Technical Report 347

Reasoning from Incomplete Knowledge in a Procedural Deduction System

Robert Carter Moore

MIT Artificial Intelligence Laboratory

REASONING FROM INCOMPLETE KNOWLEDGE IN A PROCEDURAL DEDUCTION SYSTEM

Robert Carter Moore

Massachusetts Institute of Technology

December 1975

Revised version of a thesis submitted to the Department of Electrical Engineering on October 7, 1975, in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

One very useful idea in AI research has been the notion of an explicit model of a problem situation. Procedural deduction languages, such as PLANNER, have been valuable tools for building these models. But PLANNER and its relatives are very limited in their ability to describe situations which are only partially specified. This thesis explores methods of increasing the ability of procedural deduction systems to deal with incomplete knowledge. The thesis examines in detail, problems involving negation, implication, disjunction, quantification, and equality. Control structure issues and the problem of modelling change under incomplete knowledge are also considered. Extensive comparisons are also made with systems for mechanical theorem proving.

THESIS SUPERVISOR: Gerald Jay Sussman

TITLE: Assistant Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I would like to to thank the following people for their very important contributions to this work:

Gerry Sussman, for his advice and support during the writing of the thesis.

My colleagues at the MIT AI Lab, especially Drew McDermott, for many valuable discussions of the issues raised in my research.

Mitch Marcus, Candy Bullwinkle, Drew McDermott, and Ben Kuipers, for reading and criticizing earlier versions of this report.

Suzin Jabari, for assisting in the final preparation of this report.

Most importantly, my wife Rita, for typing most of the thesis and putting up with everything for so long.

TABLE OF CONTENTS

1. Introduction

1.1 The Other Problem with PLANNER

One useful idea that has come out of AI research seems to be the notion of an explicit model of a problem situation. A model of this kind might record that a certain block is in a certain place, or that Black's knight is threatening to win White's king pawn. A problem solver can ask questions of the model in order to plan its actions and test their effects.

In early programs, these models either represented all their knowledge directly, or derived it by predetermined computations. In more complex situations, however, it is not possible to anticipate every question which might be asked. Truly intelligent programs have to be able to derive new knowledge by combining old knowledge in unexpected ways. In other words, they must be able to reason.

There are several approaches to constructing models which might be called reasoning programs. The newest of these, the "frame" approach <Minsky 1975>, is still in a preliminary stage of development. At this point it is difficult to obtain agreement on what a frame is, and even harder to evaluate its capabilities. Another approach is logical theorem proving, particularly resolution-based theorem proving <Chang and Lee 1973>. This approach suffers from at least two major problems. First, the research effort has been concentrated on proving theorems in the first-order predicate-calculus, a language which is very limited in its expressive power. Second, very little progress has been made in controlling deductions, so that a theorem prover typically wanders off making inferences that have very little to do with the problem that it is supposed to be solving.

Finally, there is "procedural deduction", an approach embodied in PLANNER <Hewitt 1972> and related programming languages. This approach starts with the more promising idea that how a fact should be used is as important as what the fact asserts. As a result of this emphasis, these languages have been given sophisticated control structures which make it possible to write far more efficient reasoning and problem solving programs than those which have been based on uniform proof procedures. But these languages are still very limited in expressive power; in some ways even more limited than first-order predicate calculus. The goal of this thesis is to investigate ways of extending the expressive power of procedural deduction systems.

To understand the claim that existing systems lack expressive power, one must distinguish between the true but uninteresting fact, "Programming languages are universal, so they can express anything;" and the more substantive question of what features are actually available for constructing descriptive models of situations. This thesis explores ways of increasing the ability of procedural deduction systems to express and reason with incomplete models of situations. We will break this down into two subproblems, modelling static situations, and modelling change. The bulk of the thesis is devoted to the first of these subproblems; some preliminary ideas about the second are presented in Appendix A. The following sections discuss the limitations of the PLANNER paradigm in these areas.

1.2 Modelling Static Situations in PLANNER

Procedural deduction systems start from the idea that a situation is modelled by a data base of procedures and symbolic statements. The data base is interrogated by representing questions as procedures which search and manipulate the data base. The basic construct that both questions and statements are built from is some representation of a relationship among individuals. In PLANNER this is done by assertions such as (ON BLOCK1 BLOCK2) where one of the elements is the relation and the remainder are its arguments. These simple structures, however, are insufficient for expressing complex information. A language for building models needs features for combining simple items to make more complex structures.

For this purpose, the predicate calculus supplies logical connectives and quantifiers. Procedural deduction systems have analogous constructs. Micro-Planner <Sussman et. al. 1971> provides THNOT, THOR, and THAND to express negation, disjunction, and conjunction; antecedent and consequent theorems to express implication; and pattern matching to handle quantification. Using these features in a straight-forward way, however, leads to difficulties when dealing with incomplete knowledge.

By incomplete knowledge or an incomplete model of a situation, we mean that there are questions within the realm of discourse of the model whose answers are unknown. Models constructed using PLANNER-like languages have almost invariably been complete in this sense. In SHRDLU <Winograd 1971> for example, there are no blocks which the robot does not know about, and the robot knows the color, shape, size, and location of every block. While the robot does not know, say, the weight of the blocks, this question is really outside the domain of the program. So, within its domain, SHRDLU is omniscient.

Now let us consider the features of Micro-Planner that make it difficult to model incompletely known situations.

Negation

The only negation operator in Micro-Planner is THNOT. (THNOT P) succeeds if and only if P fails. THNOT is typically used in the construct (THNOT (THGOAL P)) to represent (¬ P). Since (THGOAL P) fails whenever P cannot be inferred, rather than when P is known to be false, this ammounts to an assumption that whatever is not known to be true must be false. This is obviously an inference rule which is unreliable in a great many circumstances. For instance, we would not assume that a casual acquaintance was unmarried simply because we did not know that he was married.

Disjunction

Micro-Planner's disjunction operator is THOR. For the expression (THOR P Q) to succeed, either P must suceed or Q must succeed. So to answer yes to the question "Is P or Q true?" one must know that P is definitely true or that Q is definitely true. Knowing only that P or Q is true, but not knowing which, will not do. In fact, Micro-Planner really has no way of representing this state of affairs - asserting (THOR P Q) has no special meaning for the system. One might try looking for (THOR P Q) explicitly asserted, as well as evaluating it, but this fails in many cases. For example, suppose we know that P implies R, Q implies R, and P or Q. If we wanted to prove R, we could use the first fact to generate P as a subgoal and the second to generate Q as a subgoal, but the effort would fail because "P or Q" never explicitly appears as a goal. If a person knows that John owns either a dog or a cat, however, he has no trouble concluding that John owns an animal.

Quantification and Equality

Micro-Planner offers limited quantification by way of variables in its pattern matcher. A variable will match anything, so a variabilized pattern of a theorem is universally quantified. So (P $?X) in a theorem pattern means the same as (∀x (P x)). In a goal this matching convention gives variables an existential interpretation, so (THPROG (X) (THGOAL (P $?X))) means "Prove $(\exists x\ (P\ x))$".

Existential assertions and universal goals, however, are problematical. There is really no way at all of making an existential assertion like "Something is P," without specifying which thing is P. There is no operator specifically for universal goals, but in SHRDLU goals of the form "Are all P's Q?" are approximated by the expression:

```
(THNOT (THPROG (X) (THGOAL (P $?X))
                   (THNOT (THGOAL (Q $?X)))))
```

This procedure looks at all the known P's, trying to find one that is not known to be a Q. If there is no such P, then all known P's are Q and the entire expression succeeds, due to the outermost THNOT. This corresponds to replacing $(\forall x\ ((P\ x) \supset (Q\ x)))$ by $(\neg\ \exists x\ ((P\ x) \wedge (\neg\ (q\ x))))$, but because of the use of THNOT, the quantification is only over known P's. This is insufficient in many cases. If a middle-class American were asked "Do all people have telephones?" the answer would probably be no, but the person might not be able to name any. Even in situations like SHRDLU's blocks world where this procedure is reliable, it can be very inefficient. It does not permit taking direct advantage of a universal assertion to answer a universal question. Even if we had a theorem which contained the information that all cubes are blocks, to answer the question "Are all cubes blocks?" would require enumerating all cubes and using the theorem repeatedly to verify that each is a block.

Finally, the only equality operators Micro-Planner has are LISP equality operators. This makes it difficult to model situations where there is not one unique name for each individual. If MAN1 is the man we saw on the bridge yesterday and MAN2 is the man who was found in the river today, we might want to consider whether MAN1 = MAN2.

Although Micro-Planner was one of the earliest procedural deduction languages, its limitations in these respects are shared by its successors. CONNIVER <Sussman and McDermott 1972>, QA4 <Rulifson et. al. 1972>, and POPLER <Davies 1973> have made advances in control structure, data types, and multiple data bases, but they share PLANNER's biases towards requiring complete knowledge of situations being modelled. There have been suggestions as to how some of these problems might be handled (which we will consider in subsequent sections), but no systematic study has been made of these problems as a group.

1.3 Modelling Change in PLANNER

The previous section discussed the problems that arise when incomplete knowledge is introduced into PLANNER-type reasoning about static situations. We now want to consider what happens when incomplete knowledge is introduced into PLANNER's reasoning about change.

PLANNER-type systems handle change by creating a series of data bases, each being a "snap-shot" of the process being described. The typical PLANNER data base consists of theorems (procedures) which are always present, and simple assertions which may be asserted or erased as the result of performing an action. It is fairly straightforward to write procedures for each action which assert and erase the appropriate statements. The STRIPS problem solving system <Fikes and Nilsson 1971> has a similar, though less flexible, mechanism in its add and delete lists.

Several new difficulties come up when incomplete knowledge is allowed. First, instead of simple assertions, we may have arbitrarily complex pieces of knowledge which must be dealt with by the asserting and erasing procedures. For instance, suppose we are told that all pyramids in the box have been moved to the table. If we have a complete PLANNER-type model, we simply check each pyramid to see if it is in the box, and if so, erase the statement that it is in the box, and assert that it is on the table.

In an incomplete model we might not know very much about particular pyramids, but know instead "All yellow pyramids are in the box." The procedures associated with the action should erase this statement, and leave us in a situation where we can deduce "All yellow pyramids are on the table." What we need are systematic methods for dealing with complex assertions, given the effects of an action on simple assertions.

