Reset reproduction of a Ph.D. thesis proposal submitted June 8, 1978 to the Massachusetts Institute of Technology Electrical Engineering and Computer Science Department. Reset in LaTeX by Jon Doyle, December 1995. Current address: MIT Laboratory for Computer Science. Available via http://www.medg.lcs.mit.edu/doyle

Reflexive Interpreters

Jon Doyle DOYLE@MIT.EDU
Massachusetts Institute of Technology, Artificial Intelligence Laboratory
Cambridge, Massachusetts 02139, U.S.A.

Abstract

The goal of achieving powerful problem solving capabilities leads to the "advice taker" form of program and the associated problem of control. This proposal outlines an approach to this problem based on the construction of problem solvers with advanced self-knowledge and introspection capabilities.

1 The Problem of Control

Self-reverence, self-knowledge, self-control,
These three alone lead life to sovereign power.
Alfred, Lord Tennyson, *OEnone*
Know prudent cautious self-control is wisdom's root.
Robert Burns, *A Bard's Epitath*
The woman that deliberates is lost.
Joseph Addison, *Cato*

A major goal of Artificial Intelligence is to construct an "advice taker",[1] a program which can be told new knowledge and advised about how that knowledge may be useful. Many of the approaches towards this goal have proposed constructing additive formalisms for transmitting knowledge to the problem solver.[2] In spite of considerable work along these lines, formalisms for advising problem solvers about the uses and properties of knowledge are relatively undeveloped. As a consequence, the nondeterminism resulting from the uncontrolled application of many independent pieces of knowledge leads to great inefficiencies. Just the potential for nondeterminism requires organizations of problem solvers which are slower on deterministic problems than the corresponding deterministically-tailored systems.[3]

[1] The term "advice taker" is from [McCarthy 1968].

[2] The most popular additive formalism is that of the predicate calculus, interpreted by a theorem proving data base. (See [Green 1969], [Darlington 1969], [Fikes and Nilsson 1971], [Moore 1975], and [McDermott 1977a].) The additivity of such a representation stems from the celebrated Tarski [1944] definition of truth for these languages, in which the addition of new truths cannot affect the validity of previous truths. Systems violating this truth-preserving or monotonic property are called non-monotonic systems, and have been studied practically and theoretically by Rescher [1964], Kripke [1975], Doyle [1978], and [McDermott and Doyle 1978]. Several other formalisms including the Carnegie (e.g. [Rychener 1976]) and California (e.g. [Davis 1976]) flavors of production systems and the logical programming languages [Kowalski 1973] are touted as additive, but in practice these rarely are used in an purely additive fashion.

[3] Completely deterministic interpreters can be organized so that the execution of each action also sets up the next action to be performed. Interpreters made non-deterministic through additivity must expend much more effort in this regard. Since the set of successor actions is expandable, either considerable work must be done to fetch the relevant successor actions, or considerable work must be done to integrate new actions into the successor action table.

To solve the advice taker control problem, we must develop methods for incorporating diverse fragments of knowledge into efficient procedural forms.[4] This necessitates developing vocabularies[5] for describing problem solver states, actions and histories, for these are the ingredients of procedural knowledge.[6] In particular, the mechanization of this "knowledge-compilation" scheme requires an ability of the problem solver to introspect on its own knowledge, structure and behavior. We must study the problem solver itself as a domain for problem solving.[7]

2 Reflexive Interpreters

The approach investigated here is that of constructing self-modelling, self-conscious problem solving interpreters. The simplest way to describe a problem solver to itself is in its own terms.[8] This means writing the command interpreter of the problem solver in the language of descriptions and imperatives that it interprets. Such interpreters are called meta-circular interpreters.[9] The more expressive the language of the problem solver, the deeper is the understanding of itself that can be realized. In fact, several self-models can be constructed, each displaying differing levels of detail in specifying the structure of the interpreter.[10]

The self-model provided by a meta-circular specification of the interpreter gives the interpreter a static self-knowledge.[11] For the interpreter to understand its dynamical behavior requires a form of self-consciousness; the ability to introspect on current goals and activities. This ability typically requires some usage of the explict control techniques of recording and reasoning about actions and goals in a blackboard data base. One important

[4]The necessity of integrating (compiling) general purpose knowledge into special purpose procedures has been argued by Minsky [1965], Winograd [1972], Sussman [1975], and Hewitt [1975], who terms this "the procedural embedding of knowledge".

[5]McDermott [1977a,c], Rychener [1976], and Doyle [tbp] discuss the development of problem solving vocabularies with specific examples.

[6]What makes procedures different from statements of fact? Usually it is some reference to a "state" of an interpreter. Thus normal programming languages use lexical and syntactical devices to indicate modification of the program counter, connections between names, etc. AI programming languages augment these devices by ones relying on changes in contents of data bases, connections between data bases, references to the current goals, and other descriptions of the state and state changes of the interpreter.

[7]The study of the problem solver as a domain for problem solving has other applications beyond those of control. A well-understood model of the problem solver can also be used as the base of models of external agents, and the actions of external agents can be understood in terms of the analogous case of the problem solver "putting itself in their shoes".

[8]McDermott [1977a] terms this way of describing device plans in the vocabulary of the problem solver as a "machinomorphic" view of things.

[9]Meta-circular interpreters have been written for SCHEME (in great profusion in [Steele and Sussman 1978b]), first-order logic [Brown 1977], and LISP 1.5 [McCarthy, et al 1965].

[10]For example, one can write interpreters for a recursive language like SCHEME using the full recursive power of the language (such as those described in [Steele and Sussman 1978b]), or which only use a restricted form of the language (such as that presented in [Sussman and Steele 1975]). Given a logical language for interpretation, the detail of the self-model can include descriptions like plan histories, purposes and dependencies, and transformation rules between syntactic constructions. The latter might range from simplification rules like "`(CAR (CONS A B)) = A`" to recursion elimination rules (such as those described by Burstall and Darlington [1977]).

