Stanford Artificial Intelligence Laboratory
Memo AIM-315

Computer Science Department
Report No. STAN-CS-78-687

December 1978

PROLEGOMENA TO A THEORY OF FORMAL REASONING

by

Richard Weyhrauch

This paper is an introduction to the mechanization of a theory of reasoning. Currently formal systems are out of favor with the AI community. The aim of this paper is to explain how formal systems can be used in AI by explaining how traditional ideas of logic can be mechanized in a practical way. The paper presents several new ideas. Each of these is illustrated by giving simple examples of how this idea is mechanized in the reasoning system FOL. That is, this is not just theory but there is an existing running implementation of these ideas.

In this paper: 1) we show how to mechanize the notion of model using the idea of a simulation structure and explain why this is particularly important to AI, 2) we show how to mechanize the notion of satisfaction, 3) we present a very general evaluator for first order expressions, which subsumes PROLOG and we propose as a natural way of thinking about logic programming, 4) we show how to formalize metatheory, 5) we describe reflection principles, which connect theories to their metatheories in a way new to AI, 6) we show how these ideas can be used to dynamically extend the strength of FOL by "implementing" subsidiary deduction rules, and how this in turn can be extended to provide a method of describing and proving theorems about heuristics for using these rules, 7) we discuss one notion of what it could mean for a computer to learn and give an example, 8) we describe a new kind of formal system that has the property that it can reason about its own properties, 9) we give examples of all of the above.

Sponsored by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

CONTENTS

1 Introduction 1
2 FOL as a conversational program 3
3 The logic used by FOL 5
4 Simulation structures and semantic attachment 6
5 Syntactic simplifier 9
6 A general first order logic expression evaluator 10
7 Systems of languages and simulation structures 11
8 Metatheory 13
9 Reflection 16
9.1 Can a program learn 19
9.2 Using metametatheory 23
10 Self reflection 25
11 Conclusion 26
11.1 Summary of important results 26
11.2 Concluding remarks, history and thanks 27
BIBLIOGRAPHY 29
Appendix A An axiomatization of natural numbers 31
Appendix B An axiomatization of s-expressions 32
Appendix C An axiomatization of well formed formulas 33
Appendix D Examples of semantic evaluations 34
Appendix E An example of syntatic simplification 35
Appendix F An example of evaluation 39

Section 1 Introduction

The title of this paper contains both the words "*mechanized*" and "*theory*". I want to make the point that the ideas presented here are not only of interest to theoreticians. I believe that any theory of interest to artificial intelligence must be realizable on a computer.

I am going to describe a working computer program, FOL, that embodies the mechanization of the ideas of logicians *about* theories of reasoning. This system converses with users in some first order language. I will also explain how to build a new structure in which theory and metatheory interact in a particularly natural way. This structure has the additional property that it can be designed to reason about itself. This kind of self reflexive logical structure is new and a discussion of the full extent of its power will appear in another paper.

The purpose of this paper is to set down the main ideas underlying the system. Each example in this paper was chosen to illustrate an *idea* and each idea is developed by showing how the corresponding FOL feature works. I will not present difficult examples. More extensive examples and discussions of the limits of these features will be described in other places. The real power of this theory (and FOL) comes from an understanding of the interaction of these separate features. This means that after this paper is read it still requires some work to see how all of these features can be used. Complex examples will only confuse the issues at this point. Before these can be explained the logical system must be fully understood.

The FOL project can be thought of in several different ways:

1) Most important, FOL is an environment for studying epistemological questions. I look on logic as an empirical, applied science. It is like physics. The data we have is the actual reasoning activity of people. We try to build a theory of what that's like. I try to look at the traditional work on logic from this point of view. The important question is: in what way does logic adequately represent the actual practice of reasoning? In addition, its usefulness to artificial intelligence requires a stronger form of adequacy. Such a theory must be *mechanizable*. My notion of mechanizable is informal. I hope by the end of this note it will be clearer. Below, I outline the mechanizable analogues of the usual notions of model, interpretation, satisfaction, theory, and reflection principle.

2) FOL is a conversational machine. We use it by having a conversation with it. The importance of this idea cannot be underestimated. One of the recuring themes of this paper is the question: what is the nature of the conversation we wish to have with an expert in reasoning? In AI we talk about *expert systems*. FOL can be thought of as a system whose expertise is reasoning. We have tried to explore the question: what properties does an expert conversational reasoning machine have to have, independent of its domain of expertise? I believe that we will begin to call machines intelligent when we can have the kinds of discussions with them that we have with our friends. Let me elaborate on this a little. Humans are not ever likely to come to depend on the advice of a computer which has a simplistic one bit output. Imagine that you are asking it to make decisions about what stocks you should buy. Suppose it says, "I have reviewed all the data you gave me. Sell everything you own and buy stock in FOL Incorporated." Most reasonable people would like to ask some additional questions! Why did you make that choice. What theory of price behavior did you

use. Why is that better than using a dartboard. And so forth. These questions require a system that knows about more things than the stock market. For example, it needs to know how to reason about its *theories* of price movement. In FOL we have begged the question of *natural* language. The only important thing is having a sufficiently rich language for carrying out the above kinds of conversations. This paper should be looked at from this point of view.

This work has direct application in several areas. The details are referenced below.

1) Artificial Intelligence. I propose that the *language/simulation structure* pairs described below are important building blocks in a viable and mechanizable theory of knowledge representation for AI. The central idea is that FOL makes systematic use of the distinction between a language and the objects that this language describes. This distinction allows us to deal with the questions of how to manipulate theories of theory building, how to deal with modalities, how to reason about possibly inconsistent theories, how to treat "non-monotonic" reasoning and how to build a mechanizable theory of perception. By perception I mean the question of how it is possible for us to go from sense impressions to theories about what our exterior is like.

2) Mathematical Theory of Computation. FOL is an environment that can deal effectively both with a theory and its metatheory. Many aspects of program semantics are nicely expressable when this is viewed as a reasoning *system*. For a long time I have wanted to have a system in which I could develop the theory of LISP, following the ideas of Kleene[1952] when he developed recursion theory. The recent work of Cartwright[1977], and McCarthy[1978] have made this even more practicable. One main feature of this system is that it can incorporate both computation induction and the inductive assertion method in the same system. We can do this because both of these methodologies can be expressed as theorems of the metatheory. This is an example of the expressive power of the FOL system. If as above we claim that we want to be able to have discussions with FOL about anything, then programs are an interesting subject. We are currently building an "expert" system for discussing LISP programs.

3) Logic. The FOL system is not a formal system in the popular sense of the word. Logicians have used formal systems mainly to describe the sentences that are used in mathematical reasoning. I have tried on the other hand to build a structure which embodies the logicians theories of these theories and thus have a system capable of reasoning about theory building. There are several novel things about the logic of FOL that may be of interest to logicians and workers in AI. First is the way in which many sorted logic is treated. Second is the notion that simulation structures (i.e. partial models) should be represented explicitly. Third is the idea of the general purpose evaluator described below. Fourth is the use of reflection principles to connect a theory with its metatheory. Fifth is to notice that reflection and evaluation with respect to the metatheory is the technical realization of the procedural/declarative discussions which appear in the AI literature. Sixth is the discovery of META, a self reflective structure with a "locally" Tarskian semantics. This theory META is new and it has already produced some insight into the nature of meta reasoning that I will write about elsewhere.

As I reread this introduction it seems to contain a lot of promises. If they seem exaggerated to you then imagine me as a hopeless romantic, but at least read the rest of this paper. The things I describe here already exist.

Section 2 FOL as a conversational program

FOL has previously been advertised as a proof-checker. This sometimes brings to mind the idea that the way you use it is to type out a complicated formal proof, then FOL reads it and says yes or no. This picture is all wrong, and is founded on the theorem proving idea that simply stating a problem is all that a reasoning system should need to know. What FOL actually does is to have a dialogue with a user about some subject. The first step in this conversation is to establish what language we will speak to each other by establish what words we will use for what parts of speech. In FOL the establishment of this agreement about language is done by making *declarations*. This will be described below.

We can then discuss (in the agreed upon language) what facts (axioms) are to be considered true, and then finally we can chat about the consequences of these facts.

Let me illustrate this by giving a simple FOL proof. We will begin where logic began, with Aristotle[-335]. Even a person who has never had a course in formal logic understands the syllogism:

Socrates is a man
and
All men are mortal
thus
Socrates is mortal

Before we actually give a dialogue with FOL we need to think informally about how we express these assertions as well formed formulas, WFFs of first order logic. For this purpose we need an individual constant (INDCONST), Socrates, two predicate constants (PREDCONSTs), MAN and MORTAL, each of one argument, and an individual variable (INDVAR), x, to express the all men part of the second line. The usual rules for forming WFFs apply (see Kleene[1967], pp 7, 78). The three statements above are represented as

$$\mathrm{MAN}(\mathrm{Socrates})$$
$$\forall x.(\mathrm{MAN}(x) \supset \mathrm{MORTAL}(x))$$
$$\mathrm{MORTAL}(\mathrm{Socrates})$$

Our goal is to prove

$$(\mathrm{MAN}(\mathrm{Socrates}) \land \forall x.(\mathrm{MAN}(x) \supset \mathrm{MORTAL}(x))) \supset \mathrm{MORTAL}(\mathrm{Socrates})$$

As explained above the first thing we do when initiating a discussion with FOL is to make an agreement about language we will use. We do this by making declarations. These have the form

```
*****DECLARE INDCONST Socrates;

*****DECLARE PREDCONST MORTAL MAN 1;
```

```
*****DECLARE INDVAR x;
```

The FOL program types out five stars when it expects input. The above lines are exactly what you would type to the FOL system.

FOL knows all of the natural deduction rules of inference (Prawitz[1965]) and many more. In the usual natural deduction style proofs are trees and the leaves of these trees are called assumptions. The assume command looks like

```
*****ASSUME MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x));

1 MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x))  (1)
```

The first line above is typed by the user the second is typed by FOL. For each node in the proof tree there is a set of open assumptions. These are printed in parentheses after the proofstep. Notice that assumptions depend on themselves.

