A Logical Analysis of Modules in Logic Programming

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Running Title: A Logical Analysis of Modules

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: a goal $D \supset G$ is provable from a program $\mathcal{P}$ if the goal G is provable from the larger program $\mathcal{P} \cup \{D\}$. This paper explores the qualitative nature of this extension to logic programming. For example, if the formula D is the conjunction 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 aspects of abstract datatypes. Both a model theory and proof theory are presented for this logical language. In particular, we show how to build a Kripke-like model for programs by a fixed point construction and show that the operational meaning of implication mentioned above is sound and complete for intuitionistic logic. We also examine a weak notion of negation which is easily implemented in this language and show how database constraints can be represented by it.

This work has been supported by ARO grant DAAG-29-84-K-0061 and DARPA grant N000-14-85-K-0018.

1. Implications as Goals

We shall assume that our logical language is a first-order language with denumerably many constants, variables, functions symbols, and predicates (at all arities). This language will contain the following logical constants as primitives: $\wedge$ (conjunction), $\vee$ (disjunction), and $\supset$ (implication) are binary connectives, $\forall$ (for all) and $\exists$ (exists) are quantifiers, and $\perp$ (falsehood) is a 0-ary logical constant. When we have the occasion to write negation, $\sim B$, we will assume that it defined as $B \supset \perp$. This convention will be explained in Section 7 when negation is first considered. $\perp$ is not an atomic formula.

Let A be a syntactic variable which ranges over atomic formulas. Let G range over a class of formulas, called *goal* formulas, to be specified shortly. We shall assume, however, that all atomic formulas are also goal formulas. *Definite clauses*, denoted by the syntactic variable D, are defined recursively as:

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

Definite clauses, as well as atomic and goal formulas, may contain free variables. $\mathcal{P}$ will be a syntactic variable for sets of definite clauses. Such a set will often be called a *program.* We shall always assume that a program is a finite set.

Let $\mathcal{P}$ be a set of definite clauses. Define $[\mathcal{P}]$ to be the smallest set of formulas satisfying the following recursive conditions:

- ◦ $\mathcal{P} \subseteq [\mathcal{P}]$.
- ◦ If $D_1 \wedge D_2 \in [\mathcal{P}]$ then $D_1 \in [\mathcal{P}]$ and $D_2 \in [\mathcal{P}]$.
- ◦ If $\forall x\ D \in [\mathcal{P}]$ then $[x/t]D \in [\mathcal{P}]$ for all 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 goal formula and $\mathcal{P}$ a set of definite clauses, 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 subscript 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 $\vdash_O$ below. The first two are related to the structure of definite clauses.

(1) $\mathcal{P} \vdash_O A$ if $A \in [\mathcal{P}]$, and
(2) $\mathcal{P} \vdash_O A$ 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 $\forall\bar{x}(G \supset A)$ (where $\forall\bar{x}$ represents a list of universally quantified variables) 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.

To complete the description of a logic programming language, we need to describe the class of goal formulas and how non-atomic goals can be operationally proved from a program. We can define a logic equivalent to positive Horn clauses by letting goal formulas be defined as

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

and adding the following proof rules:
(3) $\mathcal{P} \vdash_O G_1 \vee G_2$ if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$.
(4) $\mathcal{P} \vdash_O G_1 \wedge G_2$ if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$.
(5) $\mathcal{P} \vdash_O \exists x G$ if there is some 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. The quantifier $\exists$ specifies an infinite non-deterministic *or* branch where the disjuncts are parameterized by the set of terms.

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 operationally equivalent to the definite clause

$$\begin{aligned} &\forall z \forall y \forall x\ [R(x,y) \wedge P(y,z) \supset P(z,y)] \wedge \\ &\forall z \forall y\ [R(z,z) \supset P(z,y)]. \end{aligned}$$

Since this normal form exists for this version of definite clauses, the literature concerning the theoretical nature of positive Horn clauses generally does not present the syntax of this logic in this more general setting. In Section 8 we generalize this normal form result for programs.

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

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

and that there is the additional proof rule
(6) $\mathcal{P} \vdash_O D \supset G$ if $\mathcal{P} \cup \{D\} \vdash_O G$.
The classes of goal formulas and definite clauses are now defined by mutual recursion.

The "proof rules" above are merely desired properties for the proof predicate $\vdash_O$. We now formalize the meaning of $\vdash_O$ by presenting a sequent-style proof system (see [9] and [18]). A *sequent* is a pair, $\Gamma \longrightarrow \Theta$, where both the *antecedent* Γ and the *succedent* Θ are possibly empty sets of formulas. When a particular sequent is being displayed, we often simply enumerate the elements of succedents and antecedents without including set brackets. Furthermore, we will often write Γ, B and B, Γ to denote $\Gamma \cup \{B\}$.

Each of the proof rules (2) — (6) can be written as *inference figures* in the following manner:

$$\frac{\mathcal{P} \longrightarrow G}{\mathcal{P} \longrightarrow A}\ (2) \qquad \frac{\mathcal{P} \longrightarrow G_i}{\mathcal{P} \longrightarrow G_1 \vee G_2}\ (3)$$

$$\frac{\mathcal{P} \longrightarrow G_1 \qquad \mathcal{P} \longrightarrow G_2}{\mathcal{P} \longrightarrow G_1 \wedge G_2}\ (4) \qquad \frac{\mathcal{P} \longrightarrow [x/t]G}{\mathcal{P} \longrightarrow \exists x\, G}\ (5)$$

$$\frac{\mathcal{P}, D \longrightarrow G}{\mathcal{P} \longrightarrow D \supset G}\ (6).$$

Here, of course, $G \supset A \in [\mathcal{P}]$, $i = 1$ or $i = 2$, and t is some term. In such inference figures, the sequent(s) appearing above the horizonal line are the *upper sequent(s)*, while the sequent appearing below the line is the *lower sequent*.

An **O**-proof for $\mathcal{P} \longrightarrow G$ is a tree whose nodes are labeled with sequents such that (i) the root node is labeled with $\mathcal{P} \longrightarrow G$, (ii) the internal nodes are instances of one of the above inference figures, and (iii) the leaf nodes are labeled with sequents representing proof rule (1), *i.e.*, with sequents of the form $\mathcal{P} \longrightarrow A$ where $A \in [\mathcal{P}]$. Such sequents are called *initial* sequents.

We shall picture proofs as growing up from their root node. The *height* of a proof is the length of the longest path from the root to some leaf. The *size* of a proof is the number of nodes in it.

Inference figures shall refer only to the syntactic objects used to build proofs while proof rules shall refer to properties of the proof predicate $\vdash_O$. The formal meaning of $\vdash_O$ can now be given as: $\mathcal{P} \vdash_O G$ if there is an **O**-proof for $\mathcal{P} \longrightarrow G$. The six proof rules given above are now obvious conclusions. Our interests throughout most of this paper will concern operational provability in its nondeterministic sense. That is, it will not matter if there are several such **O**-proofs or if a depth-first theorem prover could never find such a proof.

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

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\mathcal{P}_1, p(b) \longrightarrow p(b)}{\mathcal{P}_1, p(b) \longrightarrow r(b,a)}\ (2) \qquad \mathcal{P}_1, p(b) \longrightarrow q(a)}{\mathcal{P}_1, p(b) \longrightarrow r(b,a) \wedge q(a)}\ (4)}{\mathcal{P}_1, p(b) \longrightarrow q(f(b))}\ (2)}{\mathcal{P}_1 \longrightarrow [p(b) \supset q(f(b))]}\ (6)}{\mathcal{P}_1 \longrightarrow \exists x\ [p(x) \supset q(f(x))]}\ (5)$$

The size of this proof is 7 and its height is 6.

Although the operational notion of **O**-proof 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 an **O**-proof of $\exists x\ (p(x) \supset q)$ from $\mathcal{P}_2$? Consider the following tree of sequents.

$$\cfrac{\cfrac{\cfrac{\cfrac{\mathcal{P}_2, p(a) \longrightarrow p(a) \qquad \mathcal{P}_2, p(a) \longrightarrow p(b)}{\mathcal{P}_2, p(a) \longrightarrow p(a) \wedge p(b)}\ (4)}{\mathcal{P}_2, p(a) \longrightarrow q}\ (2)}{\mathcal{P}_2 \longrightarrow p(a) \supset q}\ (6)}{\mathcal{P}_2 \longrightarrow \exists x\ (p(x) \supset q)}\ (5)$$

This is not a proof because $\mathcal{P}_2, p(a) \longrightarrow p(b)$ is not an initial sequent. Regardless of the term used to instantiate the quantifier $\exists x$ in the root sequent, this tree cannot be extended to an **O**-proof. While it seems reasonable enough that there is no such proof, 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. (A classical proof of this formula will be presented in Section 6.) Thus classical logic is not sound with respect to the our operational semantics. The actual logical status of $\vdash_O$ will be addressed more fully in Section 6. Until then, we shall simply be concerned with how $\vdash_O$ can be used to interpret programs.

In this paper we will not discuss the specifics of how one might actually implement definite clauses into a Prolog-like language which incorporates such implementation mechanisms as backtracking, logical variables, and unification. It is useful, however, to point out two aspects of how such an implementation would need to differ from a more traditional Prolog interpreter. First, explicit quantifiers need to be used in specifying programs. In Prolog, quantifiers in Horn clauses are dropped since free variables in them can be thought of as being universally quantified. Similarly, free variables in goals can be thought of as being existentially quantified. This is not true, however, of definite clauses. For example, only the second of the two goal formulas $\exists x(\forall y\ p(x, y) \supset q)$ and $(\forall x \forall y\ p(x, y) \supset q)$ may be derived from the definite clause $p(a, c) \wedge p(b, c) \supset q$. If explicit quantifiers were dropped, there would be no way to differentiate correctly between these two goals.

If unification and logical variables are used in the standard way to delay and determine substitutions, then a second difference with traditional Prolog systems is forced, namely, that program clauses as well as goals can contain logical variables. For example, consider a goal of the form $\exists x\ (D(x) \supset G(x))$, *i.e.*, an existentially quantified implication in which the quantified variable can be free on both sides of the implication. If this quantifier is replaced with a logical (free) variable, say X, then the code $D(X)$ must be added to the current program clauses before attempting to prove $G(X)$. When unification provides substitutions for X, both program clauses and goals must be updated accordingly.

