Report 77-33
Stanford -- KSL

Scientific DataLink

Generalized Procedure Calling and
Content-Directed Invocation.
Randall Davis,
Aug 1977

Reprint of a paper appearing in:
Proceedings AI⊗PL Conference,
SIGART/SIGPLAN Combined issue, August,
1977.

Generalized Procedure Calling and Content-Directed Invocation

Randall Davis
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

We suggest that the concept of a *strategy* can profitably be viewed as *knowledge about how to select from among a set of plausibly useful knowledge sources*, and explore the framework for knowledge organization which this implies. We describe *meta rules*, a means of encoding strategies that has been implemented in a program called TEIRESIAS, and explore their utility and contribution to problem solving performance.

Meta rules are also considered in the broader context of a tool for programming. We show that they can be considered a medium for expressing the criteria for retrieval of knowledge sources in a program, and hence can be used to define control regimes. The utility of this as a programming mechanism is considered.

Finally, we describe the technique of *content-directed invocation* used by meta rules, and consider its use as a way of implementing strategies. It is also considered in historical perspective as a knowledge source invocation technique, and its advantage over some existing mechanisms like goal-directed invocation is considered.

"Generalized Procedure Calling and Content-Directed Invocation" by Randall Davis in Proceedings of the AI-PL Conference SIGART/SIGPLAN Combined Issue, August 1977.

This work was supported in part by the Bureau of Health Sciences Research and Evaluation of HEW under Grant HS-01544 and by the Advanced Research Projects Agency under ARPA Order 2494. It was carried out on the SUMEX-AIM Computer System, supported by the NIH under Grant RR-00785. The views expressed are solely those of the author.

{1} Introduction

How can we be sure that knowledge embedded in a program is indexed, retrieved, and applied appropriately? Over the years a range of different mechanisms have been proposed and used (e.g., standard procedure invocation, goal-directed invocation, etc.), each typically motivated by the attempt to develop new forms of knowledge encoding (e.g., procedures, PLANNER theorems, etc.). We consider in this paper the strengths and weaknesses of a range of these mechanisms, paying particular attention to their *expressiveness* and *validity*. This analysis brings to light certain shortcomings shared to some degree by all current mechanisms.

A number of ideas are presented as the basis for a mechanism which appears to offer a way of overcoming the problems discovered. We describe how those ideas have been implemented and tested in a rule-based system, and explore their impact on system performance, ease of construction, and flexibility. We consider also their value as a generalization of the existing notions of procedure calling.

Though the terminology may differ, some of the shortcomings we point out and some of the ideas proposed may be recognized by others who have built similar systems, where some of these ideas have been implemented in various informal ways. The purpose of this paper is not, therefore, to advocate a particular solution, but instead to provide a basis for discussion that will bring the issues into focus. We do this by examining a system with a very basic architecture, exploring in some detail its power and limitations.

Our aims are thus (a) to supply some terminology and definitions, specifying what it is that a number of programs have done to overcome the shortcomings in existing invocation mechanisms; (b) to suggest that it would be useful to elevate these techniques to the level of standard constructs, instead of hand-crafted, LISP-level tricks; (c) to try to define an environment in which using these techniques is easier than it typically is; and (d) to suggest a few points of methodology concerning some ideas we have found particularly useful in implementing our solution.

{2} The heart of the issue

The analysis and development that follows is somewhat detailed, yet the conclusions are reasonably straightforward. To insure that the main ideas are not lost in the detail, we present them here, in slightly oversimplified form.

We begin by describing meta-rules and considering them as strategies, which we define as "information about which rules to use next when more than one may be applicable." We examine the utility of this definition, and our implementation of it in meta-rules.

The discussion then broadens to consider how this same meta-rule mechanism can be used in a more fundamental way -- as a medium for specifying retrieval and invocation criteria. We propose extensions of the work we have done, considering a more general scheme which stresses the unity of the notions of retrieval criteria and strategy.

An historical review of previous invocation techniques is then used as a basis for comparison, and brings out some of the central issues. In brief, this analysis views the progression of techniques (from simple procedure calling to such things as goal-directed invocation, etc.), as an attempt to be increasingly *descriptive* in specifying the criteria which determine which procedure to invoke. In this light, standard procedure invocation is in effect simply pointing to a specific procedure, saying "use that one next". Pattern-directed invocation makes it possible to say "give me a procedure whose pattern matches this one." Goal-directed invocation interprets that pattern as the designator of a goal, and the request becomes "give me any procedure that achieves the following goal."

But note that the goal accomplished by a particular procedure is only one limited description of it. What if we wanted a procedure that both accomplished a particular goal and relied on certain specified preconditions? What if its speed, space requirements, side effects, etc., were relevant considerations? By limiting the programmer to talking about goals, or even limiting him to using *any* set of *predetermined* descriptions, we are restricting him unnecessarily.

This leads to the claim that it is useful if the programmer has available a language and

framework in which he can easily define his own invocation criteria. Such a language permits him to write code that says, in effect, "give me a procedure which fits the following description...", where the description includes any criteria he cares to use. He can thus more easily create invocation that is goal-directed, side-effect-directed, speed-directed -- in short directed by any one of or a combination of factors.

This is a generalization of established invocation techniques, and makes available an unusual degree of control over system performance. In particular, it provides an environment in which the criteria for invocation are not *predefined* and *emebedded in the language interpreter*, but are instead arbitrary descriptions available for definition by the programmer himself.

For the purposes of developing this idea, we view program execution as a process of "deciding which procedure to invoke next". Having taken this view, it then makes sense to separate out and specify clearly the criteria used in making that decision. Further issues that arise include the observations that

- *it should be possible to specify generalized invocation criteria, via a richly expressive, extensible language*

- *those criteria should be embodied as objects at the level of other program constructs* (rather than leaving them "hidden away" in the interpreter)

- *a hierarchical control framework aids in organizing the information used in specifying a control regime*, and

- *there are advantages in implementing the retrieval criteria by a technique we call content-directed invocation.*

We note that all this makes possible a programming environment in which invocation criteria can be defined by the programmer when he writes code (so he is not limited to those predefined by the language architect), and in which those criteria can be generalized to allow an arbitrary range of descriptors. The utility of these capabilities is then described.

{3} Background

The work reported here was done as part of the development of TEIRESIAS [3,5-7], a program intended to assist in building task-oriented, knowledge-based consultation systems. The knowledge base is assumed to contain inference rules of the form shown in Figure 1; both the internal (i.e., LISP) and English forms are shown. (For purposes of illustration we deal with a body of knowledge about selecting stock market investments.[1]).

