICStus Prolog has a goal named `setarg` that modifies a term. In either case, since the compiler did not generate code that stores the new object state, the new state will be lost, since the object will not be updated in the database. In order to deal with this situation, programmers need to fool the compiler. A simple way to do this is illustrated in the following clause:

```
p(X) & s(X,Y) :- q(Y), Y = Y2, s(X,Y2).
```

Suppose `Y` is unbound in the object before the call to the method. If `q(Y)` binds `Y`, we want to update the object. The goal `Y = Y2` makes a new variable and gives it the same value as `Y` (this is roughly as expensive as copying a pointer). Since the object literal in the body is not the same as the literal in the head, the translator generates code that stores the new term in the database.

There is one positive aspect to using different code to retrieve objects the translator thinks will not change. In these cases, the code that looks up an object is backtrackable, and can be used to retrieve any or all objects that match the pattern of the object literal in the head of the clause. So, the example used previously, of a goal that finds all stacks with top element greater than 10, works as advertised:

```
:- top(X,Y), Y > 10, write(X), nl, fail.
```

Logically, one should expect to be able to write a similar goal that pops the top of every stack and prints the names of those stacks that had top elements greater than 10, but pop is a method that changes the stack, and cannot be backtracked.

To experiment with broadcast messages, where a method call should be handled by any member of the class, add a method named `is_a` to each class that returns the ID (or some other attribute) of an instance currently in the database, and use it to generate IDs before the call to other methods. For example, the following method will return the ID of every stack through backtracking:

```
is_a_stack(ID) & stack(ID,S) :- stack(ID,S).
```

Now this goal can be used to send a pop message to every stack and then call `foo`, which presumably will do something with the new stack:

```
:- is_a_stack(X), pop(X,_), foo(X).
```

Of course, the items are not permanently deleted from the stack unless stack is a meta-class, since backtracking over the call to pop restores the old top item. The explanation for why methods like `top` and `is_a_stack` can be backtracked, while pop cannot, is given in section 6.

4.6 Restrictions

Several of the language features described in this manual are not implemented yet. Also, as a result of the way objects are implemented, some things will not work the way they should. The problem areas are:

∄ Commit is not implemented.

∄ Object level nondeterminism is not fully implemented in methods that change an object. If the method simply accesses the object, it will backtrack properly (see section 4.5). If a method changes an object, it is possible to backtrack within the method, but then the program cannot backtrack to another object literal. This means the first object to handle a call to a method is the only object that will ever handle it; the system will not backtrack to let another object try the message, even though such objects exist.

∄ It is possible to have an unbound logical variable in an object state, but it may not behave the way it should. Don't expect to put a variable in an object, and then bind that variable without calling a method of the class.

- If you turn on file error reporting (with the Prolog goal `fileerrors`) you will see more error messages than you want when the HOOPS runtime tries to look up a file that doesn't exist. It is best to leave this off.

- If you change the default handling of unknown predicates, so Prolog simply fails instead of trapping to the debugger, objects will not be stored and retreived properly (this is the result of a probable bug in SICStus Prolog).

5 Examples

5.1 Stacks and Expression Evaluator

The program at the end of this section evaluates infix expressions using two stacks from the stack class that was described previously. The program logic is fairly straightforward. The expression is represented by a list of tokens, where the atoms `lp` and `rp` represent left and right parentheses, respectively. The main loop of the evaluator makes one pass over the list. If it finds a number, it pushes it on the value stack. If it finds an operator, it checks the top of the operator stack. If the input symbol has a lower or equal precedence, the top of stack is applied to the values on the value stack, otherwise the new operator is pushed on the stack.

This program relies on negation as failure to determine whether an input symbol has a lower precedence than the top of the operator stack. Instead of representing a complete precedence table, we use a relation `leq(S1,S2)` that holds when symbol `S1` has lower or equal precedence than `S2`. If a call `leq(S1,S2)` fails, we assume `S1` has a higher precedence. When the input symbol has a higher precedence, it is pushed on the stack. A shortcut based on this relation can be seen in the third clause for `eval`: if the input symbol has a lower precedence than the top of stack, and the top of stack is a left parenthesis, then the input symbol must be a right parenthesis, since the only symbol `S` for which `leq(S,lp)` succeeds is `rp`. This simplifies the program structure, since there are only four cases to consider in a call to `eval`. A right parenthesis in the input stream is handled like any other operator: it has lower precedence than any other operator, so it forces all operators on the stack to be applied, until there is a balancing left parenthesis. The left parenthesis is "applied" by throwing it and the right parenthesis away, and continuing with the next input symbol.