[11]Deep understanding of the operation of the interpreter requires a logical description and analysis of the design of the interpreter, such as that produced by a plan verifier [Shrobe 1977].

consequence of this dynamic self-consciousness is the ability for concurrent goals to influence each others' solutions.[12] Another important consequence is the ability for the interpreter to redescribe its activities to itself in higher level terms. This means that global patterns of action (such as looping, making choices, committing errors, or acting inefficiently) can be recognized to guide the primary activities of the problem solver. This ability, in conjunction with the ability to modify the self-model and so the behavior, is crucial in learning skills from experience.[13]

I propose to study the construction of such "reflexive" interpreters. My thesis is that constructing a sophisticated self-conscious meta-circular problem solving interpreter is possible at this time.

The motivation for this study is that powerful advice about how to act is expressed in terms of the domain knowledge, past behavior, and current plans. The ability to express such advice is intimately connected to the ability to describe current behavior and to then act on that description. Powerful advice (advice which takes into account as many of the existing constraints as possible) is used by being very self-conscious about one's actions as they are selected and performed. The power of the advice depends crucially on the availability of rich and explict self-description capabilities. This point has been argued by McDermott and Hayes (see [Hayes 1977]) as the necessity for (in McDermott's terms) "shallow reasoning" and the impossibility of "deep reasoning". To construct an advice taker, one can use either a command language with little imperative content interpreted by an "intelligent interpreter", or one can use a rich language with many imperatives interpreted by a "fixed capacity interpreter". Although the former option is a possibility, it begs the question of constructing the intelligent interpreter in the first place. This leads to the approach of using an interpreter of limited operational complexity, in which the knowledge that guides intelligent actions can be expressed. The key word here is "guides", since the experience of the early theorem proving efforts shows that just informing a limited interpreter of the facts will not work. To be practical there must be ways of outlining procedural information.

This approach to the problem of control leads to a certain view of the structure of problem solving knowledge. This view refers to the McCarthy-Hayes [1969] distinction between epistemological and heuristic problems. Epistemological problems are those of determining the ontology of concepts in a domain, and then determining the knowledge about those concepts that describes the domain. The related heuristic problem is that of how to use this knowledge effectively. *Epistemological problems, once solved, lead to heuristic problems whose solutions in turn lead to further epistemological problems.* Obtaining a means of describing some body of knowledge leads to the problem of reasoning (effectively) using these descriptions. This heuristic problem requires the solution of a further set of epistemological problems arising in the task of reasoning about these descriptions. New

[12] McDermott [1977a] argues that many actions do not fit into the state-change model of states and actions. Good examples of this are policies or constraints which serve to influence the execution of other actions. These are actions which affect future possibilities rather than the current state. However, it might be valuable to view these actions as affecting the "mental state" of the problem solver, and dealing with histories which incorporate both world and mental actions.

[13] Sussman's [1975] HACKER debugged the programs it wrote by analyzing patterns of actions to determine the patterns of programs leading to these actions. This process of "anti-interpretation" was then used to pinpoint the erroneous parts of the programs being interpreted.

heuristic problems arise as one discovers new interactions between descriptions, and as new frequencies and importances develop in the uses of these descriptions. Mathematical logic, for example, presents a partial solution to the epistemological problem of statements and their calculus. It provides a recursive enumeration of the provable statements. This leads to the heuristic problem of using this enumeration to describe the world. This leads to epistemological problems of distinguishing objects like goals, axioms and proofs. This leads to the heuristic problem of reasoning about goals and proofs. This leads to the epistemological problems of states of the reasoning process and properties of proofs.

3 Architecture

The basic loop of a serial interpreter repeats these steps:

1. Find the applicable actions,
2. Choose one, and
3. Execute it.

Each of these steps has a wide range of realizations. Step 1 can range from that of looking at the action pointed to by the program counter, to that of fetching methods with satisfied patterns of applicability, to that of recursively invoking the problem solver to determine the current possibilities. Step 2 can range from random selection, to conflict resolution filtering strategies, to preference lists, to choice protocols which invoke the full recursive power of the problem solver. Step 3 can range from only handling primitive operators, to handling actions which must be recursively interpreted, to treating problem solving operations which require problem reduction rather than immediate execution. Each of these spectrums can be viewed as passing from locally efficient immediate operations to operations which though locally inefficient allow for globally efficient behavior. There is no problem in constructing an interpreter which captures all of these options, as long as interface conventions are defined and respected.[14]

There are several important refinements of this structure; descriptions, plans and compilation. There must be a clear structure to the descriptions maintained in the data base. The primary requirement of this structure is the ability to treat descriptions and parts of descriptions as objects to be manipulated and analyzed.[15] This requirement follows from the need to be able to state properties of descriptions (such as completeness) and relationships between descriptions (such as sharing, copying or equivalence). The type of

[14]The AMORD interpreter [de Kleer, et al 1978] displays this multi-level operation, as procedures can be mediated by the data base, interpreted by the MacLISP interpreter, or executed as compiled machine procedures. These levels need not be in a linear hierarchy, for there may be several interpreters all written in a base language, and a higher level interpreter written in terms of several of these. An earlier version of AMORD written in SCHEME displayed this organization. Such an organization is also expected of systems composed of many independent sub-interpreters, such as Tinbergen ethologies [Tinbergen 1951] and the society model of Minsky and Papert [Minsky 1977].

[15]The basic types of descriptions in LISP are those of CONS cells, which have their CARs and their CDRs, and atomic symbols with their print names, value cells, and property lists. More general types of descriptive objects have been developed by Fahlman [1977], Smith [1978], and Martin [1978]. These types of descriptions are discussed [Doyle 1977].

description most important to the interpreter is that of plan structures. Since most problem solving situations involve incomplete information (or at least a temporary lack of complete *explicit* information), much of the preparatory work for problem solving actions involves the piecemeal construction of partially specified programs - that is, plans. The incremental construction and execution of plans allows parts of plans to influence each other's execution. The structure of plans describing aspects of interpreter states (such as goals, results, actions like choosing or finding, and histories) must be part of the interpreter architecture. The self-conscious architecture of the interpreter is intimately connected with this process of plan construction and execution, for one of the major components of self-consciousness is in widely-accessible knowledge of goals and purposes as well as actions. However, to resort to plan construction and execution to accomplish every task is very inefficient, so the use and construction of compiled, special-purpose plans must be possible. Compilation of a plan involves taking the plan and some incidental information (such as past experience, or a restricted class of inputs) and producing a plan incorporating this additional information exhibiting less indeterminacy and inefficiency.[16] When ultimately reduced, the plan becomes a fully specified primitive program. The interface and use of interpretive and compiled programs must be part of the interpreter architecture and vocabulary.

