PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

SOME PRACTICAL PROPERTIES OF LOGIC PROGRAMMING INTERPRETERS

D. R. Brough[1] and Adrian Walker[2]

[1]Department of Computing, Imperial College, London, England.

[2]IBM Research Laboratory, San Jose, California, USA.

ABSTRACT

A practical problem for Prolog programmers is that, while there may be a finite number of answers to a question put to a program, a standard interpreter may in some cases produce an infinite sequence of repeated answers, or may loop indefinitely without printing an answer. While experienced programmers can often find ways to avoid this behavior, it would be better to correct the interpreter rather than to change individual programs.

This paper introduces a class of simple programs, and shows that there is an interpreter that terminates with the correct answer for each program in the class. However, the interpreter is inefficient. It is then shown that two modified top-down interpreters, relying on goal termination and on rule termination criteria respectively, are each better than a standard Prolog interpreter, in the sense that they halt and print the correct answer for a larger class of simple programs, but that neither is better than the other. However, any terminating, strictly top down, left-to-right interpreter misses some answers on a program that is outside the scope of a standard Prolog interpreter. We conclude that interpreters with a bottom up component appear promising.

1 INTRODUCTION

Logic programming (Kowalski 1979) can be viewed as the writing of executable specifications in logic. A specification, or logic program, can be executed by an interpreter, as in (Roberts 1982, Clark and McCabe 1984, Ennals 1983), or by a combined interpreter and compiler, as in (Pereira, Pereira and Warren 1978). The semantics of logic as a programming language guarantee, in principle, that an answer is derivable from a logic program if and only if it is logically implied by the program (van Emden and Kowalski 1976). However, for reasons of space-time efficiency, the standard Prolog interpreters search for answers using a strategy (top down, left-to-right, depth first) which, for certain programs, enters an infinite loop.

For a decidable problem, an experienced programmer can avoid this behaviour by changing the source program. However, to do so, it is necessary to be aware of the way the interpreter works. Thus one is no longer writing an executable specification. Also, if the source program is changed, then it becomes more complicated to generate explanations of answers (Walker 1983a,b) in terms of the original specification. So it would be better to improve the interpreter (Kowalski 1982) rather than to change individual programs.

A first approach to an improved interpreter is to modify the basic top down depth first strategy by terminating a branch in a proof tree, based on some property of the tree so far. Two criteria that have been studied are goal termination (Brough 1978) and rule termination (Walker 1983a). In goal termination a branch is stopped at a repeated goal, while in rule termination, a branch is stopped if it could only be extended by using a rule instance which covers a previously used rule instance.

This paper compares three top down depth first interpreters, denoted by I_0, I_G and I_R. I_0 corresponds to a standard Prolog interpreter, I_G is goal terminating, and I_R is rule terminating. Their behaviour is compared over a family of Prolog programs which we call Simple Knowledge Bases (SKBs). An SKB contains no function symbols, has only definite clauses (no negation in a premise), and has only ground assertions. A question put to an SKB is understood as a request to compute the set of all possible answers, which is always a finite set of terms. Our choice of SKBs corresponds to many real Prolog programs, keeps our proofs simple, and is also justified by the fact that one of our results points to a shortcoming of any strictly top down depth first interpreter. As we shall show, there is a terminating (but inefficient) algorithm that correctly answers any question put to an SKB. Hence it is not a priori unreasonable to look for an efficient interpreter which is terminating and correct for the SKBs.

For present purposes, one interpreter is better than another if, whenever the second halts and prints out a correct answer to a question put to a SKB, then so does the first, and there is at least one SKB for which the first interpreter gets the right answer and the second does not. (The right answer to a question may be the empty set.) We shall show that the goal terminating (I_G) and rule terminating (I_R) interpreters are each better than standard Prolog interpreter (I_0), but that neither is better than the other. We shall also show that there is an SKB, which is outside the scope of a standard Prolog interpreter, for which any terminating strictly top down left-to-right depth first interpreter will fail to find some answers. This result indicates that interpreters with a bottom up component (Walker 1981) may be worth further study.

The next section of this paper contains definitions. Section 3 gives some examples, Section 4 establishes our results, and Section 5 consists of conclusions. Sections 3 and 5 outline the main ideas in the paper, while Sections 2 and 4 give the technical details.