Each rule specifies an action (in this case a conclusion) which is valid if the Boolean combination of preconditions given in the premise has been met. Both preconditions and actions are stated in terms of functions on associative triples; e.g., the first precondition here is "timescale (*attribute*) of investment (*object*) is long-term (*value*)". The rules are judgmental, i.e., they can specify inexact inferences, with the strength of the inference given on a scale from 0 to 1. Conclusions made by two rules are combined using the model of confirmation theory described in [22]; the details of that model will not concern us here.

```
RULE027
If  [1] the time scale of the investment is long-term, and
    [2] the desired return on the investment is greater than 10%, and
    [3] the area of the investment is not known,
 then AT&T is a likely (.4) choice for the investment.

PREMISE     ($AND (SAME OBJCT TIMESCALE LONG-TERM)
                  (GREATER OBJCT RETURNRATE 10)
                  (NOTKNOWN OBJCT INVESTMENT-AREA))
ACTION      (CONCLUDE OBJCT STOCK-NAME AT&T .4)
```

Figure 1 - example of a rule

The rules are invoked in a goal-directed backward chaining mode that produces an exhaustive depth-first search of an and/or goal tree. Suppose, for instance, that the system is attempting to deduce which stock to invest in. It retrieves the (precomputed) list of all rules which mention STOCK-NAME in their action, and attempts to invoke each one in turn, evaluating their premises to see if the indicated conditions have been met. For the rule shown above, this means determining first whether the timescale of the investment is long-term. This is in turn set up as a subgoal and the process recurs.

The resulting search is depth-first, since each precondition is considered in turn; the tree is an and/or goal tree since premises may have disjunctions in them; and the search is exhaustive because the rules are inexact (unless an answer has been deduced with certainty, it appears appropriate to continue to collect all evidence about a given subgoal).

The ability to use an exhaustive search is of course a luxury, and can be computationally infeasible if the number of rules about a particular goal is large enough. In that case some choice would have to be made about which of the plausibly useful rules should be invoked. Note that here goal-directed invocation is insufficient. That is, it is of limited utility to know that a rule accomplishes a particular goal. We need a way of expressing a much richer set of properties about rules, and a way of effecting selection based on those properties. *Meta-rules* were created to address this problem.

{4} Meta-rules

Figure 2 below shows two meta-rules. The first of them says, in effect, that in trying to determine the best investment for a non-profit organization, rules that base their recommendations on tax bracket are not likely to be the right approach to take. The second indicates that when dealing with clients nearing retirement age, more secure stocks should be considered before more speculative ones.

```
METARULE001
If   1) you are attempting to determine the best stock to invest in, and
     2) the client's tax status is non-profit, and
     3) there are rules which mention in their premise the income-tax bracket
        of the client,
then it is very likely (.9) that each of these rules is not going to be useful.

PREMISE
    ($AND(SAME OBJCT CURGOAL STOCK-NAME)
         (SAME OBJCT STATUS NON-PROFIT)
         (THEREARE OLRULES ($AND (MENTIONS FREEVAR PREMISE BRACKET)) SET1))
ACTION
    (CONCLUDE SET1 UTILITY NO .9)


METARULE002
If   1) the age of the client is greater than 60, and
     2) there are rules which mention in their premise blue-chip risk, and
     3) there are rules which mention in theirpremise speculative risk,
then it is very likely (.8) that the former should be used before the latter.

PREMISE
    ($AND(GREATER OBJCT AGE 60)
         (THEREARE OLRULES ($AND (MENTIONS FREEVAR PREMISE BLUE-CHIP)) SET1)
         (THEREARE OLRULES ($AND (MENTIONS FREEVAR PREMISE SPECULATIVE)) SET2))
ACTION
    (CONCLUDE SET1 DOBEFORE SET2 .8)
```

Figure 2 - two meta-rules

It is important to note the character of the information conveyed by meta-rules. First, note that in both cases we have a rule which is making a conclusion about other rules. That is, where object level rules conclude about the stock market domain, meta-rules conclude about object level rules. These conclusions can (in the current implementation) be of two forms. As in the first meta-rule, they can make deductions about the likely utility of certain object level rules, or (as in the second) they can indicate a partial ordering between two subsets of object level rules.

Note also that (as in the first example) meta-rules make conclusions about the *utility* of object level rules, not their validity. That is, METARULE001 does *not* indicate circumstances under which some of the object level rules are invalid (or even "very likely (.9)" invalid). It merely says that they are likely not to be useful; i.e., they will probably fail, perhaps only after requiring extensive computation to evaluate their preconditions. This point has an impact on the issue of organization and distribution of knowledge in the system; further discussion is found in [3].

Adding meta-rules to the system requires only a minor addition to the control structure described above. As before, the system retrieves the list (call it L) of all rules relevant to the current goal. But before attempting to invoke them, it first determines if there are any meta-rules relevant to that goal[2]. If so, these are invoked first. As a result of their action, we may obtain a number of conclusions about the likely utility and desired relative ordering of the rules in L. These conclusions are used to reorder or shorten L, and the revised list of rules is then used. Viewed in tree-search terms, the current implementation of meta-rules can either reorder the branches of the tree or prune them (pruning occurs if it is possible to conclude with certainty that some rule will not be useful).

Since there is no reason to have only two levels of control, the current implementation supports an arbitrary number of levels. Each level serves to direct the use of knowledge at the next lower level. That is, the system retrieves the list (L) of object level rules relevant to the current goal.

Before invoking this, it checks for a list (L') of first order meta-rules which can be used to reorder or prune L. But before invoking this, it checks for second order meta-rules which can be used to reorder or prune L', etc. Recursion stops when there is no rule set of the next higher order, and the process unwinds, each level of meta-rules advising on the use of the next lower level.

Note that the same representational formalism (rules) is used at all levels. This uniformity of knowledge encoding helps cut down on the amount of machinery necessary. For instance, there is a single mechanism that allows rules at one level to conclude about rules at the next lower level. Similarly, there is a single (very simple) code examining technique (described in Section {9.1}), that is used by all levels of rules to examine rules at lower levels.

This hierarchical structure appears to offer a useful framework for organizing and using control information. Consider that, to the rules at any given level, all rules at lower levels appear as data. Each successive level can thus examine and reason about the control information stored at levels below it. This means that we can write (higher level) criteria which decide which (lower level) criteria to choose. Thus where first-order meta-rules decide which object-level rules to use, second order meta-rules select from among the criteria, and can help "decide how to decide".

{5} Meta-rules as strategies

Meta-rules were introduced above in the context of making a decision about which rules to invoke when there are too many to use exhaustively. We take this as a basis for our initial definition of a strategy: we view a strategy as *information about which chunk of knowledge to invoke next when more than one chunk may be applicable.* Given the importance of making that decision intelligently, strategies offer an important site for the embedding of knowledge in a system.

Note that the opportunity to do this does not arise in traditional (i.e., ALGOL-like) programs, since procedure invocation there is "well-specified", in the sense that only one procedure is considered for invocation at any given moment. It does arise in many of the programming paradigms developed in AI, however, since they admit (or even encourage) the possibility of having several alternative chunks of knowledge plausibly useful in a single situation (e.g., production rules, PLANNER-like languages, as well as the choice-point and back-tracking mechanisms present in many other languages). But note that the typical approach is exhaustive, and considers each alternative in turn, until some stopping criterion is met or all have been tried. The ability to specify a more precise form of control would be useful, since the performance of a program can be strongly improved by choosing wisely from among the alternative chunks available.