2. Storing Successful Goals

Before we examine how our extended use of implication can be used to implement modules, we first show how implications can be used to provide a scoped and temporary `assert` mechanism. We illustrate this mechanism by considering how to implement a form of memoization.

Notice that if A is atomic, $\mathcal{P} \vdash_O A \wedge G$ if and only if $\mathcal{P} \vdash_O A \wedge [A \supset G]$. The proof of this statement in the forward direction is trivial. The proof of the converse is slightly more difficult and is given below. It should be quite clear that the size of an **O**-proof of the second formula can at times be much smaller than for the first formula. With the second formula, A does not need to be reproved each time it is needed in the proof of G.

Consider the following Prolog program for computing Fibonacci numbers. In general, we shall use the syntax of [20] to represent example programs.

```
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.
```

For this example, if n is the integer value of the arithmetic expression t then the sequent $\Gamma \longrightarrow n$ `is` t, for any set Γ, will be permitted as an initial sequent. If f_n denotes the n^{th} Fibonacci number then the size of the only **O**-proof of the goal `fib(`n, f_n`)` is exponential in n.

Consider, however, the following program which employs implicational goals to store previously computed Fibonacci numbers. Here, we have introduced the symbol `=>` to represent implication (`:-` is the converse of `=>`).

```
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, there exists only one proof of `fib(`n, f_n`)` and that proof has a size proportional to n.

We now return to the formal justification of this approach to memoization.

Theorem 1. *If A is atomic, $\Gamma \vdash_O A \wedge G$ if and only if $\Gamma \vdash_O A \wedge [A \supset G]$.*

Proof. This is trivial in the forward direction. For the proof in the reverse direction, assume that $\Gamma \vdash_O A \wedge [A \supset G]$. Thus $\Gamma \vdash_O A$ and $\Gamma, A \vdash_O G$. Let T_1 and T_2 be **O**-proofs for $\Gamma \longrightarrow A$ and $\Gamma, A \longrightarrow G$, respectively. We need to construct an **O**-proof for $\Gamma \longrightarrow G$. If $A \in \Gamma$ then $\Gamma = \Gamma \cup \{A\} \vdash_O G$, so T_2 is an **O**-proof for $\Gamma \longrightarrow G$. Assume that $A \notin \Gamma$. Build a tree, T_3, by removing A from all the antecedents of sequents in T_2. This tree may not be a proof for $\Gamma \longrightarrow G$ because there may have been initial sequents of the form $\Gamma', A \longrightarrow A$ in T_2 which are now of the form $\Gamma' \longrightarrow A$. These, of course, may no longer be initial sequents. Since $\Gamma \subseteq \Gamma'$, adding the definite clauses $\Gamma' - \Gamma$ to all the antecedents of sequents in the proof T_1, we can get an **O**-proof for $\Gamma' \longrightarrow A$. Thus, if we add to the top of all such non-initial leaves of T_3 the appropriate augmented version of T_1, we will have an **O**-proof of $\Gamma \longrightarrow G$. Hence, $\Gamma \vdash_O A \wedge G$. ▮

After we prove the equivalence of $\vdash_O$ to intuitutionistic logic in Section 6, this theorem is easily proven by just noting that $A \wedge G$ is intuitutionistically equivalent to $A \wedge (A \supset G)$.

The set of formulas which can be memoized in this fashion is slightly larger than the set of atomic formulas. Such a formula must have the structure of a definite clause and of a goal formula. Let M be a syntactic variable which ranges over the set of formulas described by

$$M := A \mid M \supset A \mid M_1 \wedge M_2.$$

It is easy to show that a formula is a definite clause and a goal formula if and only if it belongs to the class of formulas satisfying this recursive definition. Notice, for example, that all propositional, positive Horn clauses are such formulas.

3. Implementing Modules

Consider now how we might reimplement Prolog's `consult` predicate with implicational goals. 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 with different programs. In particular, this single goal will cause the following goals to be attempted:

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

Implicational goals can, therefore, be used 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 programs. 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 such 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 implications. To this end, we will introduce *modules* as named collections of clauses. For example, the following is a module containing some list manipulation programs.

```
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 are considering in this paper does not encompass the cut `!` operation. We shall assume, however, that in our examples it will play the same role in controlling backtracking as it does in Prolog.

According to our definitions, a goal could be of the form $\exists x\ [D(x) \supset G(x)]$, where the variable x is free in either or both $D(x)$ and $G(x)$. This suggests that the definite clauses defining a module can 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(Small,Big), !,
       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 because we have a variable which is acting as a predicate. This could be alternatively implemented using such extra-logical tricks as replacing the atom `Order(Small,Big)` with the two goals `G =.. [Order,Small,Big], call(G)`. There is also a more direct and logical way to provide Prolog with predicate variables by placing it in a higher-order logic [14, 15]. Either approach could be used to give a meaning to this module.

Consider using the `sort` module to sort the list `[2,3,1]` in ascending order. Since `bsort` uses the `append` predicate, the `lists` module must be used along with the `sort` module. The answer substitution for x in the following goal would provide the desired sorted list.

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

It is unfortunate that in order to use the `sort` module it was necessary to explicitly reference the `lists` module, which `sort` needed to execute successfully. It should be possible for the author of the `sort` module to *import* those modules which are needed in the `sort` module. Implications can again be used to provide this importing mechanism. For example, we could rewrite the sort module as:

```
module sort(Order).
bsort(L1,L2) :-
   (lists =>
      (append(Sorted,[Big,Small|Rest],L1),
       Order(Small,Big), !,
       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 `lists` as a hypothesis. Notice that the scope of the `lists` module in the first `bsort` clause is over the entire body of that clause. Clearly, its scope could be restricted to just cover the two `append` goals. This is desirable especially because, as the code is written above, the `lists` module is imported for each recursive call to `bsort`. For the purpose of this paper, we will assume that importing code which is already imported will have no cost. This is sensible from our theoretical viewpoint because the sequent $\Gamma, D \longrightarrow D \supset G$ is provable from the sequent $\Gamma, D \longrightarrow G$.

For the convenience of later examples, we introduce the following definitions. Let the notation $\mathcal{P}(\bar{x})$ denote a finite set of definite clauses all of whose free variables are in the variable 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 and predicate symbols, 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 between module names and definite clauses can be established by the following syntactic specifications:

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

The first module is not parametric because the clauses in $\mathcal{P}_1$ contain no free variables. $\mathbf{M_1}$ will simply be 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 $\mathbf{M_1}$ 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 the body of every clause, we introduce this syntax for 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 $\mathbf{M_3}$. Of course, variable capture must be avoided here. If y is a member of the list $\bar{w}$, the bound variables would need to be changed before forming this new clause.

There are two facts that are important to point out about this development of modules. Firstly, this notion of modules is a by-product of our definite clauses. It has not been added as a separate syntactic feature that an interpreter for **O**-proofs 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 p and $\mathbf{M_2}$ which contains the definite clause $q \supset p$. Clearly, there is no **O**-proof of

$\mathbf{M_2} \supset p$ because there is no way to prove q within $\mathbf{M_2}$. If $\mathbf{M_2}$ were to import $\mathbf{M_1}$, there would still be no proof of q in $\mathbf{M_2}$. Thus, if $\mathbf{M_2}'$ is the module which contains $q \supset p$ and imports $\mathbf{M_1}$, there should be no $\mathbf{O}$-proof 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 the goal $((p \supset q) \supset p) \supset p$ has no $\mathbf{O}$-proof. 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 Peirce'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 6, we will discuss more about the connections between intuitionistic logic and $\vdash_O$.

As a final example, we give an example of how implication can be used to provide a block structured approach to writing code. The following single definite clause is a definition of the list reverse program.

```
rev(L,K) :- (all K rev1([],K,K),
             all X all L all K (rev1([X|L],K,ACC) :-
                                   rev1(L,K,[X|ACC])))
            => rev1(L,K,[]).
```

Our syntax for this definite clause cannot, of course, conform to the syntax of [20] because we need to embed quantified expressions in the body of definite clauses. Instead, we used the expression `all X P` is correspond to the logical notation $\forall x\ P$. The only time the code for the `rev1` predicate is accessible is during the evaluation of the `rev` predicate.

4. Modules as Interfaces

Modules can also be used to explicitly export and hide program code in other modules. For example, let module $\mathbf{M_1}$ contain clauses for the binary predicates `p` and `s` and for the trinary predicate `r`. Consider the following module.

```
module M₂.
import M₁.
q(X,Y) :- p(X,Y).
t(X,Y) :- r(f(Y),[],X).
```

Let $\mathbf{M_0}$ be some module which imports $\mathbf{M_2}$. $\mathbf{M_0}$ will have access to the code for predicate `p` but only through the name `q`. $\mathbf{M_0}$ will not, however, have access to the code for predicate `s` since this is not mentioned in $\mathbf{M_2}$. Also, only a certain instance of the code for the predicate `r` is provided by the predicate `t`. Such an *interfacing* module is very similar in spirit to what O'Keefe's called a *breeze brick* [17].

In this example, we have chosen to export the predicate p in $\mathbf{M_1}$ as the predicate q. We could have exported p as p. For example, let $\mathbf{M_3}$ be identical to $\mathbf{M_2}$, except that we replace the define clause `q(X,Y) :- p(X,Y)` with `p(X,Y) :- p(X,Y)`. These clauses are, of course, simply shorthand for the definite clauses $\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset q(x,y))$ and $\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset p(x,y))$. From an abstract point-of-view, this choice is immaterial; that is, we have the following equalities:

$$\{\langle t,s\rangle \mid \mathbf{M_1} \vdash_O p(t,s)\} = \{\langle t,s\rangle \mid \mathbf{M_2} \vdash_O q(t,s)\} = \{\langle t,s\rangle \mid \mathbf{M_3} \vdash_O p(t,s)\}.$$

Proofs for $\mathbf{M_3} \longrightarrow p(t,s)$ can be, however, different from those for $\mathbf{M_2} \longrightarrow p(t,s)$. For example, consider the following tree of sequents:

$$\cfrac{\cfrac{\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset p(x,y)), \mathbf{M_1} \longrightarrow p(t,s)}{\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset p(x,y)) \longrightarrow \mathbf{M_1} \supset p(t,s)}\ (6)}{\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset p(x,y)) \longrightarrow p(t,s)}\ (2).$$

The search for an **O**-proof of $p(t,s)$ from $\mathbf{M_3}$ gives rise to the search of a proof for $p(t,s)$ from $\mathbf{M_3}$ and $\mathbf{M_1}$. Since $\mathbf{M_3}$ does not affect the extension of p, this is, in the abstract sense, the same as proving $p(t,s)$ from $\mathbf{M_1}$. An interpreter for **O**-proofs could, however, loop infinitely often by repeatedly using the $\mathbf{M_3}$ code instead of using the $\mathbf{M_1}$ code. Exporting p as q is one way to get around this problem. Consider the following tree of sequents:

$$\cfrac{\cfrac{\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset q(x,y)), \mathbf{M_1} \longrightarrow p(t,s)}{\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset q(x,y)) \longrightarrow \mathbf{M_1} \supset p(t,s)}\ (6)}{\forall x \forall y\ ((\mathbf{M_1} \supset p(x,y)) \supset q(x,y)) \longrightarrow q(t,s)}\ (2).$$

In this case, an interpreter is forced to examine the code in $\mathbf{M_1}$.

Returning to our first example in this section: if `p` was implemented in $\mathbf{M_1}$ with calls to the predicate `s`, only `p`, exported as `q`, will be available from $\mathbf{M_2}$, not `s`. Hence, it is possible to hide some aspects of the actual implementation of programs. This is an essential feature in supporting the usual software engineering notion of an *abstract datatype*.

To illustrate this more directly, consider implementing binary trees which are labeled with integers in such a way that integers labeling the nodes to the left (resp. right) of a given node are smaller (larger) than the label of the given node. The following module represents a particular implementation of inserting and traversing such trees.