A final note concerns how the stack is used. Again looking at the third clause in `eval`, note that the first goal pops the top operator, on the assumption that it will be applied – either it is a left parenthesis which is now being discarded, or it is a normal operator that will be applied to the top two operands. If the precedence test fails, the call to pop is backtracked. Since there is no other way to pop a symbol from the ops stack, the method fails, and the stack is restored to its previous state before the fourth clause for `eval` is tried.

```
class stack.
```

```
new
    new_stack(ID) :- stack(ID,[]).

methods
    push(ID,X) & stack(ID,S)     :- stack(ID,[X|S]).
    pop(ID,X)  & stack(ID,[X|S]) :- stack(ID,S).
    top(ID,X)  & stack(ID,[X|S]) :- stack(ID,[X|S]).
    empty(ID)  & stack(ID,[])    :- stack(ID,[]).

end stack.

% Two-stack infix expression evaluator:

evaluate(Str,X) :-
    start_up(Str,S),        % make stacks; S is Str with parens added
    eval(S),                % reduce S
    clean_up(X).            % make sure stacks clean; return top of val

start_up(Si,[lp|So]) :-     % use atom 'lp' for left paren,
    new_stack(val),         %    'rp' for right paren
    new_stack(ops),
    append(Si,[rp],So).

clean_up(X) :-
    pop(val,X),
    empty(val),
    empty(ops), !.
clean_up(err).

% Main loop (see text for comments):

eval([]).
eval([S1|Sn]) :-
    number(S1), !,          % numbers just go on value stack
    push(val,S1),
    eval(Sn).
eval([S1|Sn]) :-            % see if input operator lower than TOS
    pop(ops,X),
    leq(S1,X), !,
    ( X = lp ->             % input lower than TOS and TOS = lp means
        eval(Sn)            %   input is rp; continue with next symbol
    ;
        apply_op(X),        % TOS is regular operator; apply it
        eval([S1|Sn])       % may have to apply again (e.g. 4-2*1-1)
    ).
eval([S1|Sn]) :-              % input not lower than TOS; push it
   push(ops,S1),
   eval(Sn).

% Apply the top operator to top two values, push the result.

apply_op(F) :-
    pop(val,V2), pop(val,V1), apply(F,V1,V2,X), push(val,X).

apply(+,X,Y,Z) :- Z is X + Y.
apply(-,X,Y,Z) :- Z is X - Y.
apply(*,X,Y,Z) :- Z is X * Y.
apply(/,X,Y,Z) :- Z is X / Y.

% Precedence relations; leq(S1,S2) means S1 is lower or equal precedence than
% S2.  When called, S1 is input, S2 on stack; if this succeeds, S2 is applied:

leq(+,+).   leq(-,+).   leq(*,*).   leq(/,*).
leq(+,-).   leq(-,-).   leq(*,/).   leq(/,/).
leq(+,*).   leq(-,*).
leq(+,/).   leq(-,/).        leq(rp,_).
```

5.2 Vectors

The next example shows one way to represent a vector as an object. A fixed length vector of N elements can be represented in Prolog as an N-ary complex term. The built-in procedure `arg(N,T,A)` unifies `A` with the `N`th argument of term `T`, giving constant-time access to any element in the vector.

The state variables of the vector are the term that represents the vector itself, and an integer used to adjust indices. `arg` counts the first argument as 1, so if the indices of the vector should be zero-based (as they are in the next example, section 5.3) then the access methods should adjust each request by adding 1.

This particular implementation uses the SICStus builtin `setarg(N,T,A)` to change the `N`th argument of `T` to `A` in the method for `vset`. This is an efficient way to change one location, but unfortunately HOOPS cannot take advantage of it yet. Since the internal database is used to store objects, the entire new object is copied to the database, and the time to update the array is proportional to the size of the array. Future versions of HOOPS will be smarter, and update only fields that change.