4 Language

Part of the problem of specifying the architecture is that of language. Although the ontology of the architecture determines the important aspects of the language, this ontology must include the language constructs themselves as objects.[17] The interpreter must be able to discuss sentences of the language and their structure to interpret and construct them.

[16] One way for a compiler to increase the efficiency of a program is to introduce as much linearity into the program steps as possible. Such sequencing of steps allows efficient retrieval of successive actions. Unfortunately, experience with new problems may show that some of these introduced orderings between steps cause program failure. One cannot then simply compile the program further with the added constraint, for the new constraint contradicts a previous ordering. Instead, the program must be debugged by determining the conflicting ordering and changing it. This is the approach used by HACKER [Sussman 1975].

[17] While sentences must be objects so that other sentences can mention them, this does not means that there need be objects designated as the "meanings" of the sentences. In this I find myself in agreement with a substantial school of thought which holds that there are only sentences, and all operations are on sentences. (See [Tarski 1944], [Davidson 1967], and [Quine 1970].)

I do not wish, however, to introduce a hierarchy of languages, such that the sentences of each language are objects of the next higher level of language. The use of such a hierarchy seems to be a carry-over from an avoidance of postulating objects external to the interpreter. While there may be reasons for this conservatism, there seems to be no reason to avoid referring to objects internal to the interpreter, such as sentences. Thus, rather than adopt an explicit hierarchy of language types such as that used by Weyhrauch in his proof checker and Tarski in his definition of truth, I favor using a single language with an implicit hierarchy of language levels determined by language use.

The major problem encountered in using a system of language types is that knowledge expressed in one language may not be easily accessible to processes using other language levels. This problem can be avoided by mapping one language into a sublanguage of another, but this translation may take time that could be better spent elsewhere. There are also more theoretical reasons for prefering hierarchies induced by language use rather than forced by language types. Some of these considerations are discussed by Backus [1973], Scott [1973] and Kripke [1975].

This same criticism applies to the "meta-levels" control system advocated by Davis [1976].

It is true that somewhat different vocabularies will be useful for describing the interpreter and for describing external domains. Indeed, Minsky [1965] suggests that the mind-body distinction drawn from introspection originates in this difference in vocabularies. Although this might be seen to indicate that different languages are appropriate for describing the structure and behavior of the interpreter and for describing external domains, several considerations argue for embedding all of these vocabularies in a single language.[18] The operation of a sophisticated interpreter is potentially as complicated as the behavior of agents and objects external to the interpreter. Thus a language adequate for describing external domains to sufficient accuracy should also suffice as a language for describing the behavior of the interpreter, and *vice versa*. This argues that there be just one language by virtue of sufficiency of a single language for the task. Another argument is that much of the reasoning useful in problem solving, debugging and learning must make reference to both descriptions of the interpreter's behavior as well as descriptions of the external world. In particular, rules expressing relevancy conditions frequently mention both the state of the problem solver and conditions in the domain model of the problem being examined. That the interpreter must be able to consider itself as a domain in this way further demonstrates that either a mulplicity of languages must be used with interfaces and translations (which themselves must be described somehow) or that one language be used which is sufficient for discussing all of these areas of knowledge.

5 Reflexive Reasoning Examples

The use of the explict control techniques encouraged by a reflexive interpreter allows many sophisticated behaviors. Occasional summarizations of actions by compiled routines can be used to trigger interrupts detecting dangerous or fruitless behaviors. For example, although when stuck on a problem it is frequently possible to reformulate the problem so as to allow progress, it is normally fruitless to try to reformulate the task of reformulation if no such rephrasing information is apparent. A rule to this effect is used in McDermott's [1977a] NASL. Other abilities can also be indicated by making observations about the state of the problem solver. For example, when provided with the capability to recognize situations in which choices between alternatives are necessary, a rule might be written to say "if more than five options are available, try to reorganize the data base to make more discriminating relevancy checks." Note that such advice does not concern which option to choose (which is something else to reason about), but rather concerns what to do based on properties of the choice situation. Alternatively, conclusions like "one of the assumptions leading to having to make this choice is wrong" or "you are in a state of mind called confusion" might have been made. NASL uses built-in advice of this form to enter the "choice" and "rephrasing" protocols. Recognizing aspects of the problem solver state is also involved in, for example, using goals to suggest methods for solution, or in avoiding conditions in the reasoning process like "be very careful if you discover a contradiction" or "find a legal way of getting rich quick".

[18] This embedding of multiple vocabularies into a single language might be taken to account for the deep confusions encountered by attempts to discuss concepts which are shared by both "mental" and "physical" domains. On the other hand, this may lead to anthropomorphic views which allow fruitful reasoning and an economical representation of knowledge. (Cf. [Minsky 1965, 1977].)

Reasoning about available knowledge has many uses also. These include ways of using facts, such as "use this implication as consequent method for proving its consequent", and ways of making decisions, such as "Assume that P is true if you cannot prove otherwise", "Could object X be identified with object Y as far as I know?", or "Could I compute (or prove) P if I had to? How much effort would be involved?". This last question might involve using proof methods not available to the computation (such as using induction), or might simply use efficiency information attached to methods as commentary to the procedures involved.

One of the most important activities requiring self-reflection is that of compilation. This depends on the ability to reason about the structure of the interpreter and language, the existing plans, and any extra constraining information available (such as case restrictions or usage patterns). The following section discusses plan compilation in more detail.

