A Theory of Modules for Logic Programming

Dale Miller

Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

June 1986

Abstract: We present a logical language which extends the syntax of positive Horn clauses by permitting implications in goals and in the bodies of clauses. The operational meaning of a goal which is an implication is given by the deduction theorem. That is, a goal $D \supset G$ is satisfied by a program $\mathcal{P}$ if the goal G is satisfied by the larger program $\mathcal{P} \cup \{D\}$. If the formula D is the conjunction of a collection of universally quantified clauses, we interpret the goal $D \supset G$ as a request to load the code in D prior to attempting G, and then unload that code after G succeeds or fails. This extended use of implication provides a logical explanation of parametric modules, some uses of Prolog's `assert` predicate, and certain kinds of abstract datatypes. Both a model-theory and proof-theory are presented for this logical language. We show how to build a possible-worlds (Kripke) model for programs by a fixed point construction and show that the operational meaning of implication mentioned above is sound and complete for intuitionistic, but not classical, logic.

1. Implications as Goals

Let A be a syntactic variable which ranges over atomic formulas of first-order logic. Let G range over a class of formulas, called *goal* formulas, to be specified shortly. We shall assume, however, that this class always contains $\top$ (true) and all atomic formulas. The formulas represented by A and G may contain free variables. Given these two classes, we define *definite clauses*, denoted by the syntactic variable D, as follows:

$$D := G \supset A \mid \forall x\ D \mid D_1 \wedge\ D_2$$

A *program* is defined to be a finite set of closed definite clauses. $\mathcal{P}$ will be a syntactic variable for programs. A clause of the form $\top \supset A$ will often be written as simply A.

Let $\mathcal{P}$ be a program. Define $[\mathcal{P}]$ to be the smallest set of formulas satisfying the following recursive definitions.

(i) $\mathcal{P} \subseteq [\mathcal{P}]$.

(ii) If $D_1 \wedge D_2 \in [\mathcal{P}]$ then $D_1 \in [\mathcal{P}]$ and $D_2 \in [\mathcal{P}]$.

This work has been supported by NSF grants MCS8219196-CER, MCS-82-07294, and DARPA N000-14-85-K-0018.

Paper to be presented at the Third IEEE Symposium on Logic Programming, Salt Lake City, Utah, 21 – 25 September 1986.

(*iii*) If $\forall x\ D \in [\mathcal{P}]$ then $[x/t]D \in [\mathcal{P}]$ for all closed terms t.
Here $[x/t]D$ denotes the result of substituting t for free occurrences of x in D.

In the case that G is a closed goal formula and $\mathcal{P}$ a program, we shall use the expression $\mathcal{P} \vdash_O G$ to mean that G can be derived from $\mathcal{P}$, or that G is an output of $\mathcal{P}$. We use the symbol O here to indicate that we are thinking about an *operational* definition of derivation, *i.e.*, one that captures an intuitive sense of computation. No *a priori* relation between $\vdash_O$ and other logical senses of derivation or validity are assumed. We present six proof rules for $\vdash_O$. The first two are related to the structure of definite clauses.

(1) $\mathcal{P} \vdash_O \top$, and
(2) If A is a closed atomic formula, $\mathcal{P} \vdash_O A$ if and only if there is a formula $(G \supset A) \in [\mathcal{P}]$ and $\mathcal{P} \vdash_O G$.

These two proof rules provide the basic elements needed to define recursive procedures. A clause of the form $\forall \bar{x}(G \supset A)$ is treated as a specification of how a procedure, the name of which is the head of A, can nondeterministically call other code, *i.e.*, the formula G.

These assumptions about $\vdash_O$ provide a general framework for looking at logic programs. A logic programming language can be completely defined by describing both the class of goal formulas and how non-atomic goals can be output from a program. For example, we can define a logic equivalent to Horn clauses by letting goal formulas be defined as

$$G := \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G$$

and adding the following proof rules:

(3) $\mathcal{P} \vdash_O G_1 \vee G_2$ if and only if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$.
(4) $\mathcal{P} \vdash_O G_1 \wedge G_2$ if and only if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$.
(5) $\mathcal{P} \vdash_O \exists x G$ if and only if there is some closed term t such that $\mathcal{P} \vdash_O [x/t]G$.

In this context, the logical connectives $\wedge$ and $\vee$ provide for the specification of non-deterministic *and* and *or* branches in the search for a derivation (computation). The quantifier $\exists$ specifies an infinite non-deterministic *or* branch where the disjuncts are parameterized by the set of closed terms.

Notice that a program in this logic programming language is equivalent to a conjunction of positive Horn clauses. For example, the definite clause

$$\forall z \forall y\ [(\exists x\ R(x,y) \wedge P(y,z)) \vee R(z,z) \supset P(z,y)]$$

is equivalent (both classically and intuitionistically) to the formula

$$\forall z \forall y \forall x\ [(R(x,y) \wedge P(y,z)) \supset P(z,y)] \wedge$$
$$\forall z \forall y\ [R(z,z) \supset P(z,y)]$$

Since this normal form exists for this version of definite clauses, the literature concerning the theoretical nature of Horn clauses generally does not present the syntax of this logic in this more general setting. The language we consider next, however, will not have such a simple normal form.

In the rest of this paper, we shall assume that goal formulas have the following syntax

$$G := \top \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G,$$

and that there is the additional proof rule

(6) $\mathcal{P} \vdash_O D \supset G$ if and only if $\mathcal{P} \cup \{D\} \vdash_O G$.

Notice that now the classes of goal formulas and definite clauses are defined by mutual recursion.

Technically, we define a $\vdash_O$-*derivation* of G from $\mathcal{P}$ to be any list of the form

$$(\mathcal{P}_1, G_1), \ldots, (\mathcal{P}_n, G_n)$$

where $n \geq 1$, $\mathcal{P}_1 = \mathcal{P}$, $G_1 = G$, and for $i = 1, \ldots, n$, $\mathcal{P}_i \vdash_O G_i$ is derivable from from 0, 1, or 2 of the later members of this list by some proof rule (1) — (6). Notice that this forces G_n to be $\top$ since this is the only goal that does not require a justification.