{5.1} Advantages of this approach

There are several advantages to this approach to encoding strategies. First, the framework it presents for knowledge organization and use appears to offer significant leverage, since much can be gained by adding to a system a store of (meta-level) knowledge about which chunk of object level knowledge to invoke next. Considered once again in tree search terms, we are talking about the difference between "blind" search of the tree, and one guided by heuristics. The advantage of even a few good heuristics in cutting down the combinatorial explosion of tree search is well known.

Consider, too, that part of the definition of intelligence includes appropriate use of information. Even if a store of (object level) information is not large, it is important to be able to use it properly. Meta-rules provide a mechanism for encoding strategies that can offer additional guidance to the system.

The power in using rules to guide rules applies at multiple levels as well: there is leverage in encoding heuristics that guide the use of heuristics. Thus, rather than adding more heuristics to improve performance, we might add more information at the next higher level concerning effective use of existing heuristics.

Second, recall that the basic control structure of the performance program is a depth-first search of the and/or goal tree sprouted by unwinding rules. The presence of meta-rules of the sort shown in Figure 2 means that this tree has an interesting characteristic: at each node, when the

system has to choose a path, there may be information stored advising about the best path to take. There may therefore be available an extensive body of knowledge to guide the search, but that knowledge is not embedded in the code of a clever search algorithm. It is instead organized around the specific objects which form the nodes in the tree; i.e., instead of a smart algorithm, we have a "smart tree".

Third, note that the rules can be judgmental. This makes it possible to write rules which make different conclusions about the best strategy to use, and then allow the underlying model of confirmation to weigh the evidence. That is, the strategies can "argue" about the best rule to use next, and the strategy that presents the best case (as judged by the confirmation model) will win out.

The judgmental character also allows the novel possibility of both inexact and conflicting statements concerning relative order. We might, for instance, have two meta-rules which offer different opinions about the order of two sorts of object level rules, indicating that there is evidence that "subset X should probably (.6) be done before subset Y", and that "subset Y should probably (.4) be done before subset X". Once again, the underlying model of confirmation will weigh the evidence and produce an answer.

Next, there are several advantages associated with the use of strategies which are goal-specific and are embedded in a representation which is the same as that of the object level knowledge. The fact that strategies are *goal-specific*, for instance, makes it possible to specify quite precise heuristics for a given goal, without imposing any overhead in the search for any other goals. That is, there may be a number of complex heuristics describing the best rules to use for a particular goal, but these will cause no computational overhead except in the search for that goal.

The use of a *uniform encoding of knowledge* makes the treatment of all levels of knowledge the same, and this offers several advantages. For example, work on TEIRESIAS included development of two capabilities involving object-level rules: explanation of system performance (via display of relevant rules) and knowledge acquisition (via mixed-initiative dialog). The uniform encoding eases the task of extending these capabilities to meta-rules as well. The first of these (explanation) has been done, and makes possible an interesting capability: in addition to being able to display the object-level rules used during a consultation, the system can similarly display the meta-rules, thereby making visible the criteria it used in "deciding how to do what it did". Knowledge in the strategies has become accessible to the rest of the system, and can be displayed in the same fashion.

There are additional advantages associated with making strategies explicit; these are described in Section {8.1}.

{6} Meta-rules as invocation criteria

The discussion so far has dealt with meta-rules as strategies, and demonstrated their use in tuning the performance of goal-directed invocation. In this section we show how to broaden this so that, rather than simply tuning an existing control structure, meta-rules can be used by the programmer to specify generalized invocation criteria. In doing this we broaden the issues somewhat, and see what arises from thinking of strategies and invocation criteria interchangeably.

As a first step in generalizing their use, consider that meta-rules can be used to tune a variety of control structures, in addition to the goal-directed variety. They can be used, for instance, to guide data-directed invocation as well, and this has been done: before execution of the set of (antecedent-style) rules triggered by making a conclusion, the system checks for meta-rules associated with that conclusion. These meta-rules can prune or reorder the set of antecedent rules to be used. This makes it possible to control the depth and breadth of the actions triggered by the addition of new information to the data base.

Next, consider that the idea of "tuning" appears to fit fairly naturally within the framework of any control structure in which multiple rules may be relevant at any one time. In general, the set of plausibly useful rules will be determined by the control structure in use, and meta-rules can then be used to "fine tune" by reordering or selecting from that set.

We can generalize this further. In systems which employ both goal-directed and data-directed

processes, for example, it is often important to be able to decide which one to use at different points in the processing. We can imagine writing meta-rules which select from this set of options as well, thereby (dynamically) controlling the selection of an appropriate control structure.

As a final generalization, consider that meta-rules can be used as a medium for expressing the criteria for retrieving a rule. Goal-directed invocation, for instance, can be expressed as "rules which mention <goal-name> in their action", while data-directed invocation is "rules which mention <data-item> in their premise". Thus, in addition to thinking of them as a tool for fine tuning a control structure, strategies can also be seen more generally as a source of information about *how or what knowledge in a system to use.*

Note that the "language" being employed to specify retrieval criteria is that of the rules, augmented by the underlying programming language (e.g., MENTIONS is a function written in LISP). Using the same rule representation at the meta and object levels offers advantages of uniformity, while augmenting this with LISP constructs provides (low-level) expressive power and extensibility. The resulting language is useful, but as yet only a single step toward the richly expressive language that we noted as desirable in Section {2}.

{6.1} The fully general scheme

To make use of this more general view of meta-rules, we require some extensions to the architecture described so far. One possible scheme is proposed below; it has not yet been implemented.

The basic execution cycle of the system would consist of two phases, each effected via the layered control technique noted above. In the first phase invocation criteria would be selected and applied; in the second the set of rules retrieved would be "fine-tuned" by the existing meta-rule mechanism. The first part of the cycle would start with the goal of deciding which invocation criteria to use (goal-directed, data-directed, etc.), and would retrieve all meta-rules which specified a criterion. But before invoking them, it would check for rules at the next level, which specified how to select from among various invocation criteria. Before invoking them, it would check for rules at the next level, etc. When this process finishes, it will have selected a particular retrieval criterion (or set of them), and retrieved the appropriate rules.

The next phase would then proceed as above, using the existing form of meta-rules to select from, or reorder the set of rules retrieved.

Note that the system is highly responsive, in the sense that it has the ability to choose a new control structure (e.g., data-driven vs goal-driven) dynamically at every cycle. The appropriate sort of rules could be included in the system to insure continuity of behavior over several cycles (e.g., "deciding which stock to invest in is important, so keep working on that goal until a decision is reached"), or to shake the system loose from a bad path (e.g., "if you're been trying to determine investment timescale for more than 5 cycles, give up and move on to something else"). The architectures described in [13] and [16] have some elements in common with this; they report favorably on the utility of the approach.

As a result, we have a very simple, multi-layered goal-directed behavior in the background ("figure out which invocation criteria to use"), and use this as framework for specifying the standard sorts of control structures. Since this approach is still speculative, we suggest it here more as a point of interest than as a recommended technique.