```
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).
```

Here a particular structuring of binary trees is chosen: the term `bt(N,T1,T2)` represents a non-terminal node labeled with `N` and which has left-subtree `T1` and right-subtree `T2`. Leaf nodes are represented by the constant `leaf`.

Such a representation is, of course, largely arbitrary and probably of little significance to the rest of the program using this module. In such a case, it would be desirable to hide the actual implementation of such binary trees. We can do this with the following interfacing module.

```
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)).
build(TreeName,[]).
build(TreeName,[X|L]) :-
         insert(TreeName,X), build(TreeName,L).
```

This module associates binary trees with user supplied names; this association is stored as a "temporary assert" using atomic facts of the form `btree(TreeName,Tree)`. It also provides the basic manipulations (insertions and traversals) for such named trees. The goal `make_btree(TreeName,Goal)` will call the goal `Goal` in an environment where `TreeName` is associated with some unspecified binary tree. This tree is initialized as a logical variable. Successive calls to `insert` are used to add integer labels to this tree. The action of inserting such labels does not produce new copies of the binary tree; the logical variables within the tree are simply specified further. In a sense, this datatype represents *monotone binary trees*: such trees can grow at the leaves, but once a node is labeled with a value, no change can be made to it. 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.

The following module describes a sorting procedure which uses the `btree` datatype.

```
module btree_sort.
import btree.
binsort(L,K) :- make_btree(sort,(build(sort,L),
                                 traverse(sort,K))).
```

The name `sort` is the name of a binary tree, and all access to that tree is made through that name. This code is independent of the actual implementation of binary trees. For example, if `btree_internal` attempted to balance binary trees on certain insertions, the meaning of the above code would not change. A user who imports `btree_sort` will only have access to the binary predicate `binsort`. No other code is made available.

The module `btree` does not explicitly import the module `btree_internal`: it was imported only locally in two of its 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. This would defeat the hiding mechanism. The import declaration often provides imported modules with too broad a scope. Explicit uses of `=>` can provide for more selective scoping.

Although this development of abstract datatypes seems to capture many of the standard software engineering notion of abstract datatypes, it does not capture all of them. For example, modules do not provide a perfect hiding mechanism. It is always possible to get at the internal structure of data objects by guessing at their implementation. For example, some module could reimplement the code within `btree_internal` and thus gain access to the representation of binary trees. Also, by simply issuing the goal `btree(Name,Tree)` in the right context, a user could get access to all the stored binary trees. Of course, this user would need to know the fact that they were stored in this fashion. It would seem that supporting a greater degree of security for abstract datatypes would require further extensions to either the logic or control primitives of the programming language we are investigating. In [15] we present a further extension to definite clauses, namely the addition of universally quantified goals, which can be used to hide access to term constructors in datatypes. A discussion of this mechanism is beyond the scope of this paper.

The code in two different modules might overlap, *i.e.*, they could both provide definitions for a common predicate. For example, assume modules $\mathbf{M_1}$ and $\mathbf{M_2}$ both contain clauses for the binary predicate p. There are occasions when the desired meaning for p is simply the union of the clauses in $\mathbf{M_1}$ and $\mathbf{M_2}$. This would be the case if these modules were parts of a database of facts and we wished to do a simple accumulation of these fact. This could be done with the following query:

$$\vdash_O \mathbf{M_1} \wedge \mathbf{M_2} \supset \exists x \exists y\, p(x,y).$$

Another way to bring these modules together is to view each one as being different versions of some program. We might then wish to find solutions to p with respect to the version in $\mathbf{M_1}$ and then with respect to the version in $\mathbf{M_2}$. This could be done by first building the module:

```
module M3.
q(X,Y) :- M1 => p(X,Y).
q(X,Y) :- M2 => p(X,Y).
```

and then using the query:

$$\vdash_O \mathbf{M_3} \supset \exists x \exists y\, q(x,y).$$

When a program is placed in a module, it is often the case that we would like to think of that program as being completely defined within that module. That is, in all environments, that module should always have the same meaning. As our one example above showed, it is possible for a second module to be used along with a given module in such a way that the clauses in the two modules accummulate. For some programs, such an addition of foreign clauses could greatly change the original program's meaning. When this is not desired, it is possible to add control primitives which specify that programs in modules cannot be extended. This could be done by explicitly using a `!, fail` combination in an interfacing module. For example, consider the following module:

