Knowledge Representation Languages and Predicate Calculus:
How to Have Your Cake and Eat It Too

Charles Rich

The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

This paper attempts to resolve some of the controversy between advocates of predicate calculus and users of other knowledge representation languages by demonstrating that it is possible to have the key features of both in a hybrid system. An example is given of a recently implemented hybrid system in which a specialized planning language co-exists with its translation into predicate calculus. In this system, various kinds of reasoning required for a program understanding task are implemented at either the predicate calculus level or the planning language level, depending on which is more natural.

Introduction

The ideas in this paper arise out of my experience in the Programmer's Apprentice project [5,8] over the past several years developing a language for representing knowledge about programs and programming. Therefore, I will begin by briefly recounting the history of the knowledge representation part of our project. Readers who have worked on other projects to build intelligent knowledge based systems will undoubtedly find many aspects of our story quite familiar; this leads me to believe that the novel step we have recently taken may be widely applicable.

The story begins in 1975 when we first perceived the need for a representation, different from the literal text of a program, which described the logical structure of a program at various levels of abstraction. The application system we were designing (the Programmer's Apprentice) would use this representation in various ways to analyze, synthesize, modify, verify and explain programs. Having no preconceived commitment to a particular formalism, we looked to our own intuitions and informal notations on blackboards and scratch pads for an appropriate framework. What we came up with was a language of boxes and arrows, with boxes inside of boxes and several different kinds of boxes and arrows. The details of this language don't matter here (it is described in detail elsewhere [6,7]) — what is important is that its structure originated in our intuitions about what we were representing and the reasoning we intended to perform.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, and in part by National Science Foundation grants MCS-7912179 and MCS-8117633.

The language, which we called *plan diagrams*, was then implemented in a straightforward way and used for various kinds of reasoning tasks over the next four or five years. For example, Shrobe [9,10] wrote a module which verified the consistency of plan diagrams; Waters [11,12] wrote a module which recognized the occurrence of certain common patterns in plan diagrams. During this period, we took a fairly ad hoc[1] approach to the semantics of our knowledge representation. This is not to say that we didn't know what plan diagrams meant, but just that ultimately the meaning of the representation was implicit in the procedures we were writing to manipulate it (some have called this an "operational" semantics).

In 1979, before embarking on a major extension and re-implementation of parts of the Programmer's Apprentice, I undertook to define a formal semantics for plan diagrams. The approach I chose for this task was to translate plan diagrams into another language which already had a well-defined formal semantics, namely a version of predicate calculus. The major benefit I expected to gain from this effort was to clarify some grey areas in the meaning of plan diagrams. As a corollary, I also expected to use the formal semantics as a kind of specification against which our manipulations of plan diagrams could be validated.[2] I did not however expect the predicate calculus to show up in any direct way in the system implementation. It turns out that I was wrong! *The predicate calculus formalization was directly usable as the basis for a practical implementation in which predicate calculus and plan diagrams co-exist.*

Plan Diagrams

In order to appreciate how this state of affairs came about, we now need to describe plan diagrams in more detail. Figure 1 shows the representation in the language of plan diagrams of how to implement a push operation on a stack implemented as an array with an index pointing to the top element.

1. Please note that I do not intend any negative connotation in the use of the term "ad hoc" here, but just the dictionary definition: "concerned with a particular end or purpose."

2. Though this is not the major point of this paper, I claim that this sort of translation into predicate calculus is almost always worth attempting. Moreover, many other people have made this observation [2,13], so it should no longer be controversial.

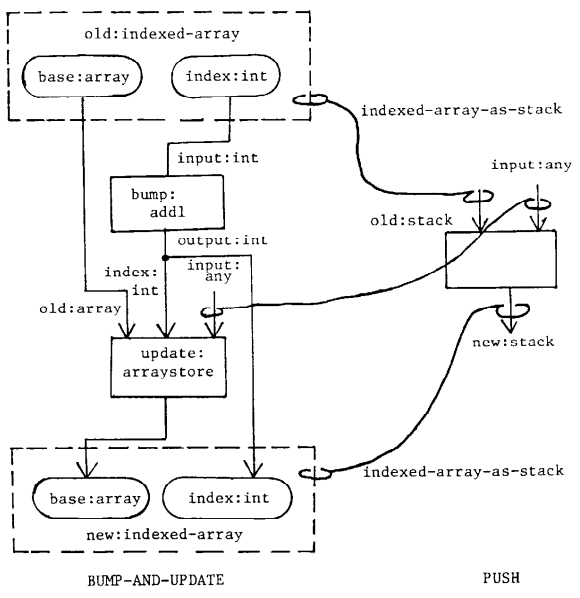
This sort of diagram is called an *overlay*. An overlay is made up of two plans (one on the right side and one on the left side) and a set of correspondences (hooked lines) between them. Each plan is a hierarchical structure with named, typed parts, called *roles*. (There are also *constraints* between the roles of a plan, but since these are already written in a logical language, their translation to predicate calculus is not part of what is interesting here.)