6 Plan Compilation

Self-conscious interpreters employing explicit control techniques as their only control mechanism are too slow. It is impossibly inefficient to continually check for all relevant information before acting, and to have to record all intermediate results in a form which allows potential interactions to be realized. The only way efficient operation can be realized is to be able to avoid these checks for relevancy and interaction, either by ruling out cases as currently impossible, or (more frequently) by ignoring them with the justification that such ignorance will allow many problems to be solved in practical times.[19] This is the task of plan compilation.

Compilation introduces locality.[20] In a globally interacting system (such as a predicate calculus based system), there is no locality; any piece of information can be used by any other; all information is globally visible. However, a particular set of knowledge may be analyzed to show that certain interactions are impossible, or that certain information is used only in a restricted way by a definite set of users, or that certain information is cheaply recomputed. This analysis can then be used to reorganize the set of knowledge so that procedural primitives are used in the cases where restrictions are possible. A good example is that of procedure integration. If one procedure asserts a call into the data base on another procedure, and if the data base can be shown to only have that single procedure as a possibility for answering the call, then the body of that other procedure can be inserted into the calling procedure to replace the call. This removes the frequently expensive data base operations of asserting a new call and fetching all potentially relevant candidates.

Plans are incompletely specified but heavily annotated programs. Thus they are composed of a nonlinear network of tasks or goals, along with the logical dependencies relating

[19] Procedures compiled in ignorance of later-discovered special cases can produce errors. The system should be designed so as not to be flustered by errors and contradictions, but rather able to analyze these malfunctions to correct their sources. (See [McDermott 1974], [Fahlman 1974], [Sussman 1975], [Goldstein 1974], [Brown 1976], [Stallman and Sussman 1977], and [Doyle 1978].)

[20] The type of locality I mean here is not that discussed by Sussman [1975, p. 5]. His locality is the locality or modularity of pieces of knowledge added to the store of general knowledge. The locality I refer to is locality of the internal structure of a procedural representation of some non-modular slice of the knowledge in the data base. Such slices representing expert procedures might be viewed as frames, with the network of assumptions involved in their invocation representing a frame-system. [Minsky 1974]

these goals. The fewer the restrictions between the parts of a plan, the more general the plan. Planning systems like Sacerdoti's [1975] NOAH system attempt to produce the most general plan possible to solve a problem. These general plans are likely to be the building blocks of a library of plans providing general techniques for solving problems in many domains. The more important or more frequently used plans should be compiled somewhat differently. Some of the inefficiency involved in executing a non-linear plan can be eliminated by imposing arbitrary orderings on the unordered components. This allows plan step setup work to be done in a predetermined fashion, and may also allow many other decisions (such as resource use) to be made at compile time. This strategy, used by Sussman's [1975] HACKER, produces more efficient but less general plans.

The reflexive nature of the reasoning involved in plan compilation is unavoidable. To attempt to compile some plan into a more refined plan requires that the interpreter and current set of knowledge be treated as objects for analysis. The current knowledge (including the history) provides the incidental constraints, and the interpreter model provides the information necessary for code expansion and symbolic evaluation. The incidental constraints must be checked during code expansion to help restrict the interpretation of the plan constructs. The knowledge of the structure of the interpreter is also important in deciding the level to which compilation will progress; whether the result of the compilation is still some level of plan (if uncertainties still exist), or an interpretable procedure (if debugging is still in progress), or a primitive procedure.

All the traditional compilation techniques, plus some extras, fit into the plan compilation process. Code expansion, as mentioned, is done through reference to a self-model. Procedure reduction occurs through the presence of multiple self-models (alternative definitions) and incidental information about their relative efficiencies. Procedure integration applies as described above, and constant folding and dead code elimination take similar forms. Other advanced techniques are possible as well. Local data structures and variables can be introduced to improve efficiency. Just as a traditional compiler might stack-allocate procedure parameters rather than making them visible to interpreted code, a plan compiler might set up procedures to pass packets of specialized data structures to their continuations, rather than transmitting their results in general form in the global data base. For example, the most standard plans might use global variables to communicate inputs, outputs, and intermediate results. (Cf. [Minsky 1977]) The self-conscious architecture also lends itself to more advanced techniques like automatic subroutinization.

Reasoning about the available information is very important in compilation, for such reasoning can be used to reduce the power of the methods remaining in the compiled procedure. For example, a procedure may contain information about default assumption, choices, or backtracking to be performed during execution. If these choices are normally correct, then substantial efficiency might be gained in routine cases by transforming the procedure so that all assumptions and choices are predetermined or partially tested with primitives, and backtracking is ignored. The new procedure can then be used under the assumption that it is applicable to the case at hand. An error signals that one of the assumptions that was lumped into the assumption of applicability is wrong, and that a more general method (perhaps the uncompiled procedure) must be used instead. Too high a frequency of failure of the procedure indicates that new information gathering (via history analysis or other means) and recompilation is necessary.

A more complex example of reasoning about the knowledge available is shown in the following. Manna and Waldinger [1977a,b] present a system for synthesizing recursive LISP programs from specifications. The system expands reduces goals to subgoals via a collection of tranformation rules. These transformation rules may have conditions of applicability. In such conditional cases, either the application condition must be proven, or a conditional test must be introduced. Similarly, recursion is introduced when a goal is seen to be an instance of the top level goal and the applicability conditions (the specifications) of the program are satisfied by proof or conditional introduction.

Unfortunately, the conditional introduction rule used in this system is to introduce a test whenever the desired condition cannot be proven. This confuses several cases which might lead to this failure of proof; in particular, those of conditions independent of the specifications, and true conditions which cannot be proven due to incompletenesses in the proof techniques. Distinguishing between these cases sometimes pays off in a more efficient program. For example, Manna and Waldinger [1977a] use the following example of a program to compute the maximum element of a list.

```
(DEFUN MAXL (X)
       (COND ((NULL (CDR X)) (CAR X))
     ((> (MAXL (CDR X)) (CAR X)) (MAXL (CDR X)))
     (T (CAR X))))
```

The entry condition to the program is that the list is nonempty. The recursive call of the program on the tail of the list is introduced after a check to see if the tail is non-empty. In this case, the conditional was introduced because the non-empty list specification to be met by the recursive calls was not provable. However, it is provable that the specification is not provable, since lists of length one and two are models of the two distinguishing cases. Here, however, the confusion between provably unprovable and lack of proof methods does not create an inefficient program (although the program is inefficient for other reasons).