```
module closedsort(Order).
import sort(Order).
binsort(L1,L2) :- bsort(L1,L2); !, fail.
```

This small module defines a version of bubble sort which cannot be extended with foreign clauses: a depth-first interpreter could never access them.

There are numerous other observations which could be made about how this notion of module could be used to structure programs and search. We shall now, however, turn away from such concerns to study some of the formal aspects of definite clauses and **O**-proofs.

5. Model Theory

In this section we shall present an alternative description of operational derivability. We shall do this by constructing a set-theoretic structure, *i.e.*, a kind of *model*, such that $\mathcal{P} \vdash_O G$ if and only if G is satisfied in that model. The main challenge to the construction of such a model is 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 of $\mathcal{P}$. From the model theory perspective, this means that an interpretation of $\mathcal{P}$ would depend on interpretations associated with larger programs.

We therefore use an approach to semantics which is inspired by possible-worlds or Kripke models: we attempt to interpret all programs simultaneously by using a single large Kripke-like interpretation. It is then possible to inductively build a single interpretation which can then be viewed as interpreting all programs. This single interpretation will be the least fixed point of a suitable operator. The references [5] and [19] contain more on Kripke models.

We shall assume that we have chosen a fixed set of non-logical constant, function, and predicate symbols. 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 cannot decrease.

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

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

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 notion of satisfiability, $I, w \models G$, for closed goal formula G in an interpretation I at a program w.

- $I, w \models A$ if and only if $A \in I(w)$.
- $I, w \models G_1 \vee G_2$ if and only if $I, w \models G_1$ or $I, w \models G_2$.
- $I, w \models G_1 \wedge G_2$ if and only if $I, w \models G_1$ and $I, w \models G_2$.
- $I, w \models \exists x\, G$ if and only if $I, w \models [x/t]G$ for some $t \in \mathcal{U}$.
- $I, w \models D \supset G$ if and only if $I, w \cup \{D\} \models G$.

An interpretation can be thought of as a large collection of models which is indexed by programs. Thus, $I, w \models G$ means that G is true in the model located in I at program w. The truth of the goal $D \supset G$ in the model located at w is given by the truth of the goal G in the model located at $w \cup \{D\}$. This is how the growth of programs is captured.

We now wish to build a single interpretation I such that $\mathcal{P} \vdash_O G$ if and only if $I, \mathcal{P} \models G$. 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:

$$T(I)(w) := \{A \mid A \in [w] \text{ or there is a closed clause } G \supset A \in [w] \text{ such that } I, w \models G\}.$$

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

We first state two lemmas concerning the predicate $\models$. 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 2. *If $I_1 \sqsubseteq I_2$ then $I_1, w \models G$ implies $I_2, w \models G$ for all $w \in \mathcal{W}$.*

Lemma 3. *Let $I_1 \sqsubseteq I_2 \sqsubseteq I_3 \sqsubseteq \ldots$ be a sequence of interpretations. If G is a goal, $w \in \mathcal{W}$, and $\bigsqcup_{i=1}^{\infty} I_i, w \models G$ then there exists a $k \geq 1$ such that $I_k, w \models G$.*

Proof. The proof is by induction on the structure of G. If G is atomic, then $\bigsqcup_{i=1}^{\infty} I_i, w \models G$ implies that $G \in (\bigsqcup_{i=1}^{\infty} I_i)(w) = \bigcup_{i=1}^{\infty} I_i(w)$. Thus there is a $k \geq 1$ such that $G \in I_k(w)$. Hence, $I_k, w \models G$. Now assume that for all increasing 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 cases.

Case 1: $G = G_1 \wedge G_2$. Since $\bigsqcup_{i=1}^{\infty} I_i, w \models G_1 \wedge G_2$, we have $\bigsqcup_{i=1}^{\infty} I_i, w \models G_1$ and $\bigsqcup_{i=1}^{\infty} I_i, w \models G_2$. By the inductive hypothesis, there are two positive integers l and j such that $I_l, w \models G_1$ and $I_j, w \models G_2$. Let k be the maximum of l and j. By Lemma 2, we have $I_k, w \models G_1$ and $I_k, w \models G_2$, and therefore $I_k, w \models G_1 \wedge G_2$.

Case 2: $G = G_1 \vee G_2$. Since $\bigsqcup_{i=1}^{\infty} I_i, w \models G_1 \vee G_2$, we have $\bigsqcup_{i=1}^{\infty} I_i, w \models G_j$ for some $j = 1, 2$. By the inductive hypothesis, there is a positive integer k such that $I_k, w \models G_j$. Thus $I_k, w \models G_1 \vee G_2$.

Case 3: $G = \exists x\, G'$. Since $\bigsqcup_{i=1}^{\infty} I_i, w \models \exists x\, G'$, we have $\bigsqcup_{i=1}^{\infty} I_i, w \models [x/t]G'$ for some $t \in \mathcal{U}$. By the inductive hypothesis, there is a positive integer k such that $I_k, w \models [x/t]G'$. Thus $I_k, w \models \exists x\, G'$.

Case 4: $G = D \supset G'$. Since $\bigsqcup_{i=1}^{\infty} I_i, w \models D \supset G'$, we have $\bigsqcup_{i=1}^{\infty} I_i, w \cup \{D\} \models G'$. By the inductive hypothesis, there is a positive integer k such that $I_k, w \cup \{D\} \models G'$. Thus $I_k, w \models D \supset G'$. ∎

Notice that $I_\perp, \mathcal{P} \models G$ holds for no $\mathcal{P}$ and G, and that $T(I_\perp), \mathcal{P} \models G$ holds if and only if G is atomic and $G \in [\mathcal{P}]$. We now show that T is a monotone and continuous function on the lattice of interpretations.

Lemma 4. *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 either $A \in [w]$, in which case $A \in T(I_2)(w)$, or there is a closed clause $G \supset A \in [w]$ such that $I_1, w \models G$. By Lemma 2, $I_2, w \models G$ 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 5. *T is continuous; that is, if $I_1 \sqsubseteq I_2 \sqsubseteq I_3 \sqsubseteq \ldots$ is a sequence of interpretations, then*

$$\bigsqcup_{i=1}^{\infty} T(I_i) = T(\bigsqcup_{i=1}^{\infty} I_i).$$

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

Since $I_j \sqsubseteq \bigsqcup_{i=1}^{\infty} I_i$ for any j, $j \geq 1$, we can apply Lemma 4 to get $T(I_j) \sqsubseteq T(\bigsqcup_{i=1}^{\infty} I_i)$. Since j was arbitrary, we have $\bigsqcup_{i=1}^{\infty} T(I_i) \sqsubseteq T(\bigsqcup_{i=1}^{\infty} I_i)$.

Let $w \in \mathcal{W}$ and $A \in T(\bigsqcup_{i=1}^{\infty} I_i)(w)$. If $A \in [w]$ then $A \in T(I_j)(w)$ for any j, $j \geq 1$ and, clearly, $A \in (\bigsqcup_{i=1}^{\infty} T(I_i))(w)$. Otherwise, there is a closed clause $G \supset A \in [w]$ such that $\bigsqcup_{i=1}^{\infty} I_i, w \models G$. By Lemma 3, there is a k such that $k \geq 1$ and $I_k, w \models G$. Thus $A \in T(I_k)(w) \subseteq (\bigsqcup_{i=1}^{\infty} T(I_i))(w)$. Since w and A are arbitrary, we conclude that $T(\bigsqcup_{i=1}^{\infty} I_i) \sqsubseteq \bigsqcup_{i=1}^{\infty} T(I_i)$. ▮

This theorem and proof would remain true if we lifted the restriction that programs were finite. Making programs infinite will not harm the continuity of T.

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 6. *If $\mathcal{P}$ is a closed program and G is a closed goal formula, then $\mathcal{P} \vdash_O G$ if and only if $T^{\infty}(I_{\perp}), \mathcal{P} \models G$.*

Proof. First we show that $\mathcal{P} \vdash_O G$ implies $T^{\infty}(I_{\perp}), \mathcal{P} \models G$. We do this by showing by induction on k that for all programs $\mathcal{P}$ and all closed goal formulas G, if the sequent $\mathcal{P} \longrightarrow G$ has a **O**-proof of height k then $T^{\infty}(I_{\perp}), \mathcal{P} \models G$.

Base case: $k = 1$. G must be atomic and $G \in [\mathcal{P}]$. Thus, $G \in T^1(I_{\perp})(\mathcal{P}) \subseteq T^{\infty}(I_{\perp})(\mathcal{P})$ and $T^{\infty}(I_{\perp}), \mathcal{P} \models G$.

Inductive case: $k \geq 1$. We need to consider the cases where the last inference in the **O**-proof of $\Gamma \longrightarrow G$ is an instance of inference figure (2) — (6). We present only the cases corresponding to inference figures (2) and (6). The other three cases are straightforward.

Assume that the last inference figure was (2). Then G is atomic and there is a $G' \supset G \in [\mathcal{P}]$ such that $\mathcal{P} \longrightarrow G'$ has an **O**-proof of height $k-1$. By the inductive hypothesis, $T^{\infty}(I_{\perp}), \mathcal{P} \models G'$. By the definition of T, $G \in T(T^{\infty}(I_{\perp}))(\mathcal{P}) = T^{\infty}(I_{\perp})(\mathcal{P})$. Hence, $T^{\infty}(I_{\perp}), \mathcal{P} \models G$.

Assume that the last inference figure was (6). Thus G is of the form $D \supset G'$ and $\mathcal{P}, D \longrightarrow G'$ has an **O**-proof of height $k-1$. By the inductive hypothesis, $T^{\infty}(I_{\perp}), \mathcal{P} \cup \{D\} \models G'$. Hence, $T^{\infty}(I_{\perp}), \mathcal{P} \models D \supset G'$.

We now prove the converse. Let $\mathcal{P}$ and G be such that $T^k(I_{\perp}), \mathcal{P} \models G$ for some $k \geq 1$ and assume that this is the smallest such value for k. If G contains $n \geq 0$ occurrences of logical connectives and quantifiers, we then attach to the pair $\mathcal{P}, G$ the ordinal measure $\omega \cdot (k-1) + n$. We now prove by induction on this measure that for all programs $\mathcal{P}$ and all closed goal formulas G, if the measure of the pair $\mathcal{P}, G$ is α then $\mathcal{P} \vdash_O G$.

Base case: the measure of $\mathcal{P}, G$ is 0 ($= \omega \cdot 0 + 0$). G is, therefore, atomic and $T^1(I_\perp), \mathcal{P} \models\!\!\models G$. But then $G \in [\mathcal{P}]$ and $\mathcal{P} \vdash_O G$.

Inductive case: the measure of $\mathcal{P}, G$ is $\omega \cdot \alpha + \beta > 0$. This case must be divided between the case where this ordinal is a limit ordinal or not, *i.e.*, $\beta = 0$ or $\beta > 0$.

Subcase $\beta = 0$. Hence, $\alpha > 0$ and G is atomic. Thus $T^{\alpha+1}(I_\perp), \mathcal{P} \models\!\!\models G$ and $G \in T^{\alpha+1}(I_\perp)(\mathcal{P})$. By the definition of T, either $G \in [\mathcal{P}]$, in which case $\mathcal{P} \vdash_O G$ is immediate, or there is a closed $G' \supset G \in [\mathcal{P}]$ such that $T^\alpha(I_\perp), \mathcal{P} \models\!\!\models G'$. In the second case, the ordinal $\omega \cdot (\alpha - 1) + \beta'$, where β' is the number of logical connectives in G', is smaller than $\omega \cdot \alpha + \beta$. By the inductive hypothesis, $\mathcal{P} \vdash_O G'$ has an **O**-proof, so $\mathcal{P} \vdash_O G$ has an **O**-proof by proof rule (2).

Subcase $\beta > 0$. In this case, G is not atomic. This part of the proof, therefore, breaks into four cases, one for each possible top-level connective of G. The proof of each of these cases is straigthforward and omitted here.

Finally, if $T^\infty(I_\perp), \mathcal{P} \models\!\!\models G$, by Lemma 3, there is a $k \geq 1$ such that $T^k(I_\perp), \mathcal{P} \models\!\!\models G$. Hence, the pair $\mathcal{P}, G$ has a measure and by the preceding inductive proof, this implies that $\mathcal{P} \vdash_O G$. ∎

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 a special case of Theorem 6.

6. Proof Theory

At this point, we would like to know if our programming language represents a new logical system or if it is an instance of some other known logical system. The purpose of this section is to address this 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 instead to discover if our logic has been studied previously in the logic literature.

We have already mentioned that if $\vdash_O$ represents a provability relation in some logic, that logic cannot be classical logic. For another illustration of this, consider the goal formula $G_1 \vee (D \supset G_2)$. Our operational semantics interprets this goal as one which will succeed if either G_1 succeeds or if G_2 succeeds given the program in D. The scope of D is clearly over only G_2. Classical logic, however, does not support this interpretation of this goal. For example, in classical logic, the following equivalences are tautologous:

$$\begin{aligned} G_1 \vee (D \supset G_2) &\equiv G_1 \vee {\sim}D \vee G_2 \\ &\equiv {\sim}D \vee G_1 \vee G_2 \\ &\equiv (D \supset G_1) \vee G_2 \\ &\equiv (D \supset G_1) \vee (D \supset G_2) \end{aligned}$$

The classical equivalence of $p \supset q$ with ${\sim}p \vee q$ undermines the intended meaning of implication as providing an environment building mechanism. We will show how two weaker logics, minimal logic and intuitionistic logic, can be used to provide a justification for our intended interpretation of the logical connectives.

We now present a different set of inference figures for sequents than those introduced in Section 1. This proof system is taken from [18].

$$\frac{\Gamma \longrightarrow \Delta, B \qquad \Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \wedge C}\ \wedge\text{-R} \qquad \frac{B, C, \Delta \longrightarrow \Theta}{B \wedge C, \Delta \longrightarrow \Theta}\ \wedge\text{-L}$$