We want to instantiate the second half of line one to the particular MAN, Socrates. First we must get this WFF onto a line of its own. FOL can be used to decide tautologies. We type TAUT followed by the WFF we want, and then the line numbers of those lines from which we think it follows.

```
*****TAUT ∀x.(MAN(x)⊃MORTAL(x)) 1;

2 ∀x.(MAN(x)⊃MORTAL(x))  (1)
```

This line also has the open assumption of line 1. We then use the ∀-elimination rule to conclude

```
*****∀E 2 Socrates;

3 MAN(Socrates)⊃MORTAL(Socrates)  (1)
```

It now follows, tautologically, from lines one and three, that Socrates must be MORTAL. Using the TAUT command again gets this result. More than one line can be given in the reason part of the TAUT command.

```
*****TAUT MORTAL(Socrates) 1,3;

4 MORTAL(Socrates)  (1)
```

This is almost the desired result, but we are not finished yet; this line still depends upon the original assumption. We close this assumption by creating an implication of the first line implying the fourth. This is done using the deduction theorem. In the natural deduction terminology this is called *implication (⊃)introduction.*

```
*****⊃I 1⊃4;

5 (MAN(Socrates)∧∀x.(MAN(x)⊃MORTAL(x)))⊃MORTAL(Socrates)
```

This is the WFF we wanted to prove. Since it has no dependencies; it is a theorem. It is roughly equivalent to the English sentence, If Socrates is a man, and for all x if x is a man, then x is mortal, then Socrates is mortal.

This example was also used in Filman and Weyhrauch[1976] and illustrates the sense in which FOL is an interactive proof constructor, not simply a proof checker.

Section 3 The logic used by FOL

The logic used by FOL is an extension of the system of first order predicate calculus described in Prawitz[1965]. The most important change is that FOL languages contain equality and allow for sorted variables where there is a partial order on the sorts. This latter facility is extremely important for making discussions with FOL more natural. The properties of this extension of ordinary logic together with with detailed examples appear in Weyhrauch[1979]. In addition there are several features which are primarily syntactic improvements. A somewhat old description of how to use FOL is found in Weyhrauch[1977].

Prawitz distinguishes between individual variables and individual parameters. In FOL individual variables may appear both free and bound in WFFs. As in Prawitz individual parameters must always appear free. Natural numbers are automatically declared individual constants of sort NATNUM. This is one of the few defaults in FOL. The only kind of numbers understood by FOL are are natural numbers, i.e. non-negative integers. -3 should be thought of not as an individual constant, but rather as the prefix operator - applied to the individual constant 3.

A user may specify that binary predicate and operation symbols are to be used as infixes. The declaration of a unary application symbol to be prefix makes the parentheses around its argument optional. The number of arguments of an application term is called its *arity*.

FOL always considers two WFFs to be equal if they can both be changed into the same WFF by making allowable changes of bound variables. Thus, for example, the TAUT rule will accept ∀x.P(x)⊃∀y.P(y) as a tautology if x and y are of the same sort.

We have also introduced the use of conditional expressions for both WFFs and TERMs. These expressions are not used in standard descriptions of predicate calculus because they complicate the definition of satisfaction by making the value of a TERM and the truth value of a WFF mutually recursive. Hilbert and Bernays[1934] proved that these additions were a conservative extensions of ordinary predicate calculus so, in some sense, they are not needed. McCarthy[1963] stressed, however, that the increased naturalness when using conditional expressions to describe functions, is more than adequate compensation for the additional complexity.

Simple derivations in FOL are generated by using the natural deduction rules described in Prawitz[1965] together with some well-known decision procedures. These include TAUT for deciding tautologies, TAUTEQ for deciding the propositional theory of equality and MONADIC which decides formulas of the monadic predicate calculus. In actual fact MONADIC decides the case of ∀∃ formulas. These features are not explained in this paper. This is probably a good place to mention that the first two decision procedures were designed and coded by Ashok Chandra and the last by William Glassmire. The important additions to the deductive mechanisms of first order logic are the syntactic and semantic simplification routines, the convenient use of metatheory and a not yet completed goal structure (Juan Bulnes[1979]). It is these later features that are described below.

Section 4 Simulation structures and semantic attachment

Here I introduce one of the most important ideas in this paper, i.e., *simulation structures*. Simulation structures are intended to be the *mechanizable* analogue of the notion of model. We can intuitively understand them as the computable part of some model. It has been suggested that I call them *effective partial interpretations*, but I have reserved that slogan for a somewhat more general notion. A full mathematical description of these ideas is beyond the scope of this paper but appears in Weyhrauch[NOTE15]. In this paper I will give an operational description, mostly by means of some examples.

Consider the first order language L, and a model M.

$$L = \langle P, F, C \rangle$$

$$M = \langle D, P, F, C \rangle$$

As usual, L is determined by a collection, P, of predicate symbols, a collection, F, of function symbols, and a collection, C, of constant symbols (Kleene[1952] pp. 83-93). M is a structure which contains a domain D, and the predicates, functions and objects which correspond to the symbols in L.

$$S = \langle \mathcal{D}, \mathcal{P}, \mathcal{F}, \mathcal{C} \rangle$$

Loosely speaking, a simulation structure, S, also has a domain, *D*, a set of "predicates", *P*, a set of "functions", *F*, and a distinguished subset, *C*, of its domain. However, they have strong restrictions. Since we are imagining simulation structures as the mechanizable analogues of models we want to be able to actually implement them on a computer. To facilitate this we imagine that we intend to use a computer language in which there is some reasonable collection of data structures. In FOL we use LISP. The domain of a simulation structure is presented as an algorithm that acts as the characteristic function of some subset of the data structures. For example, if we want to construct a simulation structure for Peano arithmetic the domain is specified by a LISP function which returns T (for true) on natural numbers and NIL (for false) on all other s-expressions. Each "predicate" is represented by an algorithm that decides for each collection of arguments if the predicate is true or

false or if it doesn't know. This algorithm is also total. Notice that it can tell you what is false as well as what true. Each "function" is an algorithm that computes for each set of arguments either a value or returns the fact that it doesn't know the answer. It too is total. The distinguished subset of the domain must also be given by its characteristic function. These restrictions are best illustrated by an example. A possible simulation structure for Peano arithmetic together with a relation symbol for "less than" is

$$S = \langle \text{natural numbers}\ ,\ \langle \{2<3, \neg 5<2\} \rangle\ ,\ \langle \text{plus} \rangle\ ,\ \{2,3\} \rangle$$

I have not presented this simulation structure by actually giving algorithms but they can easily be supplied. This simulation structure contains only two facts about "less than" - two is less than three, and it's false that five is less than two. As mentioned above, since this discussion is informal {2<3,¬5<2} should be taken as the description of an algorithm that answers correctly the two questions it knows about and in all other cases returns the fact that it cannot decide. "plus" is the name of an algorithm that computes the sum of two natural numbers. The only numerals that have interpretations are two and three. These have their usual meaning.

Intuitively, if we ask is "2<3" (where here "2" and "3" are numerals in L) we get the answer yes. If we ask is "5<2" it says, "I don't know"! This is because there is no interpretation in the simulation structure of the numeral "5". Curiously, if you ask is "2+3<2" it will say false. The reason is that the simulation structure has an interpretation of "+" as the algorithm "plus" and 5 is in the domain even though it is not known to be the interpretation of any numeral in L.

A more reasonable simulation structure for Peano arithmetic might be

$$S = \langle \text{natural numbers}\ ,\ \langle \text{lessthan} \rangle\ ,\ \langle \text{suc,pred,plus,times} \rangle\ ,\ \text{natural numbers} \rangle$$

Simulation structures are not models. One difference is that there are no closure conditions required of the function fragments. Thus we could know that three times two is six without knowing about the multiplicative properties of two and six.

Just as in the case of a model, we get a natural interpretation of a language with respect to a simulation structure. This allows us to introduce the idea of a sentence of L being satisfiable with respect to a simulation structure. Because of the lack of closure conditions and the partialness of the "predicates", etc., (unlike ordinary satisfaction) this routine will sometimes return "I don't know". There are several reasons for this. Our mechanized satisfaction cannot compute the truth or falsity of quantified formulas. This in general requires an infinite amount of computing. It should be remarked that this is exactly why we have logic at all. It facilitates our reasoning about the result of an infinite amount of computation with a single sentence.

It is also important to understand that we are not introducing a three valued logic or partial functions. We simply acknowledge that, with respect to some simulation structures, we don't have any information about certain expressions in our language.

Below is an example of the FOL commands that would define this language, assert some axioms and build this simulation structure. As mentioned above, in the FOL system one of the few defaults is

that numerals automatically come declared as individual constants and are attached to the expected integers. Thus the following axiomatization includes the numerals and their attachments by default.

```
DECLARE INDVAR n m p q ∈ NATNUM;
DECLARE OPCONST suc pred(NATNUM)=NATNUM;
DECLARE OPCONST +(NATNUM,NATNUM)=NATNUM [R←450,L←455];
DECLARE OPCONST *(NATNUM,NATNUM)=NATNUM [R←550,L←555];
DECLARE PREDCONST <(NATNUM,NATNUM) [INF];
DECLARE PREDPAR P(NATNUM);

AXIOM   Q:
        ONEONE: ∀n m.(suc(n)=suc(m)⊃n=m);
        SUCC1:  ∀n.¬(0=suc(n));
        SUCC2:  ∀n.(¬0=n⊃∃m.(n=suc(m)));
        PLUS:   ∀n.n+0=n
                ∀n m.n+suc(m)=suc(n+m);
        TIMES:  ∀n.n*0=0
                ∀n m.n*suc(m)=(n*m)+m; ;;

AXIOM   INDUCT: P(0) ∧ ∀n.(P(n)⊃P(suc(n))) ⊃ ∀n.P(n); ;

REPRESENT {NATNUM} AS NATNUMREP;
ATTACH suc  ↔ (LAMBDA (X) (ADD1 X));
ATTACH pred ↔ (LAMBDA (X) (COND ((GREATERP X 0)(SUB1 X)) (T 0)));
ATTACH +    ↔ (LAMBDA (X Y) (PLUS X Y));
ATTACH *    ↔ (LAMBDA (X Y) (TIMES X Y));
ATTACH <    ↔ (LAMBDA (X Y) (LESSP X Y));
```

The first group of commands creates the language. The second group are Robinson's axioms Q without the equality axioms (Robinson[1950]). The next is the induction axiom. The fourth group makes the semantic attachments that build the simulation structure. The expressions containing the word "LAMBDA" are LISP programs. I will not explain the REPRESENT command as it is unimportant here. The parts of the declarations in square brackets specify binding power information to the FOL parser.

