Metatheoretic Constructs in Logic Programming Languages

Frank Brown
Department of Computer Science
The University of Texas at Austin
Austin, Texas 78712

TR 140

March 1980

abstract

We axiomatize a number of meta theoretic concepts which have been used in Logic programming, including: meaning, logical truth, non-entailment, assertion and erasure, thus showing that these concepts are logical in nature and need not be defined as they have previously been defined in terms of the operations of any particular interpreter for logic programs.

Contents

1. Introduction

One of the basic theses of Logic Programming [1,2] is that programs written in Logic can be understood merely by relfecting on the intuitive meaning of the programs, without reference to any particular interpreter, or rather automatic theorem prover, that might execute the programs. This thesis would be very nice and have strong implications for programming methodology, if in fact it were true. But unfortunately, if we examine the situation closely we will see that there are features of contemporary logic programming languages [3,4,5,6,7,8,9,10.11] which do not seem to be understandable without reference to a particular interpreter. Such features are for example:

1. Meaning, such as the universal function of Prolog [6,7,9,10,11] systems
2. Logical Truth.
3. Non-Entailment, such as the Thnot function of Planner [3] and some of the uses of the slash (/) function of Prolog which simulate thnot.
4. Assertion, such as the Thassert function of Planner and the ajout functino of Prolog.
5. Erasure, such as the Therase function of Planner, and the suppress function of Prolog.

After describing in Section 2 the logical programming language that we use, we then give in this language a correct axiomatization of each of the about five semantic functions. In particular in Section 3 we axiomatize the concept of meaning, and in Section 4 we axiomatization of each of the about five semantic functions. In particular in Section 3 we axiomatize the concept of meaning, and in Section 4 we axiomatize the modal concept of logical truth which is then used to define the remaining semantic functions: non-entailment, assertion, and erasure. The theory consisting of these axioms is consistent relative to 1st order number theory. We mention this fact because the axiomatization of semantic concepts has often fallen into paradox and contradiction.

2. A Logical Programming Language

We describe the syntax of our logical programming language in Section 2.1 and then give a few examples of logic programs in Section 2.2.

2.1 Notation

We now explain our notation.

The symbols of classical logic are listed below with their English Translations:

$p \wedge q$	p and q
$p \vee q$	p or q
$p \leftrightarrow q$	p iff q
$\sim p$	not p
$\blacksquare$	true
$\square$	false
$\forall X\ \phi X$	for all objects X, ϕX holds
$\exists X\ \phi X$	for some objects X, ϕX holds
$\forall p\ \phi$	for all propositions p, ϕp holds
$\exists p\ \phi$	for some propositions p, ϕp holds

Capital letters such as X, Y, Z, S, T, A, B range over objects whereas small letters such as p, q, r range over propositions.

The symbols of modal logic are:

$\vdash p$	p is logically true
$\vdash pq$	p entails q
$\Diamond p$	p is possible
(World p)	p is a World

The last three modal symbols are defined in terms of the first one as follows:

$\vdash pq$	$=_{df}\ \vdash (p \rightarrow q)$
$\Diamond p$	$=_{df}\ \sim \vdash \sim p$
(World p)	$=_{df}\ (\Diamond p) \wedge \forall q((\vdash pq) \vee (\vdash p(\sim q)))$

Equality:

$X = Y$ X equals Y

where $X = Y \rightarrow (\phi X \leftrightarrow \phi Y)$ for all sentences ϕ including sentences containing modal connectives such as $\vdash$.