Let $\mathcal{P}_1 := \{p(b) \supset r(b,a), \forall x \forall y\ [r(x,y) \supset q(f(x))]\}$. The following is a $\vdash_O$-derivation of the goal $\exists x\ [(\top \supset p(x)) \supset q(f(x))]$ from $\mathcal{P}_1$:

$$\begin{aligned}
\mathcal{P}_1 &\vdash_O \exists x\ [(\top \supset p(x)) \supset q(f(x))] \\
\mathcal{P}_1 &\vdash_O [(\top \supset p(b)) \supset q(f(b))] \\
\mathcal{P}_1, (\top \supset p(b)) &\vdash_O q(f(b)) \\
\mathcal{P}_1, (\top \supset p(b)) &\vdash_O r(b,a) \\
\mathcal{P}_1, (\top \supset p(b)) &\vdash_O p(b) \\
\mathcal{P}_1, (\top \supset p(b)) &\vdash_O \top
\end{aligned}$$

In all further examples, we will drop references to the logical constant $\top$ and will replace expressions of the form $\top \supset A$ with just A. The use of $\top$ as a goal will make some theorems simpler to state and prove so this constant will reappear in the theorems and proofs in Section 4.

Although the operational notion of derivation is intuitive enough, it is natural to ask whether it is, in some sense, logical. Thus consider the following example. Let $\mathcal{P}_2 := \{p(a) \wedge p(b) \supset q\}$. Is there a $\vdash_O$-derivation of $\exists x\ (p(x) \supset q)$ from $\mathcal{P}_2$? Consider the following possible sequence:

$$\begin{aligned}
\mathcal{P}_2 &\vdash_O \exists x\ (p(x) \supset q) \\
\mathcal{P}_2 &\vdash_O p(a) \supset q \\
\mathcal{P}_2, p(a) &\vdash_O q \\
\mathcal{P}_2, p(a) &\vdash_O p(a) \wedge p(b) \\
\mathcal{P}_2, p(a) &\vdash_O p(a) \\
\mathcal{P}_2, p(a) &\vdash_O p(b)
\end{aligned}$$

Regardless of the term used to instantiate the quantifier $\exists x$, there is no way to complete this derivation. While it seems reasonable enough that there is no such derivation, it is important to notice that the formula

$$(p(a) \wedge p(b) \supset q) \supset \exists x\ (p(x) \supset q)$$

is classically provable. Thus classical logic is not sound with respect to the our operational semantics. This notion of definite clause includes non-Horn theories, but this is only problematic if we think of deduction as being classical. For the meantime, we shall not be worried about the logical

meaning of $\vdash_O$ but simply look at its use within programming. We return to the logical significance of $\vdash_O$ in Section 5.

2. Modules for Logic Programs

Implication within this logic language can provide a semantics for some of the side-effect mechanisms needed in a logic programming environment. We shall first consider how we might reimplement Prolog's `consult` predicate. Let `classify`, `scanner`, and `misc` be the names of files containing Prolog code. When these names appear within formulas, assume that they refer to the conjunction of universally quantified definite clauses contained in those files. Now consider the following goal.

$$\vdash_O \texttt{misc} \supset ((\texttt{classify} \supset G_1) \wedge (\texttt{scanner} \supset G_2) \wedge G_3)$$

This goal will cause each of the three goals G_1, G_2, and G_3 to be attempted but each will be attempted using different programs. In particular, this single goal will cause the following goals to be attempted:

$$\texttt{misc}, \texttt{classify} \vdash_O G_1$$
$$\texttt{misc}, \texttt{scanner} \vdash_O G_2$$
$$\texttt{misc} \vdash_O G_3$$

Notice that implication can be used in this fashion to structure the runtime environment of a program. For example, the code present in `classify` is essentially hidden during the evaluation of goal G_2. This is, of course, very desirable if it had been the case that `classify` and `scanner` were written by different people. This mechanism ensures that there will be no conflict between the predicates in these two files. Current implementations of logic programs generally require that all code be loaded into one area before it can be used. Such a lack of modularity is certainly a weakness of most logic programming implementations.

The previous discussion suggests that it would be possible to design a notion of modules for logic programming which is based entirely on the logical meaning of embedded implications. We will introduce a module as a named collection of clauses. For example, the following is a module containing some list manipulation programs. Here we have used common Prolog syntax for representing clauses [5].

```
module lists.
append([],X,X).
append([U|L],X,[U|M]) :- append(L,X,M).
member(X,[X|L]) :- !.
member(X,[Y|L]) :- member(X,L).
memb(X,[X|L]).
memb(X,[Y|L]) :- memb(X,L).
```

The theory of definite clauses we shall consider in this paper does not encompass the cut `!` operation. We shall assume, however, that in our examples it will play a similar role in controlling backtracking in a depth-first interpreter of our extended logical language.

According to our definitions, a goal could be of the form $\exists x\ [D \supset G]$, where the variable x is free in either or both D and G. This suggests that the definite clauses defining a module could contain

free variables. Thus a kind of parametric module is possible. Consider the following parametric module:

```
module sort(Order).
bsort(L1,L2) :-
       append(Sorted,[Big,Small|Rest],L1),
       Order(Big,Small), !,
       append(Sorted,[Small,Big|Rest],BetterL1),
       bsort(BetterL1,L2).
bsort(L1,L1).
```

This example, as well as others presented later, is technically not first-order since we have a variable which is acting as a predicate. This could be corrected in most Prolog implementations by replacing the atom `Order(Big,Small)` with the two goals `G =.. [Order,Big,Small], call(G)`. There is also a more direct and logical way to provide Prolog with predicate variables by basing it on higher-order logic [12]. Either approach could be used to make this a meaningful module.

Given these two modules, a very sensible way to bring them together is illustrated in the following example. Consider sorting the list `[2,3,1]`. The following goal has a $\vdash_O$-derivation, in which the substitution found for x would then be the desired sorted list.

$$\vdash_O \texttt{lists} \wedge \texttt{sort}(<) \supset \exists x\, \texttt{bsort}([2,3,1],x)$$

The module `lists` is needed here since the definition of `bsort` requires using `append`. From the point-of-view of organizing software, it is unfortunate that in order to use the `sort` module it was necessary to explicitly reference the `lists` module which it needed to execute successfully. The author of the `sort` module should be able to *import* those modules which were needed within the `sort` module. Here, embedded implications can again be used to provide this importing mechanism. For example, if we introduce the symbol `=>` to represent implication (`:-` represents the converse of `=>`), we could rewrite the sort module as:

```
module sort(Order).
bsort(L1,L2) :-
   (lists =>
      (append(Sorted,[Big,Small|Rest],L1),
       Order(Big,Small), !,
       append(Sorted,[Small,Big|Rest],BetterL1),
       bsort(BetterL1,L2)
    )).
bsort(L1,L1).
```

Now we would be able to sort our list with the simple goal

$$\vdash_O \texttt{sort}(<) \supset \exists x\, \texttt{bsort}([2,3,1],x)$$

When the body of `bsort` is attempted as a goal, it will come with the module `list` as a hypothesis.