Using these commands we can ask questions like

```
*****SIMPLIFY 2+3<pred(7);

*****SIMPLIFY 4*suc(2)+pred(3)<pred(pred(8));
```

Of course semantic simplification only works on ground terms, i.e. only on those quantifier free expressions whose only individual symbols are individual constants. Furthermore, such an expression will not evaluate unless all the constants have attachments, and there is a constant in the language for value of the expression. Thus a command like

```
*****SIMPLIFY n*0<3;
```

where n is a variable will not simplify.

This facility may seem weak as we usually don't have ground expressions to evaluate. Below I will

show that when we use the metatheory and the metametatheory we frequently do have ground terms to evaluate, thus making this a very useful tool.

Section 5 Syntactic simplifier

FOL also contains a syntactic simplifier, called REWRITE. This facility allows a user to specify a particular set of universally quantified equations or equivalences as rewriting rules. We call such a collection a *simplification set*. The simplifier uses them by replacing the left hand side of an equation by its right hand side after making the appropriate substitutions for the universal variables.

For example, ∀x y.car(cons(x,y))=x will rewrite any expression of the form car(cons(t_1, t_2)) to t_1, where t_1 and t_2 are arbitrary terms.

When given an expression to simplify, REWRITE uses its entire collection of rewrite rules over and over again until it is no longer possible to apply any. Unfortunately, if you give it a rule like

∀x y.x+y=y+x

it will simply go on switching the two arguments to "+" forever. This is because the rewritten term again matches the rule. This is actually a desired property of this system. First, it is impossible in general to decide if a given collection of rewrite rules will lead to a non-terminating sequence of replacements. Second any simple way of guaranteeing termination will exclude a lot of things that you really want to use. For example, suppose you had the restriction that no sub-expression of the right hand side should match the left hand side of a rewrite rule. Then you could not include the definition of a recursive function even if you know that it will not rewrite itself forever in the particular case you are considering. This case occurs quite frequently.

This simplifier is quite complicated and I will not describe its details here. There are three distinct subparts.

1) A matching part. This determines when a left hand side matches a particular formula.

2) An action part. This determines what action to take when a match is found. At present the only thing that the simplifier can do under the control of a user is the replacement of the matched expression by its right hand side.

3) The threading part. That is, given an expression in what order should the sub-expressions be matched.

The details of these parts are found in Weyhrauch[NOTE6]. This simplifier behaves much like a PROLOG interpreter(Warren[1977]), but treates a more extensive collection of sentences. I will say more about first order logic as a programming language (Kowalski[1974]) below.

In appendix [E] here is a detailed example which illustrates the control structure of the simplifier.

Section 6 A general first order logic expression evaluator

Unfortunately, neither of the above simplifiers will do enough for our purposes. This section describes an evaluator for arbitrary first order expressions which is adequate for our needs. I believe that the evaluator presented below is the only natural way of considering first order logic as a programming language.

Consider adding the definition of the factorial function to the axioms above.

```
DECLARE OPCONST fact(NATNUM)=NATNUM;
AXIOM FACT: ∀n.fact(n)=IF n=0 THEN 1 ELSE n*fact(pred(n)) ;;
```

Suppose we ask the semantic simplifier to

```
*****SIMPLIFY fact(3);
```

Quite justifiably it will say, "no simplifications". There are no attachments to `fact`.

Now consider what the syntactic simplifier will do to `fact(3)` just given the definition of factorial.

```
fact(3)=IF 3=0 THEN 1 ELSE 3*fact(pred(3))
       =IF 3=0 THEN 1
           ELSE 3*(IF pred(3)=0 THEN 1 ELSE pred(3)*fact(pred(pred(3))))
.
.
.
```

The rewriting will never terminate because the syntactic simplifier doesn't know anything about `3=0` or `pred(3)=0`, etc. Thus it will blindly replace `fact` by its definition forever.

The above computation could be made to stop in several ways. For example, it would stop if

(3=0)≡FALSE
∀X Y.(IF FALSE THEN X ELSE Y)=Y
pred(3)=2
fact(2)=2
3*2=6

were all in the simplification set.

Or if we stopped after the first step and the semantic attachment mechanism knew about = on integers and pred then we would get

```
syn     fact(3)=IF 3=0 THEN 1 ELSE 3*fact(pred(3))
sem            =3*fact(2)
syn            =3*(IF 2=0 THEN 1 ELSE 2*fact(pred(2)))
sem            =3*(2*fact(1))
syn            =3*(2*(IF 1=0 THEN 1 ELSE 1*fact(pred(1))))
sem            =3*(2*(1*fact(0)))
syn            =3*(2*(1*(IF 0=0 THEN 1 ELSE 0*fact(pred(0)))))
sem            =3*(2*(1*1))
                halt
```

This "looks better". The interesting thing to note is that if we had a semantic attachment to * this would have computed fully. On the other hand if we had added the definition of * in terms of + then it would have reduced to some expression in terms of addition. In this case if we didn't have a semantic attachment to * but only to + this expression would have also "computed" 6.

Notice that this combination of semantic plus syntactic simplification acts very much like an ordinary interpreter. We have implemented such an intepreter and it has the following properties.

1) It will compute any function whose definition is hereditarily built up, in a quantifier free way, out of functions that have attachments, on domains that have attachments.

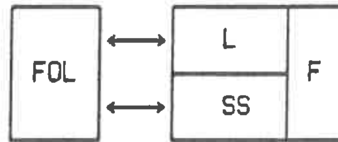
2) Every step is a logical consequence of the function definitions and the semantic attachments. This implies that as a programming system this evaluator cannot produce an incorrect result. Thus the correctness of the expression as a "program" is free.

This evaluator will be used extensively below. I would like to remark that this evaluator is completely general in that it takes an arbitrary set of first order sentences and an arbitrary simulation structure and does both semantic evaluation and syntactic evaluation until it no longer knows what to do. You should observe that the expressions you give it are any first order sentences you like. In this sense it is a substantial extension of PROLOG (Warren[1977]) that is not tied down to clause form and skolemization. In the examples below those familiar with PROLOG can see the naturalness that this kind of evaluation of first order sentences allows. Just look at the above definition of factorial. We allow for functions to have their natural definitions as terms. The introduction of semantic simplifications also gives arbitrary interpretations to particular predicates and functions.

Section 7 Systems of languages and simulation structures

As mentioned in the introduction, one of the important things about the FOL system is its ability to deal with metatheory. In order to do this effectively we need to conceptualize on what *objects* FOL is manipulating. As I have described it above, FOL can be thought of as always having its attention directed at a object consisting of a language, L, a simulation structure, SS, attachments between the two, and a set of facts, F, i. e., the finite set of facts that have asserted or proved.

We can view this as the picture.



Below I will sometimes represent these 3-tuples schematically as

<L,SS,F>

I will abuse language in two ways. Most of the time I will call these structures *LS pairs*, to emphasize the importance of having explicit representations as data structures for languages, the objects mentioned and the correspondence between the two. At other times I will call this kind of structure a *theory*.

The importance of LS pairs cannot be overestimated. I believe they fill a gap in the kinds of structures that have previously used to formalize reasoning. Informally their introduction corresponds to the recognition that we reason about objects, and that our reasoning makes use of our understanding of the things we reason about.

Let me give a mathematical example and a more traditional AI example.

Consider the following theorem of real analysis (Royden[1963]).

THEOREM: Let $<F_n>$ be a sequence of nonempty closed intervals on the real line with $F_{n+1} \subset F_n$, then, if one of the F_n is bounded, the intersection of the F_n is nonempty.

The goal I would like you to consider is: Give an example to show that this conclusion may be false if we do not require one of these sets to be bounded.

The usual counterexample expected is the set of closed intervals $[n,\infty]$ of real numbers. Clearly none of these are bounded and their intersection is empty. The idea of describing a counterexample simply cannot be made sense of if we do not have some knowledge of the models of our theories. That is, we need some idea of what objects we are reasoning about. The actualization of objects in the form of simulation structures is aimed in part at this kind of question.

As an AI example I will to use is the missionaries and cannibals puzzle. As the problem is usually posed we are asked toe imagine three missionaries, three cannibals, a river, its two banks and a boat. We then build a theory about those objects. The point here is that we have have explicitly distinguished between the objects mentioned and our theory of these objects. That is, we have (in our minds, so to speak) an explicit image of the objects we are reasoning about. This is a simulation structure as defined above.

One could argue that simulation structures are just linguistic objects anyway and we should think of them as part of the theory. I believe this is fundamentally wrong. In the examples below we make essential use of this distinction between an object and the words we use to mention it.

In addition to the practical usefulness that simulation structures have, they allow us, in a mechanized way, to make sense out of the traditional philosophic questions of sense and denotation. That is, they allow us to mention in a completely formal and natural way the relation between the the objects we are reasoning about and the words we are using to mention them. This basic distinction is exactly what we have realized by making models of a language into explicit data structures. This is more completely discussed in Weyhrauch[NOTE2] and Weyhrauch[NOTE17].

One way of describing what we have done is that we have built a data structure that embodies the idea that when we reason we need a language to carry out our discussions, some information about the object this language talks about, and some facts about the objects expressed in the language. This structure can be thought of as a the mechanizable analogue of a theory. Since it is a data structure like any other we can reason about this theory by cosidering it as an object described by some other theory. Thus we give up the idea of a "univeral" language about all objects to gain the ability to formally discuss our various theories of these objects.

Currently FOL has the facility to simultaneously handle as many LS pairs as you want. It also provides a facility for changing ones attention from one pair to another. We use this feature for changing attention from theory to metatheory as explained below.

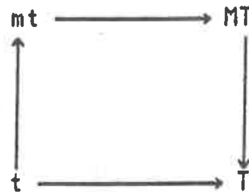
Section 8 Metatheory

I have already used the word "metatheory" many times and since it is an important part of what follows I want to be a little more careful about what I mean by it. In this note I am not concerned with the philosophical questions logicians raise in discussions of consistency, etc. I am interested in how metatheory can be used to facilitate reasoning using computers. One of the main contributions of this paper is the way in which I use *reflection principles* (Feferman[1962]) to connect theories and metatheories. Reflection principles are described in the next section.

In this section I do not want to justify the use of metatheory. In ordinary reasoning it is used all the time. Some common examples of metatheory are presented in the next section. Here I will present examples taken from logic itself, as they require no additional explanation.