A data structure of lists formed from:

Nil

(Cons X Y)

where (Cons X Y) is an ordered pair and Nil is not an ordered pair:

(Cons X Y) = (Cons U V) $\leftrightarrow$ X = U $\wedge$ Y = V

$\sim$ (Cons X Y) = Nil

We make three abbreviations as follows:

First for any expression $\alpha_1 \ldots \alpha_n$:

$[\alpha_1 \ldots \alpha_n]$=df (Cons α_1 ... (Cons α_n Nil)..)

and also:

$[\alpha_1 . \alpha_2]$=df (Cons $\alpha_1 \alpha_2$)

[ ] =df Nil

Thus $[\alpha_1 \ldots \alpha_n]$ may be thought of as being a list whose elements are $\alpha_1, \ldots, \alpha_n$; $[\alpha_1 . \alpha_2]$ may be thought of as being an ordered pair; and [] may be thought of as being an empty list.

Finally we include a method of talking about expressions of this logical language, within this language. We do this in our logical language by representing an expression as a list of the names of its symbols. Names of symbols are formed by prefixing to that symbol an accent sign: '. Thus

[['Member 'X '[]] '$\leftrightarrow$ '$\Box$]

is a name of the expression (Member X []) $\leftrightarrow$ $\Box$. It is to be understood that 'X is a constant symbol of our logical language. The apparent visual similarity between a symbol such as: X and its name such as: 'X is merely a pneumonic for the readers convenience which will also allow us to concisely state the criteria for a definition of a meaning function.

2.2 Logic as a Programming Language

This simple language may be thought of as being a programming language. That is, we can implement a system which by making logical inferences can effectively compute various things

For example a program in this language which could be used to compute whether something is a member of a list would be the three sentences:

M1: (Member X[]) ↔ $\square$

M2: (Member X (Cons X Y)) ↔ $\blacksquare$

M3: (∼(X=Z)) → ((Member X (Cons Z Y)) ↔ (Member X Y))

If our system were then defined so as to use these sentences to replace the left hand side of an equivalence by the right hand side, checking that any initial conditional sentences were true, then the system could determine that B were a member of the list [A B C] as follows:

(Member B [A B C]) :M3

↓

(Member B [B C]) :M2

↓

$\blacksquare$

As a more complex case consider the following program which computes the value of an element V in a list of pairs

$[[V_1 . \alpha_1] \ldots [V_n . \alpha_n]]$

The value of V in such a list is defined to be the first α_i whose V_i equals V.

V1: (Val V [])=Nil

V2: (Val V [[V.X].L])=X

V3: (∼U=V → (Val V [[U.X].L])=(Val V L)

Thus for example the system could determine that the value of C in [[A.B] [C.D]] is D as follows:

(Val C [[A.B] [C.D]]) :V3

↓

(Val C [[C.D]]) :V2

↓

D

More detailed expositions on the use of logic as a programming language are given in [1,2,11,12,13,14,15].

3. Meaning

One of the most important features of any reasonably general programming language is the ability to execute a program which has been constructed as a piece of data by another program. For example, a compiler essentially translates one data structure representing a program into another data-structure representing a program which is executable on some particular computer. The key step here is: how does this second data-structure actually become to be turned into a program which can be executed?

In logic, it is easy to represent both a logic program and a data structure which is a name of that logic program along the lines described in section 2. Furthermore this data structure can easily be manipulated by other logic programs, and in particular could be the result of translating a data-structure representing a program of some other language into a data-structure representing a logic program. However, the final key step of translating the data structure representing a logic program into the logic program itself is what appears to be difficult to do in logic.

In the remainder of this section we will show how this may be done in logic, and in particular in section 3.1 will we give a recursive meaning function M which maps data-structures representing logic programs into the corresponding logic programs. This meaning function M satisfies the criteria that:

$(M\ \phi) = \psi$

if ψ is obtained from ϕ by

1. eliminating the first accent sign from each symbol beginning with with an accent sign.
2. replacing all occurrences of [by (.
3. replacing all occurrences of] by).

for any expression ϕ consisting solely of the symbols: Cons, Nil, and symbols beginning with an accent sign.