{7} Analysis and historical overview

The ideas and mechanisms presented above offer a number advantages and a number of limitations. To put these in perspective we present here an historical overview, and compare meta-rules to techniques used previously. We consider subroutines (procedures), the operators used in GPS [19] and later in STRIPS [10], production rules [19,4], and the theorems in PLANNER [14] and operators in QA4 [21]. As we will see below each of these presents a different approach to invocation. This analysis is somewhat detailed, but the point is straightforward: previous invocation

mechanisms all display some shortcomings. The techniques used in meta-rules offer a step toward overcoming those problems.

Note that each of the mechanisms is considered an approach to encoding knowledge in a program, i.e., they are a source of knowledge. Thus, for the sake of a common terminology, we adopt the terms in [13,17], and refer to any of them interchangeably as a "knowledge source" (KS). For the sake of familiarity, we occasionally slip back into using the term "procedure" in a similar generic sense.

{7.1} "Handles" and "bodies"

We begin by distinguishing between the *handle* and *body* of a KS: the body is the part actually executed, while the handle is anything used as a way of retrieving the body. For a PLANNER consequent theorem, for instance, the goal pattern is the handle, while the theorem code is the body. (In this case the handle and body are distinct, but as will become clear, this is not always so.) We do not include the goal pattern in the body because the pattern is only matched against, never executed. To see the difference, consider the execution of the rule shown in Figure 1: it would be retrieved because (like a PLANNER theorem) part of it (the action) specifies that it can accomplish a particular goal (concluding about a stock). The conditions mentioned in the premise are then tested. Unlike a PLANNER theorem, however, if those preconditions evaluate successfully, the step specified in the action is carried out (i.e., it is evaluated and the conclusion is made). In PLANNER consequent theorems the goal by which the theorem was retrieved is never explicitly asserted as having been accomplished (except if the programmer puts such an assertion in the theorem body). Accomplishment of the goal is thus implicit in the theorem's succeeding. (There are of course good reasons for having done this, and a small change to the PLANNER interpreter could easily change this if desired. The point is only to show why we have chosen to draw the line between handle and body where we did.)

Since the handle provides the basis for indexing and retrieval, it is the primary element determining the character of invocation available. Our concern in the discussion that follows will be primarily with the various types of handles provided by the KSs noted above, and in particular, their *expressiveness* and *validity*.

We define *expressiveness* as the richness of the language that can be used as a handle. One crude measure is to determine if it would make any difference in program performance if every occurence of a particular handle in the program text were uniformly replaced by some arbitrary string. That is, is the handle merely a pointer, or does its structure convey information?

The *validity* of a handle is determined by the nature of the relationship between the handle and the body. A crude measure here concerns the effect of editing the body: given some KS that is invoked once during the course of program execution, what effect will editing the body have on that invocation? More precisely, is there *any part* of the body alone which we can modify and thereby produce any effect on whether or not that KS is ever invoked? Where the handle and body are disjoint, as in PLANNER for instance, the answer is "no". (These definitions will become clearer as the examples below are reviewed.)

{7.2} Reference by name and by description

One further distinction is necessary to the analysis below. A KS to be invoked can be specified either by *naming* it explicitly, or by *describing* it, perhaps via a predicate indicating required characteristics. Both of these approaches have been used in the past. A procedure is of course invoked by naming it, and PLANNER's THUSE construct illustrates the same mechanism in a slightly more sophisticated context: the effect of a statement like

```
(THGOAL (WIN POKER HAND) (THUSE BLUFF DRAWAFEW CHEAT))
```

is to specify three plausibly useful theorems and the order in which they should be tried. The GOALCLASS statement of QA4 is quite similar.

The "theorem base filter" (THTBF) mechanism in PLANNER, however, effects reference by

description: it allows the programmer to specify a predicate to be used to filter the invocation of the plausibly useful theorems. Thus the effect of

```
(THGOAL (WIN POKER HAND) (THTBF BECONSERVATIVE))
```

is to apply the BECONSERVATIVE filter to every theorem whose pattern matches the goal, and use only those for which the predicate succeeds.

Finally, note that one way to accomplish reference by description is via a set of what we call *external descriptors*, while another implementation would be to use *direct examination of KS code*. In the former case, a number of different characteristics could be chosen, and each KS described in terms of them. For a standard procedure, for instance, the descriptor set might include elements describing the procedure's main effect, preconditions for its use, etc. These would then be associated in some fashion with the procedure (e.g., in LISP, on the property list of the function name).

The alternative of direct examination of KS content is illustrated by meta-rules. Instead of relying on descriptors which are distinct from the body of a KS, they examine the code of the object level rules directly; i.e., they "go in and look" for any desired characteristics. For METARULE001, for instance, the set of "rules which mention in their premise the income-tax bracket of the client" is determined by allowing the meta-rules to examine the source code of the relevant rules to see which ones contain the desired item in the appropriate location. Because this technique relies on the content of the KSs in use, rather than any external descriptors, we refer to it as *content-directed invocation*[3]

{7.3} **Previous approaches to invocation**

Now consider the types of handles that have been used, and their relative expressiveness and validity. In historical perspective, the concept of a subroutine can be viewed as the introduction of the notion of a distinct, non-trivial KS. The only handle on it is its name, and it is purely a token, devoid of semantics (except of course in the programmer's mind). There is only one way to express which subroutine you want (by naming it), and hence this technique has minimal expressiveness. Since there is no formal connection between the name and body, arbitrary changes can be made to the body without affecting whether or not it is invoked. This handle thus has minimal validity. As a result, the programmer himself must know exactly what each subroutine contains, and select by name the proper one at the proper point in the program.

The first major departure from this came in GPS and GPS-like systems such as STRIPS. In the latter, for instance, the handle on each KS is provided by the contents of its add and delete lists. Note that part of the *definition* of the KS itself (the add list) provides the handle, and the name of the KS has become inconsequential.

Production rules are similar, since they are retrieved on the basis of symbols appearing in either their left or right hand sides, symbols which are part of the definition of the KS itself.

In both of these cases, the handle has a strong degree of validity, since it is based, as noted, on part of the definition of the KS itself. Note also that editing the body of either of these can produce changes in invocation patterns. (This is still incomplete, however, since, for example, changes made to right hand side of a rule that is accessed by its left hand side may fail to produce a different invocation pattern.) But note that expressiveness is still minimal. In both cases all of the retrieval mechanism is embedded in the system interpreter (means-ends analysis for GPS and STRIPS, varying control regimes for production rules). A single, "hardwired" mechanism effects KS retrieval, leaving the user no control over which KS is invoked. Thus, while the handle on the KS has a strong degree of validity, the user has no means of expressing his preferences of which should be retrieved.[4]

With the advent of pattern directed invocation, the subroutine acquired a *pattern* which is used as the handle. There is a limited but useful sort of expressiveness here, in a syntax which typically permits a broad range of patterns. Note that the structure of the pattern conveys information, and hence it could not be replaced by an arbitrary string without affecting program performance. Consider, however, the validity of the handle. The pattern, like a subroutine name, is

distinct from the body, hence there is no guaranteed correspondence. Arbitrary changes can be made to the body of a PLANNER theorem, for instance, without changing whether or not it is invoked. More realistically, in the course of debugging a program, so many modifications might be made to the body of a theorem that the pattern no longer describes it appropriately, yet the invocation will continue as before.