Identifying fields that change brings up another subject. Logically, it would be correct to use `vector(ID,V,X)` for the object literal in the body of `vset`, since `setarg` has modified `V`. But the HOOPS compiler doesn't know about `setarg`, and can't tell

wrapping around when it reaches either end of the vector. The other class describes double ended queues (dequeues), which can insert and remove items from either end.

The class hierarchy in this example has `qv` as a subclass of `vector`, and `dequeue` as a subclass of `qv`. In other words, each new dequeue object will inherit the state variables and methods of its superclasses. A dequeue will be a vector, with methods for accessing and changing items determined by that class, and it will also inherit the operations defined in `qv`. Note that the definitions in `qv` work equally well for single ended queues, and that a class `queue` for single ended queues could also be a subclass of `qv`. The inheritance is programmed explicitly: the object creation procedure for `dequeue` calls the object creation procedure for `qv`, which in turn calls the object creation procedure for `vector`.

Note that all three objects have the same ID. If a user program wants to find out the size of the queue with ID `q1`, it can call `size(q1,N)`, and the call is handled directly by the `vector` superclass. This ID scheme is also used in the methods for `insert` and `remove` in the queue. `insert` calls the superclass method named `vset`, passing the same ID that was passed to it, and the routines in `qv` call the `size` method for vectors.

```
class qv.

new
    new_qv(ID,N) :-
        N1 is N-1,                      % make vector to hold items,
        new_vector(ID,0,N1).            %   indexed 0..N-1

methods
    qinc(ID,X,Y) :-                     % move pointer to right by 1
        size(ID,N),
        Y is (X+1) mod N.

    qdec(ID,X,Y) :-                     % move pointer to left by 1
        size(ID,N),
        Y is (X+N-1) mod N.

    % if R >= L, number of free slots is R-L+1; instead of testing if R > L,
    % just add N to R first and then take the result mod N (two arithmetic
    % operations are faster than evaluating a conditional and branching)

    qfree(ID,L,R,X) :-                  % calculate # free slots,
        size(ID,N),                     %   given pointers L and R
        X is (R+N-L+1) mod N.

    qused(ID,L,R,X) :-                  % calculate # occupied slots
```

```
        size(ID,N),
        X is N - ((R+N-L+1) mod N) - 1.

    qempty(ID,L,R) :-               % queue empty if R next to L
        size(ID,N),
        1 is (L+N-R) mod N.

    qfull(ID,L,R) :-                % queue full if R two slots away from L
        size(ID,N),
        2 is (L+N-R) mod N.

end qv.

% Double-ended queue (dequeue).  The object literal that describes the
% current state of a queue is dequeue(ID,L,R), where L is the index of the
% left item, and R is the index of the right item.  The data structure
% that holds the items in the queue is a vector, accessed by its methods
% vset and vref.

class dequeue.

new
    new_dequeue(ID,Size) :-
        Size > 1 ->
            new_qv(ID,Size),
            dequeue(ID,1,0)
        ;
            format('dequeue ~w not made; size ~w too small~n',[ID,Size]).

methods
    empty(ID) & dequeue(ID,L,R) :-
        qempty(ID,L,R),
        dequeue(ID,L,R).

    full(ID) & dequeue(ID,L,R) :-
        qfull(ID,L,R),
        dequeue(ID,L,R).

    insert(_,ID,_) & dequeue(ID,L,R) :-
        qfull(ID,L,R), !,
        format('Dequeue ~w full; insert failed.~n',[ID]), fail.
    insert(left,ID,Val) & dequeue(ID,L,R) :-
        qdec(ID,L,L2),
        vset(ID,L2,Val),
        dequeue(ID,L2,R).
    insert(right,ID,Val) & dequeue(ID,L,R) :-
        qinc(ID,R,R2),
        vset(ID,R2,Val),
        dequeue(ID,L,R2).

    remove(_,ID,_) & dequeue(ID,L,R) :-
        qempty(ID,L,R), !,
        format('Dequeue ~w empty; remove failed.~n',[ID]), fail.
    remove(left,ID,Val) & dequeue(ID,L,R) :-
        vref(ID,L,Val),
        qinc(ID,L,L2),
        dequeue(ID,L2,R).
    remove(right,ID,Val) & dequeue(ID,L,R) :-
        vref(ID,R,Val),
        qdec(ID,R,R2),
        dequeue(ID,L,R2).

end dequeue.

% test out the queue

test(N) :-
        new_dequeue(dq,N),
        cycle.

cycle :-
        write('DEQ Test: '),
        read(T),
        (T = exit ; exec(T), !, cycle).

exec(in(D,V))  :- insert(D,dq,V), format('~w inserted at the ~w~n',[V,D]).
exec(out(D,V)) :- remove(D,dq,V), format('~w removed from the ~w~n',[V,D]).
exec(empty)    :- empty(dq) ->
                    format('dq empty~n',[]) ; format('dq not empty~n',[]).
exec(full)     :- full(dq) ->
                    format('dq full~n',[]) ; format('dq not full~n',[]).
exec(print)    :- print_objects, statistics.
exec(_).
```