It is worthwhile noting M in this criteria cannot be interpreted as being Tarski's [16] predicate E for Empirical Truth. The reason for this is that whereas Tarski's superficially similar criteria for empirical truth involves bi-implication:

$(E\ \phi) \leftrightarrow \psi$

our criteria is much stronger and in fact implies synonymity, or rather necessary bi-implication:

$\vdash ((M\ \phi) \leftrightarrow \psi)$

Thus if M in our criteria is interpreted to be empirical truth, then the criteria itself becomes contradictory. We can see this fact by simply letting ϕ be the name of any sentence ψ which is empirically true but not logically true: such as: "The Morning Star is the Evening Star."

Finally in section 3.2 we discuss some variations of this meaning function, and show why it is not easily represented in clausal form logics.

3.1 Axiomatization of Meaning

A computationally efficient meaning function M satisfying the above criteria is given below. Since all other logical symbols are definable in terms of $\wedge$, $\neg$, $\forall$, and $\vdash$ we have not bothered to list meaning axioms for any logical symbols other than these:

M0: $(M\ S) = (m\ (closure\ S)\ [\,])$

M1: $(M[S\ '\wedge\ T]A) = (m\ S\ A) \wedge (m\ T\ A)$

M2: $(m['\neg\ S]A) = \neg(m\ S\ A)$

M3: $(m['\vdash\ S]A) = \vdash(m\ S\ A)$

M4: $(m['\phi\ S_1 \ldots S_n]A) = (\phi(m\ S_1\ A) \ldots (m\ S_n\ A))$

for each non logical symbol ϕ except m or any name of m or any name of a name of m and so forth.

M5: $(m['\forall\ V\ S]A) = \forall X(m\ S([V.X].A))$

M6: $(m\ V\ A) = (Val\ V\ A)$

The variables S, T, $S_1 \ldots S_n$ range over expressions, and the variable V ranges over variables. The closure of a sentence S with free variables $'X_1 \ldots 'X_n$ is defined to be $['\forall 'X_1 \ldots ['\forall 'X_n\ S] \ldots]$.

An example of the application of this meaning function to the sentence ['∀ 'Z['Y 'movesto 'Z]] is as follows:

(M['∀ 'Z['Y 'movesto 'Z]])	:M0
(m (closure['∀ 'Z['Y 'movesto 'Z]])[])	:closure
(m ['∀ 'Y['∀ 'Z['Y 'movesto 'Z]]][])	:M5
(∀X (m['∀ 'Z['Y 'movesto 'Z]]([['Y.X]]))	:rename

We cannot immediately apply M5 again because X is a free variable in the subexpression:

(m['∀ 'Z['Y 'movesto 'Z]][['Y.X]])

and hence would be captured by the outlying quantifier: ∀X occurring in 5. Hence one of these variables must first be renamed.

(∀Y (m['∀ 'Z['Y 'movesto 'Z]][['Y.Y]]))

(∀Y (∀X (m['Y 'movesto 'Z][['Z.X]['Y.Y]]))) :M5

(∀Y (∀X ((m'Y[['Z.X]['Y.Y]]) movesto (m'Z[['Z.X]['Y.Y]])))) :M4

(∀Y (∀X ((Val 'Y[['Z.X]['Y.Y]]) movesto (Val 'Z[['Z.X]['Y.Y]])))) :M6

(∀Y (∀X (Y movesto X))) :Val

(∀Y (∀Z (Y movesto Z))) :rename

(∀Z (Y movesto Z)) :universal instantiation and generalization

We note that this meaning function consists only of sentences which recurse through syntactic structure. Thus this meaning consists entirely of recursive definitions. It is in this sense that we claim that it is a logically true meaning function.

A more detailed description of this meaning function and its philosophical implications is given in [17]. Its use by an automatic theorem prover to obtain a proof of the completeness of quantificational logic is described in [18].

A more detailed description of this meaning function and its philosophical implications is given in [4].