In its simplest form metatheory is used in the following way. Imagine that you want to prove some theorem of the theory, i.e. to extend the facts part, f, of some LS to F. One way of doing this is by using FOL in the ordinary theorem constructing way to generate a new fact about the objects mentioned by the theory. An alternative way of doing it may be to use some metatheorem which "shortens" the proof by stating that the *result* of some complicated theorem generation scheme is valid. Such shortcuts are sometimes called *subsidiary deduction rules* (Kleene[1952] p. 86).

We represent this schematically by the following diagram.



Consider the metatheorem: if you have a propositional WFF whose only sentential connective is the equivalence sign, then the WFF is a theorem if each sentential symbol occurs an even number of times. In FOL this could be expressed by the metatheory sentence

```
∀w. (PROPWFF(w)∧CONTAINS_ONLY_EQUIVALENCES(w) ⊃
        (∀s. (SENTSYM(s)∧OCCURS(s,w)⊃EVEN(count(s,w)))⊃THEOREM(w)))
```

The idea of this theorem is that since it is easier to count than to construct the proofs of complicated theorems, this metatheorem can save you the work of generating a proof. In FOLs metatheory this theorem can be either be proved or simply asserted as axiom.

We use this theorem by directing our attention to the metatheory and instantiating it to some WFF and proving that THEOREM(w). Since we are assuming that our axiomatization of the metatheory is sound, we are then justified in asserting w in the theory. The reflection principles stated below should be looked at as the *reason* that we are justified in asserting w. More detailed examples will be given in the next section.

In FOL we introduce a special LS pair META. It is intended that META is a general theory of LS pairs. When we start, it contains facts about only those things that are common to all LS pairs. Since META behaves like any other first order LS pair additional axioms, etc., can be added to it. This allows a user to assert many other things about a particular theory. Several examples will be given below.

An example of how we axiomatize the notion of well formed formulas is

```
∀ls expr.(WFF(expr,ls)≡PROPWFF(expr,ls)∨QUANTWFF(expr,ls))
```

An expression is a WFF (relative to a particular LS pair) if it is either a propositional WFF or a quantifier WFF.

```
∀ls expr.(PROPWFF(expr,ls)≡APPLPWFF(expr,ls)∨AWFF(expr,ls))
```

A propositional WFF is either an application propositional WFF or an atomic WFF.

```
∀ls expr.(APPLPWFF(expr,ls)≡PROPCONN(mainsym(expr)) ∧
        ∀n.(0<n∧n≤arity(mainsym(expr),ls)⊃WFF(arg(n,expr),ls)))
```

An application propositional WFF is an expression whose main symbol is a propositional connective, and if n is between 0 and the arity of the propositional connective then the n-th argument of the expression must be a WFF. Notice that this definition is mutually recursive with that of PROPWFF and WFF.

```
∀ls expr.(QUANTWFF(expr,ls) ≡
          QUANT(mainsym(expr)) ∧ INDVAR(bvar(expr),ls) ∧ WFF(matrix(expr),ls)
```

A quantifier WFF is an expression whose main symbol is a quantifier, whose bound variable is an individual variable and whos matrix is a WFF.

```
∀ls expr.(AWFF(expr,ls)≡SENTSYM(expr,ls)∨APPLAWFF(expr,ls)
```

An atomic WFF is either a sentential symbol or a application atomic WFF.

```
∀ls expr.(APPLAWFF(expr,ls)≡PREDSYM(mainsym(expr),ls)∧
        ∀n.(0<n∧n≤arity(mainsym(expr),ls)⊃TERM(arg(n,e),ls)))
```

An atomic application WFF is an expression whose main symbol is a predicate symbol and each argument of this expression in the appropriate range is a TERM.

```
∀ls expr.(TERM(expr,ls)≡INDSYM(expr,ls)∨APPLTERM(expr,ls))

∀ls expr.(APPLTERM(expr,ls)≡OPSYM(mainsym(expr),ls)∧
        ∀n.(0<n∧n≤arity(mainsym(expr,ls))⊃TERM(arg(n,expr),ls)))
```

A TERM is either an individual symbol or an application TERM, etc.

This is by no means a complete description of LS pairs but it does give some idea of what sentences in META look like. These axioms are collected together in appendix C. The extent of META isn't critical and this paper is not an appropriate place to discuss its details as implemented in FOL. Of course, in addition to the descriptions of the objects contained in the LS pair, it also has axioms about what it means to be a "theorem", etc.

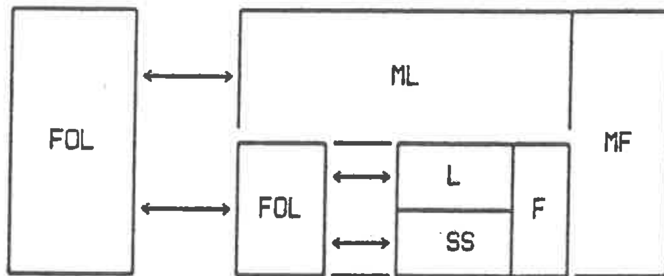
Thus META contains the proof theory and some of the model theory of an LS pair. As with any first order theory its language is built up of predicate constants, function symbols and individual constant symbols. What are these? There are constants for WFFs, TERMs, derivations, simulation structures, models, etc. It contains functions for doing "and introductions", for substituting TERMs into WFFs, constructors and selectors on data structures. It has predicates "is a well formed formula", "is a term", "equality of expressions except for change of bound variables", "is a model", "is a simulation structure", "is a proof", etc.

Suppose that we are considering the metatheory of some particular LS pair, LS0=<L,SS,F>. At this point we need to ask a critical question.

What is the *natural* simulation structure for META?

The answer is: 1) we actually have in hand the object we are trying to axiomatize, LS0, and 2) the code of FOL itself contains algorithms for the predicates and functions mentioned above.

This leads to the following picture.



It is this picture that leads to the first hint of how to construct a system of logic that can *look* at itself. The trick is that when we carry out the above construction on a computer, the two boxes labeled FOL are physically the same object. I will expand on this in the section on self reflection.

Section 9 Reflection

A *reflection principle* is a statement of a relation between a theory and its metatheory. Although logicians use considerably stronger principles (see Feferman[1962]), we will only use some simple examples, i.e., statements of the soundness of the axiomatization in the metatheory of the theory. An example of a reflection principle is

```
(in T)            \|/
                   w
              ----------------
(in MT)       Prf("\|/","w")
```

In natural deduction formulations of logic proofs are represented as trees. In the above diagram let '\|/' be a proof and 'w' be the well formed formula which it proves. Let Prf be a predicate constant in the metatheory, with Prf(p,x) true if and only if p is a proof, x is a wff, and p is a proof of x. Also, let "\|/" and "w" be the the individual constants in the metatheory that are the names of '\|/' and 'w', respectively. Then the above reflection principle can be read as: if '\|/' is a proof of 'w' in the theory we are allowed to assert Prf("\|/","w") in the metatheory and vice versa.

A special case of this rule is if 'w' has no dependencies, i.e. it is a theorem.

```
(in T)                  w    with no dependencies
                 ____________
(in META)        THEOREM("w")
```

A simpler example is

```
(in T)          an individual variable x
               __________________________
(in META)              INDVAR("x")
```

Suppose we have the metatheorem

```
ANDI:    ∀thm1 thm2.THEOREM(mkand(wffof(thm1),wffof(thm2)))
```

This (meta)theorem says that if we have any two theorems of the theory, then we get a theorem by taking the conjunction of the two wffs associated with these theorems. I need to remark about what I mean by the WFF associated with a theorem. Theorems should be thought of as particular kinds of facts. Facts are more complicated objects than only sentences. They also contain other information. For example, they include the reason we are willing to assert them and what other facts their assertion depends on. Facts also have names. Thus the above line is an incomplete representation of the metatheoretic fact whose name is ANDI. The WFF associated with this fact is just

```
∀thm1 thm2.THEOREM(mkand(wffof(thm1),wffof(thm2)))
```

Remember the reflection principle associated with THEOREM is

```
(in T)                  w    with no dependencies
                 ____________
(in META)        THEOREM("w")
```

Thus we can imagine the following scenario.

Suppose we have two theorems called T1 and T2 in the theory. These facts are represented as FOL data structures. Now suppose we want to assert the conjunction of these two in the theory. One way

to do this is to use the *and introduction* rule of FOL. This example, however, is going to do it the hard way. First we switch to the metatheory carrying with us the data structures for T1 and T2. We then declare some individual constants t1 and t2 to be of sort theorem in the metatheory, and use the semantic attachment mechanism at the metatheory level to attach the data structures for T1 and T2 to the individual constants t1 and t2 respectively. We then instantiate the metatheorem ANDI to t1 and t2. Note that the resulting formula is a ground instance of a sentence without quantifiers. This means that if we have attachments to all the symbols in the formula we can evaluate this formula. In this theorem we have the predicate constant THEOREM. In META it is the only constant in this sentence that is not likely to have an attachment. This is because being a theorem is not in general decidable. Fortunately, we can still use the reflection principle, because we understand the intended interpretation of the metatheory. So if we use the evaluator on

```
mkand(wffof(t1),wffof(t2)),
```

we can pick up the data structure computed in the model, instead of the term. Then since we know that it is a theorem we can make it into one in the theory.

This idea has been implemented in a very nice way. In FOL we have the following command.

```
*****REFLECT ANDI T1,T2;
```

The reflect command understands some fixed list of reflection principles, which includes those above. When FOL sees the word "REFLECT" it expects the next thing in the input stream to be the name of a fact in the metatheory of the current theory. So it switches to the metatheory and scans for a fact. It then finds that this fact is universally quantified with two variables ranging over facts in the theory. It switches back to the theory and and scans for two facts in the theory. In holds on to the data structures that it gets in that way and switches back to the metatheory. Once there it makes the attachments of these structures to two newly created individual constants, first checking whether or not it already has an attachment for either of these structures. We then instantiate the theorem to the relevant constants and evaluate the result. When we look at the result we notice that it will probably simply evaluate to

```
THEOREM(mkand(wffof(t1),wffof(t2)))
```

This is because we don't have an attachment to THEOREM and we also don't have a individual constant which names mkand(wffof(t1),wffof(t2)). But what we do notice is that we have reduced the theorem to the form THEOREM(-), and we know about reflection principles involving THEOREM. Thus we go back and evaluate its argument, mkand(wffof(t1),wffof(t2)), and see if it has a value in the model. In this case since it does we can reflect it back into the theory, by returning to the theory and constructing the appropriate theorem.