Another problem is that many actions have conditional effects: moving A from P1 to P2 will also move B from P1 to P2 if and only if B is on top of A. In complete models conditions such as this can always be evaluated; in incomplete models they cannot. In the above example, if we know that B is at P1 but not whether B is on A, then following the action we would like to know that B is at P1 if B was not on A, and B is at P2 if B was on A.

Finally, we would like to consider situations where performing an action results in a loss of information; that is, situations where the action itself is non-deterministic or incompletely described. If A and B are in the box, and we are told that one block has been removed, then we no longer know if A is in the box or if B is in the box. We do still know however, that either A or B is in the box.

It seems worth noting two general observations about the extensions to the PLANNER paradigm that will be required to handle examples like these. First, it will be necessary to incorporate names of world states into assertions. This is because sometimes our knowledge about the results of an action will be statable only as a relation between the state before an action and the state after the action. (Recall "B is at P1 if it was not on A.") In PLANNER and its relatives, every statement must be made completely within a data base representing a state of the world. Second, it will be necessary to examine and manipulate complex assertions. In PLANNER-like systems it is difficult to do anything except execute them as procedures. So to this extent the system must be less procedural and more declarative.

1.4 Other Work

Besides the procedural deduction systems discussed above, there are two other research efforts which are related to the question of reasoning from incomplete knowledge. The first is the work of Collins's group <Carbonell and Collins 1973> <Collins et. al. 1975>. They also deal with incompletely described situations ("open worlds" in their terms), but they attack a somewhat different problem. Rather than trying to make logically correct deductions from incomplete knowledge, they explore the use of heuristics that "fill in the gaps" in that knowledge. One of their heuristics is to classify types of assertions according to their importance. Then they assume that if you know facts of relatively little importance about a certain topic, then any assertions of significantly greater importance which are not stored are probably false. For example, in their domain of South American geography, they consider the question "Does Uruguay have any oil?". The answer to this question is not stored, so the system looks around its data base for similar assertions. It finds that Venezuela has oil and that this fact is very important. Since the system knows facts about Uruguay that are much less important, it assumes that if Uruguay did have oil, it would know this. So the system replies "No". Collins's system, however, does not deal with any of the questions we have raised such as handling disjunction, quantification, equality, etc; so there is actually very little overlap between this research and ours.

The other relevant research effort is Kowalski's <Kowalski 1973, 1974>. Kowalski has explored applying PLANNER-like control structure directly to sets of predicate calculus axioms. Since predicate calculus has representations for many of the things PLANNER doesn't handle, this might seem to be an attractive approach. Kowalski develops his theory, however, only for sets of Horn clauses. In the terminology of resolution theorem proving, a clause is a disjunction of literals, where a literal is an atomic formula (positive literal) or its negation (negative literal). So a typical clause might be $((P\ x) \vee (\neg\ (Q\ y)) \vee (R\ x\ y))$. There is a theorem of logic that any set of axioms can be turned into clausal form.

A Horn clause is a clause that contains <u>at most</u> one positive literal. Kowalski divides Horn clauses into three types. Clauses with one positive literal and no negative literals, are treated as simple assertions. Clauses which have a positive literal and some negative literals (which can be written as $A_1 \wedge \ldots \wedge A_n \rightarrow B$) are treated like PLANNER consequent theorems. Clauses with no positive literals are treated as goals. Since resolution is a refutation-based system, $((\neg\ A) \vee (\neg\ B) \vee (\neg\ C))$ can be viewed as (GOAL $(A \wedge B \wedge C)$).

This system, however, is even more restricted than PLANNER. We cannot assert a disjunction, and we do not have even THNOT-style negation or antecedent theorems. Kowalski points out these difficiencies with examples much like ours, but he seems to advocate a general purpose resolution theorem prover as the solution. We will attempt to expain later why this is not sufficient.

1.5 A Scenario

In order to put the preceding discussion in perspective, we offer the following blocks-world scenario. The reasoning necessary to answer these questions involves incomplete knowledge which systems like SHRDLU and HACKER <Sussman 1973> cannot handle.

The situation is a typical blocks-world scene having a table, a box, and some blocks. The robot is blind, so that it knows only those things about the scene which we tell it. The operation "grab" involves reaching into the box and taking out a block, but without the robot knowing which block he has grabbed. We first describe the scene to the robot.

1. A and B are green blocks in the box.
2. C is a block on the table.
3. I like one of the blocks in the scene.
4. A, B, and C are the only blocks in the scene.

The robot has to be able to represent that I like one of the blocks without knowing which one. It also has to realize that just because it does not know what color C is, it does not mean C has no color.

5. Are all the blocks in the box green? YES.

The robot must reason that any block in the box is in the scene, and so must be either A, B, or C. It then must reason by cases that it could not be C, because C is on the table and therefore is not in the box, and that if it is A or B then it is green. This question involves proving a universal and reasoning about disjunction.

6. Are all the blocks in the scene green? I DON'T KNOW.

This also involves reasoning by cases, but one of the cases, "Is C green?" cannot be answered. The robot has to know that this does not make it false.

7. Do I like any block in the box? I DON'T KNOW.

The robot can neither prove nor disprove that the block I like is in the box.

8. Is any block on the table on top of a block? NO.

Since there is only one block on the table, C, it cannot be on top of another block.

9. Put C in the box. OK.

10. Do I like any block in the box? YES.

The robot still does not know which block I like, but now all the possibilities are in the box.

11. Is any block on the table? NO.

12. Is every block in the box green? I DON'T KNOW.

13. Is at least every block in the box but one green? YES.

12. and 13. illustrate how complicated reasoning about change can become in the absence of total knowledge about the situation.

14. Grab a block from the box and put it on the table. OK.

15. Is there a green block on the table? I DON'T KNOW.

16. Is there a green block in the box? YES.

17. Is A in the box? I DON'T KNOW.

18. Is B in the box? I DON'T KNOW.

19. Is A or B in the box? YES.

These are more complications in reasoning about change, created here by the fact that "grab" results in loss of information.

20. Grab a block from the box and put it on the table. OK.

21. Is there a green block in the box? I DON'T KNOW.

22. Is there a green block on the table? YES.

2. Negation and Implication

2.1 Procedural Interpretations of Declaratives

A first step in reasoning about negation explicitly is to represent negation explicitly - like (NOT A). The question is how to satisfy goals containing negatives and how to use assertions containing negatives. We can simplify the problem by deciding to "push through" negations to the level of atomic statements. Thus (NOT (OR A B)) will become (AND (NOT A) (NOT B)), (NOT (FOR-ALL ?X (FOO ?X))) will become (EXISTS ?X (NOT (FOO ?X))), (NOT (NOT A)) will become A, and so forth.

Now the problem is reduced to how to prove negated atomic goals and how to use negated atomic assertions. The key issue is how negatives interact with antecedent and consequent theorems. Hewitt has made the point that an implication (A ⊃ B) has at least two possible uses:

1. To prove B, try to prove A.
2. If A is asserted, assert B.

If (A ⊃ B) is true, however, ((NOT B) ⊃ (NOT A)) is also true, and this fact has the corresponding uses:

3. To prove (NOT A), try to prove (NOT B).
4. If (NOT B) is asserted, assert (NOT A).

At this point we could simply say to use whatever combination of these four procedures is needed; but only three of the sixteen possible combinations seem to be really useful. It would rarely be useful to have both 1. and 2. since they perform the same inference; similarly for 3. and 4. Also whenever we have one of the positive inferences (1. or 2.), we would probably want to have one of the negative inferences (3. or 4.). If we know that all cubes are blue, and that A is green, it is trivial to see that A is not a cube. Presumably, this is done by inferring that since A is green it is not blue, and since it is not blue, it is not a cube. So although we have been given a fact in the form "All cubes are blue," we have no trouble using it in the form "Anything that is not blue is not a cube."

This eliminates all but four of the possible combinations. For the remaining useful ones, we will introduce some special notation. We will write the combination of 1. and 4. as (B <- A) and the combination of 2. and 3. as (A -> B). The idea is that the arrow expresses the direction of the inference, and the left-hand subexpression is the pattern which invokes the procedure. We do not really need both -> and <-, since (A -> B) is equivalent to ((NOT A) <- (NOT B)), but it seems convenient to have both.

As an example of a fact that might be used in this way, consider "All cats are animals". This fact can be used to infer that something is an animal if it is a cat, or that something is not a cat if it is not an animal. I would argue that the procedures to do this should be:

If "x is a cat" is asserted, assert "x is an animal".

To prove "x is not a cat", try to prove "x is not an animal".

Formally, this is expressed by:

```
((CAT ?X) -> (ANIMAL ?X))
```

or equivalently:

```
((NOT (CAT ?X)) <- (NOT (ANIMAL ?X)))
```

To see that this is the preferred form, consider how "Felix is an animal" could be inferred from "Felix is a cat". If this type of inference were done in a consequent-directed manner, it would result in a search through all the known types of animals, until "cat" was stumbled upon. The antecedent-directed method requires only that an "animal" assertion be made whenever a "cat" assertion is made. By analogy, this strategy would also require assertions for other important supersets - perhaps "living thing", "physical object", and a few others - but surely the number of these is far smaller than the number of different types of animals.

In inferring "Truck37 is not a cat," by showing "Truck37 is not an animal", similar considerations suggest that a consequent-directed strategy be followed. It would be far too costly to add assertions for all the types of animals that something is <u>not</u> when it is asserted to be a non-animal. Searching through a few important supersets seems much easier.

The third useful combination of procedures is 1. and 3. We will write this as (OR (NOT A) B). We will show later that this is consistent with the other uses of OR. We can extend this to (OR $A_1 \ldots A_n$), where any of the A_i can be proved by showing that all the others are false. Note that since any component of an OR assertion can act as a pattern, each must be either an atomic statement or its negation. The non-pattern part of an implication can, of course, be non-atomic. As an example, we might want to have:

```
(OR (NOT (PLANT ?X)) (NOT (ANIMAL ?X)))
```

This says that to prove something is not a plant, try to prove it is an animal, and to prove something is not an animal, try to prove it is a plant.

This combination would also be useful for expressing assertions that are "just facts" and have no systematic connections to our knowledge structure, such as "John owns either a dog or a cat." Since we have no *a priori* reason for treating the two components of such an assertion differently, it seems appropriate to use a construct like our OR, which treats them symmetrically.