3.2 Variations

Since:

ϕY is equivalent to ∀X(ϕZ → Y = Z)

and more generally since:

ϕ ∀X (tX) is equivalent to ∀z ϕ∀X(zX) → ∀Y(tY)=(zY)

by using these equivalences we can replace the equations of the meaning function M by implications to obtain an equivalent meaning function MA:

MAO: (M S)=Z → (m(closure S)[])=Z

MA1: (m[S'∧ T]A)=(p ∧ q) → (m S A)=p ∧ (m T A)=q

MA2: (m['∼S]A)=∼p → (m S A)=p

MA3: (m['⊢ S]=⊢p → (m S A)=p

MA4: $(m['\phi\ S_1 \ldots S_n]A)=(\phi X_1 \ldots X_n) \rightarrow (m\ S_1\ A)=X_1 \wedge \ldots \wedge (m\ S\ A)=X_n$

MA5: $(m['\forall\ V\ S]A)=\forall X(\phi X) \leftarrow \forall X(m\ S[[V.X].A\ )=(\phi X)$

MA6: $(m\ V\ A)=Z \leftarrow (Val\ V\ A)=Z$

Note that MA5 when put into skolem normal form becomes:

MA5': $(m['\forall V\ S]A)=\forall x(\phi X) \leftarrow (m\ S[[V.(Sk\ \underline{X}\ V\ S\ A(\phi X))].A\ )=(\phi(Sk\ \underline{X}\ V\ S\ A(\phi X)))$

where Sk is a skolem function.

Many contemporary logical programming languages [1,3,6,7,8,11,13] are to a large degree based on evaluating sentences of a certain form: First they are based on evaluating reverse implications ← somewhat similar to the manner in which the equivalences were evaluated as described in section 2.2. Furthermore, such systems are often defined so as to only match entire atomic propositions such as (Member A B) or (m S A)=Y rather than any embedded terms such as (m S A). And finally, such systems usually require that all sentences be initially skolemized. It will be seen that MA satisfies the first two criteria and if MA5 is replaced by MA5' then all three criteria are satisfied.

There is, however, a problem in that MA5 (and MA5') involves second order unification which usually is not available in such programming systems. Possibly, for this reason, or because MA5 is rather complicated, and therefore difficult to state, or even because of a preference for stating the object language itself in skolem normal form, sentences such as MA5 are not currently used in such programming systems. Thus quantifiers and bound variables are not allowed in the object languages.

Initial free universal object language variables could however be allowed by the simple expedient of replacing MAO by MAO':

MAO': $(M\ S)=Z \leftarrow (\forall X_1 \ldots \forall X_n (m\ S\ a))=Z$

where 'a' here is an association list:

$[V_1.X_1]\ldots[V_n.X_n]$

such that $V_1 \ldots V_n$ are all the unbound object language variables in S, and $X_1 \ldots X_n$ are distinct free universal variables. MAO' has essentially the same effect as MAO since the closure of S is $['\forall V_1 \ldots['\forall V_n\ S]\ldots]$ which by MA5 becomes: $(\forall X_1 \ldots \forall X_n (m\ S\ a))$.

Note also that MO has a similar version MO'

MO': $(MS) = (\forall X_1 \ldots \forall X_n (m\ S\ a))$

We see then that MAO', MA1, MA2, MA3, MA4, MA6 constitute a proper meaning function for a quantifier free object language. It should be noted, however, that the replacement of MAO' by MAO" does not constitute a proper meaning function:

MAO": $(M\ S) = Z \leftarrow (m\ S\ a) = Z$

The reason for this is that MAO" can, in fact, be false when MAO' is true. To see this first note that MAO" is equivalent to

MO": $(M\ S) = (m\ S\ a)$

We consider now the sentence: ['ϕ'X] in a universe of two things:

By MAO' we get:

$M['\phi'X] = \forall X(m['\phi'X][['X.X]])$

$M['\phi'X] = \forall X\phi X$

$M['\phi'X] = \phi_1 \wedge \phi_2$

whereas by LAO" we get:

$\forall X(M['\phi'X] = (m['\phi'X][['X\ X]])$

$\forall X(M['\phi'X] = \phi X)$

$M['\phi'X] = \phi_1 \wedge M['\phi'X] = \phi_2$

But clearly MAO" leads to a contradiction; for example in the case where ϕ_1 is true and ϕ_2 is false it implies that true equals false.