For a more general account of how the importing of modules works, consider the following. Let the notation $\mathcal{P}(\bar{x})$ denote a finite set of definite clauses all of whose free variables are in the list $\bar{x}$. Let the symbols $\mathbf{M}$, $\mathbf{M_1}$, $\mathbf{M_2}$, $\mathbf{M_3}$ be the names of modules. These symbols, just like function symbols and predicates, have an arity and take arguments. The arguments of a module name are used to designate the formal parameters of that module. For our purposes here, the meaning of a module, say $\mathbf{M}(\bar{x})$, is some set of definite clauses $\mathcal{P}(\bar{x})$. This association is established by such syntactic specifications as the following.

module $\mathbf{M_1}$	module $\mathbf{M_2}(x)$	module $\mathbf{M_3}(y, z)$
		import $\mathbf{M_1}$ $\mathbf{M_2}(f(y))$
$\mathcal{P}_1$	$\mathcal{P}_2(x)$	$\mathcal{P}_3(z)$

In the first module, the clauses in $\mathcal{P}_1$ contain no free variables and that module is not parametric. $\mathbf{M_1}$ is simply shorthand for the conjunction of the clauses in $\mathcal{P}_1$. The second module is parametric and the clauses $\mathcal{P}_2$ can contain various free occurrences of the variable x. Finally the third module is both parametric and explicitly imports module $\mathbf{M_1}$ and an instance of $\mathbf{M_2}$. The new syntax for importing modules is only a shorthand for writing certain embedded implications. In the `sort` module above, if `bsort` had required several clauses, each one of them may have required the `lists` module as a hypothesis. To save writing this hypothesis for every clause, we introduce the notion of imports. The intended meaning of module $\mathbf{M_3}$ is the following: For each clause of the form

$$\forall \bar{w}(G \supset A)$$

in $\mathcal{P}_3$ replace it with one of the form

$$\forall \bar{w}((\mathbf{M_1} \wedge \mathbf{M_2}(f(y)) \supset G) \supset A),$$

that is, the body of all clauses in $\mathbf{M_3}$ are relativized by the imported modules. The resulting clauses are then associated with the module name $\mathbf{M_3}$.

There are two facts that are important to point out about this development of modules. First, this notion of modules is a by-product of this extended logic. It has not been added as a separate syntactic feature that an interpreter for $\vdash_O$-derivations would need to understand. The meaning and use of modules could be reduced to uses of embedded implications. Of course, in practice this reduction would not be done so literally. A structure-sharing mechanism for modules would need to be considered. Secondly, the operational behavior of implications presented earlier is enough to guarantee that whenever a module is imported into another module, the imported module is only used privately. No additional safeguards need to be added to force private usage. In other words, if a module does not explicitly import a given module, it does not have direct access to the code in that module.

To illustrate this fact, consider the two modules $\mathbf{M_1}$, which contains just the definite clause $\top \supset p$ (or more simply p), and $\mathbf{M_2}$, which contains the definite clause $q \supset p$. Clearly, there is no $\vdash_O$-derivation of $\mathbf{M_2} \supset p$ since there is no way to prove q within $\mathbf{M_2}$. Notice that if we imported $\mathbf{M_1}$ into $\mathbf{M_2}$, there is still no proof of q in $\mathbf{M_2}$. Hence, if $\mathbf{M_2}'$ is the module which contains $q \supset p$ and imports $\mathbf{M_1}$, there should be no $\vdash_O$-derivation of p from $\mathbf{M_2}'$. This is in fact the case. The

formula represented by $\mathbf{M_2}'$ is $(p \supset q) \supset p$, and it is easy to check that $((p \supset q) \supset p) \supset p$ has no $\vdash_O$-derivation. Thus, even though $\mathbf{M_2}'$ imports a module which claims that p is true, it is not possible to prove p from that module. That is, $\mathbf{M_1}$ is used privately within $\mathbf{M_2}'$. It is worthwhile noting that the formula $((p \supset q) \supset p) \supset p$ is often called Pierce's formula and is well known as a classically true but intuitionistically false formula. The restricted nature of implication in intuitionistic logic is precisely the restriction needed to support private usage of modules when they are imported. In Section 5, we will discuss more about the connections between intuitionistic logic and $\vdash_O$.

3. Giving Side-effects a Scope

In this section we present two examples of how this programming language can utilize embedded implications in order to provide a scoped side-effect mechanism. In a sense, it is possible to simulate temporary asserts. We show two different applications of this. The first involves showing how to memoize a recursive program. The other suggests a way to implement certain kinds of abstract datatypes.

Notice that if A is atomic, $\mathcal{P} \vdash_O A \wedge G$ if and only if $\mathcal{P} \vdash_O A \wedge [A \supset G]$. In the forward direction, this is trivial. The reverse direction requires a proof reminiscent of the cut elimination theorem for first-order logic [9]. This relation to cut elimination suggests that implications may be used in logic programs to actually shorten computations. This feature can be used to "memoize" a program's execution, *i.e.*, results of previous calls to a recursive program are stored for use in later calls. Consider the following Prolog program for computing Fibonacci numbers.

```
fib(0,0).
fib(1,1).
fib(N,F) :- N1 is N-1, N2 is N-2, fib(N1,F1),
            fib(N2,F2), F is F1+F2.
```

Let f_n be the n^{th} Fibonacci number. The length of the only proof in $\vdash_O$ of the goal `fib`(n, f_n) is exponential in n. Consider, however, the following slightly modified program.

```
fib(0,0).
fib(1,1).
fib(N,F) :- fib(U,V), fib(U,V) => fib(N,F).
fib(N,F) :- N1 is N-1, N2 is N-2, fib(N1,F1),
            fib(N2,F2), F is F1+F2.
```

Notice that the third clause is a simple first-order theorem, and as such, adds nothing to the extension of the `fib` predicate. Although this clause is far too non-deterministic to be of any real use in an implementation, it does suggest the following more controlled program.

```
fib(N,M) :- memo(0,0) => memo(1,1) => fiba(N,M,2).
fiba(N,M,I) :- memo(N,M).
fiba(N,M,I) :- N1 is I-1, N2 is I-2, memo(N1,F1),
               memo(N2,F2), F is F1+F2, I1 is I+1,
               memo(I,F) => fiba(N,M,I1).
```

In this last program, a proof of `fib(`n, f_n`)` has a length proportional to n.