The last remaining combination, 2. and 4., does not seem to be very useful. the reasons for this are somewhat complex. The argument turns on the fact that 1. and 4. while expressing different inferences, perform identical or at least isomorphic computations. To see this, compare trying to prove A with asserting (NOT A), given (A <- B). In the first case we have (GOAL A) generating (GOAL B). In the second we have (NOT A) generating (NOT B). In a direct proof procedure, we succeed if (GOAL A) matches an assertion A. In an indirect proof procedure we succeed if (NOT A) contradicts an assertion A. Computationally, all we have done is substitute the symbol NOT for the symbol GOAL!

This parallel extends to AND and OR. If we have (GOAL (AND A B)) we have to match this against both A and B. If we want to prove (AND A B) indirectly, we negate it, getting (OR (NOT A) (NOT B)). To contradict this we have to match it against both A and B. For (GOAL (OR A B)) to succeed we need to match it against either A or B. Negating this we get (NOT A) and (NOT B), so that either A or B will lead to a contradiction. In each case the behavior is the same. All we have done is substitute NOT for GOAL and exchanged AND and OR. For a fuller and more formal development of these ideas, see <Kowalski 1973, 1974>.

These observations have some very interesting consequences. First and foremost, <u>there is no essential computational difference between direct proof methods and proof by contradiction</u>. It is not the case, as is sometimes argued (e.g. <Winograd, p.215>), that refutation procedures are inherently "bottom-up" rather than "goal-oriented" or "top-down". For a refutation-based procedure to be goal-oriented, it needs only to concentrate on contradicting the negation of the goal and its descendents. This is, in fact, just the set of support heuristic of resolution theorem proving <Chang and Lee, Chapter 6>.

Now let us return to the original question of whether 2. and 4. form a useful combination of procedures. If as we have argued, 1. and 4. are really the same computaion, then it is foolish to use 4. without also using 1. To use 4. without 1. would be to say "I will do this computation if it is not needed to satisfy an immediate goal (i.e. antecedently), but I will not use it if it is needed to satisfy an immediate goal (i.e. consequently)." A similar argument applies to 3. and 2. What we are saying is that we should be willing to do at least as much computation to satisfy a goal as to explore the consequences of an assertion, but the 2.-4. combination of procedures does the opposite of this.

It is important to realize that we are not claiming that top-down strategies are always better than bottom-up. We have already given examples to the contrary. What we are saying is that due to the semantics of implication, there turn out to be two distinct inferences, that can be implemented by isomorphic computations. In that case, if the antecedent implication is efficient and desirable, then surely the corresponding consequent implication is as well.

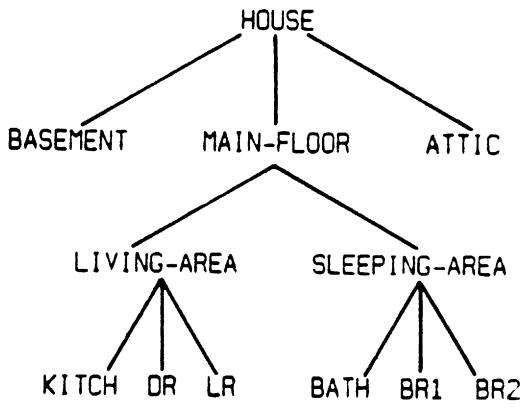
2.2 Hierarchies - An Example of Structuring Knowledge

In the preceding section we argued that the basic computational operations of a procedural deduction system are really the same as a refutation-based theorem prover (including resolution theorem provers). Is there, then, no difference at all? The answer is that, yes, there is a difference; and that difference is primarily in the control of the deductive process. We will discuss control structure issues in more detail later, but we have already developed enough structure to illustrate some of the short-comings of uniform proof procedures in this regard.

We argued above that of all the possible combinations of the four procedural interpretations of $(A \supset B)$, only two or three (depending on whether -> and <- are regarded as distinct) are useful. Virtually all resolution systems force every formula to be used according to the same procedural interpretations, regardless of whether those interpretations are appropriate for that formula. General, or unrestricted, resolution treats every formula as having all four procedural interpretations. (In the case of formulas more complex than $(A \supset B)$ there can be many more than four!) This is clearly redundant, since it treats every implication as both an antecedent and consequent-driven procedure.

More sophisticated resolution systems avoid this redundancy, but at the cost of extreme rigidity in the procedural interpretation of formulas. Most can be viewed as either totally antecedent-driven (bottom-up) or totally consequent-driven (top-down). Kowalski <1974, p. 28> gives an enumeration of which systems fall into which class. We have already given examples which demonstrate the inefficiencies of this sort of rigidity, but a more elaborate case may serve to drive home the point.

The problem of hierarchies provides a very nice domain to illustrate the various procedural interpretations of implications. Hierarchies have received some attention in AI lately <Fahlman 1975> <Grossman 1975>, as being a type of structure which should be very easy to reason about. A typical hierarchy might be the following:



This is a space hierarchy. Any object which is in one of the locations represented by a node in this tree is also in every location represented by a superior node, and is not in any locations represented by other nodes. This implies that if an object is not in the location represented by a particular node then it is not in any locations represented by inferior nodes. We will not assume this hierarchy is exhaustive; we might have forgotten to mention the garage, for example. We can efficiently implement reasoning about this hierarchy by the following assertions:

```
(INCOMP (IN ?X BASEMENT) (IN ?X MAIN-FLOOR) (IN ?X ATTIC))
((IN ?X BASEMENT) -> (IN ?X HOUSE))
((IN ?X ATTIC) -> (IN ?X HOUSE))
((IN ?X MAIN-FLOOR) -> (IN ?X HOUSE))
     (INCOMP (IN ?X LIVING-AREA) (IN ?X SLEEPING-AREA))
     ((IN ?X LIVING-AREA) -> (IN ?X MAIN-FLOOR))
          (INCOMP (IN ?X KITCH) (IN ?X DR) (IN ?X LR))
          ((IN ?X KITCH) -> (IN ?X LIVING-AREA))
          ((IN ?X DR) -> (IN ?X LIVING-AREA))
          ((IN ?X LR) -> (IN ?X LIVING-AREA))
     ((IN ?X SLEEPING-AREA) -> (IN ?X MAIN-FLOOR))
          (INCOMP (IN ?X BATH) (IN ?X BR1) (IN ?X BR2))
          ((IN ?X BATH) -> (IN ?X SLEEPING-AREA))
          ((IN ?X BR1) -> (IN ?X SLEEPING-AREA))
```

((IN ?X BR2) -> (IN ?X SLEEPING-AREA))

(INCOMP $A_1 \ldots A_n$) is an expression whose logical interpretation is that the arguments of INCOMP are pair-wise incompatible, and whose procedural interpretation is the same as (AND (OR (NOT A_1) (NOT A_2)) (OR (NOT A_1) (NOT A_3))...(OR (NOT A_{n-1}) (NOT A_n))).

These assertions form a highly structured set of procedures for using this hierarchy. Checking our definitions for -> and OR will verify that we have implemented the following algorithm: When asserting that A is in B, assert that A is in all the superiors of B. To deduce that A is in B look only for that explicit assertion. To deduce that A is not in B, prove that A is in a brother of B or (recursing) that A is not in the immediate superior of B. When asserting A is not in B, make no further assertions.

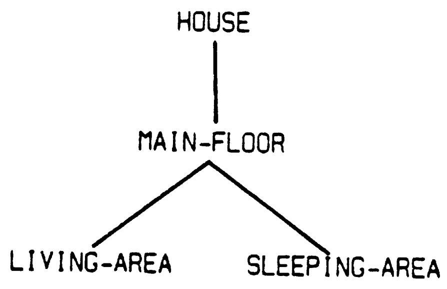
To analyze the efficiency of this algorithm, consider the case of a balanced hierarchy of depth m and branching factor n. Proving A is in B and asserting A is not in B each require only one step. Asserting A is in B requires at most m steps, one step for each superior of B. Proving that A is not in B can take up to m*n steps. In the worst case, we must check whether A is in any of the n-1 brothers of B and whether A is not in the immediate superior of B. This may be repeated m times for all the superiors of B.

Compare this with either a totally bottom-up or totally top-down strategy. For the bottom-up strategy, any proof is one step, but asserting A is in B can require m^n steps if B is at the bottom of the hierarchy. We would get assertions that A is in every superior of B and assertions that A is not in the places represented by any other node. Similarly, if B is the top node of the hierarchy, asserting that A is not in B will take m^n steps generating assertions that A is not in any node of the hierarchy.

The top down strategy is just as bad. Here, assertions require only one step, but proofs are exponentially expensive. If B is the top node of the tree, proving A is in B may require checking every node of the tree. If B is a leaf node, proving A is not in B may require checking every node. So the mixed antecedent/consequent strategy is linear in the worst case, where both the other strategies can be exponentially explosive.

Not all theorem provers are rigidly top-down cr rigidly bottom-up. Nevins's theorem prover <Nevins 1974a> and Kowalski's connection graph theorem prover <Kowalski 1974> are examples. But these, and virtually all other theorem provers which have some flexibility, rely on strictly local criteria for deciding to do an inference in a consequent or antecedent direction. For the connection graph theorem prover, Kowalski recommends the heuristic of picking the formula with the fewest connections to other formulas to do inferences from. If this formula is a goal, the system will be acting in a consequent-directed manner. If this formula is an assertion, the system will be antecedent-directed.

It is easy to show that local heuristics like this are not good enough, however. Consider some of the connections to one of the nodes of our space hierarchy, MAIN-FLOOR:



The assertions which represent these connections are:

```
((IN ?X MAIN-FLOOR) -> (IN ?X HOUSE))
((IN ?X LIVING-AREA) -> (IN ?X MAIN-FLOOR))
((IN ?X SLEEPING-AREA) -> (IN ?X MAIN-FLOOR))
```

Why did we express these inferences in this direction? Was it because of any local features of the facts about MAIN-FLOOR? The answer is no, it was a global property of the data base - the fact that these assertions were embedded in a downward-branching hierarchy - that led us to express them this way. If the overall structure had been upward-branching, relative to the piece of substructure illustrated, we would have written these assertions as:

```
((IN ?X HOUSE) <- (IN ?X MAIN-FLOOR))
((IN ?X MAIN-FLOOR) <- (IN ?X LIVING-AREA))
((IN ?X MAIN-FLOOR) <- (IN ?X SLEEPING-AREA))
```

Failure to adapt these assertions to the global nature of the structure leads to the exponential inefficiencies previously discussed, but no local heuristics could possibly hope to be sensitive to that global structure. Kowalski acknowledges that local heuristics are not generally sufficient, and advocates use of an auxiliary control language as the solution to the control problem. He cites <Hayes 1973> as presenting convincing arguments for this approach. Hayes's principal argument is that an effective deductive system has both logical and procedural aspects, and the semantics of the system are made clearer by separating the two.