2 DEFINITIONS

As a framework for our results, we need definitions of a Simple Knowledge Base, of a question, of the correct answer to a question, and of three kinds of top down interpreter. This section gives formal definitions, which are illustrated with examples in Section 3. A *Simple Knowledge Base* (SKB) is a finite set of rules and facts. A *rule* is a definite clause, i.e. it is of the form

$$A \leftarrow B_1 \,\&\ldots\&\, B_m , \qquad m > 0$$

where A, B_1 ,..., B_m are positive literals, for example P(u, y), S(c), R(a, x). a, b, c... denote constants and u, v, w... are variables. In a rule in an SKB, each variable in A occurs in at least one of the B_i's. A *fact* is a positive ground literal, for example P(a, b), S(c), R(a, d). P, S, and R are *predicate symbols*. A *question* Q is a positive literal.

The *Herbrand base* of an SKB K is defined as

$$H(K) = \{\, P(t_1, \ldots, t_n) \mid P \text{ is a predicate symbol of K, and } t_1, \ldots, t_n \text{ are constants in K} \,\}$$

A *model* of K is a subset M(K) of H(K), such that each rule and each fact of K is true in M(K), in the sense of (van Emden and Kowalski 1976). The *answer* to a question Q put to an SKB K is the set denoted by K and Q, namely

$$D(K, Q) = \{\, Q(t_1, \ldots, t_n) \mid Q(t_1, \ldots, t_n) \in \cap\ M(K) \,\}$$

where $\cap M(K)$ stands for the intersection of all of the models of K. Note that, by (van Emden and Kowalski 1976), D(K, Q) consists of exactly the instances of Q that are logically implied by K. An *interpreter* I is a procedure which, given an SKB K and a question Q either

(i) halts and outputs a (possibly empty) set A, its answer , or
(ii) does not halt.

In the first case, we write I(K, Q) = A. In the second, we say that I(K, Q) *diverges*. (Some real interpreters will output answers without halting. We choose, for simplicity of discussion, to collect all answers and only output them if the underlying computation halts.)

An interpreter I is *sound* if, for all K and Q, I(K, Q) = A implies $A \subseteq D(K, Q)$; it is *correct* if, for all K and Q, I(K, Q) = D(K, Q). An interpreter I_1 *covers* an interpreter I_2 , $I_1 \geq I_2$, if I_1 is sound and, for all K and Q, $I_2(K, Q) = A_2$ implies $I_1(K, Q) = A_1$ where A_1 is a superset of A_2 . I_1 is *better* than I_2, $I_1 > I_2$, if I_1 covers I_2, and there exist K and Q such that $I_1(K, Q) = A_1$, and either $I_2(K, Q)$ diverges or $I_2(K, Q) = A_2$ where A_2 is a proper subset of A_1.

The *most general unifier* s of the two literals Q and Q' is the most general substitution s such that s(Q)=s(Q') (Robinson 1979); we write this as mgu(Q, Q')=s. If t and s are substitutions, we write ts(Q) for t(s(Q)).

The interpreters of interest are defined below. Before giving the formal definition, we set out some notation, and indicate some of the reasons for choosing the definition.

We shall define the result I(K, Q), of running an interpreter I on an SKB K with question Q, as either a finite set of instances of Q, or as "diverges", in the case that the underlying computation loops.

The underlying computation will be done by a function J, that takes K and Q as input. If J halts, it returns a finite set of pairs of the form <B, T>, where B is an instance of Q, and T is a representation of a proof of B from K. Then, I selects just the first elements B of the pairs <B, T> to return as answers. Since there may be arbitrarily many proofs of a given B, J may not halt, in which case I diverges too.

The function J is defined recursively. If it returns a pair <B, B>, then B is a fact in the SKB K over which J is computing. If it returns a pair <B, T>, where T is different from B, then J has made at least one recursion, and has used at least one rule from K. If the right hand side of a rule has more than one literal, then these are evaluated left to

right, with substitutions arising from evaluation of the first literal applied to the second literal before it is evaluated, and so on.