Our second example illustrates an approach to approximating the notion of abstract datatypes found in several programming languages. This example is only intended to illustrate how this might be accomplished. This particular subject is certainly more complex than one example can illustrate.

Consider the problem of writing code which manipulates a labeled binary tree. We shall assume that the tree's labels are integers and that the integers labeling the nodes to the left of a given node are smaller than the given node, and that the nodes to the right are larger than the given node. Consider the following two modules:

```
module btree.
make_btree(TreeName,Goal) :-
   btree(TreeName,T) => Goal.
insert(TreeName,N)  :-
   btree_internal => (btree(TreeName,T),
                      insert_btree(N,T)).
traverse(TreeName,L) :-
   btree_internal => (btree(TreeName,T),
                      traverse_btree(T,L)).

module btree_internal.
import lists.
insert_btree(N,bt(N,T1,T2)).
insert_btree(N,bt(M,T,_))  :-
   N < M, insert_btree(N,T).
insert_btree(N,bt(M,_,T))  :-
   N > M, insert_btree(N,T).
traverse_btree(leaf,[]).
traverse_btree(bt(N,L,R), SortedList) :-
  traverse_btree(L,Left), traverse_btree(R, Right),
  append(Left,[N|Right],SortedList).
```

The module `btree_internal` contains the specific information concerning the actual implementation of binary trees. The code in module `btree` contains links into this other module. The goal `make_btree(TreeName,Goal)` will call the goal `Goal` in an environment where `TreeName` is associated to some unspecified binary tree. This tree starts as a logical variable. Successive calls to `insert` can be used to add elements to this tree. Notice that the act of inserting does not produce copies of the tree. The logical variables within the tree are simply specified further. In a sense, this datatype represents *monotone binary trees* — they can grow at the leaves, but once a node is labeled with a value, no change can be made to it. Notice also that when a binary tree is traversed, all the logical variables in the tree are instantiated to the constant `leaf` and no further insertions can be made.

Now consider the following module.

```
module btree_sort.
import btree.
build(TreeName,[]).
build(TreeName,[X|L]) :-
   insert(TreeName,X), build(TreeName,L).
binsort(L,K) :- make_btree(sort,(build(sort,L),
                                traverse(sort,K))).
```

This module shows how a binary sort algorithm could be rewritten using the `btree` data structure. The name `sort` is the name of a binary tree, and all access to that tree is made through that name.

This example uses two aspects of implication. The first is that a temporary side-effect (the binary tree) is stored in the program as the atom `btree(sort,T)`. This tree will disappear, for example, after completing the call to `binsort`. The second is the hiding mechanism of modules. If a module does not import `btree_internal` it will not have access to the code which implements binary trees. Notice that the module `btree` does not explicitly import the module `btree_internal`. It was imported only locally in two of the three clauses. If `btree_internal` had been imported over all three clauses, then the goal, `Goal`, in `make_btree(TreeName,Goal)` would have been given access to the internal representation of binary trees. The import declaration must be used carefully. Notice that it is possible for some module to reimplement the code within `btree_internal` and thus gain access to the representation of binary trees. It would seem that supporting a greater degree of security for abstract datatypes would require the implementation of control primitives which are not formally part of the logic we are investigating. Such primitives are, of course, very important for a full implementation of abstract datatypes.

4. A Model Theory

The main challenge to the construction of models for this language is capturing the fact that a program may grow during its "execution." That is, when trying to determine that $\mathcal{P} \vdash_O G$ holds, it might be necessary to determine that $\mathcal{P}' \vdash_O G'$ holds, where $\mathcal{P}'$ is some extension to $\mathcal{P}$. Thus, to understand a given program, $\mathcal{P}$, it is necessary to understand all programs larger than $\mathcal{P}$ as well. We shall do this by considering the set of all programs and attempt to supply a model for all programs simultaneously.

We shall assume that we have chosen a fixed set of non-logical constants. Over this set of non-logical constants, let $\mathcal{U}$ denote the set of all closed terms (the Herbrand universe) and let $\mathcal{H}$ denote the set of all closed, atomic formulas (the Herbrand base). Let $\mathcal{W}$ be the set of all programs and let any function $I : \mathcal{W} \rightarrow \text{powerset}(\mathcal{H})$ such that $\forall w_1, w_2 \in \mathcal{W}[w_1 \subseteq w_2 \supset I(w_1) \subseteq I(w_2)]$ be called an *interpretation*. An interpretation is simply a mapping which associates to every program a set of "true" atomic formulas which is internally "monotone," *i.e.*, if a program gets larger, the set of associated true atoms can not decrease.

We now define each of the following for interpretations I_1 and I_2.

$$I_1 \sqsubseteq I_2 := \forall w \in \mathcal{W}[I_1(w) \subseteq I_2(w)]$$
$$(I_1 \sqcup I_2)(w) := I_1(w) \cup I_2(w)$$
$$(I_1 \sqcap I_2)(w) := I_1(w) \cap I_2(w)$$

It follows quickly from the fact that the powerset of $\mathcal{H}$ is a complete lattice, that the set of all interpretations is also a complete lattice under $\sqsubseteq$. In this lattice, $\sqcup$ is the join operator and $\sqcap$ is the meet operator. The smallest interpretation, $I_\perp$, is given by setting $I_\perp(w) := \emptyset$ for all $w \in \mathcal{W}$.

We next define a weak notion of satisfiability, $I, w \not\models G$, for closed goal formula G in an interpretation I at a world w.