5.4 N Queens

The next example is the N Queens problem. This programs shows how methods can be nondeterministic – if one way to handle a message leads to a later failure, the system

- If Q is already in row R, the method should fail.
- If Q has already taken a different row, or if R has already been removed from domain, the message is ignored.
- If R is in the domain of Q, it is removed. Now there are two more cases to consider:
  - There is one row left in the domain; Q should select this row for herself, and broadcast this information to all the other queens by sending them restrict messages.
  - There are two or more rows left; no further action is required.

This class is an example of a class that has local procedures. The only "exported" procedures here are **restrict** and **choose**; everything else is hidden inside the class. If you ask for a listing at runtime, you will see that the local procedures all have names of the form **queen:**, for example **queen:assigning**.

The places where we have to be careful to put back a queen object before calling other procedures from the body of a method are in the second clause for **choose**, and the third clause of restrict. In both cases, the queen has just taken a row for herself, and is about to send **restrict** messages to all the other queens via the call to the local procedure **propagate**. It is possible that the other queens will have to send a message back to this queen, so we have to make sure there is a queen object around to handle the message. The reason we might get messages coming back is that in telling another queen to stay out of our row, we may leave her with just one location, in which case she will take it and broadcast to all other queens that she has taken that location.

The main loop of this program simply sends a **choose** message to each queen. The program also uses a meta counter to count the number of times a queen has to be removed on backtracking (see the local procedure named **assigning** that is called whenever a queen is placed).

```
class queen.

new
    new_queen(Q,N) :-
        init_domain(N,D),
        queen(Q,D).

methods

    choose(Q) & queen(Q,N) :-
        integer(N), !,          % queen may already be placed, due to
        queen(Q,N).             %    restrictions propagated by another
    choose(Q) & queen(Q,D) :-
        member(Row,D,_),         % get a candidate row
        assigning(Q,Row),        % collect stats about assignment
        queen(Q,Row),            % new object state has this row
        propagate(Q,Row).        % apply constraints to other queens

    % This method is the heart of the "forward checking" style of
    % solution.  Remove row R from the domain of queen Q; if Q has
    % already been assigned to R, we have to backtrack; if, after
    % removing R, there is only one place left for Q, assign this
    % position (and, by extension, propagate the new constraints)

    restrict(Q,R) & queen(Q,R) :-
        !, fail.                 % Q is already in row R -- fail
    restrict(Q,R) & queen(Q,X) :-
        integer(X), !,           % Q assigned already -- do nothing
        queen(Q,X).
    restrict(Q,R) & queen(Q,D) :-
        member(R,D,[D2|Dr]), !,
        ( Dr = [] ->             % empty Dr means D2 is last row;
            queen(Q,D2),         %   grab it and propagate constraints
            assigning(Q,D2),
            propagate(Q,D2)
        ;
            queen(Q,[D2|Dr])     % otherwise just continue
        ).
    restrict(Q,R) & queen(Q,D) :-
        queen(Q,D).              % R already out of D -- do nothing

    show(Q) & queen(Q,D) :-
        format('Queen ~w is in row ~w~n',[Q,D]),
        queen(Q,D).

private

    % init_domain(N,D) -- make the initial domain (complete list from
    % [1..N]), given N rows and columns on chessboard

    init_domain(0,[]) :- !.
    init_domain(N,[N|Rem]) :-
        N2 is N-1,
        init_domain(N2,Rem).