Besides the input arguments K and Q, J has two working arguments containing, repectively, a list of ancestor goals, and a list of ancestor rules. I initially calls J as J(K, Q, nil, nil), that is, with empty lists in the working arguments. If J then uses a rule Q <- R, it calls itself as J(K, R, Q.nil, (Q<-R).nil), and so on. Here the period in Q.nil is a list constructor. The goal and rule lists thus act as stacks; an item is pushed onto each one when J is called, and popped from each one when J returns. So we call them the goal stack, and the rule stack. They are used within J by a predicate called 'Stop'. 'Stop' can be defined to do nothing, or to cause goal termination, or to cause rule termination. The variations of 'Stop' lead to different behaviors for the interpreter, which we shall characterize in Sections 3 and 4.

We are now in a position to define the interpreters of interest. As the definition is somewhat terse, the reader may wish to just scan it now, then return to it after looking at the examples in Section 3.

We say that I is a *preorder interpreter* (i.e. top down left-to-right depth first interpreter), if it is of the form:

$$I(K, Q) = \begin{cases} \{B \mid \langle B, T\rangle \in J(K, Q, nil, nil)\}, \\ \quad \text{if } J(K, Q, nil, nil) \text{ halts} \\ \\ \text{diverges, otherwise} \end{cases}$$

where J(K, Q, G, R) is defined below.

$$J(K, Q, G, R) =$$

$$\{\langle s(Q), s(Q)\rangle \mid Q' \in K, mgu(Q, Q') = s\} \cup$$

$$\{\langle t_n...t_1 s(Q), t_n...t_1 s(Q' \leftarrow P_1 \ \&...\& \ P_n)\rangle \mid (Q' \leftarrow R_1 \ \&...\& \ R_n) \in K,\ mgu(Q, Q') = s,\ \text{not Stop}(s(Q' \leftarrow R_1 \ \&...\& \ R_n), G, R), \text{ and for } j = 1, ..., n\ \langle t_j...t_1 s(R_j), P_j\rangle \in J(K, t_{j-1}...t_1 s(R_j), s(Q').G, s(Q' \leftarrow R_1 \ \&...\& \ R_n).R)\}$$

We assume variables are renamed to avoid coincidental bindings. If <B, T> is in J(K, Q, nil, nil) then T is a *tree* with *result* B.

T is a *proof tree* if it satisfies the four conditions:

(a) T is a tree,

(b) each leaf of T is a fact in K,

(c) if $Q \leftarrow P_1 \ \&... \& \ P_n$ is a subtree of T, then, for j = 1..n, either

$P_j = R_j$, or

$P_j = (R_j \leftarrow R_{j,1} \ \&... \& \ R_{j,m(j)})$,

where

$Q \leftarrow R_1 \ \&... \& \ R_n$ and

$R_j \leftarrow R_{j,1} \ \&... \& \ R_{j,m(j)}$,

are instances of rules in K,

(d) each node of T is ground.

In real Prolog interpreters, facts and rules are selected in lexical order from a program. In J above, we leave the selection order open, as our results will not depend on any particular ordering of clauses in K.

The expressions for I and J define a family of preorder interpreters, from which individual interpreters can be chosen by defining the Stop predicate.

The particular kinds of interpreters for which we shall establish results are

I_0 in which Stop is always false.

I_G goal terminating ;

Stop is true if s(Q') is syntactically identical to an item in the stack of goals G

I_R rule terminating ;

Stop is true if there is a substitution t such that $ts(Q' \leftarrow R_1 \ \&...\& \ R_n)$ is syntactically identical to an item in the stack of rule instances R.

I_G has been studied by (Brough 1978) , and I_R by (Walker 1983a).

The next section gives examples to illustrate the above definitions.

3 EXAMPLES

This section gives some example SKBs, and discusses the definitions given above.

Example 1

$$K_1 = \{A(x, z) \leftarrow P(x, y) \ \& \ A(y, z),\ A(x, z) \leftarrow P(x, z),\ P(a, b),\ P(b, a)\}.$$

K_1 describes the transitive closure A of a relation P whose graph consists of a loop. The first rule for A is right recursive. Clearly, the answer to the question A(u, v) is

$D(K_1, A(u, v)) =$
$\{A(a, b), A(a, a), A(b, a), A(b, b)\}$

The tree

$A(a, a) \leftarrow P(a, b) \;\&\; (A(b, a) \leftarrow P(b, a))$

is a proof tree. Although there are simple proof trees like this for each item in $D(K_1, A(u, v))$, it is the case that $I_0(K_1, A(u, v))$ diverges. Intuitively this is because, for each element of the answer, e.g. A(a, a), there are infinitely many proof trees, and the I_0 interpreter will attempt to explore them all.