In the overlay of Figure 1, the plan on the right side represents the operation type Push, whose roles are its inputs, named Old (of type stack) and Input (any type), and its output, named New (of type stack). The plan on the left side of the overlay, called Bump-and-update, describes the pattern of operations and data structures which implements Push. This plan has four top-level roles: Bump (operation type Add1); Update (operation type Arraystore); and Old and New (composite data type Indexed-array). Each of these roles has sub-roles by virtue of being composite types: the sub-roles of Bump and Update are the input and output roles of the corresponding operation types; the sub-roles of Old and New are named Base (type array) and Index (type integer). Arrows in plan diagrams, such as between the Output of the Bump of Bump-and-update and the Input of the Update, indicate the flow of data.

Finally, the hooked correspondence lines in an overlay indicate how to map between the implementation description on the left and the more abstract description on the right. For example, the hooked line between the Input of the Update of Bump-and-update and the Input of Push indicates that the element stored in the array in the implementation view corresponds to the object which is pushed onto the stack in the more abstract view.

Generally speaking, translating from plan diagrams to predicate calculus amounts to translating from a richly structured language, i.e. one with many different primitive constructs and methods of combining them, to a much simpler language: the version of predicate calculus I use provides functions and relations as primitive notions, with boolean connectives, quantification and equality as combining methods. Thus one effect of the translation process is to make explicit certain semantic commonalities between different plan diagram constructs. For example, both overlay names and role names become function symbols in the predicate calculus; both data flow arrows and overlay correspondences become logical equalities. The surprising result (though in retrospect it seems obvious) is that these semantic commonalities can also be exploited for an elegant implementation.

FIGURE 1.



BUMP-AND-UPDATE-AS-PUSH

A Hybrid Implementation

The implementation that ensued is a hybrid system in which the predicate calculus translation of a plan diagram co-exists with an explicit representation of its box and arrow structure. Furthermore, both levels of language are used for the various kinds of reasoning that need to take place for the application domain. For example, the following is part of the predicate calculus translation of the left side of Figure 1 (plans become predicates on a domain of n-tuples).[3]

$$
\begin{aligned}
&\text{bump-and-update}(\alpha) \Rightarrow \\
&[\ \text{output}(\text{bump}(\alpha)) = \text{index}(\text{update}(\alpha)) \\
&\wedge\ \text{index}(\text{old}(\alpha)) = \text{input}(\text{bump}(\alpha)) \\
&\wedge\ \text{base}(\text{old}(\alpha)) = \text{old}(\text{update}(\alpha)) \\
&\wedge\ \text{index}(\text{new}(\alpha)) = \text{output}(\text{bump}(\alpha)) \\
&\wedge\ \text{base}(\text{new}(\alpha)) = \text{output}(\text{update}(\alpha))\ ]
\end{aligned}
$$

This formula is stored and manipulated in a predicate calculus utility package [4] which provides certain simple forms of reasoning. The basic idea of the hybrid implementation is to use the facilities of the predicate calculus package for those types of reasoning which, after the translation, are easily expressible in the language of predicate calculus.

For example, one kind of reasoning that the Programmer's Apprentice needs to do is to propagate preconditions and postconditions (part of the constraints of operation types) along data flow arrows to see if they are consistent. Given that the inputs and outputs of operations become function terms like output(bump(α)) above, and given that data flow arrows become equalities between such terms, this kind of propagation is implemented by substitution of equals in the predicate calculus. For example, if the postcondition of Add1 is as shown below,

$$\text{add1}(\alpha) \Rightarrow [\ \text{output}(\alpha) = \text{input}(\alpha) + 1\ ]$$

then one can conclude by this reasoning that the Index of the New indexed-array is one more than the Index of the Old indexed-array. Furthermore, the predicate calculus reasoning utility package which we use performs deductions based on substitution of equals very efficiently. (It also performs these deductions incrementally with retraction, which allows us to easily explore the effects of adding and removing data flow arrows.)