    % local procedure shared by 'choose' and 'restrict' -- for debugging
    % and counting purposes, print assignment (and removal, on backtracking)

    assigning(Q,R) :-
        format('Queen ~w placed in row ~w~n',[Q,R]).
    assigning(Q,R) :-
        format('Queen ~w removed from row ~w~n',[Q,R]),
        inc(backtracks),
        fail.

    % propagate(Q,R):  queen Q has just been placed in row R; restrict
    % other queens from being in squares in diagonal from square (Q,R)
    % or in same row before or after Q (6 of the 8 possible directions).
    % We can assume no other queen is in the same column.

    propagate(Q,R) :-
        check(Q,R,1,-1),        % upper right diagonal
        check(Q,R,1,0),         % same row, right
        check(Q,R,1,1),         % lower right diagonal
        check(Q,R,-1,1),        % lower left diagonal
        check(Q,R,-1,0),        % same row, left
        check(Q,R,-1,-1),       % upper left diagonal
        !.

    check(1,_,-1,_) :- !.       % boundary conditions:
    check(_,1,_,-1) :- !.              %   stop when row or column hits edge of
    check(N,_,1,_) :- val(nq,N), !.    %   board; rows and columns are labeled
    check(_,N,_,1) :- val(nq,N), !.    %   1 thru N
    check(Q,R,H,V) :-
        Q2 is Q + H,            % compute coordinates (Q2,R2) of
        R2 is R + V,            %   next square to check
        restrict(Q2,R2),        % restrict Q2 from being in row R2
        check(Q2,R2,H,V),       % continue propagation
        !.

    % member(E,L,Lr) -- succeeds if E is an item in list L, and Lr is
    % the list of all other items except E in L

    member(E,[E|L],L).
    member(E,[X|L],[X|L2]) :- member(E,L,L2).

end queen.


/*                  - - - - - top level goals - - - - -                 */
```

```
run(N) :-
        init_counters(N),
        do(1,N,new_queen(N)),
        do(1,N,choose),
        print_results(N).

do(First,Last,_) :- First > Last, !.
do(First,Last,P) :-
        P =.. [G|Args],
        Goal =.. [G,First|Args],
        call(Goal),
        Next is First + 1,
        do(Next,Last,P).

init_counters(N) :-
        new_counter(backtracks),        % meta counters
        new_counter(nq),
        set(nq,N).

print_results(N) :-
        format('All queens placed:~n',[]),
        do(1,N,show),
        val(backtracks,B),
        format('Number of backtracks = ~w~n',[B]).
```

5.5 Symbol Table

The next example is a symbol table object. This table is derived from the classic symbol table described by Warren, in his paper on writing compilers in Prolog [5].

The table itself is a list of associations, terminated by an unbound variable, for example `[foo=10,bar=3|_]`. Warren uses the usual definition of `member/2` both to look up a symbol and to insert a new symbol:

```
    member(X,[X|_]).
    member(X,[_|L]) :- member(X,L).
```

The call `member(foo=X,ST)` succeeds for one of two reasons: `foo=val` is an element of the list `ST` and `X` is unified with its value; or, `foo=X` is not currently in `ST`, but the base case in the definition of `member` succeeds, unifying the unbound variable at the end of the list to `[foo=X|L]`. In the latter case, the table is extended, and now has an entry for `foo` plus a new unbound variable to terminate the list.

The nice thing about this data structure is that it is not necessary to copy the list when new entries are added. Simply binding the variable at the end extends the original list; this is one of the classic uses of the logical variable. Another nice feature is that forward reference pointers come for free. If we call `member(foo=X,ST)` when we first encounter the symbol `foo` in an expression, but `foo` is not in the table yet, it is inserted, but not given a value yet (since `X` is unbound). Later, when we encounter the definition of `foo`, and know it has a value of 10, for example, the call `member(foo=10,ST)` binds the variable in the table to 10.