However, the I_G interpreter gets the right answer. It does this by maintaining, in the third argument of J, a stack of (copies of) the goals (i.e. node labels) above a given goal in a tree. If a goal is repeated, as is A(a, v) in the partial tree

$A(u, v) \leftarrow P(b, a) \;\&\; (A(a, v) \leftarrow P(a, b) \;\&\; (A(b, v) \leftarrow P(b, a) \;\&\; A(a, v)))$

with the stack A(b, v).A(a, v).A(u, v).nil, then the tree is discarded and never developed into a proof tree. Hence one can check, by cases, that

$I_G(K_1, A(u, v)) = D(K_1, A(u, v))$

i.e. the correct answer is computed. It is straightforward to check that an I_R interpreter also finds the correct answer. □

Example 1 indicates that, for K_1 and the question A(u, v), the I_G interpreter gets the right answer, whereas the I_0 interpreter does not. Clearly, not all of the apparatus in the definition of J is needed to get this effect. Tree construction is in J for expository reasons, while the rule stack is in the last argument of J just to describe rule termination. In the next example, rule termination appears preferable to goal termination.

Example 2

$K_2 = \{A(x, z) \leftarrow A(x, y) \;\&\; A(y, z),$
$A(x, z) \leftarrow P(x, z),\; P(a, b),\; P(b, a)\}$

This SKB is similar to K_1, except that the first rule is both left and right recursive. It is easy to check that the answer to the question A(u, v) is the same as for K_1. The left recursion causes $I_0(K_2, A(u, v))$ to diverge. The I_G interpreter will also diverge, because there are arbitrarily many trees of the form

$A(u, v) \leftarrow (A(u, y_1) \leftarrow (A(u, y_2)$
$\leftarrow (\ldots A(u, y_m) \;\&\; \ldots) \;\&\; A(y_2, y_1)) \;\&\; A(y_1, v))$

in which the goals $A(u, y_i)$ i = 1, 2... never make the stopping predicate (Stop) true.

However, the I_R interpreter keeps a rule stack. When the tree is

$A(u, v) \leftarrow A(u, y_1) \;\&\; A(y_1, v)$

the rule stack is

$(A(u, v) \leftarrow A(u, y_1) \;\&\; A(y_1, v)).\text{nil}$

When the rule instance

$A(u, y_1) \leftarrow A(u, y_2) \;\&\; A(y_2, y_1)$

is selected in J, Stop becomes true since the substitution $s = \{\langle y_1, v\rangle, \langle y_2, y_1\rangle\}$ maps this rule into the rule on the stack. One can check, by cases that $I_R(K_2, A(u, v))$ is equal to $D(K_2, A(u, v))$, i.e. that the correct answer is computed. □

So, I_G finds a correct answer for K_1, whereas I_0 does not. I_R finds a correct answer for K_2, whereas I_G does not.

Example 3

$K_3 = \{A(y, z, x) \leftarrow A(x, y, z),\; A(a, b, c)\}$

The single rule in this SKB defines the rotations A(b, c, a) and A(c, a, b) of the triple abc in the fact A(a, b, c). The answer to the question A(u, v, w) is

$D(K_3, A(u, v, w)) =$
$\{A(a, b, c), A(b, c, a), A(c, a, b)\}$

It is easy to check that $I_0(K_3, A(u, v, w))$ diverges, that $I_G(K_3, A(u,v,w))$ is equal to $D(K_3, A(u, v, w))$, but that $I_R(K_3, A(u, v, w))$ is a proper subset of $D(K_3, A(u, v, w))$. So a standard interpreter finds no answer, a goal terminating interpreter finds the correct answer, while a rule terminating interpreter finds part of the correct answer. □

This section has described the behaviour of three interpreters on three examples. A preorder interpreter I_0 (e.g. a standard Prolog interpreter) does not terminate on any of the examples. The goal terminating interpreter I_G finds the correct answer for Examples 1 and 3, but does not terminate on Example 2. The rule terminating interpreter I_R terminates on all three examples, finds the correct answer for Example 1 and 2, but finds only part of the correct answer on Example 3.