This may be true in some cases, but in regard to the antecedent/consequent distinction there are over-riding pragmatic reasons for combining the two types of knowledge. First, if the information about how an assertion is to be used is not encoded in the assertion itself, then the assertion will have to be indexed according to all its possible uses, even those that are ruled out on pragmatic grounds. So, assertions such as:

```
((CAT ?X) -> (ANIMAL ?X))
((DOG ?X) -> (ANIMAL ?X))
•
•
•
```

would all have to be indexed under ANIMAL, even though they should not be used in that way. Worse yet, if we wanted to prove (ANIMAL FRED), we might have to retrieve all these useless assertions, and use our separate control information to throw them out. Another alternative, clearly worse, would be to look at every consequent theorem, and check for patterns which match (ANIMAL FRED). Fundamentally, we need to store and access assertions according to their use. This makes separation of pragmatic and logical content impossible.

3. Disjunction

3.1 Three Principles of Disjunctive Reasoning

Our treatment of disjunction is governed by the following three reasoning principles:

1. $(A \vee B) \mid\!- C$ if $A \mid\!- C$ and $\mid\!- (\neg B)$
2. $(A \vee B) \mid\!- C$ if $A \mid\!- C$ and $B \mid\!- C$
3. $\mid\!- C$ if $A \mid\!- C$ and $(\neg A) \mid\!- C$

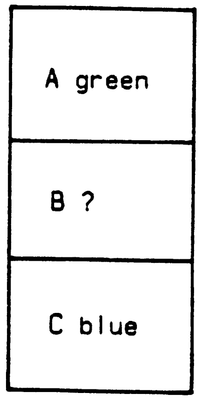
($\mid\!- C$ means "C is provable" and $A \mid\!- C$ means "C is provable given A.") 1. and 2. are the two important possible uses of a disjunctive assertion. 1. is handled by the procedural interpretation we have already given for OR. If we know (OR A B) and we generate A as a sufficient subgoal of C (which establishes $A \mid\!- C$) then we generate the goal (NOT B). We can do this completely within the framework of existing PLANNER-like systems. (OR A B) could be compiled into two consequent theorems:

```
(THCONSE A (THGOAL (NOT B)))
(THCONSE B (THGOAL (NOT A)))
```

2. is the principle of reasoning by cases. This principle is harder to handle; there is no simple compilation for it, as there is for 1. 3. is actually a special case of 2., since $(A \vee (\neg A))$ is always true. 3. seems to be more difficult to handle than 2., however. In 2., we at least have

the explicit assertion (A v B) to deal with; in 3. the (A v (¬ A)) assertion is implicit, and, of course there is an assertion of this form for any possible formula. For this reason, we will deal with 3. first.

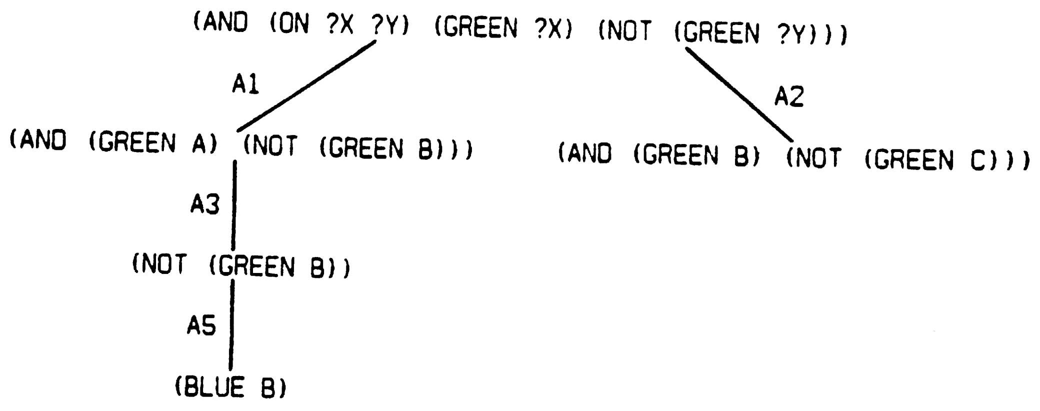
To see the usefulness of the reasoning principle involved in 3., consider the following problem. Suppose we have three blocks stacked A on B on C. A is green, C is blue, and the color of B is unknown:



We want to know whether there is a green block on top of a non-green block. In order to answer this question, we need one additional fact; being green is incompatible with being blue. We can express this by (OR (NOT (GREEN ?X)) (NOT (BLUE ?X))). So our data base has the following assertions:

```
A1. (ON A B)
A2. (ON B C)
A3. (GREEN A)
A4. (BLUE C)
A5. (OR (NOT (GREEN ?X)) (NOT (BLUE ?X)))
```

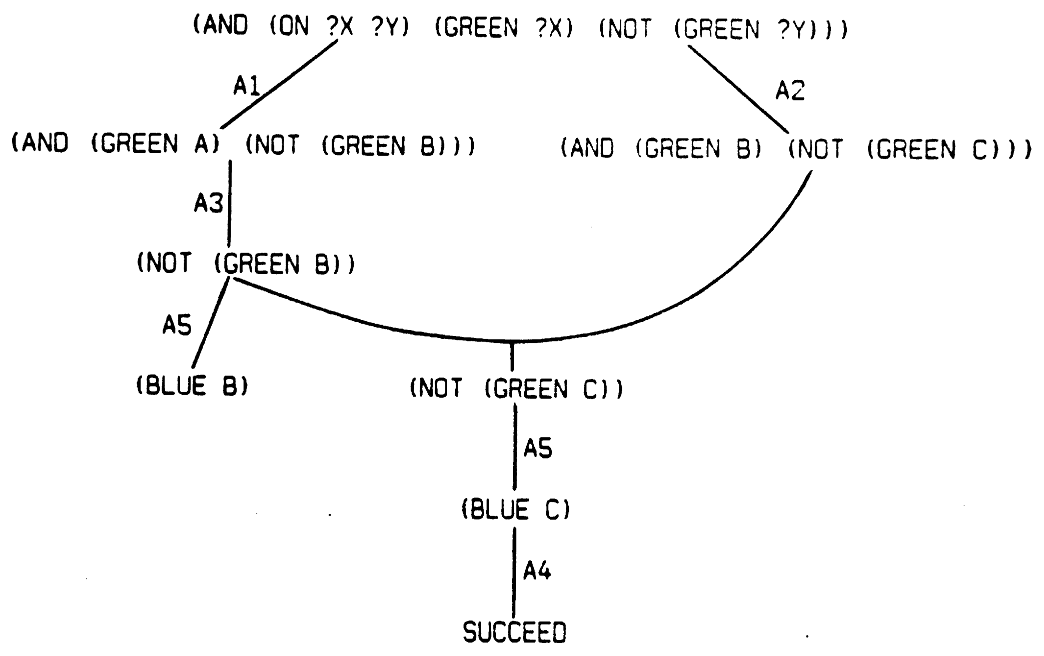
If we tried to satisfy our goal using a PLANNER-like system (given the macro-expansion for OR) we might get the following goal tree:


(AND (ON ?X ?Y) (GREEN ?X) (NOT (GREEN ?Y)))
A1
A2
(AND (GREEN A) (NOT (GREEN B)))
(AND (GREEN B) (NOT (GREEN C)))
A3
(NOT (GREEN B))
A5
(BLUE B)

The problem has been set up so that we look at pairs of blocks such that one is on top of the other, and we try to show that the first one is green, and the second is not green. The first pair we try is A and B. We know that A is green, so we go on to try to show that B is not green. We have one possible way to do this, showing that B is blue. This fails, so we go back and try another pair, B and C. We have no way of showing that B is green, so we fail. Notice, however, that one branch of the tree has the subgoal (NOT (GREEN B)) and another branch has a subgoal that contains (GREEN B). Since one of these must be true, if the other associated conditions are satisfied then we can establish our goal. We can take advantage of this fact by introducing the following rule for combining goals:

> Suppose (AND A $B_1 \ldots B_n$) is a goal and (AND (NOT A') $B_{n+1} \ldots B_m$) is an alternative goal, such that A and A' match. Then we can make (AND $B_1' \ldots B_m'$) a goal, where B_i' is B_i altered by the match between A and A'.

(It is important to note that by a goal, we mean a <u>complete</u> set of conditions sufficient to satisfy a top level request.) Applying this rule to our problem gives us the following goal structure:


(AND (ON ?X ?Y) (GREEN ?X) (NOT (GREEN ?Y)))
A1
A2
(AND (GREEN A) (NOT (GREEN B)))
(AND (GREEN B) (NOT (GREEN C)))
A3
(NOT (GREEN B))
A5
(BLUE B)
(NOT (GREEN C))
A5
(BLUE C)
A4
SUCCEED

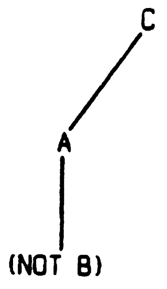
With this new rule, when we generate the goal (AND (GREEN B) (NOT (GREEN C))) we can combine it with (NOT (GREEN B)) to get (NOT (GREEN C)). We can establish this by showing that C is blue.

There are several important observations to make about this rule. First, it represents a major break from PLANNER-type control structure. All the operations in PLANNER are <u>local</u> operations on single goals. This rule represents a <u>global</u> operation on the goal tree. In order to use this rule, we must have explicitly available the failed branches of the goal tree. In PLANNER-like systems, a branch that fails is normally destroyed, so a major change in implementation would be required.