$$\frac{B, \Delta \longrightarrow \Theta \qquad C, \Delta \longrightarrow \Theta}{B \vee C, \Delta \longrightarrow \Theta}\ \vee\text{-L}$$

$$\frac{\Gamma \longrightarrow \Delta, B}{\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R} \qquad \frac{\Gamma \longrightarrow \Delta, C}{\Gamma \longrightarrow \Delta, B \vee C}\ \vee\text{-R}$$

$$\frac{\Gamma \longrightarrow \Theta, B \qquad C, \Gamma \longrightarrow \Delta}{B \supset C, \Gamma \longrightarrow \Delta \cup \Theta}\ \supset\text{-L} \qquad \frac{B, \Gamma \longrightarrow \Theta, C}{\Gamma \longrightarrow \Theta, B \supset C}\ \supset\text{-R}$$

$$\frac{\Gamma, [x/t]P \longrightarrow \Theta}{\Gamma, \forall x\ P \longrightarrow \Theta}\ \forall\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/t]P}{\Gamma \longrightarrow \Theta, \exists x\ P}\ \exists\text{-R}$$

$$\frac{\Gamma, [x/y]P \longrightarrow \Theta}{\Gamma, \exists x\ P \longrightarrow \Theta}\ \exists\text{-L} \qquad \frac{\Gamma \longrightarrow \Theta, [x/y]P}{\Gamma \longrightarrow \Theta, \forall x\ P}\ \forall\text{-R}$$

$$\frac{\Gamma \longrightarrow \Theta, \perp}{\Gamma \longrightarrow \Theta, B}\ \perp\text{-R}$$

The proviso that the variable y is not free in any formula of the lower sequent is also assumed for the $\exists$-L and $\forall$-R figures.

A *proof* for the sequent $\Gamma \longrightarrow \Theta$ is a finite tree constructed using these inference figures and such that the root is labeled with $\Gamma \longrightarrow \Theta$ and the leaves are labeled with *initial sequents*, *i.e.*, sequents $\Gamma \longrightarrow \Theta$ such that the intersection $\Gamma \cap \Theta$ contains either $\perp$ or an atomic formula.

Sequent systems of this kind generally have three *structural* figures which we have not listed. Two such figures, interchange and contraction, are not necessary here because the antecedents and succedents of sequents are sets instead of lists. Hence, the order and multiplicity of formulas in sequents are not made explicit. If an antecedent is of the form Γ, B, it may be the case that $B \in \Gamma$; that is, a formula in an antecedent or succedent has an arbitrary multiplicity. The third common structural inference figure is that of thinning: from a given sequent one may add any additional formulas to the succedent and antecedent. The following lemma, which can be proved easily by induction, establishes a form of antecedent thinning which we will find useful. We will not, however, introduce thinning as a separate inference figure.

Lemma 7. *Let Ξ be a proof of $\Gamma \longrightarrow \Theta$ and let Γ' be a set of formulas. Let $\Xi + \Gamma'$ be the tree of sequents obtained by adding Γ' to the antecedent of all sequents in Ξ. Then $\Xi + \Gamma'$ is a proof for $\Gamma \cup \Gamma' \longrightarrow \Theta$.*

We define the following three kinds of proofs. Any proof will also be called a **C**-proof. Any **C**-proof such that every sequent in it has a singleton set for it succedent is also called an **I**-proof. Furthermore, an **I**-proof in which no instance of the $\perp$-R inference figure appears is also called an **M**-proof. Sequent proofs in classical **C**, intuitionistic **I**, and minimal **M** logics are represented by, respectively, by **C**-proofs, **I**-proofs, and **M**-proofs. Finally, we write $\Gamma \vdash_C B$, $\Gamma \vdash_I B$, and $\Gamma \vdash_M B$, if the sequent $\Gamma \longrightarrow B$ has, respectively, a **C**-proof, **I**-proof, or **M**-proof. It follows immediately that $\Gamma \vdash_M B$ implies $\Gamma \vdash_I B$ implies $\Gamma \vdash_C B$. If the set Γ is empty, it will be dropped entirely from the left side of these three predicates. See [5, 9, 18, 21] for more on intuitionistic and minimal logics.

The following is a **C**-proof for a formula we considered in Section 1.

$$
\begin{prooftree}
\AxiomC{$p(a), p(b) \longrightarrow p(a)$}
\AxiomC{$p(a), p(b) \longrightarrow p(b)$}
\RightLabel{$\wedge$-R}
\BinaryInfC{$p(a), p(b) \longrightarrow p(a) \wedge p(b)$}
\AxiomC{$p(a), p(b), q \longrightarrow q$}
\RightLabel{$\supset$-L}
\BinaryInfC{$p(a) \wedge p(b) \supset q, p(a), p(b) \longrightarrow q$}
\RightLabel{$\supset$-R}
\UnaryInfC{$p(a) \wedge p(b) \supset q, p(a) \longrightarrow q, p(b) \supset q$}
\RightLabel{$\supset$-R}
\UnaryInfC{$p(a) \wedge p(b) \supset q \longrightarrow p(a) \supset q, p(b) \supset q$}
\RightLabel{$\exists$-R}
\UnaryInfC{$p(a) \wedge p(b) \supset q \longrightarrow p(a) \supset q, \exists x\, (p(x) \supset q)$}
\RightLabel{$\exists$-R.}
\UnaryInfC{$p(a) \wedge p(b) \supset q \longrightarrow \exists x\, (p(x) \supset q)$}
\end{prooftree}
$$

Similarly, the following is a **C**-proof of Peirce's formula, mentioned at the end of Section 3.

$$\dfrac{\dfrac{\dfrac{p \longrightarrow p, q}{\longrightarrow p, p \supset q}\;\supset\text{-R} \qquad p \longrightarrow p}{(p \supset q) \supset p \longrightarrow p}\;\supset\text{-L}}{\longrightarrow ((p \supset q) \supset p) \supset p}\;\supset\text{-R.}$$

Neither of these formulas have **I**-proofs.

In classical proofs, there may be more than one formula in the succedent, and the "processing" of any one of these formulas may interact with the others in the succedent. This is very clear if we return to the goal formula, $G_1 \vee (D \supset G_2)$, presented earlier. Consider the following tree of sequents:

$$\dfrac{\dfrac{\dfrac{D \longrightarrow G_1, G_2}{\longrightarrow G_1, D \supset G_2}\;\supset\text{-R}}{\longrightarrow G_1, G_1 \vee (D \supset G_2)}\;\vee\text{-R}}{\longrightarrow G_1 \vee (D \supset G_2)}\;\vee\text{-R}$$

This reduction of the root sequent to $D \longrightarrow G_1, G_2$ loses the scoping restriction of D over G_2. Permitting more than one formula on the right works against our intented interpretation of the logical connectives.

Theorem 8. *Let $\mathcal{P}$ be a set of definite clauses and let G be a goal formula. If $\mathcal{P} \vdash_O G$, then $\mathcal{P} \vdash_M G$, $\mathcal{P} \vdash_I G$, and $\mathcal{P} \vdash_C G$.*

Proof. Each of the inference figures (2) through (6) are derivable in **M** in the following way. Figure (2) is a combination of $\forall$-L, $\wedge$-L, $\supset$-L, and an initial sequent; figure (3) is $\vee$-R; figure (4) is $\wedge$-R; figure (5) is $\exists$-R; figure (6) is $\supset$-R. An initial sequent of $\vdash_O$ is derivable from $\forall$-L, $\wedge$-L, and an initial sequents of this system. Since all these inference figures are derivable in the weakest system, **M**, they are also derivable in **I** and **C**. ▮

The proofs of the next two lemmas are straightforward inductions on the height of proofs.

Lemma 9. *Let $\mathcal{P}$ be a set of definite clauses and Θ be a set of goal formulas. Any proof of a sequent $\mathcal{P} \longrightarrow \Theta$ contains no instances of the inference figures $\vee$-L, $\exists$-L, and $\forall$-R.*

Lemma 10. *Let $\mathcal{P}$ be a set of definite clauses. There is no* **I***-proof of the sequent $\mathcal{P} \longrightarrow \bot$.*

From Lemma 10 it follows that any **I**-proof of $\mathcal{P} \longrightarrow \Theta$ cannot contain any instances of the $\bot$-R inference figure. Hence, for a set of definite clauses $\mathcal{P}$ and a goal formula G, $\mathcal{P} \vdash_I G$ if and only if $\mathcal{P} \vdash_M G$.

We now wish to show that if $\mathcal{P} \longrightarrow G$ is intuitionistically provable, it is then operationally provable. There are several overlaps between the **O** and **I** proof systems, but there are two substantial differences. First, there are no **O**-proof inference figures which correspond to $\forall$-L and $\wedge$-L. This difference is compensated by the use of $[\mathcal{P}]$ instead of $\mathcal{P}$ in proof rules (1) and (2). Secondly, and more importantly, the match between inference figures (2) and $\supset$-L is not direct. Given the sequent $G' \supset A, \mathcal{P} \longrightarrow G$, (2) is applicable only if $G = A$ while $\supset$-L has no such restriction. In fact, $\supset$-L generates an entire subproof for $A, \mathcal{P} \longrightarrow G$ which is not present in instances of inference figure (2). As a result of this difference, we need to make the following definitions and observations.

Instances of $\supset$-L in a proof are at the root of two smaller proofs. These two proofs are called the *left subproof* and *right subproof* of this instance of $\supset$-L. An instance of $\supset$-L in a proof is *simple* if its right subproof has height 1. Otherwise, the instance is *complex.* A proof in which all instances of $\supset$-L are simple is a *simple proof.* It is simple instances of $\supset$-L which correspond to uses of inference figure (2). Lemma 12 establishes that simple **I**-proofs are sufficient. First, however, we need to prove the following lemma.

Lemma 11. *Assume that Ξ is an **I**-proof of the form*

$$\dfrac{\begin{array}{c}\Xi_1\\ \mathcal{P} \longrightarrow G'\end{array} \qquad \begin{array}{c}\Xi_2\\ A, \mathcal{P} \longrightarrow G\end{array}}{G' \supset A, \mathcal{P} \longrightarrow G}\ \supset\text{-}L,$$

*where Ξ_1 and Ξ_2 are simple **I**-proofs. There exists a simple **I**-proof for*

$$G' \supset A, \mathcal{P} \longrightarrow G.$$

Proof. Our proof shows how a complex instance of $\supset$-L can be converted to simple instances by moving it backwards through the inference figures in Ξ_2. To do this, we need to show that $\supset$-L "commutes" with all inference figures in which Ξ_2 can terminate. Formally, the proof is by induction on the height of Ξ_2. If the height is 1, then the instance of $\supset$-L above is simple and we are finished. Assume that the height of Ξ_2 is greater than 1. We now show that this complex instance of $\supset$-L commutes with the last inference figure in Ξ_2. We consider the cases for each such inference figure.

Cases $\wedge$-R, $\supset$-R, $\vee$-R, and $\exists$-R: Assume that the root inference figure in Ξ_2 is $\wedge$-R. Hence, Ξ is of the form

$$\dfrac{\begin{array}{c}\Xi_1\\ \mathcal{P} \longrightarrow G'\end{array} \qquad \dfrac{\begin{array}{c}\Xi_2'\\ A, \mathcal{P} \longrightarrow G_1\end{array} \qquad \begin{array}{c}\Xi_2''\\ A, \mathcal{P} \longrightarrow G_2\end{array}}{A, \mathcal{P} \longrightarrow G_1 \wedge G_2}\ \wedge\text{-R}}{G' \supset A, \mathcal{P} \longrightarrow G_1 \wedge G_2}\ \supset\text{-L}.$$

Consider the following **I**-proof:

$$\dfrac{\dfrac{\overset{\Xi_1}{\mathcal{P} \longrightarrow G'} \qquad \overset{\Xi_2'}{A, \mathcal{P} \longrightarrow G_1}}{G' \supset A, \mathcal{P} \longrightarrow G_1}\;\supset\text{-L} \qquad \dfrac{\overset{\Xi_1}{\mathcal{P} \longrightarrow G'} \qquad \overset{\Xi_2''}{A, \mathcal{P} \longrightarrow G_2}}{G' \supset A, \mathcal{P} \longrightarrow G_2}\;\supset\text{-L}}{G' \supset A, \mathcal{P} \longrightarrow G_1 \wedge G_2}\;\wedge\text{-R.}$$

The two instances of $\supset$-L above both have left and right subproofs which are simple and are such that their right subproofs are shorter than Ξ_2. The inductive hypothesis can be applied to each subproof to finish this case. The case when the last inference figure of Ξ_2 is either $\supset$-R, $\vee$-R, or $\exists$-R is similar.

Cases $\forall$-*L and* $\wedge$-*L*: Assume that the root inference figure in Ξ_2 is $\forall$-L. Hence, Ξ is of the form

$$\dfrac{\overset{\Xi_1}{\forall x\, D, \mathcal{P}' \longrightarrow G'} \qquad \dfrac{\overset{\Xi_2'}{A, [x/t]D, \mathcal{P}' \longrightarrow G}}{A, \forall x\, D, \mathcal{P}' \longrightarrow G}\;\forall\text{-L}}{G' \supset A, \forall x\, D, \mathcal{P}' \longrightarrow G}\;\supset\text{-L,}$$

where $\mathcal{P} = \mathcal{P}' \cup \{\forall x\, D\}$. Consider the following **I**-proof:

$$\dfrac{\dfrac{\overset{\Xi_1 + \{[x/t]D\}}{[x/t]D, \forall x\, D, \mathcal{P}' \longrightarrow G'} \qquad \overset{\Xi_2' + \{\forall x\, D\}}{A, [x/t]D, \forall x\, D, \mathcal{P}' \longrightarrow G}}{G' \supset A, \forall x\, D, [x/t]D, \mathcal{P}' \longrightarrow G}\;\supset\text{-L}}{G' \supset A, \forall x\, D, \mathcal{P}' \longrightarrow G}\;\forall\text{-L.}$$

The instance of $\supset$-L above has left and right subproofs which are simple and is such that its right subproof is shorter than Ξ_2. The inductive hypothesis can be applied to this instance of $\supset$-L to finish this case. The case when the last inference figure of Ξ_2 is $\wedge$-L is similar.

Case $\supset$-*L*: For our final case, assume that the last inference figure of Ξ_2 is $\supset$-L. Ξ is then of the form

$$\dfrac{\overset{\Xi_1}{G_1 \supset A_1, \mathcal{P}' \longrightarrow G'} \qquad \dfrac{\overset{\Xi_2'}{A, \mathcal{P}' \longrightarrow G_1 \qquad A, A_1, \mathcal{P}' \longrightarrow G}}{A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G}\;\supset\text{-L}}{G' \supset A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G}\;\supset\text{-L,}$$