The next section gives some general results about the three interpreters I_0, I_G and I_R.

4 RESULTS

Section 2 defined a simple knowledge base (SKB), a question, the correct answer to a question put to an SKB, and the interpreters I_0 , I_G and I_R. Section 2 also defined the notion that an interpreter I_2 may be better than an interpreter I_1 , written $I_2 > I_1$, over the simple knowledge bases. The definitions were illustrated with examples in Section 3.

This section first shows that there exists an interpreter which, for any question Q put to an SKB K, halts and produces the correct answer. The interpreter described in the proof is clearly too inefficient to be of practical interest, but serves to show that a terminating and correct algorithm exists for answering questions put to SKBs.

Next, it is shown that the interpreters I_0 , I_G, and I_R are sound, i.e. each item in an answer is logically implied by the knowledge base. It is then shown that $I_G > I_0$, and $I_R > I_0$, but that neither $I_G > I_R$ nor $I_R > I_G$. The last result in this section characterizes a limitation of any preorder interpreter.

The first result shows that correct termination is possible over the SKBs.

Theorem 4.1 There exists an interpreter I, that takes as input any SKB K and question Q, and halts with output D(K, Q).

Proof Since K is an SKB, it consists of a finite set of rules and facts, built from a finite set of predicate symbols and a finite set of constants. The required interpreter I operates in the following steps, each of which is clearly effectively computable and terminating, (albeit inefficient):

(i) construct the Herbrand base

$$H(K) = \{ P(t_1, ..., t_n) \mid P \text{ is a predicate symbol of } K,\ t_j \text{ is a constant, } j = 1,...,n \}$$

(ii) construct $M = \{ S \mid S \text{ is a subset of } H(K), S \text{ is a model of } K \}$

(iii) construct $\cap M(K)$ from M.

(iv) $D(K, Q) = \{ Q(t_1, ..., t_n) \mid Q(t_1, ..., t_n) \in \cap M(K) \}$ □

Thus, terminating and correct interpreters for SKBs are possible in principle. Next, we establish soundness of the three interpreters I_0, I_G and I_R.

Theorem 4.2 Each of the interpreters I_0, I_G, I_R is sound.

Proof Let I be an interpreter (as defined in section 2), and suppose I(K, Q) = A. Suppose B is a member of A. Then, by our definition of an interpreter, there is a T such that <B, T> ∈ J(K, Q, nil, nil). By Lemma 4.2.1 below, T is a proof tree with result B. From T, it is easy to construct a derivation of B, (see Lemma 4.2.2 below), in the sense of (van Emden and Kowalski 1976); hence, by their result that B is derivable iff it is in ∩ M(K), it follows that B is in D(K, Q). □

For the following Lemma, we refer to the definition of an interpreter I in terms of a set-valued function J.

Lemma 4.2.1 If <B, T> ∈ J(K, Q, nil, nil), then T is a proof tree.

Proof Let <B, T> ∈ J(K, Q, nil, nil). Then

(A) By the definition of an interpreter, T is a tree. So T satisfies part (a) of the definition of a proof tree.

(B) By definition of J, P is a leaf of T only if P=s(Q), for some s and Q, and there is a fact Q' in K such that P=s(Q)=s(Q'). By definition of K, Q' is ground, so P is ground also. Hence, each leaf of T is a fact in K, so T satisfies part (b) of the definition of a proof tree.

(C) Suppose S is a subtree of T. Then, by the definition of J, S is of the form

$$t_n...t_1s(Q' <\text{-} P_1 \ \&...\& \ P_n) \quad .....(1)$$

Also by definition of J, there is a rule

$$Q' <\text{-} R_1 \ \&...\& \ R_n \quad .....(2)$$

in K, such that, for j=1...n,

$$<t_j...t_1s(R_j), P_j> \in J(K, t_{j-1}...t_1s(R_j), ..., ...) \quad .....(3)$$

If the match in the call to J in (3) is with a fact, then P_j is an instance of R_j (see the definition of J). If the call to J in (3) is recursive, then there is a straightforward induction to show that P_j is an instance of $(R_j <\text{-} ...)$.