However, there are two awkward situations. If we want to use `member` to scan the list to return all symbols and their values via backtracking, e.g. the call is `member(S=X,L)`, the call to member at the end of the list succeeds instead of fails, and it adds a new entry to the table, with both the symbol part and value part unbound. Another problem is that if we just call `member` each time we find a new symbol definition, we don't know if we have a redefined symbol. `member` will just add the new definition at the end of the list.

The class shown below remedies these two situations, while keeping the same overall structure of the table. The table is a list of associations terminated with an unbound variable. The method named `insert(S,V)` is used to add a new definition; it will fail if `S` is already in the table with a different value. `assoc(S,V)` can be used either to look up a symbol/value pair, or generate all associations through backtracking. It fails if `S` is not in the table. As an extra feature, the class keeps track of the number of symbols in the table.

What this example demonstrates is that unbound variables can be used profitably in instance variables of objects. However, as discussed in sections 3.4 and 4.6, users have to be careful with unbound variables in the current implementation. This example works because the term that represents the current state of the table is fetched from the internal database, modified by binding one of the variables in the table, and then returned to the database. Note that the new table literal in the clause for `insert` is different that the head literal (it has `N2` instead of `N`) so the compiler forces the runtime system to store the new value. In future versions, the system will correctly handle this term, and not have to copy the table to and from the internal database.

```
class symbol_table.

new
    new_symbol_table :-
        symbol_table(_,0).

methods
```

```
    % insert(+Sym,?Val) -- Unify Sym=Val with entries in the table;
    % the difference between this and 'member' is that 'insert' fails if
    % Sym is in the table with a different value, and Sym must be bound

    insert(Sym,Val) & symbol_table(Tbl,N) :-
        nonvar(Sym),
        find(Sym=Val,Tbl,Res),
        ( Res = found ->
            N2 = N
        ;
            N2 is N+1
        ),
        symbol_table(Tbl,N2).

    % assoc(?Sym,?Val) -- Look up Sym in the table; can be used to traverse
    % the table if Sym is unbound, without adding new table entry at end.

    assoc(Sym,Val) & symbol_table(Tbl,N) :-
        symbol_table(Tbl,N),
        member(Sym=Val,Tbl).

    % size(?N) -- there are N symbols currently in the table

    size(N) & symbol_table(Tbl,N) :-
        symbol_table(Tbl,N).

private

    % find(X,L,R) -- assumes X is of the form S=V; acts like member/2,
    % but fails (doesn't scan rest of L) if S=V2 is in the list and
    % V =\= V2.  Unify Res with 'new_sym' if S=V added to the list, or
    % 'found' if S is already in the list.

    find(X,L,new_sym) :-
        var(L), L = [X|_].
    find(S=V,[S=V2|L],found) :-
        !, (V = V2  ;  fail).
    find(X,[_|L],Res) :-
        find(X,L,Res).

    % local version of member, used to traverse list without adding new entries

    member(X,Y) :- var(Y), !, fail.         % var table entry ends search
    member(X,[X|_]).
    member(X,[_|L]) :- member(X,L).

end symbol_table.
```

5.6 Class Methods

Some object oriented languages allow programmers to write *class methods*, or procedures that operate on class instead of any one object that is an instance of the class. It is easy to write this kind of method in HOOPS, as the following example shows.