The next step in the evolution of invocation mechanisms was to provide a set of interpretations for patterns, adding further expressiveness to the language. Goal- and event-directed invocation are the two best-known examples here. In PLANNER, for instance, there are three classes of patterns while QA4 offers goal patterns plus an extensive set of event descriptors in a general demon mechanism. But the validity remains the same (i.e., minimal), because there is still no formal link between the handle and body[5].

We thus have, in retrospect, two lines of development pushing in different directions. The GPS-STRIPS-production rule line offers retrieval with strong validity because the handle is part of the KS body, but minimal expressiveness, since the control regime is predetermined and inflexible. The subroutine-PLANNER-QA4 line, relying on the external descriptor approach, offers handles with minimal validity, but with increased expressiveness.[6]

Even so, that expressiveness is limited, because the set of available descriptors is *predetermined*, and *hardwired*. As noted, PLANNER offers three interpretations of a pattern in a theorem (goal, assertion event and erasing event), and there is no way for the programmer to add to this set. The problem, in general, is that the available set of descriptors is in all cases predetermined by the language designer, and not modifiable by the user.

Consider also that the "goal" of a procedure is only one description of it. What about its speed, space requirements, side-effects, other procedures it calls, etc. All of these may be relevant in deciding which procedure to invoke next, yet no existing mechanism offers a way of expressing such preferences explicitly.

But more generally, why have any predetermined set of descriptors? Why not make it possible for the user to define his own invocation criteria, by offering him a richly expressive language, and allowing him to describe the characteristics of the KS to be invoked next? And why not insure the validity of those descriptions by effecting them not via external descriptors, but via direct reference to the KS content itself. As we have seen, the ideas on which meta-rules are based take a step toward achieving this.

A number of informal techniques have been employed in the past to obtain some of the same effects. Some theorem provers (e.g., QA3 [12]) have expressed their control strategies in terms of a wide range of descriptors about the clauses they manipulated. While this achieves the effect of allowing generalized invocation criteria, the resulting systems had less flexibility than the one described here, since they used external descriptors (typically property list entries). Consider, for instance, the problem of adding a new descriptor to the established vocabulary. It might be a sizable task to go through the system and add the appropriate entry every place it was needed. Content-based description may make this unnecessary: as long as the new descriptor concerns a property computable from the content of the clauses, it would be possible to define that property, and allow the system to determine which items met its description.

The advantage of using generalized invocation criteria is clear, whether implemented as here in meta-rules, or in other schemes: it makes available a high degree of control over system performance. Consider once again the historical perspective. The programmer using procedures effectively says "give me that KS next", indicating it by name. In GPS-STRIPS, and traditional production systems, the user has little or no control over which KS is invoked next. PLANNER made it possible to say "give me any KS whose pattern matches this one"; in using that pattern as a designator of a goal the request becomes "give me any KS that achieves the goal designated." Finally, generalized invocation criteria make it possible to say "give me any KS that fits the following description." By writing the proper sort of description, we can have invocation that is goal-directed, side-effect-directed, speed-directed -- in short directed by any one of or a

combination of factors.

The idea of content-directed retrieval means that the "handles" effected by these criteria have a strong degree of validity. Because direct reference is made to the body of the KS, there is a formal link between the KS and its description. Where (as in meta-rules) these two ideas of generalized invocation criteria and content-directed retrieval are combined, we have available a mechanism which can be highly expressive, and yet retains a strong degree of validity.

{8} Implications

As we have noted, it takes some effort to make non-trivial use of the the mechanisms described above. Once used successfully, however, they offer a number of additional benefits besides the direct utility of making it easier to define new control regimes.

One of these has been mentioned previously: the use of second and higher order meta-rules as "criteria that choose criteria". As we have noted, each successive order of rules in the system treats all levels below it as data. This makes it possible to write successively higher order "advice", starting by defining a control regime, then moving on to rules which decide which regime to use, deciding how to decide which to choose, etc.

Other benefits follow from the concepts of generalized invocation criteria and content-directed invocation.

{8.1} Generalized invocation criteria

One important benefit that arises from user-definable invocation criteria is the opportunity to express those criteria *explicitly*, rather than using a range of implicit or indirect techniques to get the effect desired. The latter situation often arises when the available set of invocation mechanisms is incomplete for a particular problem, and the programmer resorts to various devious techniques. One popular approach is that of getting a multitude of effects from a single mechanism. Where KSs are retrieved via pre-computed index lists, for instance, a commonly used approach is to hand-tool the ordering of these lists to achieve effects not otherwise available in the existing formalism. For example, where goal-directed invocation is accomplished by using pre-computed lists of operators, hand-tooling those lists can add a range of other control regimes. In [24], for instance, the GOALCLASS lists were hand-ordered to insure that the fastest operators were invoked first; the analogous lists in MYCIN [23,8] have in the past been hand-tooled to effect a number of different partial orderings on the rules that are invoked.

A similar example arises when using a multiple priority level agenda of the sort described in [2]. Suppose, for example, we wanted to insure a particular partial ordering of processes to be put on the agenda. Note that there is no way to say explicitly, *make sure that these processes (in set A) are executed before those (in set B)*. Instead, we have to be indirect, and could for instance, assign a priority of 6 to the rules in set A, and a priority of 5 to those in B.

There are a number of problems associated with trying to do these sorts of indirect encodings, all of which seem to arise from the fact that information is unavoidably lost by the indirection involved. Note that in all the cases above, the intent of the hand-tooling and indirect priority setting is nowhere recorded. The resulting system is both opaque, and prone to bugs. For instance, in the agenda example, after the priorities have been set, it will not be apparent *why* the processes in A were given higher priority than those in B. Were they more likely to be useful, or is it desirable that those in A precede those in B no matter how useful they each may be? After a while, even the programmer who set those priorities may forget what motivated the particular priorities chosen.

Bugs can arise in this setting due both to execution-time events, and events in the long-term development of the program. Consider for instance what happens if, during a run, before we get a chance to invoke any of the processes in A, an event occurs which makes it clear that their priority ought to be reduced (for reasons unrelated to the desired partial ordering). If we adjust only the priority of those in A, an execution-time bug arises, since the desired relative ordering may be lost. Yet there is no record of the necessary connection of priorities to remind us not to adjust only one

set of them. A similar problem can arise during the long-term development of the program, if we attempt to introduce another indirect effect by juggling priorities, and end up modifying those in set A without the necessary adjustments to those in B.

The problem is that this approach tries to use a single invocation mechanism to accomplish a number of effects. It does this by reducing a number of different, incommensurate factors to a single number, *with no record of how that number was reached.* Meta-rules offer one mechanism for making these sorts of considerations explicit, and for leaving a record of why a set of processes has been queued in a particular order.