Now consider writing an evaluator for a LISP-like language. We will not give the specification or the exact details of such a process, but instead present three different such evaluator fragments and indicate how different reasoning strategies would have produced the differences.

First of all, we have the least complicated version of the major function EVAL. This function takes an expression and an environment, and returns the evaluation of the expression in the environment. If the expression is an atom, its value is looked up in the environment. If the expression is non-atomic, then the function position of the expression is checked to see what type of combination is indicated. If the function position is either atomic or a random form, it is evaluated and applied to the list of evaluated arguments. If the function is a lambda expression, however, it is applied directly to the list of evaluated arguments. One can easily imagine how the simple conjunction and recursion introduction rules used by Manna and Waldinger could produce the something like the following function.

```
(DEFUN EVAL (X ENV)
       (COND ((ATOM X) (LOOKUP X ENV))
     ((ATOM (CAR X))
      (APPLY (EVAL (CAR X) ENV) (EVLIS (CDR X) ENV)))
```

```
     ((EQ (CAAR X) 'LAMBDA)
      (APPLY (CAR X) (EVLIS (CDR X) ENV)))
     (T (APPLY (EVAL (CAR X) ENV) (EVLIS (CDR X) ENV)))))
```

Thus, direct recursions are introduced in the fourth and seventh lines of the program. Note, however, that in the fourth line one additional constraint on the argument to the recursive call is known that is not present for the call on the function in general. That is, the argument is known to be atomic, and so the recursion can be replaced by that particular case of the specifications, namely by a simple lookup. This produces the following program.

```
(DEFUN EVAL (X ENV)
       (COND ((ATOM X) (LOOKUP X ENV))
     ((ATOM (CAR X))
      (APPLY (LOOKUP (CAR X) ENV) (EVLIS (CDR X) ENV)))
     ((EQ (CAAR X) 'LAMBDA)
      (APPLY (CAR X) (EVLIS (CDR X) ENV)))
     (T (APPLY (EVAL (CAR X) ENV) (EVLIS (CDR X) ENV)))))
```

This is not all that can be done, however. In the seventh line, the function position is already known to be non-atomic, so the atom test that is performed in the recursive invocation is wasted. Noticing this allows the function to be made more efficient by rewriting it as two functions as follows.

```
(DEFUN EVAL (X ENV)
       (COND ((ATOM X) (LOOKUP X ENV))
     (T (EVAL1 X ENV))))

(DEFUN EVAL1 (X ENV)
       (COND ((ATOM (CAR X))
      (APPLY (LOOKUP (CAR X) ENV) (EVLIS (CDR X) ENV)))
     ((EQ (CAAR X) 'LAMBDA)
      (APPLY (CAR X) (EVLIS (CDR X) ENV)))
     (T (APPLY (EVAL1 (CAR X) ENV) (EVLIS (CDR X) ENV)))))
```

In this version, the extra atom test is avoided.

7 Research Directions

I have alluded to many very difficult problems in this paper, too many for a thesis. However, there are several projects I hope to pursue in investigating the topics of reflexive interpreters and plan compilers. One task is to write a compiler which takes as input a meta-circular interpreter and a program, and then symbolically evaluates[21] the actions of the interpreter on the program to produce a compiled version of the program. The benefit of this would be that small modifications to the interpreter could easily be accomodated without extensive

[21] For the simpler languages, the process of symbolic evaluation can be made more like that employed in [Doyle 1976] rather than the more complex methods employed by [Rich and Shrobe 1976].

changes to the compiler. The compiler could then be used to compile itself for a particular interpreter, producing a compiler which has only an implicit model of the interpreter.[22] This study can be carried out using a small language such as a subset of SCHEME.[23]

Another task is to write a compiler for a rule-based language, such as AMORD or NASL. This would involve describing three types of function calling disciplines, one for explicit calls mediated by the data base, one for interpreted code, and one for primitive compiled functions. This would also require settling on a representation for plans and the history of the interpreter in symbolically executing these plans. With these, the next step would be to use the meta-circular techniques of the first project to produce a reflexive interpreter and plan compiler.

These projects would be followed by increasing the sophistication of the interpreters involved. This means developing a more expressive and wide ranging vocabulary for talking about control actions, interpreter states, and interpreter histories. This would also include developing a library of plans for the action of the interpreter. The only plans that could be developed to any depth would be those strictly concerned with the operation of the interpreter proper, such as language analysis and compilation strategy plans. For example, program transformation techniques of the form discussed by Burstall and Darlington [1977] might be formalized as compilation strategies and interpreter plans.

8 Overview of Related Work

This research builds on many foundations. Minsky [1965] presented an early description of problems of self-modelling interpreters. He develops the theses that the mind-body distinction derives from the different vocabularies used in discussing mental and physical activities, and that the concept of free will derives from incomplete analyses of our internal self-models. Other questions concerning the structure and use of these self-models (descriptions of behavior and control in particular) are discussed in [Minsky 1962].

The construction of meta-circular interpreters has been studied in great detail by Steele and Sussman [1978b]. Related studies in the foundations of self-descriptive language systems have been carried out by Backus [1973], Brown [1977], Scott [1973], Smullyan [1957], Montague [1963] and Kripke [1975].

Hayes has long advocated careful formalization of the control processes of problem solving systems. Towards this end, he has been working for several years on the GOLUX system, a self-descriptive theorem prover, and on related representational problems. [Hayes 1970, 1971a, 1971b, 1973, 1974, 1977] There seem to be few details on this system, which Hayes has indicated (in his 1977 IJCAI lecture) is still unimplemented.