Note however, that a sentence similar to MAO" with the (m S)=X replaced by a relation could, however, be consistent if this relation were intended to capture some concept of partial meaning. For example:

(i S A X) means that: X is the meaning in A of some instance of S.

(P S A X) means that: X is implied by the meaning of S in A.

We see that both

$(I\ S\ X) \leftarrow (i\ S\ aX)$

and $(P\ S\ X) \leftarrow (P\ S\ aX)$

are probably true.

Note that although I and P are transitive and perhaps reflexive relations, neither is a symmetric relation. Generally they are related to M as follows:

$((M\ X)=X \rightarrow (I\ S\ X)) \wedge ((I\ S\ X) \rightarrow (P\ S\ X))$

A meaning-of-some-instance relation (I) has been implemented in a logical programming system by D. Warren [9]. The sentences of this relation may be obtained from MAO", MA1, MA2, MA3, MA4, MA6 by replacing $(m\ \alpha)=\beta$ by $(I\ \alpha\ \beta)$ and $(m\ A\ \alpha)=\beta$ by $(i\ \alpha\ A\ \beta)$.

4. Modality

Another important feature of any reasonably general programming language is the ability to execute a program within a particular context. For example, one might wish to evaluate a particular expression using certain function definitions which are quite different from the function definitions which are active at the top level context In a logical languate the analogy of this would be to evaluate some expression using a certain database of axioms which could be quite different from the data base of axioms being assumed at the top level

At first glance, it would appear that it is impossible to define such a construct in logic because even though one could use the $b \rightarrow p$ concept to create a new context containing additional axioms b which can be used when evaluating p, there does not seem to be any way to stop the use of any axioms a of the top level context such as in $a \rightarrow (b \rightarrow p)$.

However, modal logic offers a way to solve this problem. If we let $\vdash$ be the modal symbol which captures the notion of logical truth, then it is easy to see that $\vdash(b \rightarrow p)$ means that p is to be evaluated using only the axioms b. Thus, for example, if the top level context is a, $a \rightarrow \vdash (b \rightarrow p)$ still only allows p to be evaluated using the axioms b.

The problem is to find a correct axiomatization for the modal concept of logical truth: $\vdash$, for it is certainly clear that there must be special axioms of modality in addition to the normal axioms of classical logic. For example the sentence:

$$\vdash(b \rightarrow (p \wedge q)) \leftrightarrow (\vdash(b \rightarrow p) \wedge \vdash(b \rightarrow q))$$

is intuitively valid although it is not derivable solely from the normal axioms of classical logic.

In section 4.1 we give a correct axiomatization of the modal concept of logical truth. Then in section 4.2 we use this concept to define the semantic functions of non-entailment, assertion, and erasure.

4.1 Axiomatization of Logical Truth

The logical axioms of the modal logic which captures the notion of logical truth [18] includes any complete and reasonable axiomatization of classical quantificational logic, with propositional quantifiers plus the following inference rule and axioms about modality:

RO: from P infer $\vdash p$

A1: $\vdash p \rightarrow p$

A2: $\vdash(p \rightarrow q) \rightarrow (\vdash p \rightarrow \vdash q)$

A3: $\vdash p \vee \vdash \neg \vdash p$

A4: $(\forall r((\text{World } r) \rightarrow \vdash(r \rightarrow p))) \rightarrow \vdash p$

RO, A1, A2, and A3 are essentially the inference rule and axioms ofS5 modal logic. Axiom A4 which we call Leibniz's postulate expresses his intuition that something is logically true if it is true in all possible worlds.