- $I, w \not\models \top$.
- $I, w \not\models A$ if $A \in I(w)$.
- $I, w \not\models G_1 \wedge G_2$ if $I, w \not\models G_1$ and $I, w \not\models G_2$.
- $I, w \not\models G_1 \vee G_2$ if $I, w \not\models G_1$ or $I, w \not\models G_2$.
- $I, w \not\models \exists x\, G$ if $I, w \not\models [x/t]G$ for some $t \in \mathcal{U}$.
- $I, w \not\models D \supset G$ if $I, w \cup \{D\} \not\models G$.

We now wish to build a single interpretation I such that $\mathcal{P} \vdash_O G$ if and only if G is "true" in I at $\mathcal{P}$, *i.e.*, $I, \mathcal{P} \not\models G$. Such an interpretation will be the result of building a least fixed point. To this end, let T be a function from interpretations to interpretations defined as follows:

$$T(I)(w) := \{A \mid \text{there is a } G \supset A \in [w] \text{ such that } I, w \not\models G\}.$$

It is easy to show that $T(I)$ is an interpretation whenever I is an interpretation.

We first prove two lemmas concerning the predicate $\not\models$. Both these lemmas are proved using induction on the structure of goal formulas. We present the proof for only the second lemma. The proof of the first is similar and simpler.

Lemma 1. *If $I_1 \sqsubseteq I_2$ then $I_1, w \not\models G$ implies $I_2, w \not\models G$.*

Lemma 2. *Let $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ be a sequence of interpretations. If G is a goal, $w \in \mathcal{W}$, and $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G$ then there exists a $k \geq 0$ such that $I_k, w \not\models G$.*

Proof. The proof is by induction on the structure of G. If G is $\top$ then set $k = 0$. If G is atomic, then $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G$ implies that $G \in (\bigsqcup_{i=0}^{\infty} I_i)(w) = \bigcup_{i=0}^{\infty} I_i(w)$. Thus there is a $k \geq 0$ such that $G \in I_k(w)$. Hence, $I_k, w \not\models G$. Now assume that for all non-decreasing sequences of interpretations and all $w \in \mathcal{W}$, the lemma is true for all goal formulas with a given bounded size. We need to consider the following four inductive cases.

Case 1: $G = G_1 \wedge G_2$. Since $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G_1 \wedge G_2$, we have $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G_1$ and $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G_2$. By the inductive hypothesis, there are two non-negative integers l and j such that $I_l, w \not\models G_1$ and $I_j, w \not\models G_2$. Let k be the maximum of l and j. By Lemma 1, we have $I_k, w \not\models G_1$ and $I_k, w \not\models G_2$, and therefore $I_k, w \not\models G_1 \wedge G_2$.

Case 2: $G = G_1 \vee G_2$. Since $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G_1 \vee G_2$, we have $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G_j$ for some $j = 1, 2$. By the inductive hypothesis, there is non-negative integer k such that $I_k, w \not\models G_j$. Thus $I_k, w \not\models G_1 \vee G_2$.

Case 3: $G = \exists x\ G'$. Since $\bigsqcup_{i=0}^{\infty} I_i, w \not\models \exists x\ G'$, we have $\bigsqcup_{i=0}^{\infty} I_i, w \not\models [x/t]G'$ for some $t \in \mathcal{U}$. By the inductive hypothesis, there is non-negative integer k such that $I_k, w \not\models [x/t]G'$. Thus $I_k, w \not\models \exists x\ G'$.
Case 4: $G = D \supset G'$. Since $\bigsqcup_{i=0}^{\infty} I_i, w \not\models D \supset G'$, we have $\bigsqcup_{i=0}^{\infty} I_i, w \cup \{D\} \not\models G'$. By the inductive hypothesis, there is a non-negative k such that $I_k, w \cup \{D\} \not\models G'$. Thus $I_k, w \not\models D \supset G'$. ∎

We now show that T is a monotone and continuous function on the lattice of interpretations.

Lemma 3. *T is monotone; that is, if $I_1 \sqsubseteq I_2$ then $T(I_1) \sqsubseteq T(I_2)$.*
Proof. Assume that $I_1 \sqsubseteq I_2$ and let $w \in \mathcal{W}$ and $A \in T(I_1)(w)$. Thus there is a $G \supset A \in [w]$ such that $I_1, w \not\models G$. By Lemma 1, $I_2, w \not\models G$ so $A \in T(I_2)(w)$. Since w and A were arbitrary, we can conclude that $T(I_1) \sqsubseteq T(I_2)$. ∎

Lemma 4. *T is continuous; that is, if $I_0 \sqsubseteq I_1 \sqsubseteq I_2 \sqsubseteq \ldots$ is a sequence of interpretations, then*

$$\bigsqcup_{i=0}^{\infty} T(I_i) = T(\bigsqcup_{i=0}^{\infty} I_i).$$

Proof. To prove this equality, we prove inclusion in two directions.

Since for any non-negative j, $I_j \sqsubseteq \bigsqcup_{i=0}^{\infty} I_i$, we can apply Lemma 3 to get $T(I_j) \sqsubseteq T(\bigsqcup_{i=0}^{\infty} I_i)$. Since j was arbitrary, we have $\bigsqcup_{i=0}^{\infty} T(I_i) \sqsubseteq T(\bigsqcup_{i=0}^{\infty} I_i)$.

Let $w \in \mathcal{W}$ and $A \in T(\bigsqcup_{i=0}^{\infty} I_i)(w)$. Thus, there is a $G \supset A \in [w]$ such that $\bigsqcup_{i=0}^{\infty} I_i, w \not\models G$. By Lemma 2, there is a non-negative k such that $I_k, w \not\models G$. Thus $A \in T(I_k)(w) \subseteq (\bigsqcup_{i=0}^{\infty} T(I_i))(w)$. Since w and A are arbitrary, we conclude that $T(\bigsqcup_{i=0}^{\infty} I_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} T(I_i)$. ∎

By the Knaster-Tarski theorem [1], the least fixed point of T is given by the equation

$$T^{\infty}(I_{\perp}) := T(I_{\perp}) \sqcup T^2(I_{\perp}) \sqcup T^3(I_{\perp}) \sqcup \ldots.$$

The following theorem can now be proved.