[22] The technique of symbolically evaluating a meta-circular interpreter for a language as a compiler does not seem to have been investigated. Instead, related efforts have concentrated on using separate fixed languages to describe the language and machine for which the compiler is being developed. Thus compiler-compilers are typically seen as accepting a description of the language being compiled, producing a syntax analyzer and compiler-proper for the language, and then accepting a machine description and producing a code generator for the machine. (See for compiler-compilers [McKeeman, Horning and Wortman 1970], for code generator generators [Fraser 1977] and [Cattell 1978], for language description languages [Cleaveland and Uzgalis 1977], and for machine description languages [Bell and Newell 1971].

[23] Many small interpreters for SCHEME subsets are presented in [Steele and Sussman 1978b]. The reference description of SCHEME is [Sussman and Steele 1978a].

McDermott [1977a, 1977b, 1977c, 1978] has constructed NASL, a self-conscious program for designing electronic circuits. NASL is an interpreter of hierarchically organized plans which it constructs in a predicate-calculus theorem-proving data base. NASL is a single-level system (everything goes through the data base - there is no provision for compilation) which uses only a fragmentary self-model and an undeveloped representation of the history of the interpreter's actions. This leads to several inadequacies in the power and performance of the program. One of McDermott's major developments is a demonstration of the expressibility of control information in the extended predicate calculus. He has developed a vocabulary for discussing tasks, task states (pending, active, finished, blocked, enabled, reduced, unreduced, and others), task actions and methods, relations between tasks (successor, subtask, maintasks, scope, sequencing, iteration, and others), and actions for controlling the interpreter (infer, find, continue, finish, rephrasing, choosing, etc.) I hope to use much of this vocabulary, and much of my thought about reflexive interpreters has used NASL as a model.

Rychener [1976] has also developed a vocabulary of control contructs. He uses these to implement a version of GPS as a production system.

Weyhrauch [personal communication] has developed a proof checking system for a first-order logic of programs which can explain and reason about its own operation. Proofs and related concepts are among its ontology, and it can explain any of the actions it takes in checking a proof. It is not a problem solver, and thus has a very restricted set of actions for description. It is based on a rigid hierarchy of language types, such that statements in one level of the hierarchy refer to objects representing statements at the next lower level of the type structure. It uses no concept of hierarchy of action types or of hierarchical descriptions.

Milner and his colleagues have a proof checker based on an extended lambda-calculus, augmented by a simple vocabulary for constructing tactics for guiding the proof checking. [Gordon, et al 1977]

The work of myself and my collaborators has involved studies of the representation and manipulation of hierarchical proofs and programs suitable for self-conscious interpreters. The major topics investigated have been the description of rules for using explicit control assertions [de Kleer, et al 1977], recording and maintaining program and proof histories [Doyle 1978], and search and control methods employing these techniques [Doyle 1978, Stallman and Sussman 1977].

Sussman [1975] investigated the problem of compilation of procedures for the use of knowledge. The major topics developed in his study were the processes of automatic subroutinization and the debugging or modification of overconstrained compiled programs. Other topics in the compilation of procedures from general knowledge have also been studied by Sandewall [1976], Manna and Waldinger [1977b], Clark and Sickel [1977], and Brown [1978].

There are many subtopics involved in the study of sophisticated reflexive interpreters. The description and use of non-linear and hierarchical plans of actions have been studied in many ways by Sacerdoti [1975], Tate [1976], Barstow [1977], Rich and Shrobe [1976], and McDermott [1977a]. Problems of reasoning about knowledge and actions has been studied by Hintikka [1962], Rescher [1964], Mackie [1965], McCarthy and Hayes [1969], Hayes [1971a,b], Moore [1975, 1977], McCarthy [1977], Doyle [1978], Pratt [1976], and

Harel [1978]. The particular problems of representing hierarchical and other structures of knowledge have been studied by Fahlman [1977], Martin [1978], Smith [1978], and Sussman [1977].

Acknowledgements

Gerald Jay Sussman, Drew McDermott and Marvin Minsky have provided considerable inspiration for this work. Johan de Kleer, Howard Shrobe, Scott Fahlman, Tomas Lozano-Perez, Guy Steele, and Richard Weyhrauch have helped with many fruitful discussions. Don Petersen, Shelly Lieber and Mike Leitch have provided substantial moral support. The Fannie and John Hertz Foundation has provided the graduate fellowship making my work possible.

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643 and in part by NSF Grant MCS77-04828.

References

[**Backus 1973**]
John Backus, "Programming Language Semantics and Closed Applicative Languages," Proc. Symp. on Principles of Programming Languages, 1973, pp. 71-86.

[**Barstow 1977**]
David R. Barstow, "Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules," Stanford AI Laboratory, Memo AIM-308, November 1977.

[**Bell and Newell 1971**]
C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, New York: McGraw-Hill, 1971.

[**Brown 1976**]
Allen Brown, "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures," MIT AI Lab, TR-362, December 1976.

[**Brown 1977**]
Frank Brown, "The Theory of Meaning," University of Edinburgh, Department of Artificial Intelligence Research Report 35.

[**Brown 1978**]
Richard Brown, "Automatic Synthesis of Numerical Computer Programs," MIT Ph.D. proposal, May 1978.

[**Burstall and Darlington 1977**]
R. M. Burstall and John Darlington, "A Transformation System for Developing Recursive Programs," *JACM*, V. 24, No. 1, (January 1977), pp. 44-67.

[**Cattell 1978**]
R. G. G. Cattell, "Formalization and Automatic Derivation of Code Generators," Carnegie-Mellon University, Computer Science Department, April 1978.

[**Clark and Sickel 1977**]
Keith Clark and Sharon Sickel, "Predicate Logic: A Calculus for Deriving Programs," *IJCAI 5*, pp. 410-411.

[**Cleaveland and Uzgalis 1977**]
J. C. Cleaveland and R. C. Uzgalis, *Grammars for Programming Languages*, New York: Elsevier, 1977.

[**Darlington 1969**]
J. L. Darlington, "Theorem Proving and Information Retrieval," in *Machine Intelligence 4*, Meltzer and Michie, editors, p. 173.

[**Davidson 1967**]
Donald Davidson, "Truth and Meaning," *Synthese*, XVII, (1967), pp. 304-323.