An efficient sequent calculus proof procedure based on theorems derived from these modal axioms is described in [18,19].

The consistency problem of modal logic is that from the logical axioms of modal logic we cannot prove certain elementary facts about the possibility of conjunctions of distinct possible negated atomic expressions consisting of non-logical symbols. For example, if we have a theory formulated in our modal logic which contains the non-logical atomic expression (ON A B) then since $\sim$(ON A B) is not logically true, it follows that (ON A B) must be possible. Yet $\Diamond$(ON A B) is not a theorem of our modal logic.

Thus, for any theory expressed in modal logic, a certain number of non-logical axioms dealing with possibility should also be added. For example, in the case of the propositional logic, or in the case of the quantificational logic over a finite domain since it reduces to propositional logic, one sufficient but inefficient axiomatization would be to assert the possibility of all consistent disjunctions of conjunctions of literals as additional non-logical axioms:

$\Diamond(\vee(\wedge \text{ Literals}))$

A more computationally efficient axiomatization which is obtained by noting that the possibility of a disjunction of sentences is implied by the possibility of any one of those sentences:

$\Diamond p \rightarrow \Diamond(p \vee q)$

is to assert only the possibility of all consistent conjunctions of literals:

$\Diamond(\wedge \text{ literals})$

Using our meaning function [4] this may be done in a finite manner by adding the single axiom:

(Conj S) $\wedge$ (Consist S) $\rightarrow \Diamond$(M S)

where Conj and Consist are recursive functions defined as follows:

(Conj S) = df (Lit S) $\vee$ $\exists$T $\exists$R (S=[T'$\wedge$ R] $\wedge$ (Lit T) $\wedge$(Conj R))

(Lit S) = df ($\exists$T S=['$\sim$T] $\wedge$ (Atomicsent T)) $\vee$ (Atomicsent S)

(Consist [ ]) = df $\blacksquare$

(Consist [S,L]) = df (Consist2 S L) $\wedge$ (Consist L)

(Consist2 S [ ]) = df $\blacksquare$

(Consist2 S[T.L]) = df $\sim$(Opp S T) $\wedge$ (Consist2 S L)

(Opp S T) = df ($\exists$R S=['$\sim$R]$\wedge$ T=R) $\vee$ ($\exists$R T=['$\sim$R]$\wedge$ S=R)

4.2 Non-entailment, Assertion, and Erasure

Having now axiomatized the concept of logical truth, it is easy to define non-entailment and assertion:

D1: (Not-Entail a p) $\leftrightarrow$ df $\neg \vdash ap$

D2: (Assert a p) $\leftrightarrow$ df $a \wedge p$

That is p is not entailed by a, iff a implies p is not logically true. And the assertion of p to database a is simply (a and p).

The definition of erasure is, however, slightly more complex, and in general there will be more than one reasonable resulting database b which is obtained by erasing a proposition p from a given database 'a'.

D3: (Erase a p b) $\leftrightarrow$ $\vdash a\, b \wedge (\vdash b\, p \rightarrow \vdash p)$

$\wedge\ \forall q\, (\vdash a\, q \wedge \vdash q\, b \wedge (\vdash b q \rightarrow \vdash q\, p))$

That is b is obtained by erasing p from a iff a entails b, b entails p only if p is logically true, and no proposition stronger than b can be obtained from a by deleting p.

We can see that D3 is indeed a reasonable definition of erasure by noting the following theorem:

T1: $\neg \vdash ap \rightarrow ((\text{Erase } a\ p\ b) \leftrightarrow \vdash a \leftrightarrow b)$

That is, if p is not entailed by a then erasing p from a merely results in a itself.

This definition of erasure is closely related to Stalnaker's Theory of Conditionals [20] and to Schwind's Theory of Action [21].