So, from (1) and (2), and the just outlined proof that each P_j is either an instance of an R_j, or is an instance of an $(R_j <\text{-} ...)$, T satisfies part (c) of the definition of a proof tree.

(D) Since each fact in K is ground, and since, in each rule $A <\text{-} B_1 \ \& \ ... \ \& \ B_m$ in K, each variable in A occurs in some B_i, it follows easily from (B) and (C) that each node of T is ground. □

Lemma 4.2.2 below will verify the statement "From T, it is easy to construct a derivation of B" in the proof of Theorem 4.2.

Lemma 4.2.2 Let T be a proof tree, constructed by J using a knowledge base K, with result B. Then there is a derivation of B from K.

Proof We exhibit below the top level of a Prolog program which, given a proof tree T with result B, computes the relation derivation(T, Deriv), where Deriv is a derivation of B. It is straightforward to check that, given a proof tree T, constructed by J from an SKB K, the result Deriv is indeed a valid derivation of B from K.

```
op('=>', rl, 5).

derivation(T, T => '<empty>') <- leaf(T).

derivation(T, G => RemDeriv) <-
   absorbLeft(T, RemT) & currGoals(T, G) &
   derivation(RemT, RemDeriv).

currGoals(T<-*, T).

currGoals(T&Ts, G&Gs) <-
   currGoals(T, G) & currGoals(Ts, Gs).

currGoals(T, T) <- leaf(T).

absorbLeft(Goal <- Ts, Ts).

absorbLeft(T & Ts, Ts) <- leaf(T).

absorbLeft(T & Ts, SubTs & Ts) <-
   absorbLeft(T, SubTs).

leaf(T) <- ( T/= (*&*) ) & ( T /= (*<-*) ).
```

(Note: this version of the derivation program was written by A. van Gelder). □

Theorem 4.2 assures us that our interpreters only print items that are logically implied by an SKB. It is then of interest to compare the items printed by different interpreters. It can be shown that if I_0 halts and produces an answer, then I_G halts and produces the same answer.

Theorem 4.3 If $I_0(K, Q) = A$ then $I_G(K, Q) = A$.

Proof We refer to the definition of an interpreter I, in terms of a function J, in section 2. Since

$I_0(K, Q) = A$

J(K, Q, nil, nil) halts. Suppose, during a computation by J, that there is a call of the form

J(K, G_2, P...P..., ...)

Then, from the definition of J, there is a computation of the form

```
...
...
J(K, P1, ..., ...)
J(K, G1, s1(P1)..., ...)
...
J(K, P2, ...s1(P1)..., ...)
J(K, G2, s2(P2)...s1(P1)..., ...)
```

where $s_1(P_1) = s_2(P_2) = P$.

Since the definition of J requires all possible rules to be tried at each call of J, there is, in particular, such a computation with $G_1 = G_2$. It is clear that this computation extends to a computation with arbitrarily many calls to J, contradicting the fact that J(K, Q, nil, nil) halts. So it is the case that there is no call of the form J(K, G2, P...P..., ...) while I_0 is computing.

The only difference between I_0 and I_G, is that Stop is always false in I_0, but is true in I_G if a call of the form J(K, G2, P...P..., ...) is about to occur. Since we have established that there are no such calls, the computations by I_0 and I_G yield the same results. □

Theorem 4.3 assures us that I_G covers I_0, $I_G \geq I_0$. Recall that I_G finds a correct answer for the SKB K_1 of Example 1 whereas I_0 does not. So I_G is better than I_0, $I_G > I_0$.

Corollary 4.4 The goal terminating interpreter (I_G) is better than the interpreter without a stopping criterion (I_0).

One can establish a similar result for the rule terminating interpreter I_R as follows.

Theorem 4.5 If $I_0(K, Q) = A$, then $I_R(K, Q) = A$.