3. This presentation suppresses details of the formalization having to do with mutable data structures. See [6,7] for a more complete treatment.

Another attractive feature of performing some deductions in the predicate calculus translation is that different plan diagram constructs which are translated into similar predicate calculus forms can share the implementation of some of the reasoning that is required with them. For example, as mentioned earlier, both data flow arrows and overlay correspondences become equalities. The following is part of the predicate calculus translation of the overlay in Figure 1 (overlays become functions from n-tuples to n-tuples).

$$\beta = \text{bump-and-update-as-push}(\alpha) \Rightarrow$$
$$[\ \text{old}(\beta) = \text{indexed-array-as-stack}(\text{old}(\alpha))$$
$$\wedge\ \text{input}(\beta) = \text{input}(\text{update}(\alpha))$$
$$\wedge\ \text{new}(\beta) = \text{indexed-array-as-stack}(\text{new}(\alpha))\ ]$$

Given this formalization, substitution of equals also serves to map information between corresponding plans in an overlay. For example, anything that is asserted to be true of the Input of the Update of a Bump-and-update plan automatically becomes true of the Input of the corresponding Push, and vice versa.

We have also implemented a general mechanism in the predicate calculus utility package which makes deductions based on the domain and range of functions. Given that both roles and overlays become functions, this single mechanism takes care of both enforcing type restrictions on the roles of a plan and asserting the appropriate abstract plan type corresponding to an implementation plan in an overlay. For example, asserting that the bump function has domain bump-and-update and range add1 causes the following deduction to occur.

$$\text{bump-and-update}(\alpha) \Rightarrow \text{add1}(\text{bump}(\alpha))$$

Similarly, asserting that the bump-and-update-as-push function has domain bump-and-update and range push causes the following deduction to occur by the same mechanism.

$$\text{bump-and-update}(\alpha) \Rightarrow \text{push}(\text{bump-and-update-as-push}(\alpha))$$

Other kinds of reasoning used in the Programmer's Apprentice are *not* easy to express as direct manipulations on predicate calculus formulae. Therefore we explicitly store the original plan diagram as extra-logical annotation of the predicate calculus translation. For example, as far as the predicate calculus reasoning mechanisms are concerned, both bump and bump-and-update-as-push are just function symbols. However some reasoning procedures need to treat these two symbols very differently because one names a role and the other an overlay. The best way to think of these procedures is as operating on plan diagrams.

For example, a powerful method of program analysis used by the Programmer's Apprentice is analysis by inspection. The key step in analysis by inspection is to recognize familiar patterns in a program. This recognition is achieved in the Programmer's Apprentice by translating a program into a plan diagram and then trying to match it against a library of standard plan diagrams. This matching algorithm [1] is most naturally written in the language of plan diagrams, not in terms of the predicate calculus formulae directly.[4] What I mean here by "natural" is: first, I doubt whether one could ever discover the right matching algorithm by thinking in terms of the predicate calculus formulae; and second, although it is theoretically possible to implement a plan diagram matching algorithm operating directly on the predicate calculus formulae, this approach does not lead to a convenient or an efficient implementation.

Similarly, an important part of program synthesis reasoning in the Programmer's Apprentice requires breadth-first traversal of a program's plan diagram (e.g. the tree shown in Figure 2), looking for the next implementation decision to make. This procedure is also most naturally written in the language of plan diagrams.

Discussion

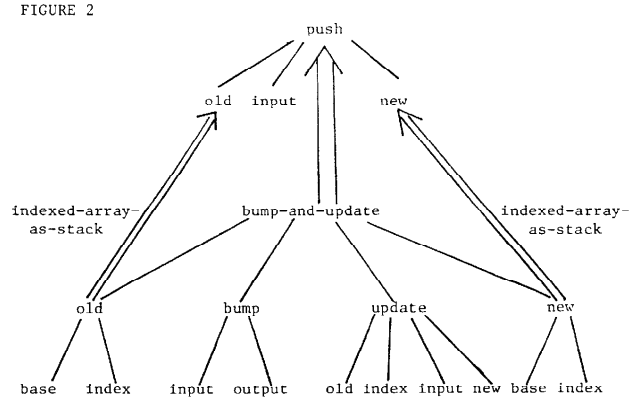
Before going any further it is crucial to understand that this paper is about pragmatic rather than philosophical issues in knowledge representation. What we have developed via our experience with plan diagrams and predicate calculus is a new methodology for implementing intelligent knowledge based systems. This methodology has significant bearing on how to structure the necessary reasoning processes in such systems;[5] it does not however have anything to say about the important questions of the meaning and expressive power of knowledge representations (since we are only talking about knowledge representations which can be translated into predicate calculus).