Theorem 5. *If $\mathcal{P}$ is a program and G is a closed goal formula, then $\mathcal{P} \vdash_O G$ if and only if $T^{\infty}(I_{\perp}), \mathcal{P} \not\models G$.*
Proof. We shall only hint at the proof of this theorem. First, assume that $\mathcal{P} \vdash_O G$. Thus there is a $\vdash_O$-derivation of G from $\mathcal{P}$, say $(\mathcal{P}_1, G_1), \ldots, (\mathcal{P}_n, G_n)$, where $\mathcal{P}_1 = \mathcal{P}$ and $G_1 = G$. Prove by induction on $i < n$ that $T^{\infty}(I_{\perp}), \mathcal{P}_{n-i} \not\models G_{n-i}$. Setting $i := n - 1$, yields $T^{\infty}(I_{\perp}), \mathcal{P} \not\models G$.

Secondly, prove by induction on $k \geq 0$ that if $T^k(I_{\perp}), \mathcal{P} \not\models G$ then there is a $\vdash_O$-derivation of G from $\mathcal{P}$. Now assume that $T^{\infty}(I_{\perp}), \mathcal{P} \not\models G$. By Lemma 2, there is a $k \geq 0$ such that $T^k(I_{\perp}), \mathcal{P} \not\models G$, and by the just mentioned fact, we have $\mathcal{P} \vdash_O G$. ∎

A similar theorem for the classical theory of positive Horn clauses is given in [1]. The fixed point result in that paper can be viewed as special case of this fixed point theorem. The interpretations we have presented in this section are often referred to as Kripke models [15]. These models are frequently used to explain the semantics of modal and intuitionistic logic. Since we shall show in the next section that our logic is essentially a sublogic of intuitionistic logic, the presence of Kripke models here is not surprising.

5. Connections to Proof Theory

For the past several decades, logicians have developed and studied many different kinds of logics, including classical, intuitionistic, minimal, modal, relevance, first-order, and higher-order, to name only a fraction. It is a very natural question to ask whether our programming language belongs to this list or if it is a new kind of logic. The purpose of this section is to address that question.

It is worthwhile noting that our emphasis here is different than what is generally found in theoretical discussions of logic programming. The more common approach starts with a specific logic, namely classical first-order logic, and then examines the programming language significance of that logic's meta-theory. Our approach in this paper is the reverse, that is, we first fixed a natural and interesting programming language and then looked for a logic whose meta-theory includes its operational semantics. This step of looking for such a logic is not meant to justify the programming language — it is justified to the extent that it has formal properties and implements important programming features. We wish to make a connection to previously studied logics in order to make connections to existing logic literature.

We have already mentioned that if $\vdash_O$ represents a provability relation in some logic, that logic can not be classical logic. This is also apparent since the model of the last section has a structure quite different from models of classical logic. As we showed in Section 1, classical logic fails to represent our operational semantics since it is too strong, *i.e.*, it proves more formulas than our proposed interpreter would prove. Several weaker versions of classical logic which have been studied, and we shall be concerned with two such logics, namely, minimal logic and intuitionistic logic. We can not present these logics in very much detail. The interested reader is referred to [14] or [16] for more information. Theorems in the rest of this paper will be stated without proof.

Let $\vdash_C, \vdash_I, \vdash_M$ represent provability in, respectively, classical **C**, intuitionistic **I**, and minimal **M** logic. The difference between these provability relations can be summarized roughly as follows. In **M**, negation and implication are rather weak connectives. In particular, it is not possible to prove in **M** any formula of the form

$$(p \wedge {\sim}p) \supset q \qquad (Ax1),$$

where q is different from p and ${\sim}p$. Thus contradictions are, in a sense, local. If all instances of $(Ax1)$ are added to **M**, we would then have **I**. However, this logic is still weaker than **C** since formulas of the form

$$p \vee {\sim}p \qquad (Ax2)$$

are not provable in **I**. However, if we add all forms of $(Ax2)$ to **I**, we would than have **C**.

Theorem 6. *Let $\mathcal{P}$ be a program and G be a closed goal formula. If $\mathcal{P} \vdash_O G$, then $\mathcal{P} \vdash_M G$, $\mathcal{P} \vdash_I G$, and $\mathcal{P} \vdash_C G$.*

This theorem follows immediately from the fact that all the operational "proof rules" (1) through (6) of Section 1 remain true if $\vdash_O$ is replaced with $\vdash_M$, $\vdash_I$, and $\vdash_C$. That is, the operational proof rules we have used are weak enough to be contained in all of these logics.

These three systems of logic, of course, have more proof rules than just (1) through (6). For example, they each contain a rule for arguing from cases; that is, to show that a formula A follows from $B \vee C$, show that A follows from B and A follows from C. The syntax of our programming

language is so restrictive, however, that no proof from a disjunctive statement is ever attempted. If we focus on just our restricted syntax, the additional rules of inference may not be relevant. This is the case for both **M** and **I** but not for **C**. In particular, we can show the following.

Theorem 7. *Let $\mathcal{P}$ be a program and let G be a closed goal formula. $\mathcal{P} \vdash_O G$ if and only if $\mathcal{P} \vdash_M G$ if and only if $\mathcal{P} \vdash_I G$. On the other hand, there exist goal formulas, for example,*

$$(p(a) \wedge p(b) \supset q) \supset \exists x\ (p(x) \supset q) \text{ and}$$

$$((p \supset q) \supset p) \supset p$$

which are provable in **C** *but not in our programming system, and hence, not in* **M** *or* **I**.

It is important to notice that if we restrict programs and goals to the language we described in Section 1 which was equivalent to Horn clause logic, then there is no separation between provability operationally, minimally, intuitionistically, or classically. Only when we strengthen our programming language to include embedded implications do we manage to make some distinctions between these logical languages. In Section 6, we will extend our programming language again slightly. In that extension, both **C** and **I** are too strong, while **M** correctly represents its operational behavior.

Since classical logic is too strong to capture our operational semantics, resolution theorem proving is clearly an inappropriate paradigm for building an actual interpreter for our programming language. Instead, many of the techniques used within natural deduction theorem provers would need to be used. Actually, the description of $\vdash_O$ comes close to describing a complete theorem proving procedure. The only problems occur with rules (2) and (5). Here, a search procedure would need to guess at a closed term, thereby causing potentially infinite branching. The cure for this problem is, of course, to introduce unification to compute most general unifiers and to delay the determination of substitution terms. The approach taken in many implementations of Prolog to the role of unification could be used within this setting. A main difference with this approach, however, is that free variables might appear within program clauses. Hence, the variables within program clauses would need to be marked as being either bound or free. This one difference would, in fact, cause several other changes to the way one looks at Prolog implementations. More discussion of this can be found in [7]. A simplified version of the tableau proof procedure presented by McCarty in [11] could also be used to implement an interpreter for this language.