Meta-rules also offer the advantage of *localizing* all of the control information. Note that juggling priorities means trying to achieve a global effect via a number of scattered local adjustments. This is often quite difficult, and can be very hard to change or update. Localizing each such invocation criterion in a single meta-rule makes subsequent modifications easier, since all of the information is in one place -- changing a criterion can be accomplished by editing the relevant meta-rule, rather than searching through a program for all the places priorities have been set to effect that criterion.

Our historical overview provides another observation. Consider viewing the progress from standard procedure calling, to techniques like goal-directed invocation, as making it possible to be less precise about the role of a given procedure in a program. Where invocation by name requires that we decide exactly *where* and *when* the code is to be invoked, goal-directed invocation requires only that we specify *how* it is to be used. The perspective suggested here moves us another step along that line, by considering retrieval separately from the writing of a KS. The programmer can, if he desires, write a KS without specifying how it is to be used, and can leave that up to the invocation criteria to decide.

The point here is not that the programmer shouldn't specify such information, since it can be very useful for cutting down combinatorics. For inference rules, for instance, it is important to know which are more useful in the goal-directed mode, and which more useful in the data-directed mode. Such information can cut down the search space when trying to achieve a goals, and control the number of forward inferences drawn from a new assertion.

We claim only that if the programmer does want to specify such things, it is better if he is not forced to make that information synonymous with the handle used to retrieve the code. (This occurred in PLANNER, for instance, where the indication that a theorem ought to be used in a goal-directed mode for a particular goal becomes the sole way of indexing it (by the single goal pattern)). If we keep these two things separate, we allow information about appropriate use to become a piece of advice rather than a constraint.

One other advantage may result from this: unexpected uses for code. Recall that we allow as specifications arbitrary computable predicates. It is possible (although we have not observed it as yet) that a KS written early in program development may later on be examined by a newly added invocation criterion which finds in it characteristics that may not have occurred to the programmer who wrote the code. As a result, the system may find unexpected applications for its code.

{8.2} Content-directed invocation

In addition to the *validity* of the retrieval process as described above, content-directed invocation offers an unusual degree of *flexibility* in a program. Consider, for example, two kinds of changes that might be made in a program over the course of its development, and the amount of effort necessary in the three different retrieval schemes described in Section {7.2}. Table I briefly summarizes the interactions discussed here, organized by the different retrieval schemes (the rows; reference by name, reference by description using external descriptors, reference by description using content reference), and the two kinds of changes (the columns; changing or adding an object-level rule; changing or adding a meta-rule). Note that meta-rule can also be interpreted as "invocation criterion", and object-level rule as "knowledge source".

Consider the effect of editing (or adding) an object-level rule, and imagine that meta-rules

used the reference by name approach (i.e., they said something like `If ... then there is evidence that RULE002, RULE456, ... are likely to be useful`). After editing an object-level rule in such a system, first, all meta-rules that mention it must be retrieved and examined to see if they still apply, and edited accordingly. Next, it is also possible that the revised object-level rule should now be mentioned in other meta-rules, so the rest would also have to be examined. In the external-descriptor approach, we need only update the appropriate descriptors, which would be stored with the object-level rule. In addition, the updating required should be evident from the sort of editing done on the object-level rule itself. All relevant meta-rules will then automatically adjust to these changes. With content-reference there is no additional effort of even updating descriptors, since the meta-rules will adjust to the changes found in the edited rule (assuming that suitably powerful code examination techniques exist; see {9.1} for comments on this).

Table I: Flexibility Benchmarks

retrieval basis	edit or add object-level rule	edit or add meta-rule
reference by name	check all meta-rules to see which should name it	check all rules to see which it should name
reference by description, via external descriptors	update its descriptors	possibly no addl effort, possibly have to add new descriptor
reference by description, via content-reference	no additional effort	no additional effort

Adding a new meta-rule to the system (or revising an old one) also causes problems for the reference by name approach: it necessary to review all the object level rules to determine which the new or revised meta-rule should mention. Using external-descriptors, it is possible that no additional effort is required, if the description in the new meta-rule uses the available "vocabulary" of descriptors. If, however, it requires a descriptor which is not yet in that vocabulary, we have the formidable task of reviewing all existing object level rules, and adding to each the appropriate entry for the new descriptor. Note that the entire operation may be transparent when content-reference is used.

One interesting additional benefit that arises from the techniques we use is that we appear to have taken another step toward transferring more of the burden for system upkeep to the computer. This long-term trend is noted in [20], where it is illustrated by using the example of high level languages, which have taken increasing amounts of the responsibility for storage management out of the hands of the programmer. This has numerous benefits, among them the fact that the programmer has been given a chance to avoid yet another uninteresting tasking that presents numerous opportunities for error. This same effect results several times in the work reported here. The assured validity of retrieval criteria, for example, avoids the problems noted of having to maintain KS descriptors separately from the body they describe. The opportunity to write code without having to specify how it is to be used presents a similar advantage, as does the flexibility arising from content-directed invocation when making changes to a large system. The result is that the programmer can focus more on what it is he wants to say, rather than exactly how it is to be

said.

{9} Limitations

There are of course a number of limitations encountered in attempting to use the techniques we have outlined here. We consider here the nature of the limitations encountered, and attempt to characterize the sorts of problems for which this technique seems well-suited.

{9.1} Difficulties

We have dealt s mewhat blithely in previous sections with the idea of having invocation criteria that examine code. This is of course a very difficult problem, for several reasons. First, it can be difficult even for a programmer to read an arbitrary chunk of code and understand it. Second, even if we could read it, it is not always clear what to look for; try to specify for instance how to detect by reading its code what goal a procedure achieves.

TEIRESIAS currently has only the simplest form of code examining ability, made possible by several useful shortcuts. We rely first on the nature of the representation in use. The organization of information of rules makes it possible to say that, e.g., "*rules which mention* STOCKNAME *in their conclusion*" is an example of goal-directed retrieval. Second, the rules are viewed as a task-specific high level language: the primitive terms they use are both domain-specific, and reasonably abstract (e.g., income tax BRACKET, etc.). This makes their code much easier to decipher than, say, an assembly code version of the same thing. Finally, the code is strongly stylized (the predicates and associative triples), allowing us to associate with each predicate function a template that helps decipher its code. The template describes the format of a call to the function (the order and generic type of its arguments), much like a simplified procedure declaration. Each predicate function thus carries a description of its own calls, and by referring to that description (i.e., retrieving the template associated with the CAR of a LISP form), we can dissect that call into its components (see [3] for a full description of this process).

We have, of course, thus far used this ability in only the most basic, syntactic ways. We have not yet developed means of deriving more interesting, semantic content by examining code, and this is clearly a difficult problem. While the shortcuts described can be used with representations other than rules, they are not universally applicable. But while this whole problem is difficult, it is also a separable issue. That is, the extent of the current capability to examine code is extremely elementary, but even the simplest form of it makes available the interesting capabilities displayed above.