This example is a particularly simple one, but the feature described is very general. I will give some more examples below. One thing I want to emphasis here is that what we have done is *to change theorem proving in the theory into evaluation in the metatheory*. I claim that this idea of using reflection with evaluation is the most general case of this and that this feature is not only an extremely useful operation but a fundamental one as well. It is the correct technical realization of how we can *use*

declarative information. That is, the only thing you expext a sentence to do is to take its intended interpretation seriously.

The metatheorem we use in reflection does not need to be of the form THEOREM(-). This is the reason for needing the evaluater rather than simply either the syntatic or the semantic simplification mechanisms alone. Consider the following general metatheorems *about* the theory of natural numbers. If you find it hard to read it is explained in detail in the next subsection.

```
∀vl x.(LINEAREQ(wffof(vl),x) ⊃ THEOREM(mkequal(x,solve(wffof(vl),x))));

∀w x.(LINEAREQ(w,x) ≡
        mainsym(w)=Equal ∧
        (mainsym(lhs(w))=Sum ∨ mainsym(lhs(w))=Diff) ∧
        larg(lhs(w))=x ∧
        NUMERAL(rarg(lhs(w))) ∧
        NUMERAL(rhs(w))) ∧
        (mainsym(lhs(w))=Sum ⊃ mknum(rhs(w))>mknum(rarg(lhs(w))));

∀w x.(solve(w,x)=IF mainsym(lhs(w))=Sum
                   THEN mknumeral(mknum(rhs(w))-mknum(rarg(lhs(w))))
                   ELSE mknumeral(mknum(rhs(w))+mknum(rarg(lhs(w)))) );;
```

These axioms together with the reflection mechanism extend FOL, so that it can solve equations of the form x+a=b or x-a=b, when there is a solution in natural numbers. We could have given a solution in integers or for n simultaneous equations in n unknowns. Each of these requires a different collection of theorems in the metatheory.

This axiomatization may look inefficient but let me point out that solve is exactly the same amount of writing that you would need to write code to solve the same equation. The definition of LINEAREQ is divided into two parts. The first five conjunctions are to do type checking, the sixth conjunct checks for the existence of a solution before you try to use solve to find it. The above example actually does a lot. It type checks the argument, guarantees a solution and then finds it.

Section 9.1 Can a program learn

In this section I want to digress from the stated intent of the paper and speak a little more generally about AI. It is my feeling that it is the task of AI to explain how it might be possible to build a computer individual that we can interact with as a partner in some problem solving area. This leads to the question of what kinds of conversations we want to have with such an individual and what the nature of our interactions with him should be.

Below I describe a conversation with FOL about solving linear equations. As an example it has two purposes. First it is to illustrate the sense in which FOL is a conversational machine that can have rich discussions (even if not in natural language). And second to explore my ideas of what kinds of dialogues we can have with machines that might be construed as the computer individual learning. I believe that after the discussion presented below we could reasonably say that FOL had learned to

solve linear equations. That is, by having this conversation with FOL we have taught FOL some elementary algebra.

Imagine that we have told FOL about Peano arithmetic. We could do this by reading in the axioms presented in appendix B. We can then have a discussion about *numbers*. For example, we might say

```
*****ASSUME n+2=7;

1 (n+2)=7  (1)
```

and we might want to know what is the value of n. Since we are talking about numbers in the language of Peano arithmetic the *only* way we have of discussing this problem is by using facts about numbers. Suppose that we already know the theorems

```
THM1:   ∀p q m.(p=q⊃p-m=q-m)
THM2:   ∀p q m.(p+q)-r=p+(q-r)
THM3:   ∀p.(p+0)=p
```

Then we can prove that n=5

```
*****∀E THM1 n+2,7,2;

2 (n+2)=7⊃((n+2)-2)=(7-2)

*****EVAL  BY {THM2,THM3};

3 (n+2)=7⊃n=5

*****⊃E 1,3;

4 n=5  (1)
```

In this case what we have done is proved that n=5 by using facts about *arithmetic*. To put it in the perspective of *conversation*, we are having a discussion about numbers.

If we were expecting to discuss with FOL many such facts, rather than repeating the above conversation many times we might choose to have a single discussion about *algebra*. This would be carried out by introducing the notion of *equation* and a description of how to *solve* them. What is an equation? Well, it simply turns out to be a special kind of *atomic formula* of the theory of arithmetic. That is, we can discuss the solution to equations by using metatheory.

In FOL we switch to the metatheory. We make some declarations and then define what it means to be a linear equation with a solution by stating the axiom

```
∀w x.(LINEAREQ(w,x) ≡
        mainsym(w)=Equal ∧
        (mainsym(lhs(w))=Sum ∨ mainsym(lhs(w))=Diff) ∧
        larg(lhs(w))=x ∧
        NUMERAL(rarg(lhs(w))) ∧
        NUMERAL(rhs(w))) ∧
        (mainsym(lhs(w))=Sum ⊃ mknum(rhs(w))>mknum(rarg(lhs(w))))
```

Here w is a (meta)variable ranging over WFFs, and x is a (meta)variable ranging over individual variables. Spelled out in english this sentence says that a well formed formula is a linear equation if and only if: i) it is an equality, ii) its left hand side is either a sum or a difference, iii) the left hand argument of the left hand side of the equality is x, iv) then right hand argument of the left hand side of the equality is a numeral, v) the right hand side of the equality is a numeral and vi) if the left hand side is a sum then the number denoted by the numeral on the right hand side is greater than the number denoted by the numeral appearing in the left hand side.

In more mathematical terminology it is: that the well formed formula must be either of the form $x+a=b$ or $x-a=b$ where a and b are numerals and x is an individual variable. Since here we are only interested in the natural numbers, the last restriction in the definition of LINEAREQ is needed to guarantee the existence of a solution.

We also describe how to find out what is the *solution* to an equation.

```
∀w x.(solve(w,x)=IF mainsym(lhs(w))=Sum
                   THEN mknumeral(mknum(rhs(w))-mknum(rarg(lhs(w))))
                   ELSE mknumeral(mknum(rhs(w))+mknum(rarg(lhs(w)))) );;
```

This is a function definition in the meta theory. Finally we assert that if we have an equation in the theory then the numeral constructed by the solver can be asserted to be the answer.

```
∀vl x.(LINEAREQ(wffof(vl),x) ⊃ THEOREM(mkequal(x,solve(wffof(vl),x))));
```

We then tell FOL to remember these facts in a way that is convenient to be used by FOL's evaluator.

This then is the conversation we have with FOL about equations. Now we are ready to see how FOL can use that information, so we switch FOL's attention back to the theory. Now, whenever we want to solve a linear equation, we simply remark, using the reflect command, that he should remember our discussion about solving equations.

We can now get the effect of the small proof above by saying

```
*****REFLECT SOLVE 1;

5 n=5  (1)
```

In effect FOL has learned to solve simple linear equations.

We could go on to ask FOL to prove that the function solve actually provides a solution to the equation, rather than our just telling FOL that it does, but this is simply a matter of sophistication. It has to do with the question of what you are willing to accept as a justification.

One reasonable justification is that the teacher told me. This is exactly the state we are in above. On the other hand if that is not satisfactory then it is possible to discuss with FOL the justification of the solution. This could be accomplished by explaining to FOL (in the metatheory) not to assert the solution of the equations in the theory, but rather to construct a proof of the correctness of the solution as we did when we started. Clearly this can be done using same machinery that was used here. This is important because it means that our reasoning system does not need to be expanded. We only have to tell it more information.

A much more reasonable alternative is to tell FOL (again in the metatheory) two things. One is what we have above, i.e., to assert the solution of the equation. Second is that if asked to justify the solution, then to produce that proof. This combines the advantages each of the above possibilities. I want to point out that this is very close to the kinds of discussions that you want to be able to have with people about simple algebra.

Informally we always speak about solving *equations*. That is, we think of them as syntactic and learn how to manipulate them. This is not thinking about them as relations, which is their usual first order interpretation. In this sense going to the metatheory and treating them as syntactic objects is very close to our informal use of these notions.

I believe that this is exactly what we want in an AI system dealing with the question of *teaching*. Notice that we have the best of both worlds. On the one hand, at the theory level, we can "execute" this learning, i.e. use it, and on the other hand, at the metatheory level, we can reason about what we have learned about manipulating equations. In addition the correct distinction between equations as facts and equations as syntactic objects has been maintained. The division between theory and metatheory has allowed us to view the same object in both these ways without contradiction or the possibility of confusion.

As is evident from the above description, one of the things we have here is a very general purpose programming system. In addition it is extendable. Above we have showed how to introduce any new subsidiary deduction rule that you chose, "simply" by telling FOL what you would like it to do. This satisfies the desires of Davis and Schwartz[1977] but in a setting not restricted to the theory of hereditarily finite sets. As I said above: we are using first order logic in what I believe is its most general and natural setting.

There are hundreds of examples of this kind where their natural description is in the metatheory. In a later paper I will discuss just how much of the intent of natural language can only be understood if you realize that a lot of what we say is about our use of language, not about objects in the world. This kind of conversation is most naturally carried out in the metatheory with the use of the kind of self-reflective structures hinted about below.

Section 9.2 Using metametatheory

We can take another leap by allowing ourselves to iterate the above procedure and using metametatheory. This section is quite sketchy but would require a full paper to write out the details.

We can use the metametatheory to describe declaratively what we generally call heuristics. Consider an idealized version of the Boyer and Moore[1979] theorem prover for recursive functions. This prover looks at a function definition and tries to decide whether or not to try to prove some property of the function using either CAR-induction or CDR-induction, depending on the form of the function definition.