6. Minimal Logic Negation

Let us now introduce a new logical constant $\perp$ to denote false. Formulas of the form $B \supset \perp$ will be abbreviated as $\sim B$. Notice that the negation of a goal formula is a definite clause and that the negation of a definite clause is a goal formula. It is very easy to show that $p \wedge \sim p \vdash_O \perp$ while it is not the case that $p \wedge \sim p \vdash_O q$. That is, this view of negation is that of minimal logic and not that of intuitionistic logic. More precisely, let $\vdash_{O'}$ be the relation between programs $\mathcal{P}$ and closed goal formulas G which satisfies all the proof rules of Section 1, except that (2) is modified as follows: If A is a closed atomic formula, $\mathcal{P} \vdash_{O'} A$ if and only if either $\mathcal{P} \vdash_{O'} \perp$ or there is a formula $G \supset A \in [\mathcal{P}]$ and $\mathcal{P} \vdash_{O'} G$. Derivations using $\vdash_{O'}$ are more costly that those using $\vdash_O$ since the former requires repeatly checking to see if the current program is "inconsistent," *i.e.*, if the current program proves $\perp$. We have the following theorem:

Theorem 8. *$\mathcal{P} \vdash_O G$ if and only if $\mathcal{P} \vdash_M G$. $\mathcal{P} \vdash_{O'} G$ if and only if $\mathcal{P} \vdash_I G$.*

One way to model negation in a logic programming system is through a meta-logical principle called negation-by-failure. This principle states that a closed atom can be taken as being false if there is no proof of it. While the negation-by-failure principle is much stronger than the negation we have just introduced, we can make a useful partial connection between them. A program $\mathcal{P}$ will be called *consistent* if there is no proof of $\bot$ from $\mathcal{P}$, or equivalently, that $\bot \notin T^\infty(I_\bot)(\mathcal{P})$. Notice that if $\mathcal{P}$ is consistent and $\mathcal{P} \vdash_O \sim A$ then there is no proof of A from $\mathcal{P}$. This suggests the possibility of replacing certain forms of negation-by-failure with the search for proofs of negations. This strategy is illustrated in the following example.

Let us consider a very simple database program. Facts within our database will be simple, closed atomic formulas. Definite clauses of the form $G \supset \bot$ will be used to represent constraints. For example, consider the following few clauses.

```
enrolled(jane,102).
enrolled(bill,100).
⊥ :- enrolled(X,101),enrolled(X,102).
```

This tiny database asserts that Jane and Bill are/have been enrolled in 102 and 100, respectively. There is also a constraint that states that it is inconsistent for the same person to be enrolled in both 101 and 102. Now consider the following simple database program.

```
db :- read(Command), do(Command), db.
do(enter(F)) :- F => db.
do(retract) :- fail.
do(commit) :- repeat.
do(check(F)) :-
   (F, write(yes),nl,!;
    F => ⊥, write(no),nl,!;
    write('no, but it could be true'),nl).
do(consis) :- (not ⊥, write(yes),!; write(no)), nl.
```

Here, `db` represents an infinitely looping database query and updating program. Providing the command `enter(F)` to `db` makes an update to the current database by calling `db` after it has made `F` an hypothesis. It is possible to retract such updates by using the `retract` command. This command simply fails. All updates will be undone unless a `commit` command had been issued at some previous point. Notice that the `commit` command will always re-succeed (given a depth-first interpreter).

It is the `check` command which is most interesting here. It has a three-valued behavior. Assume for the time being that the database is consistent. The command `check(F)` will first look for a proof of `F`, and if one is found, prints `yes`. Otherwise, a proof for the negation of `F`, *i.e.*, `F =>` $\bot$ is searched for. If a proof is found, then `no` is printed. If neither the positive nor negative form of `F` can be proved, then the database does not contain `F` although it is consistent for `F` to be a fact in some extension of that database. If the current database is not consistent, the conclusions drawn

by the `check` command could be wrong. With respect to the above database, the three commands

```
check(enrolled(jane,102))
check(enrolled(jane,101))
check(enrolled(bill,101))
```

would print the answers "yes," "no," and "no, but it could be true," respectively.

The `consis` command uses negation-by-failure to determine if the current database is consistent. The use of `not` here is meta-logical also and not accounted for by the theory we have presented.

Notice that the model presented in Section 4 is rich enough to model this three valued behavior. Let $\mathcal{P}$ be a consistent program and let A be a closed atom. Clearly we have either $A \in T^{\infty}(I_{\perp})(\mathcal{P})$ or $A \notin T^{\infty}(I_{\perp})(\mathcal{P})$ (the meta-logic of this paper is very classical). The first case is true when `check(`A`)` prints "yes." The later case, however, can be broken into two additional cases. Clearly, there is some program larger than $\mathcal{P}$ in which A is true ($\mathcal{P} \cup \{A\}$, for example). Given our classification of worlds into consistent and inconsistent, we can make further distinctions: Either the only extensions of the world $\mathcal{P}$ which contain A are inconsistent, or this is not so. The first case is true when `check(`A`)` prints "no," and the later case is true when `check(`A`)` prints "no, but it could be true."

7. Related Work

Gabbay and Reyle in [7] have presented a logical language very similar to the one presented here. Their motivation for selecting this logic was largely based on the observation that this language captures more of its own metatheory. For example, the `demo` predicate of [2] could be encoded directly using implication. That is, the goal, `demo(`D, G`)`, which should succeed if the goal G is provable from D, is equivalent to the goal $D \supset G$.

A stronger logical language, which includes full intuitionistic negation and goals which are universally quantified, is investigated by McCarty in [11]. McCarty uses this logic not in a programming language context but as the basis for building knowledge representation and common sense reasoning programs. He also presents a fixed point construction and a tableau proof procedure for his logic.

Warren in [17] investigated a simpler version of this logic as a basis of a "pure" implementation of a database updating program, such as the one in Section 6. He essentially used implications within goals only when the hypothesis of that implication is atomic. His "modal" operator `assume(`A`)@`G notation could be implemented within our language as $A \supset G$. Warren also provides a partial semantics for this operator using possible worlds semantics.