There is also a problem with the expressiveness of our current language. The combination of the meta-rules and LISP is at best, adequate (in the sense of being able to express any computable predicate), and extensible only because, for example, new predicate functions can be added to the rule language by writing them in LISP. It would be better, of course, if the meta-rule language already included a large set of well-designed primitives that provided a good foundation for expressing invocation criteria. Then we could suggest not only that "it's useful to be able to define your own, generalized invocation criteria", but might also say, "and here's a well-designed language that will help get you started, without restricting you since it's also extensible." It would be useful to be able to offer the user an initial set of descriptors (like goals, side effects, speed and space requirements, etc.) which was rich enough to allow him to express interesting criteria easily (i.e., without having to discover and implement that set of primitives on his own).

Nor do we advocate meta-rules as a preferred language for strategies or invocation criteria. A rule-based representation does present some advantages for encoding reasonably small, modular chunks of knowledge. But as we haved noted elsewhere [4], it is not particularly well suited to expressing larger, more complex constructs. The design of a good strategy language is still very much an open issue.

The discussion above also dealt blithely with the idea of replacing reference by name with reference by description. Yet it is not always clear how to generalize from a specific procedure to be invoked, to a general description of the capabilities desired. Consider the PLANNER fragments in

Section {7.2}, for example, and note the difficulty of replacing the THUSE advice with more general characterizations.

There is also the issue of the amount of compute time required to do this form of indirect referencing, compared to the speed of using names. If we are willing to assume that no new KSs are created during program execution, then, most of the computational cost can be paid in a background mode computation between performance runs. During that time the system could compute the sets of KSs determined by the various descriptions.[7] The results might either be saved for execution time use, or, in a form of "pre-compiling", the source code might be re-written, replacing the descriptions with the sets they define. The idea of computing such indexing lists before execution is of course a standard technique. We are simply suggesting that the basis for doing the indexing might be more sophisticated than the simple literal-matching done in production rules, PLANNER's discrimination net, etc. (Some recent work described in [1] takes a step in this direction.) By doing this, we obtain both the flexibility of reference by description, and the speed of reference by name.

We can even back off a little on our assumption about no new KSs being created during program execution. The worst that happens is that the original descriptions would have to be retrieved, and applied to the new KS, to see if it meets their criteria. This could be done either "on demand" (as each invocation criterion is about to be used), or the system might pause to "recompile". In either case, the time to accomodate a single new KS might not be unreasonable.

Finally, we made the point earlier that this approach helps provide a supportive environment for the programmer, by discouraging some varieties of indirect or implicit coding tricks. The mechanisms we have provided can still, of course, be misused. One might, for example, write predicates specifically tailored to the body of a particular KS, perhaps by testing for a uniquely named local variables, etc. This might achieve some other desired effect via an accidental correlation with that local variable, and hence retrieval is again indirect. But note that there is one small improvement at least: the retrieval criterion will appear explicitly, and be clearly visible. If it seems bizarre, that's a hint that perhaps something devious is going on. We cannot prevent this from happening, but at least we can make it highly visible.

{9.2} Appropriate problems

We have explored here a particular perspective on programming, and examined a number of mechanisms based on it. Since this perspective is not universally applicable, we consider here the character of the problems for which it appears to be useful.

It must first of all be possible to decompose the problem into parts of roughly the right size. If the problem doesn't decompose, then we have no set of KSs to choose from; if it is decomposed into parts that are too small, we may be hard-pressed to produce non-trivial descriptions of them.

There must also be some degree of "ill-structuredness" [18] or indeterminacy to the problem. This approach seems to make little sense if the problem can be broken down into a totally specified sequence of operations, with no uncertainty about what to do next. In that case it makes more sense to apply the more traditional, (i.e., ALGOL style) programming. This view is more appropriately applied to the kinds of problems typically attacked with the more recent AI languages which permit a degree of indeterminacy. In that case the ability to specify invocation criteria can be a useful approach.

There is some question, too, of whether this very general "deciding how to decide" process is always the appropriate thing to do. Note, however, that we need not adopt it in all its full-blown generality. It might be used to tune an existing control structure (as in MYCIN), or applied selectively to choose between control regimes. The more the problem is ill-structured, the more general an implementation we might want to use. In more structured contexts, parts of the technique may be adopted as needed.

Our techniques can also be useful when building large systems subject to frequent developmental changes. Recall the discussion in Section {8.2}, which demonstrated that using content-directed invocation made the system in part self-adjusting: changes to KSs and invocation

criteria were automatically accomodated. This can be a useful property as the number of KSs in the system gets large, and the burden of reviewing them begins to be unmanageable.

Thus in general, our view appears well suited to problems which can be appropriately decomposed, and for which there is some degree of ill-structuredness, arising either out of the nature of the task-domain, or the problems of incremental construction of large systems.

{10} Status of the current implementation; related work

The current implementation of TEIRESIAS serves mainly as a feasibility study of the ideas described above. There are a half-dozen meta-rules, all first order. This is primarily due to the fact that MYCIN's knowledge base is still small enough to permit exhaustive invocation. Thus in their present environment, meta-rules have something of the character of a solution looking for a problem.

The current implementation contains all of the features mentioned above (the capacity for arbitrary levels of meta-rules, the use of meta-rules to guide data-directed invocation, etc.), with the exception of the "pre-compilation" mentioned in Section {9.1}. In addition, as noted, there is as yet in the system only a very primitive ability to examine code, and very basic language for expressing invocation criteria. Both of these are important foci for additional work.

The ideas we have described have found application recently in another system: PSI [11] a rule-based program synthesis system. One of the modules in the system is LIBRA [15], the "efficiency expert" which oversees the use of the rules in generating programs. In LIBRA, first and second order meta-rules have been employed to control the retrieval and application of the object-level rules in the system. The concept of a layered control structure has also proven to be an effective framework for organizing the necessary information.

The work reported here was influenced by some of the early developments in the HEARSAY II [17] system. While much of that work proceeded in parallel with this, their generalized control structures using blackboards, and multiple, competing knowledge sources helped to provide a useful framework for thinking about the issues that meta-rules raised.

{11} Conclusions

In the preceding sections we have explored the concept of a strategy, viewing it initially as information about which process to invoke when more than one is applicable. This was illustrated in a rule-based system, and we noted that this implementation provides a number of advantages arising from the use of multiple levels of control, and a uniformity of knowledge representation across all levels.

This framework was then generalized to show that the same mechanisms could be used to specify invocation criteria, leading to the more general view of strategies as any information about how or what knowledge in a system is to be used.

We then adopted the view that program execution can be viewed as a process of "deciding which procedure to invoke next", and saw that this view makes sense where it is possible to structure a program as a collection of distinct, non-trivial procedures (or other form of KS). Its value lies in the way it encourages both a broad perspective on the process of invocation, and the explicit specification of the knowledge used in that process.