Proof For literals, or rules, P_1 and P_2, write $P_1 \leq P_2$ if there is a substitution f such that $f(P_1) = P_2$. Since $I_0(K, Q) = A$, J(K, Q, nil, nil) halts. Consider a computation $I_0(K, Q)$, and suppose that, during the sub-computation by J, there is a call

J(K, G, .., $s_2(G_2<\text{-}B_2)...s_1(G_1<\text{-}B_1)$..)

such that

$s_2(G_2<\text{-}B_2) \leq s_1(G_1<\text{-}B_1)$ .....(1)

Then, by the definition of J, there is a computation of the form

...
...
J(K, G_1, .., ...)
J(K, .., .., $s_1(G_1<\text{-}B_1)$...)
...
J(K, G_2, .., ...$s_1(G_1<\text{-}B_1)$...)
J(K, .., .., $s_2(G_2<\text{-}B_2)$...$s_1(G_1<\text{-}B_1)$...)
...

That is, the instance $s_1(G_1)$ of G_1 matches some rule R_1, and the instance $s_2(G_2)$ of G_2 matches some rule R_2. Note that, by the hypothesis (1) above, $s_2(G_2) \leq s_1(G_1)$. Hence the computation above may, in particular, select $R_2 = R_1$, with $s_2(R_1) \leq s_1(R_1)$. So it is clear that there is a computation of the above form with arbitrarily many steps in which R_1 is matched to goal instances of the form $s_{i+1}(G_{i+1}) \leq s_i(G_i)$, contradicting the fact that J(K, Q, nil, nil) halts. So, during the computation by J, there is no call

$$J(K, G, .., s_2(G_2<\text{-}B_2)...s_1(G_1<\text{-}B_1)..)$$

such that

$$s_2(G_2<\text{-}B_2) \leq s_1(G_1<\text{-}B_1).$$

The only difference between I_0 and I_R, is that Stop is always false in I_0, but is true in I_R if a call of the form

$$J(K, G, .., s_2(G_2<\text{-}B_2)...s_1(G_1<\text{-}B_1)..)$$

such that

$$s_2(G_2<\text{-}B_2) \leq s_1(G_1<\text{-}B_1)$$

is about to occur. Since we have established that there are no such calls, the computations by I_0 and I_R yield the same results. □

Theorem 4.5 assures us that I_R also covers I_0, i.e. $I_R \geq I_0$. Recall that I_R finds a correct answer for the SKB K_2 of Example 2, whereas I_0 does not. So I_R is better than I_0, $I_R > I_0$.

Corollary 4.6 The rule terminating interpreter I_R is better than the interpreter I_0 without a stopping criterion.

However, it is not the case that $I_G > I_R$ or that $I_R > I_G$. I_G is correct on Example 3, while I_R is not. I_R is correct on Example 2, but I_G is not. Given this situation, we might wish to look for better preorder interpreters for SKBs. The next result shows a limitation of any preorder interpreter, namely, that if such an interpreter terminates on all SKBs, then there is an SKB for which it finds less than the whole correct answer.

Theorem 4.7 If I is a preorder interpreter such that I(K, Q) terminates for for all SKBs K and questions Q, then there is an SKB K' and a question Q' such that I(K', Q') = A' and A' is a proper subset of D(K', Q').

Proof Our proof will make use of a family F of SKBs defined as follows. Let $F = \{K_4(n) \mid n \geq 2\}$ where

$$K_4(n) = \{A(x, z) <\text{-} A(x, y) \,\&\, P(y, z),\ A(x, y) <\text{-} P(x, y)\} \cup \{P(a_j, a_{j+1}) \mid 1 \leq j < n\}.$$

In $K_4(n)$, the rules express the transitive closure of the relation P. The graph of P is a simple straight line with n-1 arcs. The first rule is left recursive.

Since $I(K_4(n), A(u, v))$ terminates for any n, it is clear that the Stop predicate must terminate the tree

$$A(u, v) <\text{-} (A(u, y_1) <\text{-} (A(u, y_2) <\text{-} (...A(u, y_m) \,\&\, ...) \,\&\, P(y_2, y_1)) \,\&\, P(y_1, v)).$$

at some node $A(u, y_m)$. But then

$$I(K_4(m+3), A(u, v)) = A'$$

and, although it is the case that

$$A(a_1, a_{m+3}) \in D(K_4(m+3), A(u, v))$$

we have $A(a_1, a_{m+3}) \notin A'$. Thus the theorem holds with $K' = K_4(m+3)$ and $Q' = A(u, v)$. □

This section has shown that a correct, terminating interpreter for the SKBs exists in principle. Each of the interpreters I_0, I_G, and I_R is sound. The goal terminating and rule terminating interpreters are each better than the interpreter without a Stop criterion. However, neither is better than the other, and any strictly preorder interpreter that terminates must miss part of an answer on an SKB.