[**Davis 1976**]
Randall Davis, "Applications of Meta Level Knowledge to the Construction, Maintainance and Use of Large Knowledge Bases," Stanford AI Lab Memo AIM-283, July 1976.

[**de Kleer, Doyle, Steele and Sussman 1977**]
Johan de Kleer, Jon Doyle, Guy L. Steele Jr., and Gerald Jay Sussman, "Explicit Control of Reasoning," MIT AI Lab, Memo 427, June 1977.

[**de Kleer, Doyle, Rich, Steele and Sussman 1978**]
Johan de Kleer, Jon Doyle, Charles Rich, Guy L. Steele Jr., and Gerald Jay Sussman, "AMORD: A Deductive Procedure System," MIT AI Lab, Memo 435, January 1978.

[**Doyle 1976**]
Jon Doyle, "Analysis by Propagation of Constraints in Elementary Geometry Problem Solving," MIT AI Lab, WP 108, June 1976.

[**Doyle 1977**]
Jon Doyle, "Hierarchy in Knowledge Representations," MIT AI Laboratory, Working Paper 159, November 1977.

[**Doyle 1978**]
Jon Doyle, "Truth Maintenance Systems for Problem Solving," MIT AI Lab TR-419, January 1978.

[**Doyle tbp**]
Jon Doyle, "Refining State Descriptions," forthcoming.

[**Fahlman 1974**]
Scott E. Fahlman, "A Planning System for Robot Construction Tasks," *Artificial Intelligence*, *5* (1974), pp. 1-49.

[**Fahlman 1977**]
Scott E. Fahlman, "A System for Representing and Using Real World Knowledge," MIT Ph.D. Thesis, September 1977.

[**Fikes and Nilsson 1971**]
R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence 2*, (1971), pp. 189-208.

[**Fraser 1977**]
Christopher W. Fraser, "A Knowledge-Based Code Generator Generator," Proc. Symp. on Artificial Intelligence and Programming Languages, SIGART Newsletter, No. 64, August 1977, pp. 126-129.

[**Goldstein 1974**]
Ira P. Goldstein, "Understanding Simple Picture Programs," MIT AI Lab, TR-294, September 1974.

[**Gordon, Milner, Morris, Newey, and Wadsworth 1977**]
M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, "A Metalanguage for Interactive Proof in LCF," Proc. Fifth Symposium on Principles of Programming Languages, pp. 199-130.

[**Green 1969**]
C. Cordell Green, "Theorem-Proving by Resolution as a Basis for Question-Answering Systems," in Meltzer and Michie, *Machine Intelligence 4*, pp. 183-205.

[**Harel 1978**]
David Harel, "Logics of Programs: Axiomatics and Descriptive Power," MIT LCS TR-200, May 1978.

[**Hayes 1970**]
P. J. Hayes, "Robotologic," in Meltzer and Michie, editors, *Machine Intelligence 5*, pp. 533-554.

[**Hayes 1971a**]
P. J. Hayes, "A Logic of Actions," in Meltzer and Michie, editors, *Machine Intelligence 6*, pp. 495-520.

[**Hayes 1971b**]
P. J. Hayes, "The Frame Problem and Related Problems in Artificial Intelligence," Stanford AI Lab, AIM 153, 1971. Also in Elithorn and Jones, editors, *Artificial and Human Thinking*, pp. 45-59.

[**Hayes 1973**]
P. J. Hayes, "Computation and Deduction," Proc. MFCS Symposium, Czech. Acad. of Sciences 1973.

[**Hayes 1974**]
P. J. Hayes, "Some Problems and Non-Problems in Representation Theory," Proceedings of the AISB Summer Conference, 1974, pp. 63-79.

[**Hayes 1977**]
P. J. Hayes, "In Defence of Logic," *IJCAI 5*, pp. 559-565.

[**Hewitt 1975**]
Carl Hewitt, "How to Use What You Know," *IJCAI4*, September 1975, pp. 189-198.

[**Hintikka 1962**]
Jaakko Hintikka, *Knowledge and Belief*, Ithica: Cornell University Press, 1962.

[**Kowalski 1973**]
Robert Kowalski, "Predicate Logic as a Programming Language," University of Edinburgh, Department of Computational Logic, DCL Memo 70, 1973.

[**Kripke 1975**]
Saul Kripke, "Outline of a Theory of Truth," *Journal of Philosophy*, Vol. 72, No. 19, (November 6, 1978), pp. 690-716.

[**Mackie 1965**]
J. L. Mackie, "Causes and Conditions," *American Philosophical Quarterly*, 2.4 (October 1965), pp. 245-255 and 261-264.

[**Manna and Waldinger 1977a**]
Zohar Manna and Richard Waldinger, "The Automatic Synthesis of Recursive Programs," Proc. Symposium on Artificial Intelligence and Programming Languages, SIGART Newsletter, No. 64, August 1977, pp. 29-36.

[**Manna and Waldinger 1977b**]
Zohar Manna and Richard Waldinger, "The Automatic Synthesis of Systems of Recursive Programs," *IJCAI 5*, pp. 405-411.

[**Martin 1978**]
William A. Martin, "Descriptions and the Specialization of Concepts," MIT LCS TM-101, March 1978.

[**McCarthy, et al 1965**]
J. McCarthy, et al, *LISP 1.5 Programmer's Manual*, Cambridge: MIT Press, 1965.

[**McCarthy 1968**]
John McCarthy, "Programs With Common Sense," in Minsky, *Semantic Information Processing*, pp. 403-418.

[**McCarthy 1977**]
John McCarthy, "Epistemological Problems of Artificial Intelligence," *IJCAI 5*, pp. 1038-1044.

[**McCarthy and Hayes 1969**]
J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," in Meltzer and Michie, *Machine Intelligence 4*, pp. 463-502.

[**McDermott 1974**]
Drew McDermott, "Assimilation of New Information by a Natural Language-Understanding System," MIT AI Lab, AI-TR-291, February 1974.

[**McDermott 1977a**]
Drew McDermott, "Flexibility and Efficiency in a Computer Program for Designing Circuits," MIT AI Lab TR-402, June 1977.