where $\mathcal{P} = \{G_1 \supset A_1\} \cup \mathcal{P}'$. Since, the upper most instance of $\supset$-L is simple, the sequent $A, A_1, \mathcal{P}' \longrightarrow G$ is initial. Hence, either $G \in \mathcal{P}'$, $G = A$, or $G = A_1$. If $G \in \mathcal{P}'$ then a simple **I**-proof for the sequent

$$G' \supset A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G$$

is the one node proof labeled with just this sequent. If $G = A$, then a simple **I**-proof for this sequent would be

$$\frac{\begin{array}{c}\Xi_1\\ G_1 \supset A_1, \mathcal{P}' \longrightarrow G'\end{array} \qquad A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G}{G' \supset A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G} \supset\text{-L.}$$

Finally, if $G = A_1$, then consider the following **I**-proof:

$$\frac{\dfrac{\begin{array}{c}\Xi_1\\ G_1 \supset A_1, \mathcal{P}' \longrightarrow G'\end{array} \qquad \begin{array}{c}\Xi_2' + \{G_1 \supset A_1\}\\ A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G_1\end{array}}{G' \supset A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G_1} \supset\text{-L} \qquad A_1, G' \supset A, \mathcal{P}' \longrightarrow G}{G' \supset A, G_1 \supset A_1, \mathcal{P}' \longrightarrow G} \supset\text{-L.}$$

The lower instance of $\supset$-L is simple while the upper instance is not necessarily simple. Since that instance's right subproof is shorter than Ξ_2, the inductive hypothesis can be used to complete this case. ▮

Lemma 12. *Let $\mathcal{P}$ be a set of definite clauses and let G be a goal formula. If $\mathcal{P} \longrightarrow G$ has an* **I***-proof, it has a simple* **I***-proof.*

Proof. This follows by induction on the number of complex instances of $\supset$-L in an **I**-proof of $\mathcal{P} \longrightarrow G$. If the number of such instances is greater than 0, choose one which has only simple left and right subproofs. Using the preceding lemma, that instance can be removed. The resulting **I**-proof has one fewer complex instances. In this fashion, all complex instances can be removed. ▮

Theorem 13. *Let $\mathcal{P}$ be a set of definite clauses and let G be a goal formula. $\mathcal{P} \vdash_O G$ if and only if $\mathcal{P} \vdash_M G$ if and only if $\mathcal{P} \vdash_I G$.*

Proof. We need only show one remaining implication, namely, if $\mathcal{P} \vdash_I G$ then $\mathcal{P} \vdash_O G$. We first prove by induction on the height of Ξ that if Ξ is a simple **I**-proof for $\mathcal{P}' \longrightarrow G$ and $[\mathcal{P}'] \subseteq [\mathcal{P}]$, then there exists an **O**-proof for $\mathcal{P} \longrightarrow G$. We may assume that no internal sequents in Ξ are instances of initial sequents, since if there was such a sequent, the proof could be simplified by removing the subproofs above it. If the height of Ξ is one, *i.e.*, it is simply an initial sequent, then G is atomic and $G \in [\mathcal{P}']$. Hence, $G \in [\mathcal{P}]$, and $\mathcal{P} \longrightarrow G$ is an initial sequent for $\vdash_O$. This single sequent is the desired **O**-proof.

Now assume that the height of Ξ is greater than 1, and let Ξ_1 and, if necessary Ξ_2, be the **I**-proofs which arise from deleting the last inference figure of Ξ. We need to consider the following 7 cases — one for each inference figure in which Ξ can terminate.

Cases $\vee$*-R,* $\wedge$*-R,* $\exists$*-R*: Let the last inference figure of Ξ be $\wedge$-R. Here, G is $G_1 \wedge G_2$; so Ξ_1 is a proof of $\mathcal{P}' \longrightarrow G_1$ and Ξ_2 is a proof of $\mathcal{P}' \longrightarrow G_2$. By the inductive hypothesis, let T_1 and T_2 be **O**-proofs for, respectively, $\mathcal{P} \longrightarrow G_1$ and $\mathcal{P} \longrightarrow G_2$. The necessary **O**-proof for $\mathcal{P} \longrightarrow G_1 \wedge G_2$ results from appending the inference figure (4) to the trees T_1 and T_2. The cases for $\vee$-R and $\exists$-R are similar. These cases use inference figures (3) and (5), respectively, instead of (4).

Case $\supset$*-R*: Let the last inference figure of Ξ be $\supset$-R. Here, G is $D \supset G_1$; so Ξ_1 is a proof of $\mathcal{P}', D \longrightarrow G_1$. By the inductive hypothesis, let T_1 be an **O**-proof for $\mathcal{P}, D \longrightarrow G_1$. We may apply the inductive hypothesis since $[\mathcal{P}'] \subseteq [\mathcal{P}]$ implies $[\mathcal{P}', D] \subseteq [\mathcal{P}, D]$. The necessary **O**-proof for $\mathcal{P} \longrightarrow D \supset G_1$ results from appending the inference figure (6) to T_1.

Cases $\forall$*-L and* $\wedge$*-L*: Let the last inference figure of Ξ be $\forall$-L. Hence, $\mathcal{P}' = \{\forall x\, D\} \cup \mathcal{P}''$, for some set $\mathcal{P}''$, and Ξ_1 is a proof of $\mathcal{P}'', [x/t]D \longrightarrow G$, for some term t. The inductive hypothesis immediately supplies the necessary **O**-proof for $\mathcal{P} \longrightarrow G$. We may apply the inductive hypothesis since $[\mathcal{P}'] \subseteq [\mathcal{P}]$ implies $[\mathcal{P}'', [x/t]D] \subseteq [\mathcal{P}]$. The case for inference figure $\wedge$-L is similar.

Case $\supset$*-L*: Let the last inference figure of Ξ be $\supset$-L. Hence, $\mathcal{P}' = \{G' \supset A\} \cup \mathcal{P}''$ for some $\mathcal{P}''$. Ξ must then be of the form:

$$\frac{\begin{array}{c}\Xi_1\\ \mathcal{P}'' \longrightarrow G'\end{array} \qquad \begin{array}{c}\Xi_2\\ A, \mathcal{P}'' \longrightarrow G\end{array}}{G' \supset A, \mathcal{P}'' \longrightarrow G} \supset\text{-L}.$$

Since Ξ is simple, $A, \mathcal{P}' \longrightarrow G$ must be an initial sequent. This forces $G = A$, since otherwise $G \in \mathcal{P}''$ and the root sequent of Ξ would be an initial sequent, contradicting our assumption about Ξ. By the inductive hypothesis, let T_1 be an **O**-proof for $\mathcal{P} \longrightarrow G'$. We may apply the inductive hypothesis since $[\mathcal{P}'] \subseteq [\mathcal{P}]$ implies $[\mathcal{P}''] \subseteq [\mathcal{P}]$. Since $G' \supset A \in \mathcal{P}'$, $G' \supset A \in [\mathcal{P}]$ and the necessary **O**-proof results from appending an instance of the inference figure (2) to T_1.

Finally, let $\mathcal{P} \vdash_I G$. Then $\mathcal{P} \longrightarrow G$ has a simple **I**-proof. By the proof above and since $[\mathcal{P}] \subseteq [\mathcal{P}]$, $\mathcal{P} \longrightarrow G$ has an **O**-proof and $\mathcal{P} \vdash_O G$. ▮

Throughout this section, the differences between minimal logic and intuitionistic logic proofs were not evident. This is because goal formulas and definite clauses contain no instances of negations or $\perp$. In the next section, we introduce negation to our programming logic. As we shall see, there are two natural interpretation of this negation; one interpretation is equivalent to minimal logic and the other to intuitionistic logic.

7. Minimal Logic Negation

Let us now permit the logical constant $\perp$ to occur in both goals and definite clauses as if it were considered an atomic formula. More precisely, for this section, consider definite clauses and goal formulas to be defined by the following mutually recursive definition:

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

$$G := \perp \mid A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid \exists x G \mid D \supset G.$$

Formulas of the form $B \supset \perp$ will be abbreviated as $\sim B$. Thus, the negation of a goal formula is a definite clause, and the negation of a definite clause is a goal formula. Operationally, let us treat $\perp$ just as if it was an atomic formula. Hence, with this simple or "minimal" view of $\perp$, we can easily add the following two proof rules to handle the case where $\perp$ is a goal.

(7) $\mathcal{P} \vdash_O \perp$ if $\perp \in [\mathcal{P}]$.

(8) $\mathcal{P} \vdash_O \perp$ if there is a formula $(G \supset \perp) \in [\mathcal{P}]$ and $\mathcal{P} \vdash_O G$.

We need to add to the **O**-proof system two inference figures corresponding to (7) and (8). This extension, however, is really not an extension at all. A reasonable implementation of rules (1) — (8) would, in fact, treat $\perp$ as a non-logical predicate symbol and simply use rules (1) — (6).

Proof rule (8) asserts that $\perp$ follows from $\mathcal{P}$ if there is a formula G for which both $\sim G \in [\mathcal{P}]$ and G follows from $\mathcal{P}$. For this reason, we shall say that $\mathcal{P}$ is *inconsistent* if $\mathcal{P} \vdash_O \perp$. If $\mathcal{P}$ is not inconsistent, it is *consistent*.

The inconsistency of a set of definite clauses $\mathcal{P}$ does not necessarily mean that $\mathcal{P}$ **O**-proves all (goal) formulas. For example, It is very easy to show that $p \wedge \sim p \vdash_O \perp$ while it is not the case that $p \wedge \sim p \vdash_O q$. Hence, inconsistencies are essentially "local." As the next theorem shows, this view of negation is that of minimal logic and not that of intuitionistic logic.

Theorem 14. *$\mathcal{P} \vdash_O G$ if and only if $\mathcal{P} \vdash_M G$. There are goals, for example $(p \wedge \sim p) \supset q$, which are intuitionistically provable but not provable in* **O**.

Proof. Let Ξ be an **O**-proof for $\mathcal{P} \longrightarrow G$. Let $\Xi', \mathcal{P}'$, and G' be the result of replacing every instance of $\perp$ in $\Xi, \mathcal{P}$, and G by some 0-ary predicate symbol, say F, which does not occur in Ξ. Clearly, Ξ' is an **O**-proof of the sequent $\mathcal{P}' \longrightarrow G'$. By Theorem 13, there is an **M**-proof, T for this same sequent. Finally, if we replace all the occurrences of F with $\perp$

in T to get T', then T' is an **M**-proof for $\mathcal{P} \longrightarrow G$. The proof of the converse is similar. ▮

It is a simple matter to strengthen the proof system **O** so that $\perp$ and negation are equivalent to that of intuitionistic logic. This is done by adding a proof rule which encodes the $\perp$-R inference figure. That is, we need to add the rule which says that any goal formula is provable if $\perp$ is provable. Let **O**$'$ be the proof system which is the result of adding the following proof rule to those of **O**.

(9) $\mathcal{P} \vdash_{O'} G$ if $\mathcal{P} \vdash_{O'} \perp$.

It is easy to see, in fact, that $p, {\sim}p \vdash_{O'} q$. In fact, we have the following theorem.

Theorem 15. *$\mathcal{P} \vdash_{O'} G$ if and only if $\mathcal{P} \vdash_I G$.*

Proof. The proofs for Lemmas 11 and 12 and for Theorem 13 can be extended to deal with **I**-proofs which contain the inference figure $\perp$-R. These extended lemmas and theorem prove this theorem because proof rule (9) corresponds exactly to $\perp$-R for **I**-proofs. ▮

The choice of which interpretation of $\perp$ and negation that is made would seem to depend very much on what applications are written in this language. In an application where it is very important to know whenever a program becomes inconsistent, the minimal logic interpretation is probably the most appropriate. In such a case, it might seem meaningless to be deriving all goal formulas from a program which is inconsistent. Such an application would probably be more concerned with dealing with the inconsistency itself than with continuing to do inferencing with it. On the other hand, if there are some reasoning processes which involve many instances of assuming hypotheses during a single process, then the intuitionistic interpretation might be more natural. This is because the construction of nested proofs often makes use of contradictions to help draw conclusions. For example, the goal formula

$$(({\sim}p \supset r) \supset q) \supset (p \supset q)$$

has an **I**-proof but no **M**-proof. The proof of this goal contains a subproof which proves a contradiction. That contradiction does not, however, reflect on the environment in which the query is initially asked.

For the rest of this paper, we shall assume that we are using the minimal logic interpretation of $\perp$.

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 either the minimal logic or intuitionistic logic interpretation of negation, we can make a useful partial connection between them. If $\mathcal{P}$ is consistent and $\mathcal{P} \vdash_O {\sim}A$ then there can be no proof of A from $\mathcal{P}$. In a consistent program, therefore, $\mathcal{P} \vdash_O {\sim}A$

implies `not(`A`)` (negation-by-failure). This suggests the possibility of replacing certain forms of negation-by-failure with the search for proofs of negations. Negation-by-failure is not completely removed from this consideration because determining that a program is consistent requires determining that there is no proof of $\perp$. The following example demonstrates these two negation principles.

Let us consider a very simple database program. Facts within our database will be simple, closed atomic formulas. Definite clauses of the form $G \supset \perp$ 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(Fact)) :- Fact => db.
do(retract) :- fail.
do(commit) :- repeat.
do(check(Query)) :-
   (Query, write(yes),nl,!;
    Query => ⊥,write(no),nl,!;
    write('no, but it could be true'),nl).
do(consis) :- (not ⊥, write(yes),!; write(no)), nl.
```

