HCPRVR: AN INTERPRETER FOR LOGIC PROGRAMS

Daniel Chester
Department of Computer Sciences
University of Texas at Austin

ABSTRACT

An overview of a logic program interpreter written in Lisp is presented. The interpreter is a Horn clause-based theorem prover augmented by Lisp functions attached to some predicate names. Its application to natural language processing is discussed. The theory of operation is explained, including the high level organization of the PROVE function and an efficient version of unification. The paper concludes with comments on the overall efficiency of the interpreter.

I INTRODUCTION

HCPRVR, a Horn Clause theorem PRoVeR, is a Lisp program that interprets a simple logical formalism as a programming language. It has been used for over a year now at the University of Texas at Austin to write natural language processing systems. Like Kowalski [1], we find that programming in logic is an efficient way to write programs that are easy to comprehend. Although we now have an interpreter/compiler for the logic programming language Prolog [2], we continue to use HCPRVR because it allows us to remain in a Lisp environment where there is greater flexibility and a more familiar notation.

This paper outlines how HCPRVR works to provide logic programming in a Lisp environment. The syntax of logic programs is given, followed by a description of how such programs are invoked. Then attachment of Lisp functions to predicates is explained. Our approach to processing natural language in logic programs is outlined briefly. The operation of HCPRVR is presented by giving details of the PROVE and MATCH functions. The paper closes with some remarks on efficiency.

II LOGIC PROGRAM SYNTAX

A logic program is an ordered list of axioms.

An axiom is either an atomic formula, which can be referred to as a fact, or an expression of the form

(<conclusion> < <premiss1> ... <premissN>)

where both the conclusion and the premisses are atomic formulas. The symbol "<" is intended to be a left-pointing arrow.

An atomic formula is an arbitrary Lisp expression beginning with a Lisp atom. That atom is referred to as a relation or predicate name. Some of the other atoms in the expression may be designated as variables by a flag on their property lists.

III CALLING LOGIC PROGRAMS

There are two ways to call a logic program in HCPRVR. One way is to apply the EXPR function TRY to an atomic formula. The other way is to apply the FEXPR function ? to a list of one or more atomic formulas, i.e., by evaluating an expression of the form

(? <formula1> ... <formulaN>)

In either case the PROVE function is called to try to find values for the variables in the formulas that makes them into theorems implied by the axioms. If it finds a set of values, it displays the formulas to the interactive user and asks him whether another set of values should be sought. When told not to seek further, it terminates after assigning the formulas, with the variables replaced by their values, to the Lisp atom VAL.

IV PREDICATE NAMES AS FUNCTIONS

Occasionally it is useful to let a predicate name be a Lisp function that gets called instead of letting HCPRVR prove the formula in the usual way. The predicate name NEQ*, for example, tests its two arguments for inequality by means of a Lisp function because it would be impractical to have axioms of the form (NEQ* X Y) for every pair of

---

* This work was supported by NSF Grant MCS 79-24918.

constants X and Y such that X does not equal Y. Predicate names that are also functions are FEXPRs and expect that their arguments have been expanded into lists in which all bound variables have been replaced by their values. These predicate names must be marked as functions by having the Lisp property FN set to T, e.g., by executing (PUT '<predicate name> 'FN T), so that HCPRVR will interpret them as functions.

V PARSING NATURAL LANGUAGE

As an example of an HCPRVR application, we show how it can be made to parse sentences. Parsing is done by axioms that combine syntax with semantic and pragmatic tests. Each syntactic category is a predicate name that requires three terms after it. The first term is a list of words. The second term is a representation for an initial segment of the word list which makes a phrase belonging to the syntactic category. The last term is the remainder of the word list after the phrase is removed. Thus, in the atomic formula (NP (THE CAT IS ON THE MAT) (CAT DET THE) (IS ON THE MAT)) the second term, (CAT DET THE) represents the NP phrase THE CAT.

By letting syntactic categories be predicates with three arguments, we can make axioms that pull phrases off of a list of words until we get a sentence that consumes the whole list. In addition, arbitrary tests can be performed on the phrase representations to check whether they can be semantically combined. Usually the phrase representation in the conclusion part of an axiom tells how the component representations are combined, while the premisses tell how the phrase should be factored into the component phrases, what their representations should be, and what restrictions they have. Thus, the axiom

```
((S X (U ACTOR V . W) Z) < (NP X V Y)
                           (VP Y (U . W) Z)
                           (NUMBER V N1)
                           (NUMBER U N2)
                           (EQ N1 N2))
```

says that an initial segment of word list X is a sentence if first there is a noun phrase ending where word list Y begins, followed by a verb phrase ending where word list Z begins, and both phrases agree in number (singular or plural). Furthermore, the noun phrase representation V is made the actor of the verb U in the verb phrase, and the rest of the verb phrase representation, W, is carried along in the representation for the sentence.

After suitable axioms have been stored, the sentence THE CAT IS ON THE MAT can be parsed by typing

```
(? (S (THE CAT IS ON THE MAT) X NIL)
```

The result of this computation is the theorem

```
(S (THE CAT IS ON THE MAT)
   (IS ACTOR (CAT DET THE)
       LOC (ON LOC (MAT DET THE)))
   NIL)
```

VI THEORY OF OPERATION

A. General Organization

HCPRVR works essentially by the problem reduction principle. Each atomic formula can be thought of as a problem. Those that appear as facts in the list of axioms represent problems that have been solved, while those that appear as conclusions can be reduced to the list of problems represented by the premisses. Starting from the formula to be proved, HCPRVR reduces each problem to lists of subproblems and then reduces each of the subproblems in turn until they have all been reduced to the previously solved problems, the "facts" on the axiom list. The key functions in HCPRVR that do all this are PROVE and MATCH.