Having broadened the standard view of the invocation process, the inadequacy of existing mechanisms soon becomes clear. This leads to the observation that the programmer should be able to state generalized invocation criteria in a richly expressive, extensible language. In place of the usual set of predetermined invocation modes hardwired into a language interpreter, we want instead the ability to specify a wide range of descriptions. The language provided by meta-rules is a step toward that goal. Ultimately we want a language that is both richly expressive (to aid the programmer in writing invocation descriptions), and extensible (since we don't want to limit him to a particular vocabulary of invocation modes). We saw that this presents advantages in terms of making possible explicit rather than implicit expressions of control.

We noted that invocation criteria can be embodied as objects at the level of other program constructs. Where they are typically "hidden away" in the language interpreter, in the sense that they are inaccessible to a program, we have proposed instead that, as illustrated by meta-rules, they can be embodied at the level of other program constructs. This makes it possible for invocation criteria to control the use of other invocation criteria, as illustrated by meta-rules at higher levels examining and reasoning about those at lower levels.

The layering of control employed here aided in organizing the information used in specifying a control regime. As noted, each level controls the use of information at the level below it, and this provides guidance about where in the hierarchy a particular piece of information ought to go. It also means that a high-level rule in this scheme can have a powerful focussing effect, analogous to having a good heuristic available when making the early choices in a tree search.

Finally, we saw that content-directed invocation is a useful technique for implementing retrieval criteria. By removing the distinction between the content of a KS and the structures by which that KS is retrieved, content-directed invocation makes possible a useful degree of validity. In addition, its use can make a system more flexible in the face of some standard sorts of modifications. While it requires advances in the field of code understanding to implement fully, the benefits of using it are incremental, and it offers a useful ability even at the simple levels of implementation illustrated here.

Acknowledgments

Cordell Green and David Barstow made a number of very useful suggestions on earlier drafts of this paper.

Notes

(1) This stock market system is modelled after the MYCIN system [23,8], which provided the context within which TEIRESIAS was developed and meta-rules were implemented. We have abstracted out here just the essential elements of MYCIN's design, and shifted the domain from medicine to the stock market via a few word substitutions (e.g., *E.coli* became *AT&T*, *infection* became *investment*, etc.). This shift was done to keep the discussion phrased in terms familiar to a wide range of readers, and to emphasize that neither the problems attacked nor the solutions suggested are restricted to a particular domain of application.

(2) That is, are there meta-rules directly associated with that goal. Meta-rules can also be associated with other objects in the system; the range and utility of such indexing points is a sizable topic considered in more detail in [3].

(3) There has been a parallel evolution in access to memory locations, from absolute binary, to symbolic addressing in assemblers, to relocatable core images, on up to content addressable memories (e.g., LEAP [9]). Content directed invocation can thus been seen as the procedural analogue of content addressable memory.

(4) It is possible to force specified interactions by anticipating the operation of the retrieval mechanism, but this is often difficult for a large system, and in any case contrary to the "spirit" of the formalism. See [4], especially Section {5}, for a discussion of the issue.

(5) We can also consider some of the recent work on task agendas (e.g., [16]) in this light. Rather than attempting to specify a procedure at all, this approach instead "posts" a task list, and the various procedures examine the list to see which they might be able to achieve. This inverts the usual approach: rather than having to describe the kind of procedure to invoke, each procedure carries an expression describing the kinds of tasks it can accomplish.

(6) Indeed, there is the potential for being seduced by a false sense of power that these interpretations imply. The KS handle is no longer a token, but conveys information. When that information is advertised as a "purpose", it's easy to start believing that every KS is sure to achieve its advertised "purpose", and easy to forget that it is purely the programmer's responsibility to insure that this happens. For further comments on this issue, see [3].

(7) There are non-trivial problems associated with the appearance of free variables in the description parts. See [3] for ways of dealing with this.

References

[1] Barstow D R, A knowledge-base for organizing rules about programming, *Proc Pattern-Directed Inference Systems Workshop*, to appear.

[2] Bobrow D, Winograd T, An overview of KRL, *Cognitive Science*, vol 1, pp 3:47, January 1977.

[3] Davis R, Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases, Stanford HPP Memo 76-7, July 1976.

[4] Davis R, King, J J. An overview of production systems, in *MI 8: Machine Representations of Knowledge*, (Elcock and Michie, eds), John Wiley, 1977.

[5] Davis R, Knowledge about representations as a basis for system construction and maintenance, to appear in *Pattern-Directed Inference Systems*, (Waterman and Hayes-Roth, eds.), Academic Pres, (in press).

[6] Davis R, Interactive transfer of expertise, to appear in *Proc 5th IJCAI*, Aug 1977.

[7] Davis R, Buchanan B G, Meta-level knowledge: overview and applications, to appear in *Proc 5th IJCAI*, Aug 1977.

[8] Davis R, Buchanan B G, Shortliffe E H, Production rules as a representation for a knowledge-based consultation system, *Artificial Intelligence*, 8:15-45, Spring 1977.

[9] Feldman J, et. al, Recent developments in SAIL, an ALGOL-based language for artificial intelligence, Stanford AI Memo 176, November 1972.

[10] Fikes R, Nilsson N, STRIPS- a new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, 2:189-208, Winter 1971.

[11] Green C C, The design of the PSI program synthesis system, *Proc 2nd Internatnl Conf on Software Engineering*, Oct 1976.

[12] Green C C, The application of theorem proving to question answering systems, Stanford AI Memo 96, August 1969.

[13] Hayes-Roth F, Lesser V R, Focus of attention in the HEARSAY II speech understanding system, Carnegie-Mellon Computer Science Dept Report, January 1977.

[14] Hewitt C, Description and theoretical analysis of PLANNER, PhD Thesis, Department of Mathematics, MIT, 1972.

[15] Kant E, The selection of efficient implementations for a high-level language, *Proc AI/PL Conf*, to appear.

[16] Lenat D B, AM: an AI approach to discovery in mathematics as heuristic search, Stanford AI Memo 286, July 1976.

[17] Lesser V R, Fennell R D, Erman L D, Reddy D R, Organization of the HEARSAY II speech

understanding system, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-23, February 1975, pp 11-23.

[18] Newell A, Heuristic programming: ill-structured problems, in *Progress in Operations Research*, (Aronofsky, ed.), 3:362-414.

[19] Newell A, Simon H A, *Human Problem Solving*, Prentice Hall, 1972.

[20] Pratt V R, The Competence/Performance Dichotomy in Programming. Fourth ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Santa Monica, CA, Jan. 1977. pp. 194-200.

[21] Rulifson J F, et. al., QA4: A procedural calculus for intuitive reasoning, Stanford Research Institute Technical Note 73, November 1972.

[22] Shortliffe E H, Buchanan B G, A model of inexact reasoning in medicine, *Mathematical Biosciences* 23 (1975) pp 351-379.

[23] Shortliffe E H, *MYCIN: Computer-based Medical Consultations*, American Elsevier, 1976.

[24] Waldinger R, Levitt K N, Reasoning about programs, *Artificial Intelligence*, 5:235-316, Fall 1974.