The class is named foo. Instances of the class are like counters, handling messages that increment, decrement, and set the value of the counter. The class methods are bag, unbag, mgu, count, and sum. bag is a meta-level predicate, turning objects into terms; it collects all the instances of foo that exist at the time of the call and returns the corresponding terms in a list. Note that no instances exist after this call – a call to inc or any other instance method will fail. unbag does the opposite, making foo ojects for each item in the input list. As a simple security check, it makes sure each item has the proper shape. count returns the number of foo instances. sum and mgu replace all current instances with a single new instance, and return the ID of the new instance. The new ID is chosen at random from the IDs of the earlier instance. In sum, the new instance is a counter with a value that is the sum of all the previous instances. mgu creates a new instance that is the most general instance of all the previous objects.

```
class foo.

new
    new_foo(X) :- foo(X,0).

methods

    % instance methods -- methods that operate on objects of the class -- are
    % 'inc' (add 1 to state of the object), 'dec' (substract 1), and 'set':

    inc(X)   & foo(X,Y) :- Z is Y+1, foo(X,Z).
    dec(X)   & foo(X,Y) :- Z is Y-1, foo(X,Z).
    set(X,Y) & foo(X,_) :- foo(X,Y).

    % the interesting stuff -- the following are class methods:

    bag(L) :- collect([],L).    % call local method, with initial list

    unbag([]).
    unbag([F1|Fn]) :-
```

```
        F1 = foo(X,Y) ->          % verify form of new object
            foo(X,Y),
            unbag(Fn).

    mgu(X) :-
        bag(L),                   % collect all foo's
        unify(L,foo(X,Y)),        % boil down to single element
        foo(X,Y).                 % assert it; return its ID to caller

    count(N) :-
        bag(L),
        length(L,N),
        unbag(L).

    sum(X,Y) :-
        bag(L),
        foosum(L,0,foo(X,Y)),
        foo(X,Y).

private

    % collect all foo's into a list; in this case, return a term that looks
    % exactly like the object, but in general we might want to hide some stuff

    collect(Li,Lo) & foo(X,Y) :-
        collect([foo(X,Y)|Li],Lo).
    collect(L,L).                          % base case; invoked when no foo's

    % reduce list of foo's to most general common instance

    unify([],_).
    unify([X|Y],U) :-
        X = U,
        unify(Y,U).

    % sum values in all foo's in list, return new foo with sum

    foosum([foo(X,Y)],N,foo(X,Z)) :-         % id of new foo is id of last
        !, Z is N+Y.
    foosum([foo(X,Y)|Rem],N,F) :-
        Z is N+Y,
        foosum(Rem,Z,F).

  end foo.
```

```
% test out some stuff:

make_foos([]).
make_foos([F1|Fn]) :- new_foo(F1), make_foos(Fn).

set_foos([]).
set_foos([F=V|Rem]) :- set(F,V), set_foos(Rem).

utest(F) :-
        make_foos([a,a,a,a]),          % all have to have same ID...
        set_foos([a=10,a=A,a=B,a=C]),
        mgu(F).

sumtest(X,Y) :-
        make_foos([a,b,c,d]),
        set_foos([a=5,b=3,c=1,d=10]),
        sum(X,Y).

bagtest(L2) :-
        make_foos([a,b,c,d]),
        set_foos([a=5,b=3,c=1,d=10]),
        bag(L1),
        inc_foos(L1,L2),
        unbag(L2).

inc_foos([],[]).
inc_foos([foo(X,Y)|L1],[foo(X,Z)|L2]) :- Z is Y+1, inc_foos(L1,L2).
```

6 Implementation

6.1 The HOOPS Translator

The transformation from object clause to Horn clause is simple. For an object clause with head `p & s` create a Horn clause with head `p`. The body of the Horn clause will have the same body as the object clause, with the addition of a new goal `$get(s)` at the front. Every object literal `s` in the body is replaced it with the goal `$put(s)`.

`$get` and `$put` are HOOPS runtime system goals that manage objects in the internal database; they are described in the next section. As their names imply, these goals insert and remove object instances. These actions are undone on backtracking. If an object is defined by a meta class, the translator compiles code that accesses the database directly, without calling `$get` or `$put`, since changes made by meta classes are not undone when the method fails.

As an optimization, if the translator determines that the input object and output object in a method are identical (via the goal `In == Out`), then `$get` is replaced by `$state`, and there is no call to `$put`.

The translator enforces the scope rules of class names by refusing to make a goal of the form `$put(s)` or `$get(s)` outside the class for object `s`. Private methods and procedures are implemented by giving them names that depend on the current class name. If there is a definition of a method or procedure named `foo` in the private section of a class `s`, the translator makes a new name of the form `s:name`.[7]

6.2 $get and $put

Instances of objects are stored in Prolog's internal database, using the principle functor of the object literal as the database key. The Prolog procedure `recorda(K,T,R)` is used to record a term `T` using key `K`.