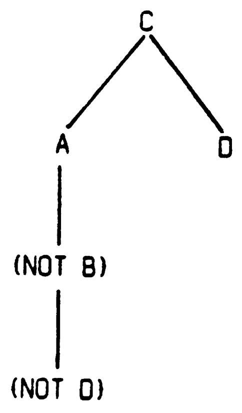
Second, this rule is really an application of the resolution-principle. If we consider how this rule would work in a refutation system, it would mean combining (OR A $B_1 \ldots B_n$) with (OR (NOT A) $B_{n+1} \ldots B_m$) to give (OR $B_1 \ldots B_m$). It is, however, a highly restricted form of resolution. It applies only to pairs of goals, and since it applies only to goals we can restrict it even further - only the left-most components of goals are candidates for matching. The reason for this restriction is quite simple. Suppose we had the goals (AND A C) and (AND B (NOT C)). We could combine them to get (AND A B), but this is unnecessary. In the goal (AND A C), A is the component we are currently trying to solve. Unless we can solve A, however, there is no point in worrying about C, since applying our new rule to C (or any other rule, for that matter) does not get rid of A. A similar argument applies to B. If A and B can be satisfied, we will eventually be left with C and (NOT C) as goals. We can apply our rule at this point to produce the proof we wanted. This is a very strong restriction. If we have goals with m and n components respectively, there are m*n possible ways to combine them. With this restriction, we need only consider one of those ways. We will call this procedure restricted goal resolution (RGR).

Finally, we want to ask whether RGR is necessary. Even with the restrictions we have placed on it, RGR is a very powerful rule. It is especially suspicious because it is entirely syntactic; the semantics of the problem domain do not enter into its application. Despite these reservations, it seems there is no getting around the need for this rule. Look back at our example. In order to solve this problem, it is necessary to consider two cases; either B is green or it is not. Is there any aspect of the problem statement or the domain that would have told us that these were the two cases to consider? If there is, I cannot see it. Rather it appears that it is the topology of the goal tree, produced by this particular question interacting with these particular facts, that made this dichotomy the important one. If this is true, then we can do no better than a rule that looks at the form of the goal trees.

We have presented procedures which implement principles 1. and 3. of disjunctive reasoning. We now turn our attention to principle 2. It turns out that we do not need an additional procedure to handle 2. Suppose in attempting to prove C we generate A as a goal and we have (OR A B) as an assertion. By principle 2., we can establish our goal by showing B |- C. But we must also try principle 1., because B might simply be false. So we need our definition of OR to generate the goal (NOT B). So far our goal tree looks like this:



Now suppose that, indeed B |- C. If this is the case there must be some intermediate D, such that asserting B causes D to be asserted by antecedent processes, and the goal C generates D as a subgoal by consequent processes. D can, of course, be much more complex than an atomic assertion, but the argument should generalize. It can also be the case that D = B or D = C, in which case the path from B to C is either totally antecedent, or totally consequent. Using our procedural interpretations of ->, <-, and OR, whenever there is an antecedent path from B to D, there will also be a consequent path from (NOT B) to (NOT D). So the (NOT B) in our goal tree will eventually generate (NOT D) as a subgoal. If this branch fails, we will eventually get around to generating D as a subgoal of C. This leaves us with the following goal tree:

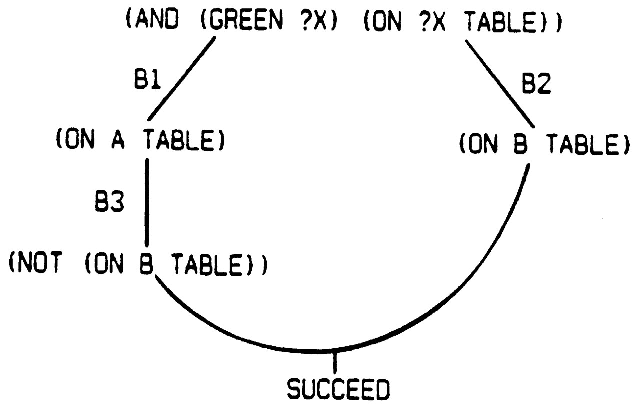


Now we can apply our RGR principle to D and (NOT D) and we are done.

For a simple concrete example, suppose that A and B are two blocks, and we know that A and B are both green, and either A is on the table or B is on the table. We might represent these facts as:

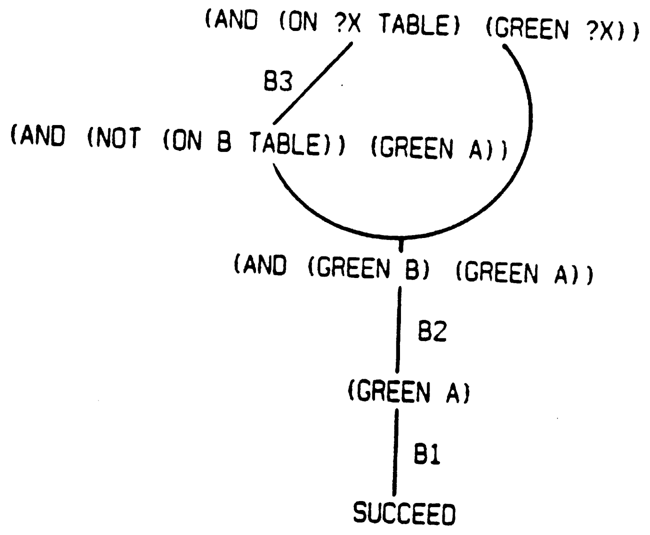
```
B1. (GREEN A)
B2. (GREEN B)
B3. (OR (ON A TABLE) (ON B TABLE))
```

Trying to show that there is something green on the table might generate this goal tree:



We start by looking for green things and we find A. Then we try to show that A is on the table by showing that B is not. We have no way to do this so we fail. Looking for more green things, we find B. We generate the goal of showing that B is on the table, but we notice that we already have a goal showing B is not on the table; so it really does not matter which is true - we are done.

It is interesting to observe that if we reverse the order of the goals - looking for things on the table before green things - we get a less intuitive, but no less efficient solution:


(AND (ON ?X TABLE) (GREEN ?X))
B3
(AND (NOT (ON B TABLE)) (GREEN A))
(AND (GREEN B) (GREEN A))
B2
(GREEN A)
B1
SUCCEED

Starting out looking for things on the table, we try to show that A is on the table, by showing that B is not on the table. This gives us the goal (AND (NOT (ON B TABLE)) (GREEN A)). We can match this against an instance of the original goal, (AND (ON B TABLE) (GREEN B)). Combining these two yields the goal (AND (GREEN B) (GREEN A)), which can be satisfied immediately by assertions.

3.2 Interactions among Complex Assertions

The inference procedures we have described so far do not form a logically complete deductive system, even for propositional logic. (We have not yet considered either quantification or equality, of course.) Part of this is by design. If we have (A -> B) and ((NOT A) -> C), then we have in effect chosen not to be able to deduce (OR B C), although it logically follows from these assertions. A case where this is clearly desirable is a definition like A if and only if EXP, where EXP is some complex expresion. We might choose to represent this as (A <- EXP) and (A -> EXP). If we used conjuctive normal form, as in resolution, we would have $(A \vee (\neg EXP))$ and $((\neg A) \vee EXP)$ which would allow us to make such useless inferences as $(A \vee (\neg A))$ and $(EXP \vee (\neg EXP))$.

In fact, we would argue that being <u>necessarily</u> complete is a very undesirable feature for a deduction system, contrary to the presumption of most theorem proving reasearch. For instance, a number theory theorem prover, if asked to disprove Fermat's Last Theorem, might try whatever special tricks it knows; but it should certainly give up rather than try all possible counter-examples. What we want is for the user of the system to have control over how complete or incomplete the deductive process is. We attempt to provide this control by supplying different notations for each of the useful procedural interpretations of an assertion.

There is, however, one large class of inferences which we have completely omitted up to this point - antecedent inferences made from two complex assertions. We will show examples where such inferences are necessary, but this type of inference must be approached with great restraint. In relatively unrestricted resolution systems, these inferences can be a source of tremendous inefficiency. Many resolution systems, given the equivalent of $(A \supset B)$ and $(B \supset C)$ will produce $(A \supset C)$. This may seem innocent enough - It gives us a shorter way to prove C given A. Suppose, however, we want to prove C, but don't know A. An exhaustive search would end up trying two equivalent search paths, C to B to A, and C directly to A, before failing.

In more complex cases, this can get really horrible. To go back to our study of hierarchies, assume we know P and we want to show $(\neg Q)$ where P and Q are bottom nodes of the hierarchy, and their only common superior is the top node. Using the techniques for reasoning about hierarchies which we discussed, there would be only one deductive path between P and $(\neg Q)$, having length twice the height of the tree (say 2m). If we allowed unrestricted inferences of the form $(A \supset B), (B \supset C) \vdash (A \supset C)$, then there would be 2^{2m} such paths, one path for each possible combination of intermediate nodes. So called "deletion strategies" can cut this search space back to a managable size by checking each subgoal or assertion to see if it has been previously generated, but the point is that this enormous, redundant search space should never have been generated in the first place.

We noted previously that, at least in the case of hierarchies, completely top-down deduction could be just as bad as completely bottom-up deduction. Yet, in the folk-wisdom of theorem proving, there seems to be a sense that top-down strategies are more efficient. I think that this may be due in part to the fact that bottom-up systems normally include those inferences which combine assertions in ways that redundantly expand the search space. It is interesting to note that some of the more efficient theorem proving systems which do allow bottom-up inferences have restrictions which block combinations of this sort. Nevins's system, while allowing $(A \supset B)$ to be used in either a consequent or antecedent direction (which is itself redundant), does not allow (A ⊃ B) to combine with (B ⊃ C). One of Bledsoe's systems <Bledsoe and Bruell 1973> allows antecedent deductions, but only if one of the assertions contains no variables. The deductions we want to block are basically those which involve two procedures, both of which would normally contain variables.

There are, as we mentioned, some cases where it is desirable to allow antecedent deductions involving two complex assertions. One way this situation can arise is in using our second princple of disjunctive reasoning:

$(A \vee B) \vdash C$ if $A \vdash C$ and $B \vdash C$

We have shown how our rules for OR will implement this principle once either A or B is generated as a subgoal. The problem is that if the paths from A to C and from B to C both include antecedent inferences, neither A or B will ever be generated as a subgoal. In the simplest case, using only the rules we have developed so far, from (A -> B), (C -> B), and (OR A C), we would be unable to conclude B. If either of the first two assertions were expressed using <-, we would be able to generate either A or C as a subgoal and complete the deduction. Deducing "Fred is an animal," from "Fred is either a cat or a dog," is an instance of this reasoning schema.