B. The PROVE Function

PROVE is the function that controls the problem reduction process. It has one argument, a stack of subproblem structures. Each subproblem structure has the following format:

```
( <list of subproblems>.<binding list> )
```

where the list of subproblems is a sublist of the premisses in some axiom and the CAR of the binding list is a list of variables occurring in the subproblems, paired with their assigned values. When PROVE is initially called by TRY, it begins with the stack

```
( ( ( <formula> ) NIL ) )
```

The algorithm of PROVE works in depth-first fashion, solving subproblems in the same left-to-right order as they occur in the axioms and applying the axioms as problem reduction rules in the same order as they are listed. PROVE begins by examining the first subproblem structure on its stack. If the list of subproblems in that structure is empty, PROVE either returns the binding list, if there are no other structures on the stack, i.e., if the original problem has been solved, or removes the first structure from the stack and examines the stack again. If the list of subproblems of the first subproblem structure is not empty, PROVE examines the first subproblem on the list. If the predicate name in it is a function, the function is applied to the arguments. If the function returns NIL, PROVE fails; otherwise the subproblem is removed from the list and PROVE begins all over again with the modified structure.

When the predicate name of the first subproblem in the list in the first subproblem stucture is not a function, PROVE gets all the axioms that are stored under that predicate name and assigns them to the local variable Y. At this point PROVE goes into a loop in which it tries to apply each axiom in turn until one is found that leads to a solution to the original problem. It does this by calling the MATCH function to compare the conclusion of an axiom with the first subproblem. If the match fails, it tries the next axiom. If the match succeeds, the first subproblem is removed from the first subproblem structure, then a new subproblem structure is put on the stack in front of that structure. This new subproblem structure consists of the list of premisses from the axiom and the binding list that was created at the time MATCH was called. Then PROVE calls itself with this newly formed stack. If this call returns a binding list, it is returned as the value of PROVE. If the call returns NIL, everything is restored to what it was before the axiom was applied and PROVE tries to apply the next axiom.

The way that PROVE applies an axiom might be better understood by considering the following illustration. Suppose that the stack looks like this:

(((C1 C2).<blist>) ...)

The first subproblem in the first subproblem structure is C1. Let the axiom to be applied be

(C < P1 P2 P3)

PROVE applies it by creating a new binding list blist', initially empty, and then matching C with C1 with the call (MATCH C <blist'> C1 <blist>). If this call is successful, the following stack is formed:

(((P1 P2 P3).<blist'>) ((C2).<blist>) ...)

Thus problem C1 has been reduced to problems P1, P2 and P3 as modified by the binding list blist'. PROVE now applies PROVE to this stack in the hope that all the subproblems in it can be solved.

In the event that the axiom to be applied is (C), that is, the axiom is just a fact, the new stack that is formed is

((().<blist'>) ((C2).<blist>) ...)

When PROVE is called with this stack, it removes the first subproblem stucture and begins working on problem C2.

C. The MATCH Function

The MATCH function is a version of the unification algorithm that has been modified so that renaming of variables and substitutions of variable values back into formulas are avoided. The key idea is that the identity of a variable is determined by both the variable name and the binding list on which its value will be stored. The value of a variable is also a pair: the term that will replace the variable and the binding list associated with the term. The binding list associated with the term is used to find the values of variables occurring in the term when needed. Notice that variables do not have to be renamed because MATCH is always called (initially) with two distinct binding lists, giving distinct identities to the variables in the two expressions to be matched, even if the same variable name occurs in both of them.

MATCH assigns a value to a variable by CONSing it to the CAR of the variable's binding list using the RPLACA function; it also puts that binding list on the list bound to the Lisp variable SAVE in PROVE. This is done so that the effects of MATCH can be undone when PROVE backtracks to recover from a failed application of an axiom.

VII EFFICIENCY

HCPRVR is surprisingly efficient for its simplicity. The compiled code fits in 2000 octal words of binary programming space and runs as fast as the Prolog interpreter. Although the speed can be further improved by more sophisticated programming, we have not done so because it is adequate for our present needs. A version of HCPRVR has been written in C; it occupies 4k words on a PDP11/60 and appears to run about half as fast as the compiled Lisp version does on a DEC KI10.

The most important kind of efficiency we have noticed, however, is program development efficiency, the ease with which logic programs can be written and debugged. We have found it easier to write natural language processing systems in logic than in any other formalism we have tried. Grammar rules can be easily written as axioms, with an unrestricted mixture of syntactic and non-syntactic computations. Furthermore, the same grammar rules can be used for parsing or generation of sentences with no change in the algorithm that applies them. Other forms of natural language processing are similarly easy to program in logic, including schema instantiation, question-answering and text summary. We have found HCPRVR very useful for gaining experience in writing logic programs.

REFERENCES

[1] Kowalski, R. A. "Algorithm = logic + control." CACM 22, 7, July, 1979, 424-436.

[2] Warren, D. H., L. M. Pereira, and F. Pereira. "PROLOG - the language and its implementation compared with lisp." Proc. Symp. AI and Prog. Langs., SIGPLAN 12, 8/SIGART 64, August, 1977, 109-115.