[**McDermott 1977b**]
Drew McDermott, "A Deductive Model of Control of a Problem Solver," Proc. Workshop on Pattern-Directed Inference Systems, SIGART Newsletter, No. 63, (June 1977), pp. 2-7.

[**McDermott 1977c**]
Drew McDermott, "Vocabularies for Problem Solver State Descriptions," *IJCAI 5*, August 1977.

[**McDermott 1978**]
Drew McDermott, "Circuit Design as Problem Solving," Proc. IFIP Working Conference on Artificial Intelligence and Pattern Recognition in Computer Aided Design, Grenoble, France, March 1978.

[**McDermott and Doyle 1978**]
Drew McDermott and Jon Doyle, "Semantical Considerations on Non-Monotonic Logics," forthcoming 1978.

[**McKeeman, Horning and Wortman 1970**]
W. M. McKeeman, J. J. Horning and D. B. Wortman, *A Compiler Generator*, Englewood Cliffs: Prentice-Hall, 1970.

[**Minsky 1962**]
Marvin L. Minsky, "Problems of Formulation for Artificial Intelligence," *Proc. Symp. on Mathematical Problems in Biology*, American Mathematical Society, Providence, RI, 1962, pp. 35-46.

[**Minsky 1965**]
Marvin L. Minsky, "Matter, Mind, and Models," Proc. IFIP 65, pp. 45-49.

[**Minsky 1974**]
Marvin Minsky, "A Framework for Representing Knowledge," MIT AI Lab, AI Memo 306, June 1974.

[**Minsky 1977**]
Marvin Minsky, "Plain Talk about Neurodevelopmental Epistemology," *IJCAI 5*, pp. 1083-1092.

[**Montague 1963**]
Richard Montague, "Syntactical Treatments of Modality, with Corollaries on Reflection Principles and Finite Axiomatizability," *Acta Philosophica Fennica*, V. 16 (1963), pp. 153-167.

[**Moore 1975**]
Robert Carter Moore, "Reasoning From Incomplete Knowledge in a Procedural Deduction System," MIT AI Lab, AI-TR-347, December 1975.

[**Moore 1977**]
Robert C. Moore, "Reasoning About Knowledge and Action," *IJCAI 5*, August 1977.

[**Pratt 1976**]
Vaughan R. Pratt, "Semantical Considerations on Floyd-Hoare Logic," MIT Laboratory for Computer Science, TR-168, September 1976.

[**Quine 1970**]
W. V. Quine, *Philosophy of Logic*, Englewood Cliffs: Prentice Hall, 1970.

[**Rescher 1964**]
N. Rescher, *Hypothetical Reasoning*, Amsterdam: North Holland 1964.

[**Rich and Shrobe 1976**]
Charles Rich and Howard E. Shrobe, "Initial Report on a LISP Programmer's Apprentice," MIT AI Lab, TR-354, December 1976.

[**Rychner 1976**]
Michael D. Rychner, "Production Systems as a Programming Language for Artificial Intelligence Applications," 3 Volumes, CMU CS Department, December 1976.

[**Sacerdoti 1975**]
Earl D. Sacerdoti, "A Structure for Plans and Behavior," SRI AI Center, TN 109, August 1975.

[**Sandewall 1976**]
Erik Sandewall, "Conversion of Predicate-Calculus Axioms to Corresponding Deterministic Programs," *IEEE Transactions on Computers*, Vol. C-25, No. 4, (April 1976), pp. 342-346.

[**Scott 1973**]
D. Scott, "Models for Various Type-Free Calculi," in *Logic, Methodology and Philosophy of Science IV*, P. Suppes, L. Henkin, A. Joja, Gr. C. Moisil editors, Amsterdam: North-Holland 1973.

[**Shrobe 1977**]
Howard Shrobe, "Plan Verification in a Programmer's Apprentice," MIT AI Laboratory, Working Paper 158, January 1977.

[**Smith 1978**]
Brian C. Smith, "Levels, Layers, and Planes: The Framework of a Theory of Knowledge Representation Semantics," MIT Masters Thesis, January 1978.

[**Smullyan 1957**]
Raymond M. Smullyan, "Languages in which Self-Reference is Possible," *J. Symb. Logic*, V. 22, (1957), pp. 55-67.

[**Stallman and Sussman 1977**]
Richard M. Stallman and Gerald Jay Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis," Artificial Intelligence, Vol. 9, No. 2, (October 1977), pp. 135-196.

[**Steele and Sussman 1978a**]
Guy Lewis Steele Jr. and Gerald Jay Sussman, "The Revised Report on SCHEME, a Dialect of LISP," MIT AI Laboratory, Memo 452, January 1978.

[**Steele and Sussman 1978b**]
Guy Lewis Steele Jr. and Gerald Jay Sussman, "The Art of the Interpreter, or The Modularity Complex," MIT AI Laboratory, Memo 453, May 1978.

[**Sussman 1975**]
Gerald Jay Sussman, *A Computer Model of Skill Acquisition*, American Elsevier Publishing Company, New York, 1975.

[**Sussman 1977**]
Gerald Jay Sussman, "Slices: At the Boundary Between Analysis and Synthesis," MIT AI Lab, Memo 433, July 1977.

[**Sussman and Steele 1975**]
Gerald Jay Sussman and Guy Lewis Steele Jr., "SCHEME: An Interpreter for Extended Lambda Calculus," MIT AI Lab Memo 349, December 1975.

[**Tarski 1944**]
Alfred Tarski, "The Semantic Conception of Truth and the Foundations of Semantics," *Philosophy and Phenomenological Research IV*, 3 (1944), pp. 341-375.

[**Tate 1976**]
Austin Tate, "Project Planning Using a Hierarchic Non-Linear Planner," Edinburgh DAI, Research Report #25, August 1976.

[**Tinbergen 1951**]
N. Tinbergen, *The Study of Instinct*, Oxford: Clarendon Press, 1951.

[**Winograd 1972**]
Terry Winograd, *Understanding Natural Language*, Academic Press, New York, 1972.