5. Conclusion

We have axiomatized a number of basic semantic concepts for a logical programming language. An efficient automatic theorem prover based on a sequent calculus [18,19] derived from these axioms is currently running at Edinburgh, and is being used to prove rather difficult theorems in meta mathematics.

Our semantic theory also forms the basis of Brown and Schwind's [22,23,24] theory of natural language understanding which is currently being developed.

References

1. Kowalski, R. "Predicate Logic as programming language." Proceedings IFIP, 1974.

2. Bibel, W. "Predicative Programming," Institutsbericht, TU München Abteilung Mathematik, 1974.

3. Hewitt, C. "Description and Theoretical Analysis of PLANNER:" A Language for Proving Theorems and Manipulating Models in a Robot AI-TR-258 1972.

4. Boyer, R. S. and Moore J. 'The Sharing of Structure in Theorem Proving Programs, Machine Intelligence' ed meltzes and Michie, 1972.

5. Moore, J. "A Programming Language for Structure Sharing" Chapter 6 of "Computational Logic: Structure Sharing and Proof of Program Properties Part 1 DCL Memo No. 67, 1973.

6. Battani, G. and Meloni H. "Interpreteur dur language de programation PROLOG" Groupe de l'Intelligence Artificiells U.E.R. de Luming, Marseille 1973.

7. Roassel P. "PROLOG: Manuel de Reference et d'utilisation" Groupe d'Intelligence Artificielle, U.E.R. de Luming, Marseille, 1975.

8. Tärnlund, S. A. "An Interpreter for the Programming Language Predicate Logic, Proc. of IJCAIS. Tibihsi 1975

9. Warren, D. "Implementing Prolog-Compiling Predicate Logic Programs" DAI report 39 University of Edinburgh 1977.

10. Perevia L. M., Perevia F. C. N. and Warren D. H. P. Users Guide to Dec System-10 PROLOG DAI report. University of Edinburgh 1978

11. Futo, I., Darvas, F., Szeredi, P. "The Application of Prolog to the Development of QA and DBM systems" Logic and Databases ed Ga-laire, H and Minker J.

12. Gallaire, H. and Minker J. Logic and Databases. Plenum Press New York 1978.

13. Tarnlund, Sten dke "Logic Information Processing" tech report. Department of Information Processing and Computer Science. The Royal Institute of Technology and the University of Stockholm. 1975.

14. Clark K. and Tranlund S. A. "A First Order Theory of Data and Programs IFIPP77 Toronto 1977.

15. Hansson A. and Tarnlund S-A "A Natural Programming Calculus" IJCAI6 Tokyo 1979.

16. Tarski, A. "The Concept of Truth in Formalized Languages," (1931), Logic Semantics, Metamathematics, trans. by J. H. Woodger, Oxford, Clarendon Press, 1956.

17. Brown, F. M. "The Theory of Meaning." DAI Research Report 35, 1977.

18. Brown, F. M. "A Theorem Prover for Metatheory" Proceedings of the Fourth Workshop on Automated Deduction." Austin, Texas 1979.

19. Brown, F. M. "A Sequent Calculus for Modal Quantificational Logic," 3rd AISB/GI Conference Proceedings, Hamburgh, July, 1978.

20. Stalnaker, R. C. "A Theory of Conditionals," Causation and Conditionals, ed. E. Sora, Oxford University Press, 1975.

21. Schwind, C. "Representing Actions by a Modal Tense Logic," 3rd AISB/GI Conference Proceedings, Hamburgh, 1978.

22. Brown, F. M. and Schwind, C. "Analyzing and Representing Natural Language in Logic," 3rd AISB/GI Conference Proceedings, Hamburgh, 1978.

23. Brown, F. M. and Schwind, C. "Towards an Integrated Theory of Natural Language Understanding," to appear in Computational Linguistics Conference Proceedings, Bergen, 1978.

24. Brown, F. M. and Schwind, C. "Outline of an Integrated Theory of Natural Language Understanding" to appear as DAI Research Report 50.