CAR and CDR inductions are axiom schemas, which depend on the form of the function definition and the WFF being proved. Imagine that these had already told to FOL in the metatheory. Suppose we had called them CARIND and CDRIND. Then using the facilities described above we could use these facts by reflection. For example,

```
*****ASSUME ∀u.counta(u)=if atom(u) then u else counta(car(u));

1 ∀u.counta(u)=if atom(u) then u else counta(car(u))  (1)

*****REFLECT CARIND 1 ∀u.ATOM(counta(u));

2 ∀u.ATOM(counta(u))  (1)

****ASSUME ∀u.countd(u)=if null(u) then 'NIL else countd(cdr(u));

3 ∀u.countd(u)=if null(u) then 'NIL else countd(cdr(u))  (3)

*****REFLECT CDRIND 3 ∀u.countd(u)='NIL;

4 ∀u.countd(u)='NIL  (3)
```

The use of this kind of command can be discussed in the metametatheory. We introduce a function, T_reflect, in the metametatheory, which we attach using semantic attachment to the FOL code that implements the above reflect command. Thus T_reflect takes a fact, vl and a list of arguments, and if it succeeds returns a new proof whose last step is the newly asserted fact and if it fails returns some error. Suppose also that Carind and Cdrind are the metametatheory's name for CARIND and CDRIND respectively. Then suppose in the metametatheory we let

```
WFF1=mkforall(T_u,mkapplw1(T_ATOM,(mkapplt1(T_counta,T_u))))

WFF2=mkforall(T_u,mkequal(mkapplt1(T_countd,T_u),mksexp('NIL)))
```

that is, ∀u.ATOM(counta(u)) and ∀u.countd(u)='NIL, respectively. We prefix things refering to the theory by "T_". The effect of the above commands (withhout actually asserting anything) is gotten by using the FOL evaluator on

T_reflect(Cdrind,<T_fact(1),WFF1>) and

```
T_reflect(Cdrind,<T_fact(3),WFF1>).
```

Now suppose that vl ranges over facts in the theory, f ranges over function symbols, and w ranges over WFFs. The micro Boyer and Moore theorem prover can be expressed by

```
∀vl f w.
  (IS_T_FUNDEF(vl,f) ⊃
     (CONTAINS_ONLY_CAR_RECURSION(vl,f) ∧ NOERROR(T_REFLECT(Carind,<vl,w>)) ⊃
          T_THEOREM(last_T_step(T_REFLECT(Carind,<vl,w>))))  ∧
     (CONTAINS_ONLY_CDR_RECURSION(vl,f) ∧ NOERROR(T_REFLECT(Cdrind,<vl,w>)) ⊃
          T_THEOREM(last_T_step(T_REFLECT(Cdrind,<vl,w>))))  )
```

In the metametatheory we call this fact BOYER_and_MOORE. It is read as follows: if in the theory, vl is a function definition of the function symbol f, then if this function definition only contains recursions on car, and if when you apply reflection from the theory level to the metatheorem called Carind you don't get an error, then the result of this reflection is a theorem at the theory level, similarly for cdr induction.

As explained in the previous sections, asserting this in the metametatheory allows it to be used at the theory level by using the same reflection device as before.

When our attention is directed to the theory we can say

```
*****MREFLECT BOYER_and_MOORE 1,counta,∀u.ATOM(counta(u));

5 ∀u.ATOM(counta(u))  (1)

*****MREFLECT BOYER_and_MOORE 3,countd,∀u.countd(u)='NIL;

6 ∀u.countd(u)='NIL  (3)
```

Here MREFLECT simply means reflect into the metametatheory.

This example shows how the metametatheory, together with reflection, can be used to drive the proof checker itself. Thus we have the ability to declaratively state heuristics and have them effectively used. The ability to reason about heuristics and prove additional theorems about them provides us with an enormous extra power. Notice that we have once again changed theorem proving at the theory level into computation at the metametatheory level. This is part of the leverage that we get by having all of this machinery around simultaneously.

This example, as it is described above has not yet been run in FOL. It is the only example in this paper which has not actually been done using the FOL system, but it is clear that it will work simply given the current features.

A good way of looking at all of this is that the same *kind* of language that we use to carry on ordinary conversations with FOL can be used to discuss the control structures of FOL itself. Thus it can be used to discuss its own actions.

Section 10 Self reflection

In the traditional view of metatheory we start with a theory and we axiomatize that theory. This gives us metatheory. Later we may axiomatize that theory. That gives us metametatheory. If you believe that most reasoning is at some meta level (as I do) then this view of towers of metatheories leads to many questions. For example, how is it that human memory space doesn't overflow. Each theory in the tower seems to contain a complete description of the theory below thus exponentiating the amount of space needed!

In the section on metatheory, I introduced the LS pair, META. Since it is a first order theory just like any other, FOL can deal with it just like any other. Since META is the general theory of LS pairs and META is an LS pair this might suggest that META is also a theory that contains facts about itself. That is, by introducing the individual constant Meta into the theory META and by using semantic attachment to attach the theory (i. e., the actual machine data structure) META to Meta we can give META its own name. The rest of this section is somewhat vague. We have just begun to work out the consequences of this observation.

FOL handles many LS pairs simultaneously. I have already showed how given any LS pair we can direct FOL's attention to META using reflection. Once META has an individual constant which is a name for itself and we have attached META to this constant, then META is FOL's theory of itself. Notice several things: 1) If META has names for all of the LS pairs known to FOL then it has the entire FOL system as its simulation structure; 2) Since META is a theory about any LS pair, we can use it to reason about itself.

We can illustrate this in FOL by switching to META and executing the following command.

```
*****REFLECT ANDI ANDI ANDI;

1  ∀thm1 thm2.THEOREM(mkand(wffof(thm1),wffof(thm2))) ∧
        ∀thm1 thm2.THEOREM(mkand(wffof(thm1),wffof(thm2)))
```

The effect we have achieved is that when FOL's attention is directed at itself, then when we reflect into its own metatheory we have a system that is capable of reasoning about itself.

When looking at human reasoning I am struck by two facts. First, we seem to be able to apply the meta facts that we know to any problems that we are trying to solve, and second, even though it is possible to construct simple examples of use/mention conflicts, most people arrive at correct answers to questions without even knowing there is a problem. Namely, although natural language is filled with apparent puns that arise out of use/mention confusions, the people speaking do not confuse the names of things with the things. That is, the *meaning* is clear to them.

The above command suggests one possible technical way in which both of these problems can be addressed. The structure of FOL *knew* that the first occurrence of ANDI in the above command was a "use" and that the second and third were "mentions". Furthermore, the same routines that dealt effectively with the ordinary non self reflective way of looking at theory/metatheory relations also dealt with this case of self reflection without difficulty.

Notice that I said, "this case". It is possible with the structure that I have described above to ask META embarrassing questions. For example, if you ask META twice in a row what the largest step number in its proof is you will get two different answers. This would seem to lead to a contradiction.

The source of this problem is in what I believe is in our traditional idea of what it means to be a *rule of inference*. Self reflective systems have properties that are different from ordinary systems. In particular, whenever you "apply a rule of inference" to the facts of this system you change the structure of META itself and as a result you change the attachment to Meta. This process of having a rule of inference change the models of a theory as well as the already proven facts simply does not happen in traditional logics. This change of point of view requires a new idea of what is a valid rule of inference for such systems.

The extent of the soundness of the structure that I propose here is well beyond the scope of this elementary paper. Also FOL was built largely before I understood anything about this more general idea of rule of inference, thus the current FOL code cannot adequately implement these ideas. Some of the technical details of what I know appear in Weyhrauch[NOTE15]. One of my main current research interests is in working out the consequences of these self reflective structures.

META has many strange properties which I have just begun to appreciate and is a large topic for further research.

Section 11 Conclusion

Section 11.1 Summary of important results

I want to review what I consider to be the important results of this paper.

One is the observation that, when we reason, we use representations of the objects we are reasoning about as well as a representation of the facts about these objects. This is technically realized by FOL's manipulation of LS pairs using semantic attachment. It is incorrect to view this as a procedural representation of facts. Instead we should look at it as an ability to explicitly represent procedures. That is, simulation structures give us an opportunity to have a machine representation of the objects we want to reason about as well as the sentences we use to mention them.

Second, the evaluator I described above is an important object. When used by itself it represents a mathematical way of describing algorithms together with the assurance that they are correctly implemented. This is a consequence of the fact that the evaluator only performs logically valid transformations on the function definitions. In this way we could use the evaluator to actually generate a proof that the computed answer is correct. In these cases evaluation and deduction become the same thing. This is similar in spirit to the work of Kowalski[1974], but does not rely on any normalization of formulas. It considers the usual logical function definitions and takes their intended interpretation seriously. This evaluator works on any expression with respect to any LS pair and its implementation has proved to be only two to three times slower than a lisp interpreter.

Third is the observation that the FOL proof checker is itself the natural simulation structure for the theory META of LS pairs. This gives us a clean way of saying what the intended interpretation of META is. This observation makes evaluation in META a very powerful tool. It is also the seed of a theory of self reflective logic structures that, like humans, can reason about themselves.

Fourth is the use of reflection principles to connect an LS pair with META. This, together with the REFLECT command, is a technical explanation of what has been called the declarative/procedural controversy. Consider the META theorem ANDI described above. When we use the REFLECT command to point at it from some LS pair, ANDI is viewed procedurally. We want it to do an *and introduction*. On the other hand when we are reasoning in META, it is a sentence like any other. Whether a sentence is looked at declaratively or procedurally depends on your point of view, that is, it depends where you are standing when you look at it.

I have presented here a general description of a working reasoning system that includes not only theories but also metatheories of arbitrarily high level. I have given several examples of how these features, together with reflection can be used to dynamically extend the reasoning power of the working FOL system. I have made some references to the way in which one can use the self reflective parts of this system. I have given examples of how heuristics for using subsidiary deduction rules can be described using these structures. In addition, since everything you type to FOL refers to some LS pair, all of the above things can be reasoned about using the same machinery.

Section 11.2 Concluding remarks, history and thanks

I have tried in this paper to give a summery of the ideas which motivate the current FOL system. Unfortunately this leaves little room for complex examples so I should say a little about history, the kinds of things that have been done and what is being done now.

FOL was started in 1972 and the basic ideas for this system were already known in the summer of 1973. Many of the ideas of this system come directly out of taking seriously John McCarthy's idea that before we can ever expect to do interesting problem solving we need a device that can represent the ideas involved in the problem. I started by attempting to use ordinary first order logic and set theory to represent the ideas of mathematics. My discovery of the explicit use of computable partial models (i. e. simulation structures) came out of thinking about a general form for what McCarthy[1973] called a "computation rule" for logic, together with thinking about problems like the one about real numbers mentioned above. The first implementation of semantic evaluation was in 1974 by me. Since then it has been worked on by Arthur Thomas, Chris Goad, Juan Bulnes, and most recently by Andrew Robinson. The first aggressive use of semantic attachment was by Bob Filman[1978] in his thesis.