Here, `db` represents a looping database query and updating program. Providing the command `enter(Fact)` to `db` makes an update to the current database by calling `db` after it has made `Fact` a hypothesis. It is possible to retract such updates by using the `retract` command. This command simply fails. All updates will be undone backwards to the point of the last `commit` command. The `commit` command will always re-succeed (given, of course, a depth-first interpreter).

It is the `check` command which is most interesting here. It has a three-valued behavior. Assume that the database is consistent. The command `check(Query)` will first look for a proof of `Query`, and if one is found, prints "yes". Otherwise, a proof for the negation of `Query`, *i.e.*, `Query =>` $\perp$ is searched for. (This goal means: "if `Query` were true, do we have an inconsistency?") If a proof is found, then "no" is printed. If neither the positive nor negative form of `Query` can be proved, then the database does not contain `Query` although it is consistent for `Query` to be a fact in some extension of that database. If the current database is inconsistent, 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 and not accounted for by the theory we have presented.

The model presented in Section 5 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 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."

8. Removing Disjunctions From Programs

In this section, we show that disjunctions are not needed in writing programs. That is, for every program there is a second program, generally much larger, which proves the same atomic formulas and which contains no occurrences of disjunctions. This is true for simple Horn clauses as well as definite clauses.

We define two functions, dnf which maps goal formulas to sets of goal formulas, and dfnf which maps definite clauses to sets of definite clauses, by mutual recursion. The expressions $\bigwedge \mathcal{B}$ and $\bigvee \mathcal{B}$ denote, respectively, the conjunction and disjunction of the formulas in $\mathcal{B}$ in some fixed but arbitrary order. The two acronyms, dnf and dfnf, stand for the *disjunctive normal form* and the *disjunction-free normal form*, respectively. These are defined as follows:

$$\begin{aligned}
\mathit{dnf}(A) &= \mathit{dfnf}(A) = \{A\}\\
\mathit{dnf}(G_1 \vee G_2) &= \mathit{dnf}(G_1) \cup \mathit{dnf}(G_2)\\
\mathit{dnf}(G_1 \wedge G_2) &= \{G' \wedge G'' \mid G' \in \mathit{dnf}(G_1), G'' \in \mathit{dnf}(G_2)\}\\
\mathit{dnf}(\exists x\ G) &= \{\exists x\ G' \mid G' \in \mathit{dnf}(G)\}\\
\mathit{dnf}(D \supset G) &= \{(\bigwedge \mathit{dfnf}(D)) \supset G' \mid G' \in \mathit{dnf}(G)\}\\
\mathit{dfnf}(G \supset A) &= \{G' \supset A \mid G' \in \mathit{dnf}(G)\}\\
\mathit{dfnf}(D_1 \wedge D_2) &= \mathit{dfnf}(D_1) \cup \mathit{dfnf}(D_2)\\
\mathit{dfnf}(\forall x\ D) &= \{\forall x\ D' \mid D' \in \mathit{dfnf}(D)\}
\end{aligned}$$

If $\mathcal{P}$ is a set of definite clauses, then we shall also write $\mathit{dfnf}(\mathcal{P}) = \bigcup\{\mathit{dfnf}(D) \mid D \in \mathcal{P}\}$. It is easy to see that the formulas in the sets $\mathit{dnf}(G)$ and $\mathit{dfnf}(\mathcal{P})$ contain no occurrences of disjunctions. The following Lemma has a straightforward proof which is omitted.