So, in order to reason effectively and efficiently with complex assertions, we must separate them into two types; first, the procedures which form the basic knowledge structures of the domain, and second, the assertions which are "just facts" - situation specific assertions which may be haphazard connections between otherwise unrelated concepts. We might just happen to know that either John's father is a plumber or his wife made a blunder in the commodities market. (We conclude this from the ammount of scrap copper in his basement.) From now on we will refer to the former as procedures, the latter as facts, and both as assertions. We will assume that the programmer declares which assertions are of which type.

Procedures should not be allowed to interact with each other, whereas facts can interact with each other and with procedures. It should be the case that in most common sense situations, the great majority of complex assertions will be procedures rather than facts. The decision to restrict inferences involving procedures is supported by the observation that in a well structured set of procedures, these inferences are superfluous. Suppose that (A -> B), (C -> B), and (OR A C) are all procedures. What do we do about the fact that B cannot be inferred? The answer is that this problem should never have come up. If these assertions really are procedures, they should not be expressed this way, but rather by something like (OR A C) and B. Since B is always true, (A -> B) and (C -> B) become irrelevant. Any system is doomed to hopeless inefficiency if it holds open the possibility of radically restructuring its basic view of the world every time it is asked a question.

Given that we need antecedent deductions from complex facts, how do we go about it? Let's return to our prototype example - procedures (A -> B) and (C -> B), and the fact (OR A C). The difficulty is that we have selected, presumably for good reason, to infer B from A antecedently; but the A in (OR A C) is blocked from triggering (A -> B) because it is not a simple assertion. One way to get around this problem is the following: Compile (OR $A_1 \ldots A_n$) into the set of consequent-directed procedures that it embodies. These would be (A_1 <= (AND (NOT A_2)...(NOT A_n))), (A_2 <= (AND (NOT A_1) (NOT A_3)...(NOT A_n))), etc. (We use <= to indicate a purely consequent procedure.) For (OR A C) this would give us (A <= (NOT C)) and (C <= (NOT A)). We then supply the following rule:

> If (A -> EXP1), or equivalently ((NOT A) <- (NOT EXP1)), is an assertion and (A' <= EXP2) is a fact, then assert (EXP1' <= EXP2') as a fact, where EXP1' and EXP2' are derived from the match between A and A'.

The idea is that (A' <= EXP2) is just the simple assertion A' with an extra condition EXP2, so that it is treated like A' except that the extra condition must be added if A' is used to satisfy a goal. In particular, when A' triggers the antecedent procedure (A -> EXP1), EXP2 is dragged along.

Using this rule, we can do the deduction "Fred is an animal," from "Fred is a cat or a dog." We start with these assertions:

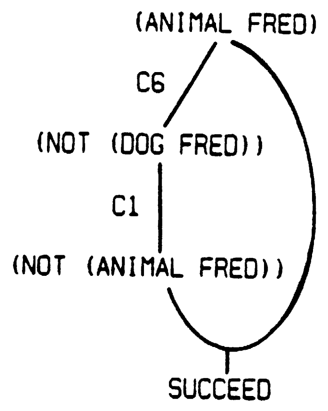
```
C1. ((DOG ?X) -> (ANIMAL ?X))
C2. ((CAT ?X) -> (ANIMAL ?X))
C3. (OR (CAT FRED) (DOG FRED))
```

Applying the new rules we have just created will eliminate the OR statement and add the following assertions:

```
C4. ((CAT FRED) <= (NOT (DOG FRED)))
```

```
C5. ((DOG FRED) <- (NOT (CAT FRED)))
C6. ((ANIMAL FRED) <- (NOT (DOG FRED)))
C7. ((ANIMAL FRED) <- (NOT (CAT FRED)))
```

The goal tree for showing that Fred is an animal would then be:



One way of showing that Fred is an animal is to show that he is not a dog. Recalling the double interpretation of ->, we can do this by showing that he is not an animal. This is the negation of an earlier goal, so by RGR we are done.

Since OR has no antecedent interpretation, there will be no interactions between OR statements. Also, note that there will be no more assertions generated by (OR A1...An) than by A1...An asserted separately. This is in contrast to the combinatorial increase of some resolution methods.

There may also be occasions when we will want to represent a complex fact using -> or <-. In that case we need the following rule:

> If (A -> EXP1) is an assertion and (A' <- EXP2) is a fact, then assert (EXP1' <- EXP2') as a fact, where EXP1' and EXP2'are derived from the match between A and A'.

Of course, any assertion using -> or <- may be replaced by the equivalent assertion using the other.

If we put the expressions involved in this rule in disjunctive form, the rule becomes ((¬ A) v EXP1), (A' v (¬ EXP2)) |- (EXP1' v (¬ EXP2')). This turns out to be just the same procedure as RGR, applied to assertions rather than goals. If we expressed all assertions using only -> and <-, and used this procedure as our only inference rule, we would have a system very much like Boyer's lock resolution <Chang and Lee, Chapter 6>.

We have ended up in a rather peculiar situation. We have attempted to come up with a set of rules that would implement a heuristic deduction system based on common sense reasoning principles, but it turns out that each of our rules can be viewed as some version of the resolution principle. The net result, however, is something with a very different structure from the typical resolution theorem prover. To be specific, the type of system that our rules would produce would be sensitive to the following semantic features of a problem domain:

1. The distinction between goals and assertions.
2. The distinctions among the various procedural interpretations of complex assertions.
3. The distinction between facts and procedures.

Some traditional theorem proving systems share some of these properties. Most take account of 1., but then impose a rigid top down search strategy. Lock resolution provides some limited capabilities in regards to 2. In lock resolution, the literals in a clause have numerical indicies, and only the literal with the lowest index can be resolved upon. So $(A_1 \vee (\neg B_2))$ is equivalent to $(A \leftarrow B)$, and $(A_2 \vee (\neg B_1))$ is equivalent to $(B \rightarrow A)$. But lock resolution provides no way to implement OR in our sense, nor does it take account of 1. or 3. Other systems, like Kowalski's connection graph system, give different procedural interpretations to different formulas, but do so on the basis of purely local features. Finally, Nevins's system takes some account of 3., as we mentioned before, but not 1. or 2.

So, as far as we know, no traditional theorem proving system takes account of all these semantic features, although all are necessary for effective reasoning. We have given examples which show that ignoring 2. and 3. can lead to gross inefficiency, and 2. of course, depends on 1. PLANNER-type systems on the other hand are sensitive to all these aspects of reasoning, but they are limited in the types of situations which they can handle. These past sections illustrate the kind of synthesis of the two approaches which we hope to achieve.

3.3 Splitting

There is another technique for reasoning with disjunction called <u>splitting</u>, which seems to be more intuitive than the types of rules we have proposed, and seems to be quite different from resolution. The technique combines our principles 1. and 2. of disjunctive reasoning in a very straightforward manner.

> To prove $((A \vee B) \supset C)$ set up two distinct data bases. In the first try to prove either $(\neg A)$ or $(A \supset C)$. In the second try to prove $(\neg B)$ or $(B \supset C)$. If both subproofs succeed, the entire proof succeeds.

Two comments - First, this procedure obviously generalizes to disjunctions involving more than two terms. Second, in a refutation based procedure asserting A is the same as trying to prove $(\neg A)$. The only difference is whether to regard the formula A as a goal or not. If A is a goal an attempt is made to involve A in the contradiction to be derived. So by assuming a refutation procedure (or a procedure with rules like our own AGR), we can state the method more simply as:

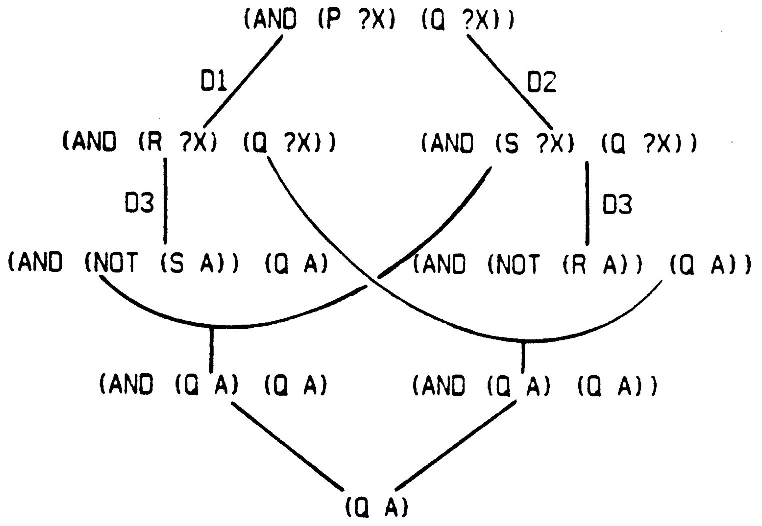
> To prove $((A \vee B) \supset C)$ set up two distinct data bases. In the first try to prove $((\neg A) \vee C)$. In the second try to prove $((\neg B) \vee C))$.

We have made the rule less intuitive, but since we have already argued that the computations involved in asserting A are just a subset of those involved in proving $(\neg A)$, it would be foolish to duplicate these computations. The rule, more or less in this form, is used in the theorem provers of Bledsoe <1971>, Reiter <1973>, and Nevins. Bledsoe uses this procedure on $(A \vee B)$ only if A and B are independent - i.e. they have no variables in common. Reiter and Nevins use techniques which can handle dependent sets of disjuncts as well. Nevins seems to make the most sophisticated use of this idea, so we will use his procedure for more detailed analysis.