This idea of attaching algorithms to function and predicate letters is not new to AI. It appears first in Green[1969] I believe, but since then in too many places to cite them all. What is new here is that we have done it uniformly, in such a way that the process can be reasoned about. We have also arranged it so that there can be no confusion between what parts of our data structure is code and what parts are sentences of logic.

The real push for metatheory came from several directions. One was the realization that most of mathematical reasoning in practice was metatheoretic. This conflicted with most current theorem proving ideas of carring out the reasoning in the theory itself. Second was my desire to be able to do the proofs in Kleene[1952] about the correctness of programs. In the near future we are planning to carry out this dream. Carolyn Talcott and I plan to completely formalize LISP using all the power of the FOL system described above. In addition there will be people working on program transformations in the style of Burstall and Darlington[1977]. The third push for metatheory was a desire to address the question of common sense reasoning. This more than anything needs the ability to be able to reason about our theories of the world. One step in this direction has been taken by Carolyn Talcott and myself. We have worked out D. Michie's Keys abd boxes problem using this way of thinking and are currently writing it up.

The desire to deal with metatheory led to the invention of the FOL reflection command. Metatheory is pretty useless without a way of connecting it to the theory. I believe that I am the first to use reflection in this way.

All of the above ideas were presented at the informal session at IJCAI 1973. This panel was composed of Carl Hewitt, Allen Newell, Alan Kay, and myself.

The idea of self reflection grew out of thinking about the picture in the section on metatheory.

It has taken several years to make these routines all work together. They first all worked in June 1977 when Dan Blom finished the coding of the evaluator. I gave some informal demos of the examples in this paper at IJCAI 1977.

I suppose that here is as good a place as any to thank all the people that helped this effort. Particularly John McCarthy for his vision and for supporting FOL all these years. I would not have had as much fun doing it alone. Thanks.

I hope to write detailed papers on each of these features with substantial examples. In the meantime I hope this gives a reasonable idea of how FOL works.

Bibliography

Aristotle (-350) Organan

Boyer, R. S., and Moore, J S. (1979) A Computational Logic, to be published in the ACM Monograph Series, Academic Press.

Bulnes, J. (1978) GOAL: A Goal Oriented Command Language for Interactive Proof Construction Forthcoming Ph.D. thesis, Stanford University, Stanford.

Burstall, R.M. and Darlington, J. (1977) *A transformation system for developing recursive programs*, JACM, vol 24, no 1, pp.44-67.

Cartwright, R. (1977) Practical Formal Semantic Defintion and Verification Systems', Ph.D. thesis, Stanford University, Stanford.

Davis, M. and Schwartz, J.T. (1977) Correct-Program Technology/Extensibility of Verifiers, Two Papers on Program Verification, Courant Computer Science Report #12, New York University.

Diffie, W. (1973) PCHECK: operation of the proof checker, unpublished.

Feferman, S. (1962) *Transfinite recursive progressions of axiomatix theories*, Journal of Symbolic Logic, vol. 27, pp. 259-316.

Filman, R.E. and Weyhrauch, R.W. (1976) A FOL Primer, Stanford Artificial Intelligence Laboratory Memo AIM-228, Stanford University, Stanford.

Filman, R.E. (1978) The Interaction of Observation and Inference, fourthcoming Ph.D. Thesis, Stanford University, Stanford.

Green, C. (1969) The Application of Theorem Proving to Question-Answering Systems, Ph. D. thesis, Stanford University, Stanford.

Kelley, J.L. (1955) General Topology, D. Van Nostrand Company, Inc., Princeton, 298 pp.

Kleene, S.C. (1952) Introduction to metamathematics, D. Van Nostrand Company, Inc., Princeton, 550 pp.

Kleene, S.C. (1967) Mathematical Logic, John Wiley & Sons, Inc., New York, 398 pp.

Kowalski, R. (1974) *Predicate logic as a programming language*, Proc. IFIP Congress 1974.

Kreisel, G. (1971a) Five notes on the application of proof theory to computer science, Stanford University: IMSSS Technical Report 182, Stanford.

Kreisel, G. (1971b) *A survey of proof theory,II* in (J.E.Fenstad,ed.) **Proceedings of the Second Scandinavian Logic Symposium**, North-Holland, Amsterdam.

McCarthy, J. (1963) *A basis for a mathematical theory of computation*, in **Computer Programming and Formal Systems**, North-Holland, Amsterdam.

McCarthy, J. and Hayes, P.J. (1969) *Some Philosophical Problems from the Viewpoint of Artificial Intelligence*, in (D.Michie,ed.) **Machine Intelligence,7**, Edinburgh U.P., Edinburgh.

McCarthy, J. (1973) appendix to **PCHECK: operation of the proof checker by W. Diffie**, unpublished.

McCarthy, J. (1978) *Representation of Recursive Programs in First Order Logic*, in **Proceedings the International Conference on Mathematical Studies of Information Processing**, Kyoto, Japan.

Prawitz, D. (1965) **Natural Deduction - a proof-theoretical study**, Almqvist & Wiksell, Stockholm.

Robinson, R. M. (1950) *An essentially undecidable axiom system* in **Proc. Int. Cong. Math.**, Cambridge, Mass., vol 1, pp.729-730.

Royden, H. L. (1963) **Real Analysis**, Macmillan Company, New York.

Warren, D. (1977) *Implementing PROLOG - compiling predicate logic programs*, vol 1 and vol 2, DAI Research Report No. 39 and 40, Edinburgh.

Weyhrauch, Richard W. (1977) **FOL: A Proof Checker for First-order Logic**, Stanford Artificial Intelligence Laboratory Memo AIM-235.1, Stanford University, Stanford.

Weyhrauch, Richard W. (1978) **Lecture notes on the use of logic in artificial intelligence and mathematical theory of computation**, Summer school on the foundations of artificial intelligence and computer science (FAICS), Pisa.

This series of notes refers to my workiing papers which are sometiimes available from me. This AI memo is Informal Note 8.

Weyhrauch, Richard W. [NOTE2] **The physiology of a computer individual**, Informal Note 2, unpublished.

Weyhrauch, Richard W. [NOTE6] **FOL: a reasoning system**, Informal Note 6, unpublished.

Weyhrauch, Richard W. [NOTE15] **The logic of FOL**, Informal Note 15, unpublished.

Weyhrauch, Richard W. [NOTE17] **What is real?**, Informal Note 17, unpublished.

Appendix A An axiomatization of natural numbers

The commands below repeat those given in section 4. They will be used in the examples below. One should keep in mind that this is an axiomatization of the natural numbers (including 0), not an axiomatization of the integers.

```
DECLARE INDVAR n m p q ∈ NATNUM;
DECLARE OPCONST suc pred(NATNUM)=NATNUM;
DECLARE OPCONST +(NATNUM,NATNUM)=NATNUM [R←450,L←455];
DECLARE OPCONST *(NATNUM,NATNUM)=NATNUM [R←550,L←555];
DECLARE PREDCONST <(NATNUM,NATNUM) [INF];
DECLARE PREDPAR P(NATNUM);

AXIOM   Q:
        ONEONE: ∀n m.(suc(n)=suc(m)⊃n=m);
        SUCC1:  ∀n.¬(0=suc(n));
        SUCC2:  ∀n.(¬0=n⊃∃m.(n=suc(m)));
        PLUS:   ∀n.n+0=n
                ∀n m.n+suc(m)=suc(n+m);
        TIMES:  ∀n.n*0=0
                ∀n m.n*suc(m)=(n*m)+m; ;;

AXIOM   INDUCT: P(0) ∧ ∀n.(P(n)⊃P(suc(n))) ⊃ ∀n.P(n); ;

REPRESENT {NATNUM} AS NATNUMREP;
ATTACH suc  ↔ (LAMBDA (X) (ADD1 X));
ATTACH pred ↔ (LAMBDA (X) (COND ((GREATERP X 0)(SUB1 X)) (T 0)));
ATTACH +    ↔ (LAMBDA (X Y) (PLUS X Y));
ATTACH *    ↔ (LAMBDA (X Y) (TIMES X Y));
ATTACH <    ↔ (LAMBDA (X Y) (LESSP X Y));
```

Appendix B An axiomatization of s-expressions

These commands describe to FOL a simple theory of s-expressions. In addition it contains the definitions on the functions @, for appending two lists, and rev, for reversing a list.

```
DECLARE INDVAR x y z ∈ Sexp;
DECLARE INDVAR u v w ∈ List;
DECLARE INDCONST nil ∈ Null;

DECLARE OPCONST car cdr 1;
DECLARE OPCONST cons(Sexp,List)=List;
DECLARE OPCONST rev 1;
DECLARE OPCONST @ 2 [inf];

DECLARE SIMPSET Basic;
DECLARE SIMPSET Funs;

MOREGENERAL Sexp ≥ {List, Atom, Null};
MOREGENERAL List ≥ {Null};

REPRESENT {Sexp} AS SEXPREP;

AXIOM CAR: ∀x y. car(cons(x,y))=x;;
AXIOM CDR: ∀x y. cdr(cons(x,y))=y;;
AXIOM CONS: ∀x y. ¬Null(cons(x,y));;

Basic ← {CAR,CDR,CONS};

AXIOM REV: ∀u. (rev(u) = IF Null(u) THEN u ELSE rev(cdr(u)) @ cons(car(u),nil));;
AXIOM APPEND: ∀u v. (u@v = IF Null(u) THEN v ELSE cons(car(u),cdr(u)@v);;

Funs ← {REV,APPEND};
```

Appendix C An axiomatization of well formed formulas

This is an example of how WFFs are axiomatized in META. It simply collects together the formulas of section 8

```
∀ls expr.(WFF(expr,ls)≡PROPWFF(expr,ls)∨QUANTWFF(expr,ls))

∀ls expr.(PROPWFF(expr,ls)≡APPLPWFF(expr,ls)∨AWFF(expr,ls))

∀ls expr.(APPLPWFF(expr,ls)≡PROPCONN(mainsym(expr)) ∧
        ∀n.(0<n∧n≤arity(mainsym(expr),ls)⊃WFF(arg(n,expr),ls)))

∀ls expr.(QUANTWFF(expr,ls) ≡
          QUANT(mainsym(expr)) ∧ INDVAR(bvar(expr),ls) ∧ WFF(matrix(expr),ls)

∀ls expr.(AWFF(expr,ls)≡SENTSYM(expr,ls)∨APPLAWFF(expr,ls)

∀ls expr.(APPLAWFF(expr,ls)≡PREDSYM(mainsym(expr),ls)∧
        ∀n.(0<n∧n≤arity(mainsym(expr),ls)⊃TERM(arg(n,e),ls)))

∀ls expr.(TERM(expr,ls)≡INDSYM(expr,ls)∨APPLTERM(expr,ls))

∀ls expr.(APPLTERM(expr,ls)≡OPSYM(mainsym(expr),ls)∧
        ∀n.(0<n∧n≤arity(mainsym(expr,ls))⊃TERM(arg(n,expr),ls)))
```

Appendix D Examples of semantic evaluations

We give two sets of examples of semantic evaluation.