Several papers have dealt with designing modules for logic programming languages. For example, Bowen and Weinberg in [3] have extend the work of Bowen and Kowalski in [2] and presented several very interesting programs using a notion similar to modules. Chomicki and Minsky in [4] have shown the importance of introducing modularity into Prolog programs and developed a rule-based security system for controlling access among various fragments of code. Neither of these papers, however, provided a logical analysis of their respective notions of modules. The theoretical analysis in this paper should provide a basis for such an analysis.

The paper by O'Keefe [13] presents a formal approach to developing modules for Prolog. Much of what he presents in that paper can be captured by the theory presented in this paper. For example, let $\mathbf{M_1}$ be a module containing the definitions (axioms) for the binary predicates `p`, `r`, and `s`. Say we wish to use in module $\mathbf{M_2}$ the predicate `p` but not the predicate `s`, and we wish to define a new binary relation `q` which is the converse of the relation `r`. In our theory, this is possible by having $\mathbf{M_2}$ import a third module $\mathbf{M_3}$, defined below, instead of importing $\mathbf{M_1}$.

```
module M3.
import M1.
p(X,Y) :- p(X,Y).
q(X,Y) :- r(Y,X).
```

Such a module roughly corresponds to one of O'Keefe's *breeze bricks*.

Goguen and Messequer in [10] presented a notion of module for a sorted theory of Horn clauses with equality. Their modules had associated with them Horn clauses and they provided a mechanism of module importing called *enriching*. Their notion of importing is one of accumulation; that is, if module $\mathbf{M_1}$ imports $\mathbf{M_2}$, the clauses associated with $\mathbf{M_2}$ are part of the meaning of $\mathbf{M_1}$. Hence, modules are not imported for simply private use. Instead, modules form a *use hierarchy* which shows which modules are parts of other modules. With the accumulation approach, searching for a clause whose head matches a given atomic goal requires searching through all modules which are reachable in the use hierarchy from the current module no matter how remote such modules are. Using the more restrictive approach of this paper, only those explicitly imported modules are searched. Such a search can be much smaller.

Several researchers have investigated extensions of positive Horn theories in an entirely classical logic setting. The resulting operational and model-theoretic semantics are quite different than those investigated here. For example, the HORNLOG system of Gallier and Raatz [8] permits programs to be general Horn clauses; that is, programs can contain any number of negative clauses. As such, the database constraints described in Section 6 can be written directly as negative Horn clauses. Queries asked of such a system, however, may have "disjunctive" answer substitutions. For example, if HORNLOG were given the query

$$(\sim p(a) \vee \sim p(b)) \supset \exists x \sim p(x),$$

it would succeed and provide the disjunctive answer substitution claiming that x would get either a or b. The corresponding query in the logic presented here would be written as

$$(p(a) \wedge p(b) \supset \bot) \supset \exists x\ (p(x) \supset \bot)$$

and would not be provable. Fitting in [6] used classical logic to investigated an extension of Horn clauses which contained negations within the body of clauses. His model theory for such clauses used *partial models* (also attributed to Kripke). The operator used in building partial models as fixed points, however, was not continuous. Weakening logic from classical to intuitionistic is one way to preserve both simple answer substitutions and continuous fixed point operators in extensions to Horn clause logic.

8. Acknowledgements

We are grateful to Greg Hager, Thorne McCarty, Gopalan Nadathur and Wayne Snyder for valuable discussions and comments on this paper.

9. References

[1] Krzysztof R. Apt and M. H. van Emden, "Contributions to the Theory of Logic Programming" *J.ACM* **29** (1982), 841 – 862.

[2] K. A. Bowen, R. A. Kowalski, "Amalgamating language and metalanguage in logic programming," in *Logic Programming*, ed. Clark and Tarnund, Academic Press, 1982, 153 – 172.

[3] Kenneth A. Bowen, Tobias Weinberg, "A Meta-Level Extension of Prolog," 1985 Symposium on Logic Programming, Boston, 48 – 53.

[4] Jan Chomicki, Naftaly H. Minsky, "Towards a programming environment for large Prolog programs," 1985 Symposium on Logic Programming, Boston, 230 – 241.

[5] W. Clocksin and C. Mellish, *Programming in PROLOG*, Springer – Verlag, Berlin, 1981.

[6] Melvin Fitting, "A Kripke-Kleene Semantics for Logic Programming," Journal of Logic Programming **2** (1985), 295 – 312.

[7] D. M. Gabbay, U. Reyle, "N-Prolog: An Extension of Prolog with Hypothetical Implications. I," Journal of Logic Programming **1** (1984), 319 – 355.

[8] Jean H. Gallier, Stan Raatz, "Logic Programming and Graph Rewriting," 1985 Symposium on Logic Programming, Boston, 208 – 219.

[9] Gerhard Gentzen, "Investigations into Logical Deductions" in *The Collected Papers of Gerhard Gentzen* edited by M. E. Szabo, North-Holland Publishing Co., Amsterdam, 1969, 68 – 131.

[10] J. A. Goguen, J. Meseguer, "EQLOG: Equality, Types and Generic Modules for Logic Programming," Journal of Logic Programming, **1** (1984), 179 – 209.

[11] L. Thorne McCarty, "Fixed Point Semantics and Tableau Proof Procedures for a Clausal Intuitionistic Logic," Technical Report LRP-TR-18, Rutgers University, 1986.

[12] Dale A. Miller, Gopalan Nadathur, "Higher-Order Logic Programming," Third International Logic Programming Conference, Imperial College, London, 1986.

[13] Richard O'Keefe, "Towards an Algebra for Constructing Logic Programs," 1985 Symposium on Logic Programming, Boston, 152 – 160.

[14] Dag Prawitz, *Natural Deduction,* Almqvist & Wiksell, Uppsala, 1965.

[15] Craig Smorynski, "Applications of Kripke models," Chapter V in [16].

[16] Anne S. Troelstra, *Metamathematical investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics 344, Springer-Verlag, Berlin, 1973.

[17] David S. Warren, "Database Updates in Prolog," Proceedings of the International Conference on Fifth Generation Computer Systems, 1984, 244 – 253.