Lemma 16. *Let $\mathcal{P}$ be a set of definite clauses. Then each of the following is true.*

(1) $A \in [\mathcal{P}]$ *if and only if* $A \in [\mathit{dfnf}(\mathcal{P})]$.

(2) $G \supset A \in [\mathcal{P}]$ *if and only if* $G' \supset A \in [\mathit{dfnf}(\mathcal{P})]$ *for each* $G' \in \mathit{dnf}(G)$.

(3) $G' \in \mathit{dnf}([x/t]G)$ *if and only if there is a* $\exists x\ G'' \in \mathit{dnf}(\exists x\ G)$ *such that* $G' = [x/t]G''$.

The main result concerning dnf and dfnf is given by the following theorem.

Theorem 17. $\mathcal{P} \longrightarrow G$ *has an* **O***-proof if and only if for some* $G' \in \mathit{dnf}(G)$, $\mathit{dfnf}(\mathcal{P}) \longrightarrow G'$ *has an* **O***-proof.*

Proof. Let Ξ be an **O**-proof for $\mathcal{P} \longrightarrow G$. We proceed by induction on the height of Ξ.

Case: Ξ is an initial sequent. In that case, G is atomic and $G \in [\mathcal{P}]$. By Lemma 16, $G \in [\mathit{dfnf}(\mathcal{P})]$ and $\mathit{dfnf}(\mathcal{P}) \longrightarrow G$ is also an initial sequent.

Case: Ξ ends in inference figure (2). Thus, G is atomic and there is a $G' \supset G \in [\mathcal{P}]$ such that $\mathcal{P} \longrightarrow G'$ has a shorter **O**-proof. By the inductive hypothesis, there is a $G'' \in \mathit{dnf}(G')$, such that $\mathit{dfnf}(\mathcal{P}) \longrightarrow G''$ has an **O**-proof. Again, by Lemma 16, $G'' \supset G \in [\mathit{dfnf}(\mathcal{P})]$, so $\mathit{dfnf}(\mathcal{P}) \longrightarrow G$ has an **O**-proof.

Case: Ξ ends in inference figure (3). Thus, G is a disjunction, $G_1 \vee G_2$, and for $i = 1$ or $i = 2$, $\mathcal{P} \longrightarrow G_i$ has a shorter **O**-proof. Hence, for some $G' \in \mathit{dnf}(G_i)$, $\mathit{dfnf}(\mathcal{P}) \longrightarrow G'$ has an **O**-proof. Since $G' \in \mathit{dnf}(G)$, we are finished.

Case: Ξ ends in inference figure (4). Thus, G is a conjunction, $G_1 \wedge G_2$, and for $i = 1$ and $i = 2$, $\mathcal{P} \longrightarrow G_i$ has a shorter proof than Ξ. By the inductive hypothesis, there are formulas $G_1' \in \mathit{dnf}(G_1)$ and $G_2' \in \mathit{dnf}(G_2)$ such that $\mathit{dfnf}(\mathcal{P}) \longrightarrow G_1'$ and $\mathit{dfnf}(\mathcal{P}) \longrightarrow G_2'$ have **O**-proofs. Thus, $\mathit{dfnf}(\mathcal{P}) \longrightarrow G_1' \wedge G_2'$ also has an **O**-proof. Since $G_1' \wedge G_2' \in \mathit{dnf}(G)$, we are finished.

Case: Ξ ends in inference figure (5). Thus, G is an existentially quantified formula, $\exists x\ G_0$, and for some term t, $\mathcal{P} \longrightarrow [x/t]G_0$ has a shorter **O**-proof. Hence, for some

$G' \in \mathit{dnf}([x/t]G_0)$, $\mathit{dfnf}(\mathcal{P}) \longrightarrow G'$ has an **O**-proof. By Lemma 16, G' is of the form $[x/t]G''$, where $\exists x\, G'' \in \mathit{dnf}(\exists x\, G_0)$. This completes this case.

Case: Ξ ends in inference figure (6). Thus, G is an implication, say $D \supset G_0$. Since $\mathcal{P}, D \longrightarrow G_0$ has a smaller **O**-proof, there is a $G' \in \mathit{dnf}(G_0)$ such that $\mathit{dfnf}(\mathcal{P}, D) \longrightarrow G'$ has an **O**-proof. Since $\mathit{dfnf}(\mathcal{P}, D) = \mathit{dfnf}(\mathcal{P}) \cup \mathit{dfnf}(D)$, the sequent $\mathit{dfnf}(\mathcal{P}), \bigwedge \mathit{dfnf}(D) \longrightarrow G'$ has an **O**-proof. By applying an instance of inference figure (6) to this proof, we have an **O**-proof for $\mathit{dfnf}(\mathcal{P}) \longrightarrow [\bigwedge \mathit{dfnf}(D)] \supset G'$. Since $[\bigwedge \mathit{dfnf}(D)] \supset G' \in \mathit{dnf}(G)$, we have completed this case.

The proof of the converse is similar and is omitted. ∎

A simply corollary of this Theorem is that the two programs $\mathcal{P}$ and $\mathit{dfnf}(\mathcal{P})$ prove the same atomic formulas.

9. Related Work

Many of the results in this paper were first presented in [12]. The papers [13] and [15] are direct extensions of the logic presented in this paper. In particular, they introduce a class of formulas, call hereditary Harrop formulas, which properly contains both Horn clauses and the logic of this paper. Hereditary Harrop formulas can be defined for both first-order and higher-order logic.

Gabbay and Reyle in [7] have described 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 $D \supset G$.

Warren in [22] 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 7. He essentially used implications within goals only when the hypothesis of that implication is atomic. His “modal” operator `assume(`A`)@`G notation could be approximated within our language as $A \supset G$. Warren also provides a semantics for this operator using possible worlds semantics.

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.

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 results in this paper should provide a basis for such an analysis.

O'Keefe in [17] 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. See, for example, our earlier discussion in Section 4 regarding breeze bricks. Nait Abdallah in [16] presents a logical approach to introducing module-like procedures. His approach is to base logic programming within a fragment of second-order logic and to use second-order quantifiers as an abstraction mechanism for procedures (predicates). This approach is quite different from our development here and to our other work on higher-order logic programming [14].

Goguen and Meseguer in [10] presented a notion of module for a sorted theory of Horn clauses with equality. Their modules have 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 also associated with $\mathbf{M_1}$. 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, of course, be more constrained.

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 from 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 7 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

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

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 because there is no "simple" answer substitution. Fitting in [6] used classical logic to investigate 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, is 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.

10. Acknowledgments

We are grateful to Greg Hager, Thorne McCarty, Gopalan Nadathur and Wayne Snyder for valuable comments on a previous version of this paper. John Hannan and a reviewer provided valuable comments on this version of the paper.

11. References

1. Apt, K. R. and van Emden, M. H., Contributions to the Theory of Logic Programming *JACM*, 29:841 – 862 (1982).
2. Bowen, K. A. and Kowalski, R. A., Amalgamating language and metalanguage in logic programming, in: K. L. Clark and S. A. Tarnund (eds.), *Logic Programming*, Academic Press, 1982, 153 – 172.
3. Bowen, K. A. and Weinberg, T., A Meta-Level Extension of Prolog, 1985 Symposium on Logic Programming, Boston, 48 – 53.
4. Chomicki, Jan and Minsky, Naftaly H., Towards a programming environment for large Prolog programs, 1985 Symposium on Logic Programming, Boston, 230 – 241.
5. Fitting, M. C., *Intuitionistic Logic Model Theory and Forcing*, North-Holland Pub. Co., 1969.
6. Fitting, M. C., A Kripke-Kleene Semantics for Logic Programming, Journal of Logic Programming, 2:295 – 312 (1985).
7. Gabbay, D. M. and Reyle, U., N-Prolog: An Extension of Prolog with Hypothetical Implications. I, Journal of Logic Programming 1:319 – 355 (1984).
8. Gallier, J. H., and Raatz, S., HORNLOG: A Graph-Based Interpreter for General Horn Clauses, Journal of Logic Programming 4:119 – 155 (1987).
9. Gentzen, G., Investigations into Logical Deductions, in: M. E. Szabo (ed.), *The Collected Papers of Gerhard Gentzen*, North-Holland Publishing Co., Amsterdam, 1969, 68 – 131.
10. Goguen, J. A. and Meseguer, J., EQLOG: Equality, Types and Generic Modules for Logic Programming, Journal of Logic Programming 1:179 – 209 (1984).
11. McCarty, L. T., Fixed Point Semantics and Tableau Proof Procedures for a Clausal Intuitionistic Logic, Technical Report LRP-TR-18, Rutgers University, 1986.

12. Miller, D., A Theory of Modules for Logic Programming, IEEE Symposium on Logic Programming, Salt Lake City, September 1986, 106 – 115.

13. Miller, D., Hereditary Harrop Formula and Logic Programming, VIII International Congress of Logic, Methodology, and Philosophy of Science, Moscow, August 1987.

14. Miller, D. and Nadathur, G., Higher-Order Logic Programming, 1986 International Conference on Logic Programming, London, 448 – 462.

15. Miller, D., Nadathur, G., and Scedrov, A., Hereditary Harrop Formulas and Uniform Proof Systems, Second Annual Symposium of Logic in Computer Science, Cornell University, June 1987, 98 – 105.

16. Nait Abdallah, M. A., Procedures in Horn-Clause Programming, 1986 International Conference on Logic Programming, London, 433 – 447.

17. O'Keefe, R., Towards an Algebra for Constructing Logic Programs, 1985 Symposium on Logic Programming, Boston, 152 – 160.

18. Prawitz, D., *Natural Deduction,* Almqvist & Wiksell, Uppsala, 1965.

19. Smorynski, C., Applications of Kripke models, Chapter V in [21].

20. Sterling, L., Shapiro, E., *The Art of Prolog*, MIT Press, 1986.

21. Troelstra, A. S., *Metamathematical investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics 344, Springer-Verlag, Berlin, 1973.

22. Warren, D. S., Database Updates in Prolog, Proceedings of the International Conference on Fifth Generation Computer Systems, 1984, 244 – 253.