In the theory of s-expressions

```
*****DECLARE OPCONST length(Sexp)=Sexp;

*****ATTACH length ↔ LENGTH;

length attached to LENGTH

*****SIMPLIFY length('(A B));

1 length('(A B))='2

*****SIMPLIFY length('(A B))=2;

2 length('(A B))=2≡'2=2

*****SIMPLIFY '2=2;

Can't simplify

*****SIMPLIFY '2='4;

3 ¬('2='4)
```

In the theory of natural numbers

```
%6*****SIMPLIFY 2+3<pred(7);

1 2+3<pred(7)

*****SIMPLIFY 4*suc(2)+pred(3)<pred(pred(8));

2 ¬4*suc(2)+pred(3)<pred(pred(8))

*****SIMPLIFY n*0<3;

no simplifications
```

Appendix E An example of syntatic simplification

After

```
*****simplify Null(nil);
1  Null(nil)
```

The command

```
REWRITE rev cons(x,nil) BY Basic U Funs U {1} U LOGICTREE;
```

produces the result

```
2 rev(cons(x,nil))=cons(x,nil)
```

by a single syntatic simplification. The exact details of what the simplifier did are recorded below. The numbers on the left refer to notes below the example.

```
    Trying to simplify
    |  rev(cons(x,nil))
    |
    succeeded using REV yielding
    |  IF Null(cons(x,nil))
    |     THEN cons(x,nil)
    |     ELSE (rev(cdr(cons(x,nil))) * cons(car(cons(x,nil)),nil))
    |
    Trying to simplify
    |  IF Null(cons(x,nil))
    |     THEN cons(x,nil)
    |     ELSE (rev(cdr(cons(x,nil))) * cons(car(cons(x,nil)),nil))
    |
    failed
    →→→Trying to simplify the condition
       |  Null(cons(x,nil))
       |
       suceeded using CONS yielding
       |  FALSE
       |
    poping up
    Trying to simplify
    |  IF FALSE
    |     THEN cons(x,nil)
    |     ELSE (rev(cdr(cons(x,nil))) * cons(car(cons(x,nil)),nil))
    |
    suceeded using LOGICTREE yielding
    |  (rev(cdr(cons(x,nil))) * cons(car(cons(x,nil)),nil))
    |
    Trying to simplify
    |  (rev(cdr(cons(x,nil))) * cons(car(cons(x,nil)),nil))
    |
1   |while trying to match *, SORT scruples do not permit me to bind u
    |     to  rev(cdr(cons(x,nil)))
    |
    →→→Trying to simplify argument 1
       |  rev(cdr(cons(x,nil)))
```

```
          |
          |while trying to match rev, SORT scruples do not permit me to bind u
          |     to  cdr(cons(x,nil))
          |
          →→→Trying argument 1
             |  cdr(cons(x,nil))
             |
             succeeded using CDR yielding
             |  nil
             |
          poping up
          Trying to simplify
2         |  rev(nil)
          |
          succeeded using REV yielding
          |  IF Null(nil) THEN nil ELSE (rev(cdr(nil)) * cons(car(nil),nil))
          |
      popping up
      Trying to simplify
      |  (IF Null(nil) THEN nil ELSE (rev(cdr(nil)) * cons(car(nil),nil))   *
      |      cons(car(cons(x,nil),nil)))
      |
3     |while trying to match *, SORT scruples do not permit me to bind u
      |     to  IF Null(nil) THEN nil ELSE rev(cdr(nil)*cons(car(nil),nil)
      |
      →→→Trying to simplify argument 1
         |  IF Null(nil) THEN nil ELSE rev(cdr(nil)*cons(car(nil),nil)
         |
         failed
         →→→Trying to simplify condition
            |  Null(nil)
            |
            succeeded using line 1 yielding
            |  TRUE
            |
         poping up
         Trying to simplify
         |  IF TRUE THEN nil ELSE rev(cdr(nil)*cons(car(nil),nil)
         |
         suceeded using LOGICTREE yielding
         |  nil
      popping up
      Trying to simplify
      |  nil * cons(car(cons(x,nil)),nil)
      |
4     |while trying to match *, SORT scruples do not permit me to bind v
      |     to  cons(car(cons(x,nil)),nil)
      |
      →→→Trying to simplify argument 1
         | nil
         |
5        failed but we are at a leaf: argument 1 completely simplified
      popping up
      →→→Trying to simplify argument 2
         |  cons(car(cons(x,nil)),nil)
         |
         failed
         →→→Trying to simplify argument 1
            |  car(cons(x,nil))
            |
            succeeded using CAR yielding
            |  x
            |
         poping up
```

```
     Trying to simplify
     |  cons(x,nil)
     |
     failed
     →→→Trying to simplify argument 1
        |  x
        |
        failed but we are at a leaf: argument 1 completely simplified
     popping up
     →→→Trying to simplify argument 2
        |  nil
        |
        failed but we are at a leaf: argument 2 completely simplified
     popping up
     argument 2 completely simplified
  popping up
  Trying to simplify
  |  nil * cons(x,nil)
  |
  succeeded using APPEND yielding
  |  IF Null(nil) THEN cons(x,nil) ELSE  cons(car(nil),(cdr(nil)*cons(x,nil)))
  |
  Trying to simplify
  |  IF Null(nil) THEN cons(x,nil) ELSE  cons(car(nil),(cdr(nil)*cons(x,nil)))
  |
  failed
  →→→Trying to simplify condition
     |  Null(nil)
     |
     succeeded using line 1 yielding
     |  TRUE
     |
  poping up
  Trying to simplify
  |  IF TRUE THEN cons(x,nil) ELSE  cons(car(nil),(cdr(nil)*cons(x,nil)))
  |
  suceeded using LOGICTREE yielding
  |  cons(x,nil)
  |
  Trying to simplify
  |  cons(x,nil)
  |
6 this node already maximally simplified
  return cons(x,nil)

  11 substitutions were made
  26 calls were made to SIMPLIFY
```

Note 1: This is the FOL sort checking mechanism at work. FOL knows that x is an Sexp (by declaration) and that nil is a List because nil is of sort Null and Lists are moregeneral than Nulls. This means that it knows by declaration that cons(x,nil) is a List. Unfortunately, it knows nothing about the cdr of a List. Thus since the the definition of rev requires that u be instantiated to a Lists, this attempted replacement fails, and we try to simplify its arguments.

Note 2: Notice that the argument to rev actually simplifies to something that FOL can recognize as a List. This means that sort scruples do not prohibit the instantiation of the definition of rev.

Note 3: Unfortunately we have the same problem as in Note 1.

Note 4: This time the first argument to * is ok, but the second is not. Again we try to simplify the arguments.

Note 5: This time when we try to simplify nil nothing happens. In this case as a subterm it is completely simplified and gets marked in such a way that the simplifier never tries to do this again.

Note 6: It is very clever and remembers that it saw this before and since it is at the top level with a maximally simplified formula it stops.

Appendix F An example of evaluation

This is an abbreviated trace of the evaluation of fact(2).

```
eval
| fact(2)
|
interpreting
| fact
fails
|
→→→Syntactic simplification succeeds, yielding
   | IF 2=0 THEN 1 ELSE 2*fact(pred(2))
   |
   eval
   | IF 2=0 THEN 1 ELSE 2*fact(pred(2))
   |
   →→→eval
      | 2=0
      semantic evaluation succeeds, yielding
      | FALSE
   poping up
   semantic evaluation succeeds, yielding
   | 2*fact(pred(2))
   |
   interpreting
   | *
   succeeds evaluating args
     1  eval
        | 2
        semantic evaluation succeeds, yielding
        | 2
        |
     2  eval
        | fact(pred(2))
        |
        interpreting
        | fact
        fails
        |
        Syntactic simplification succeeds, yielding
        | IF pred(2)=0 THEN 1 ELSE pred(2)*fact(pred(pred(2)))
        |
        →→→eval
           | pred(2)=0
           semantic evaluation succeeds, yielding
           | FALSE
        poping up
        semantic simplification succeeds, yielding
        | pred(2)*fact(pred(pred(2)))
        |
        eval
        | pred(2)*fact(pred(pred(2)))
        |
        interpreting
        | *
        succeeeds evaluating args
          1  eval
             | pred(2)
             semantic simplification succeeds, yielding
             | 1
```

```
          |
        2 eval
          | fact(pred(pred(2)))
          |
          interpreting
          | fact
          fails
          |
          Syntactic simplification succeeds, yielding
          | IF pred(pred(2))=0
          |     THEN 1 ELSE pred(pred(2))*fact(pred(pred(pred(2))))
          |
          eval
          | IF pred(pred(2))=0
          |     THEN 1 ELSE pred(pred(2))*fact(pred(pred(pred(2))))
          |
          eval
          | pred(pred(2))=0
          semantic evaluating succeeds, yielding
          | TRUE
          |
          semantic evaluation succeeds, yielding
          | 1
          |
        Evaluating 1 gives 1
       Evaluating IF pred(pred(2))=0 THEN 1 ELSE pred(pred(2))*fact(pred(pred(pred(2)))) gives 1
      Evaluating fact(pred(pred(2))) gives 1
     Evaluating pred(2)*fact(pred(pred(2))) gives 1
    Evaluating IF pred(2)=0 THEN 1 ELSE pred(2)*fact(pred(pred(2))) gives 1
   Evaluating fact(pred(2)) gives 1
  Evaluating 2*fact(pred(2)) gives 2
 Evaluating IF 2=0 THEN 1 ELSE 2*fact(pred(2)) gives 2
Evaluating fact(2) gives 2

1 fact(2)=2
```