The question we really want to answer is whether splitting can be used more efficiently than the other techniques we have presented. In the following example it would seem so. Suppose we want to prove $(\exists x ((P x) \wedge (Q x)))$ and we know $((R x) \supset (P x))$, $((S x) \supset (P x))$, and $((R a) \vee (S a))$. We might represent these facts as:

```
D1. ((P ?X) <- (R ?X))
D2. ((P ?X) <- (S ?X))
D3. (OR (R A) (S A))
```

Then if we attempt to prove (AND(P ?X) (Q ?X)) we get the following goal tree:



We start out trying to satisfy (P ?X). One way to do this is to satisfy (R ?X). This matches (R A) in (OR (R A) (S A)), so we try to prove (NOT (S A)). We have no way to do this so we fail and backup. Another way to satisfy (P ?X) is to satisfy (S ?X). We have two ways to satisfy (S ?X). One way is to use RGR with the goal that contains (NOT S A). The result of applying RGR to these two goals is (AND (Q A) (Q A)). This simplifies to (Q A), but we have no way to satisfy this goal, so we again backup. The other way to satisfy (S ?X) is to use (OR (R A) (S A)) and try to show (NOT (R A)). The only thing we can do with this is to apply RGR with (AND (R ?X) (Q ?X)). This again generates (AND (Q A) (Q A)) which simplifies to (Q A). If we are clever, we recognize this as a goal we have previously generated and we fail.

We failed, as we should have, since we could not prove (Q A); but it seems like we went through a lot more manipulations than necessary. In fact we did two nearly equivalent deductions. In one we got rid of (R ?X) using (OR (R A) (S A)), and (S ?X) by RGR. In the other we got rid of (S ?X) using (OR (R A) (S A)) and (R ?X) by RGR. The suspicious aspect seems to be that (OR (R A) (S A)) was invoked twice in what turns out to be an example of reasoning by cases.

Compare this with what happens when we use splitting. If we split (OR (R A) (S A)), we assert D4. (R A) and then try to prove (AND (P ?X) (Q ?X)). The worst that can happen is that we select (S ?X) first as a way to satisfy (P ?X):

$$\begin{tikzcd}
 & \text{(AND (P ?X) (Q ?X))} \arrow[dl, dash, "\text{D2}"'] \arrow[dr, dash, "\text{D1}"] & \\
\text{(AND (S ?X) (Q ?X))} & & \text{(AND (R ?X) (Q ?X))} \arrow[d, dash, "\text{D4}"] \\
 & & \text{(Q A)}
\end{tikzcd}$$

Since we cannot prove (Q A) we do not have to go on to the second case, (S A).

What makes the splitting method more efficient than the resolution method? We can answer this question by looking at the problem in the following way: There are basically three ways that (A v B) can be used to prove C.

A |- C and |- (¬ B)

B |- C and |- (¬ A)

A |- C and B |- C

The two methods cover these three possibilities differently. Splitting covers them with (B |- C or |- (¬ B)) and (A |- C or |- (¬ A)). For our resolution based method, invoking (OR A B) by matching A invokes (A |- C and |- (¬ B)). RGR invokes B |- C whenever |- (¬ B) is invoked, so we get A |- C and (|- (¬ B) or B |- C). Since (OR A B) can also be invoked by matching B, we also get B |- C and (|- (¬ A) or A |- C), so the entire method can be summarized by:

(A |- C and (|- (¬ B) or B |- C)) or (B |- C and (|- (¬ A) or A |- C))

This expression covers the A |- C, B |- C case two different ways. Therefore, whenever reasoning by cases applies, there will be two search paths that yield the same result. The splitting method, on the other hand, covers each possibility in only one way and avoids this redundancy.

While splitting is more efficient in this respect, it has some problems of its own. Suppose (Q A) were provable. In that case, we would have gone on to the second data base, where D5. (S A) is asserted and the goal tree would have looked like this:

$$\begin{tikzcd}
\text{(AND (P ?X) (Q ?X))} \arrow[d, dash, "\text{D2}"] \\
\text{(AND (S ?X) (Q ?X))} \arrow[d, dash, "\text{D5}"] \\
\text{(Q A)}
\end{tikzcd}$$

One minor point is that the top part of the goal tree is repeated in each case. Nevins handles this by postponing splitting until after all other methods have been exhausted, and the goals established at this point are inherited by both of the split data bases. Another problem is that in each data base we have to prove (Q A). If the proof of (Q A) is long and involved, it would be quite inefficient to repeat it. In the non-splitting method we faced this problem also, when we generated the goal (AND (Q A) (Q A)). Here it was a simple matter of eliminating the repeated subgoal. Perhaps a similar technique could be developed for the splitting method, but it looks like a more difficult problem.

The most serious problem with splitting, however, is deciding what formulas to split. Suppose we have a goal A which we are unable to prove without splitting. If we decide to split (B_1 v ... v B_n), and this turns out to be irrelevant to the proof of A, then the same proof of A will be repeated n times. If we pick another formula (C_1 v ... v C_m) which is also irrelevant, then for each B_i, we will go through all the C_i's, giving us m*n data bases, and m*n identical proofs. We have a serious danger of a combinatorial explosion, so picking only relevant assertions to split is very important.

This typically is less of a problem in mathematical theorem proving, because disjunctive assertions are not added to a problem statement unless they are relevant to the goal. It would be considered misleading to ask for a proof of $((A \vee B) \supset C)$ if C is provable by itself. In common sense world modelling the situation is quite different. Here we expect to have a data base that can be used to answer many different questions. Only some of the assertions are used to answer any one particular question.

The obvious criteria for a split being relevant is that at least one of the disjuncts should contribute to solving a goal. We faced this problem before with our other interpretation of (OR A B). If the paths from A and B to the goal are antecedent paths, then we will be unable to invoke (OR A B) for splitting. We can handle this problem by a technique similar to that used before. For each of the disjuncts A_i in (OR $A_1 \ldots A_n$) we can generate the antecedent deductions that would follow from A_i, and combine them into a conjunction, EXP_i. Then (OR $A_i \ldots A_n$) would be replaced by (OR (AND A_1 EXP_1)...(AND A_n EXP_n)). Then the split would be triggered by a goal matching any subexpression in this longer disjunction. Furthermore, all of the antecedent deductions for each case would be pre-computed.

Nevins has a quite different solution to the relevance problem which is very interesting and possibly more elegant than the solution proposed above. Nevins's idea is to postpone checking the relevance of a split until a case has been solved. Then if the split was not involved in the solution, the other cases of the split do not have to be tried. For a simple example suppose we know (A ⊃ B), (C ⊃ B), (A ∨ C), and (D ∨ E), and we want to prove B. Without splitting we generate the goals B, A, and C. Since none of these goals has been solved, we have to use splitting. Suppose we chose to split (D ∨ E) first. We would add D to the data base and try again. We still cannot get anywhere, so we try another split, (A ∨ C). We add A to the data base and find we have a proof. Then we try C and we also have a proof. So now we have "solved" the case where D is true. If we look at the solution, however, we discover that it did not depend on D, so we can go back to the next higher level (in this case, the top level) without bothering with the other pseudo-case, E. Splitting (D ∨ E) has cost us only whatever inferences are made from D alone, and the combinatorial explosion of cases is avoided.

It turns out that there is a mechanism which works much like Nevins's technique, but without the overhead of the multiple data bases required by splitting. In fact, it amounts to nothing more than treating a fact (A ∨ B) with the deductive procedures we have associated with the fact (A <- (NOT B)). Recall that in section 3.2 we introduced the rule that if (A -> EXP1) is an assertion and (A <- EXP2) is a fact, we assert (EXP1 <- EXP2) as a fact. We needed this rule because of the problem of antecedent deductions from complex facts.

If we treat the fact (OR A B) in a corresponding fashion, we would have the following rules:

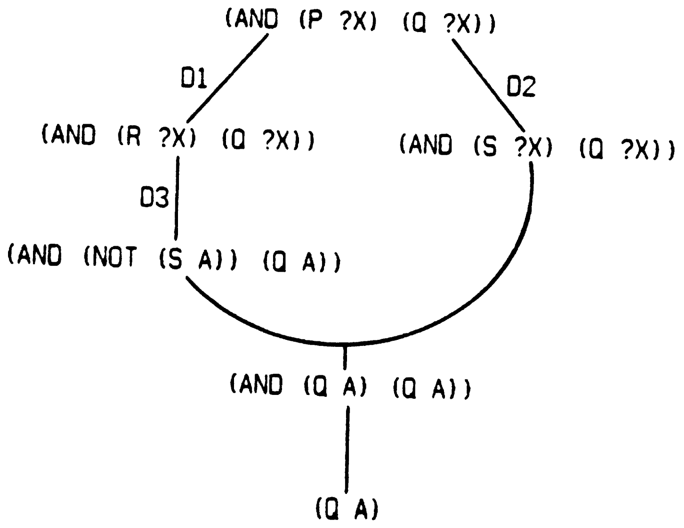
1. If (OR A $B_1 \ldots B_n$) is a fact and (AND A $C_1 \ldots C_m$) is a goal, allow (AND $C_1 \ldots C_m$ (NOT B_1)...(NOT B_n) to be a goal.
2. If (OR A $B_1 \ldots B_n$) is a fact and (OR (NOT A) $C_1 \ldots C_m$) is a fact, assert (OR $C_1 \ldots C_m$ $B_1 \ldots B_n$) as a fact.
3. a. If (OR A $B_1 \ldots B_n$) is a fact and (A -> EXP) is a procedure, assert (OR EXP $B_1 \ldots B_n$) as a fact.

   b. If (OR A $B_1 \ldots B_n$) is a fact and ((NOT A) <- (NOT EXP)) is a procedure, assert (OR EXP $B_1 \ldots B_n$) as a fact.

Of course, these rules generalize to inferences that require matching between A and some A'. Also note that 3.a. and 3.b. are the same rule because of the procedural equivalence of (A -> EXP) and ((NOT A) <- (NOT EXP)). Finally these rules should be understood to cover degenerated cases such as (OR A B), (NOT A) |- B (a special case of rule 2.). We will use these rules for <u>facts</u> of the form (OR $A_1 \ldots A_n$). We still need procedures which embody the multiple-consequent-theorem idea, however, so we will keep our old interpretation for (OR $A_1 \ldots A_n$) as a <u>procedure</u>.