5 CONCLUSIONS

One application of logic programming is to retrieve all of the answers to a question from a knowledge base consisting of facts (ground assertions) and rules (definite clauses). The correct answer to a question is the set of instances of the question that are contained in the intersection of the models of the knowledge base.

For reasons of efficiency, most logic interpreters answer a question by a top down search process, starting from the question. Standard Prolog interpreters search for answers top down, left-to-right, and depth first, i.e. in preorder. While the search

space so defined yields weakly complete behaviour in theory (each correct answer is in the space), a preorder interpreter can enter an unbounded recursion before completing the search. The practical consequence of this behaviour is that no final answer is returned at all. A top down breadth first interpreter can also suffer from termination problems when searching for all answers to a question.

This paper has defined a class of Simple Knowledge Bases. We have supplied a notation for characterizing top down interpreters, and we have shown that there is a terminating (but inefficient) algorithm that finds the correct answer to a question put to a Simple Knowledge Base. Two modified preorder interpreters, using goal termination and rule termination, have been studied for Simple Knowledge Bases. The results are that each is better than the standard Prolog interpreter. However, goal termination permits some of the unbounded recursions of Prolog. Rule termination has no unbounded recursions, but can miss some answers on programs that are outside the scope of a standard Prolog interpreter. In fact, neither goal termination nor rule termination is uniformly better than the other at finding all of the answers to a question put to a Simple Knowledge Base.

These results naturally raise the question of whether there is an efficient interpreter that is better than either the preorder goal terminating or preorder rule terminating interpreters. This paper has shown that any preorder interpreter which terminates, and which does so just by examining its partial proof tree, misses some answers. So, while there may be better preorder interpreters, there is a fundamental limit to the strict preorder approach.

Top down preorder interpretation was originally chosen for computational efficiency, but, as we have shown, it is of limited use when all of the answers to a question are needed. Hence it appears interesting to look further at the backchain-iteration method of (Walker 1981) that combines top down and bottom up execution of a logic program. In implementations of backchain-iteration, there can be less computational overhead for termination checking than is needed for the preorder interpreters we have described here. All of the examples in this paper are correctly executed by backchain-iteration.

6 ACKNOWLEDGEMENTS

It is a pleasure to acknowledge conversations with Keith Clark, Ron Fagin, Bob Kowalski, John Lloyd, Alan van Gelder, and David H. D. Warren about this work, and to thank the FGCS84 conference referees for their comments.

7 REFERENCES

Brough, D. Loop Trapping in Logic. Unpublished note, Imperial College, London 1978

Clark, K., and McCabe, F. *Programming in Logic.* Prentice-Hall International, 1984.

Ennals, R. *Beginning micro-Prolog.* Ellis Horwood, 1983.

Kowalski, R. *Logic for Problem Solving.* North Holland, 1979.

Kowalski, R. Logic Programming. Report, Department of Computing, Imperial College, London, 1982.

Pereira, L. M., Pereira, F. C. M. and Warren, D. H. D. User's Guide to DEC System-10 Prolog. Occasional Paper No. 15, Department of Artificial Intelligence, University of Edinburgh, 1978.

Robinson, J. A. *Logic: Form and Function.* North Holland, 1979.

Roberts, G. Waterloo Prolog User's Manual. Department of Computer Science, University of Waterloo, 1982.

van Emden, M.H. and Kowalski, R. The Semantics of Predicate Logic as a programming language. Jour. Assoc. Comp. Mach., 23, 4, 1976, 733-742.

Walker, A. Syllog: A Knowledge Based Data Management System. Report No. 034, Computer Science Department, New York University, 1981.

Walker, A. Prolog/Ex1: An Inference Engine which Explains both Yes and No Answers. Proc 8th Int. Joint Conf. Artificial Intelligence, 1983a.

Walker, A. Syllog: an Approach to Prolog for Non-Programmers. Report RJ 3950, IBM Research Laboratory, San Jose, California, 1983b. To appear in: Logic Programming and its Applications, M. van Caneghem and D. H. D. Warren (Eds.), Ablex, 1984.