When a new object instance is created, HOOPS stores it in the database. When the system backtracks to this point, the item has to be erased from the database. Similarly, when a method is called, the system erases from the database an object that matches the object literal in the head, and if the method fails, the system has to put that object back in the database. The procedure in the HOOPS runtime system that records a new object, and then erases it on backtracking, is named `$put`. The procedure that erases an existing object on entry into a method, and puts it back on backtracking, is called `$get`.

It is not a trivial matter to code `$get` and `$put` in Prolog. At first glance, one would think the following structure would work:

```
'$get'(Obj,State) :- recorded(Obj,State,Ref), erase(Ref).
'$get'(Obj,State) :- recorda(Obj,State), fail.

'$put'(Obj,State) :- recorda(Obj,State).
'$put'(Obj,State) :- recorded(Obj,State,Ref), erase(Ref), fail.
```

This definition for `$get` works when there is an object with the correct form in the database, but has a serious problem if there is no matching object. In a peculiar form of wishful thinking, it creates an object instance that does match the call. If the user program makes a call such as `top(a,5)` to see if it is true that the top of stack `a` is 5, and 5 is not currently on the stack, the system will create a new stack with ID `a`, put 5 on the top, and install it as an extra instance of stack `a`.

This problem can be solved with a bit of convoluted coding, but a more serious problem remains. What if there is a cut in the body of the method? The second clause

[7]The name is an atom, it just has a funny symbol stuck in the middle.

for **$get** is cut along with the alternatives for the method itself, and if the method fails the object is never put back in the database. This problem, which also applies to **$put**, is solved by using **undo** to post goals to undo the effects of forward execution. undo(G) is a NOP when encountered as a goal, but the system puts G on the trail. Later, when the trail is unwound during backtracking, and the system finds G on the trail, it is executed. Since the trail has to be used even when alternatives have been cut, the goals that remove or reassert objects are safe.

Here is the actual code for **$get**, **$put**, and **$state**:

```
'$get'(Obj,State) :-
     recorded(Obj,State,R),          % is there a matching object?
     instance(R,Pattern),            % yes; get the recorded version
     erase(R),                       % zap it
     !,                              % get only one object per call
     undo(recorda(Obj,Pattern,_)).   % put object back on backtracking

'$put'(Obj,State) :-
     recorda(Obj,State,R),
     undo((recorded(Obj,State,R2),erase(R2))), !.

'$state'(Obj,State) :- recorded(Obj,State,_).
```

The cut in the body of **$get** is what prevents it from finding multiple object instances that match the state of the object described in the head of the method. It is tempting to leave this out, so that the choice point in **recorded** would find the next object that matches. However, by the time we backtrack to **recorded**, the postponed goal in the argument to **undo** has been executed, and it has just asserted a new object. It is too much to expect the internal database to keep straight the different objects that are retracted and reasserted, and to expect them to be brought out once each.

Since **$state** is just a call to **recorded**, it can be expected to generate all matching objects on backtarcking. This is why methods that do not change objects, *i.e.* the methods with head object literals that are translated to **$state** instead of **$get**, can be nondeterministic.

The call to **instance** in **$get** is needed in case the object contains an unbound variable. We could skip this call, and use **State** in the call to **recorda** in the goal that puts the object back on backtracking, except there will be a problem if the object contains an unbound variable. If it does, **State** will be the unification of the object in the call and the object in the database, and the object that is restored will be this unified object. In other words, even though the method fails, the object put back is a less general instance of the original object.

One would think that the call to `recorded` in the postponed goal in `$put` is redundant. Why not use the database reference `R` returned by the call to `recorda`, and just call `erase(R)` on backtracking? The reason is, by the time the call to `erase` is executed, some other method may have been applied to the object, deleting it from the database. Instead of finding the original term with reference `R`, it could find a completely different object, or no term at all.

References

[1] Carlsson, M. and Widén, J. *SICStus Prolog User's Manual*. Research Report R 88007, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden, Feb. 1988.

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

[3] Conery, J.S. Logical objects. In *Proceedings of the Fifth Conference/Symposium on Logic Programming*, (Seattle, WA, Aug 15–19), 1988.

[4] Sterling, L. and Shapiro, E. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

[5] Warren, D.H.D. Logic programming and compiler writing. *Software – Practice and Experience 10*, (1980), 97–125.