We can summarize the methodology as follows: Use a hybrid system in which an ad hoc knowledge representation language co-exists with a predicate calculus translation. At the predicate calculus level, provide as much reasoning power as can be naturally expressed and efficiently implemented in the language of predicate calculus. At the higher level, implement those reasoning procedures which naturally exploit the structure of the ad hoc knowledge representation language. Finally, provide explicit connections between the two levels so that changes at one level can be incrementally translated to the other.

4. Note however, that once a match has been found, it is recorded in the the predicate calculus utility package as an assertion of the plan type predicate. Thus from point of view of the predicate calculus utility package, the matching algorithm is a derived implication.

5. It would be nice if we had precise ways of talking about the structure of reasoning processes (and hopefully we eventually will). For the meantime, however, the reader will have to be satisfied by examples and weak arguments about "naturalness."

FIGURE 2



This hybrid approach resolves some of the controversy [3] between advocates of predicate calculus and the users of other knowledge representation languages by demonstrating that you can have the key features of both. A typical argument made by advocates of predicate calculus is that a given knowledge representation language is not interesting because it can be translated into ("is a notational variant of") predicate calculus. This argument misses the important practical issue, namely what happens if you actually try to use the predicate calculus translation to implement the task for which the original knowledge representation language was designed. What happens is that, in order to write effective reasoning procedures, you end up reinventing essentially the same knowledge representation language as an ad hoc set of conventions and annotations on top of the predicate calculus.

On the other hand, ad hoc knowledge representation languages by themselves typically are not designed to facilitate the small, simple deductions, such as implications and substitutions, which are often needed to mediate between their associated special purpose reasoning procedures. These small, simple deductions are just the kind of deductions for which existing predicate calculus machinery is very effective.

Finally, it is interesting to note that one can arrive at this methodology from either of two directions. In our experience, we started with an ad hoc language and reasoning procedures and then added the predicate calculus level for semantic clarity. I could also imagine starting with predicate calculus as the language of expression and then developing a higher level language as the set of conventions and annotations required to write effective reasoning procedures. In either case, the important conclusion is that both levels of language are useful for building practical systems.

Acknowledgements

Many of the ideas in this paper were developed in collaboration with Dan Brotsky. Also, we would probably never have tried to do things this way if not for David McAllester's Reasoning Utility Package.

References

[1] D. Brotsky, "Program Understanding Through Cliche Recognition", (M.S. Proposal), MIT/AI/WP-224, December, 1981.

[2] R. Fikes and G. Hendrix, "A Network-Based Knowledge Representation and its Natural Deduction System", *Proc. of 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, pp. 235-246.

[3] P. Hayes, "In Defence of Logic", *Proc. of 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, pp. 559-565.

[4] D.A. McAllester, "Reasoning Utility Package User's Manual", MIT/AIM-667, April, 1982.

[5] C. Rich, H.E. Shrobe, and R.C. Waters, "An Overview of the Programmer's Apprentice", *Proc. of 6th Int. Joint Conf. on Artificial Intelligence*, Tokyo, Japan, August, 1979.

[6] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), December, 1980.

[7] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice", *Proc. of 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, Canada, August, 1981.

[8] C. Rich and R.C. Waters, "Abstraction, Inspection and Debugging in Programming", MIT/AIM-634, June, 1981.

[9] H.E. Shrobe, "Explicit Control of Reasoning in the Programmer's Apprentice", *Proc. of 4th Int. Conf. on Automated Deduction*, February, 1979.

[10] H.E. Shrobe, "Dependency Directed Reasoning for Complex Program Understanding", (Ph.D. Thesis), MIT/AI/TR-503, April, 1979.

[11] R.C. Waters, "Automatic Analysis of the Logical Structure of Programs", MIT/AI/TR-492, (Ph.D. Thesis), December, 1978.

[12] R.C. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Eng.*, Vol. SE-5, No. 3, May 1979, pp. 237-247.

[13] W. Woods, "What's in a Link", *Representation and Understanding*, Academic Press, 1975.