To see the relationship between this method and Nevins's, think of (OR A $B_1 \ldots B_n$) as a split where A is asserted with a pointer to the other cases $B_1 \ldots B_n$. It really behaves this way, because only the left most expression, A, can trigger inferences. The other cases $B_1 \ldots B_n$ are appended to the results of these inferences, and when the A case is "solved" the next case, B_1, becomes active with the others held in reserve. If the solution to the problem does not involve A, then just as in Nevins's system, the other cases associated with A will never be considered. To see this work, look again at one of our previous examples. D1. ((P ?X) <- (R

?X)) and D2. ((P ?X) <- (S ?X)) are procedures, D3. (OR (R A) (S A)) is a fact, and (AND (P ?X) (Q ?X)) is the goal:



Comparing this to our previous resolution-based method, we can no longer generate the double search path, because the (S A) in (OR (R A) (S A)) cannot be invoked until the (R A) has been eliminated. The cost of this is seen in the following example: Suppose we know (OR A B) and (NOT A), and we want to prove B. Before, we could have matched the goal B against the (OR A B) and generated the goal (NOT A). Now, however, we have to do the inference (OR A B), (NOT A) |- B in the antecedent direction so we can get to B for solving the goal. On the other hand, the antecedent inferences that we would have had to do involving $B_1 \ldots B_n$ in (OR A $B_1 \ldots B_n$), now do not have to be done until all the preceding possibilities have either been contradicted by an assertion, or used to solve some goal.

It is worth pointing out that since almost all resolution theorem provers, except lock resolution, allow (OR A B) to be invoked either by matching A or B, they all share the multiple path problem in situations where a case analysis applies. Nevins does not seem to be aware of this advantage of his system over resolution, but he does cite two other advantages. The first is that by using multiple data bases, the formula generated in solving one case can be deleted before the next case. Nevins claims that this is more efficient, but that would be true only in an unindexed data base (which Nevins uses). In the type of indexed data base used in PLANNER-like systems, these extra formulas would get looked at only if they matched some goal. There is no a priori reason to believe that these matches are less useful than others; they simply represent inferences that Nevins's system cannot do.

The other point that Nevins makes is that his system does not require putting all expressions into conjunctive normal form. If we know something of the form $(A \vee (B \wedge C))$, resolution systems force us to express this as two assertions $(A \vee B)$ and $(A \vee C)$. If there are a lot of inferences to be made from A, then they have to be made twice. Equivalently, if $(A \vee (B \wedge C))$ is a goal, this gets expanded to two goals, $(A \vee B)$ and $(A \vee C)$; so the subgoal A may have to be solved twice. This is required by neither Nevins's system nor ours, so both are superior to traditional resolution systems in this respect.

4. Quantification and Equality

Subsequent to Micro-Planner and SHRDLU, there has been one idea which has come up repeatedly as a solution to the problem of quantification in procedural deduction systems. The idea is that to prove a universal assertion (∀x (P x)), create a previously unused name, say G0123, and prove the assertion with this new name substituted for the quantified variable, (P G0123). Since we have no information about G0123, if we can prove (P G0123), then P must be true of everything. The analogous idea for assertions is to represent (∃x (P x)) by (P G3210); i.e., just assign an arbitrary name to the individual whose existence is being asserted. Variations on this idea are found in <Moore 1973>, <Davies 1974>, <Hewitt 1975>, and <Rich and Shrobe 1975>.

In fact this technique is nothing more than eliminating quantities by use of Skolem functions, as is done in resolution. This is clearly the case in the treatment of existential assertions. The case of universal goals may be less clear, but recall that in a refutation system the negation of the goal is asserted. So (∀x (P x)) becomes (¬ (∀x (P x))), which becomes (∃x (¬ (P x))). Eliminating the quantifier now gives us (¬ (P G0123)). Since we previously showed that (¬ EXP) in a refutation system is equivalent to (GOAL EXP) in a direct-proof system, we end up with the same goal that we had before, (P G0123).

This technique handles quantification very nicely, but it requires having an effective mechanism for reasoning about equality, since an expression introduced by Skolemization might turn out to name the same individual as some expression already in the data base. PLANNER-like systems, however, assume that every individual has one unique name. This makes reasoning about equality trivial; two expressions are equal only if they are identical. This is not sufficient for the more complex situations we want to model.

The techniques for handling equality which we will propose are all based on one key idea - the principal role of equality in common sense reasoning is to establish the identity of individuals. If we know ($\exists$x (P x)) and we represent this as (P G0123), we will be concerned with <u>which</u> individual G0123 actually is. Or, we might know that John's only friends are Bill and Mary, represented by ((FRIEND JOHN ?X) -> (OR (?X = MARY) (?X = BILL))). Then if we are asked a question about all of John's friends, we know we only have to consider Bill and Mary.

In other types of reasoning, equality is treated differently. In proving mathematical theorems, an equality is frequently regarded as an expression to be transformed into some other expression, and the question of which individual is referred to is meaningless. In engineering or other applications of algebraic problem solving, an equality is a constraint to be satisfied, and special techniques are required which manipulate sets of constraints to produce solutions. The techniques required to handle these situations are much more sophisticated and specialized than anything we will consider.

If we want to use equality to talk about the identity of individuals, we have to have some way to indicate when we _know_ the identity of an individual. To do this we will have two types of names - _identified constants_ whose identity will be considered known, and _unidentified constants_ whose identity is unknown. An identified constant will always be unequal to any other identified constant; an unidentified constant might be equal to other constants of either type. Skolem constants are, of course, unidentified.

This distinction has two practical consequences. First, in doing a deduction we need to consider looking for an equality substitution only if an unidentified constant is involved. Second, we do not need explicit assertions that identified constants are distinct. A common-sense world model will normally contain many identified constants and only a few unidentified constants. In SHRDLU's blocks world, it would be incredibly inefficient to have to have assertions that each known block is distinct from every other known block, or when trying to prove something about block A, to have to consider whether block A might be the same as block B, C, D, etc. On the other hand, for some unknown object G0123, this may be exactly what we want to do. The distinction between identified and unidentified constants allows us to do this sort of thing only when necessary.

We can extend this idea to handle functions as well. If a function is one-to-one, we can take an application of the function to be the canonical name of its value. If we want to reason about lists, we will need a function (call it CONS) to add an item to a list. Now if we know what A and B are, we do not have to ask what the value of (CONS A B) is; it is simply (CONS A B). We can, of course, introduce other notations for convenience; say, let [A B C] be an abbreviation for (CONS A (CONS B (CONS C NIL))). So, we can have identified expressions based on identified functions. Two identified expressions are unequal unless they have the same function and their respective arguments are equal. So if CONS and FOO are identified functions, (CONS A B) and (FOO C D) are implicitly unequal. (CONS A B) and (CONS C D) are equal only if A equals C and B equals D. Many-to-one functions and all Skolem functions would be unidentified.

The basic operation in reasoning with equality is simply substitution of equals for equals. The simplicity of this rule makes it very dangerous, however. There are so many symmetries that the possibilities for redundancy are enormous. The first symmetry to consider is the fact that given (G1 = G2) we have a choice of whether to substitute G1 for G2 or G2 for G1. If we know (P G1) and (G1 = G2), and we want to prove (P G2), we can either change the assertion to (P G2) or the goal to (P G1). To try both would be redundant. We will adopt the convention that the left arqument is to be replaced by the right argument.

In most common sense situations, it will be obvious which direction the substitution should go. For instance, an unidentified expression should always be replaced by an identified expression. There are two reasons for this. The first reason is that by reasoning with the identified expression the possibilities for further equality substitutions are reduced. Only unidentified expressions are candidates for matching, where any expression would be a possibility if we used the unidentified expression. The other reason is that we probably know much less about the unidentified expression, so that fewer substitutions will actually have to be done.

Another typical case where the direction of the substitution is obvious is a simplification or evaluation rule - ((MINUS (MINUS ?X)) = ?X), ((CAR (CONS ?X ?Y)) = ?X), or ((FACT (SUC ?X)) = (TIMES ?X (FACT ?X))). The last of these gives a good illustration of the need for ascribing procedural significance to the order of the arguments of =. The most obvious local heuristic to use would be that the more complex expression should be replaced by the simpler expression. This works fine for MINUS and CAR, but would make the wrong substitutions for FACT. It is the semantics of the domain which tells us that FACT is defined in terms of TIMES, and not the other way around.

Another choice we have to make is whether to do equality substitutions in a bottom-up or top-down manner. The bottom-up approach is the obvious substitution: if (P G1) and (G1 = A) are asserted, assert (P A). The top-down approach would be: if (P A) is a goal and (P G1) is an assertion, propose (G1 = A) as a goal. There is a dual choice with respect to proving inequalities. The bottom-up method is: if (P G1) and (NOT (P A)) are assertions, assert (NOT (G1 = A)). The top-down method is: if (NOT (G1 = A)) is a goal and (P G1) is an assertion, propose (NOT (P A)) as a goal. As usual, the bottom-up method for equalities is computationally equivalent to the top-down method for inequalities, and vice versa. In refutation systems they are the same.

The usual equality rule for resolution systems, paramodulation, embodies the bottom-up method for using equalities and the top-down method for proving inequalities. Without the concept of an identified expression, this is the only reasonable approach. Suppose we want to prove (P A) and we know (P B), (P C), (P D), etc. We would not want to genarate (A = B), (A = C), (A = D), etc. as goals, unless we had some reasonable hope of proving one of them. With identified expressions cutting down the number of such matches, however, this approach (call it "anti-paramodulation") becomes more attractive. Of course, if there is a very large number of assertions (P A), (P B), P C), etc. then anti-paramodulation can be inefficient. But if there are powerful methods for proving expressions equal - say ((?X = ?Y) <- (BIG-HAIRY-EXP ?X ?Y)) - then bottom-up equality substitution can also be inefficient.

There are other problems. Suppose we know E1. ((P ?X) <- (Q ?X)), E2. ((Q ?X) <- (R ?X)), and E3. (OR (G1 = A) S), and we want to prove (P G1):

$$\begin{tikzcd}
 & & & & \text{(P G1)} \arrow[dll, dash, "\text{E1}"'] \arrow[d, dash, "\text{E3}"] \\
 & & \text{(Q G1)} \arrow[dll, dash, "\text{E2}"'] \arrow[d, dash, "\text{E3}"] & & \text{(AND (P A) S)} \arrow[d, dash, "\text{E1}"] \\
\text{(R G1)} \arrow[d, dash, "\text{E3}"] & & \text{(AND (Q A) S)} \arrow[d, dash, "\text{E2}"] & & \text{(AND (Q A) S)} \arrow[d, dash, "\text{E2}"] \\
\text{(AND (R A) S)} & & \text{(AND (R A) S)} & & \text{(AND (R A) S)}
\end{tikzcd}$$

Basically, we have two ways to prove (P G1). We can prove (P G1) directly, or we can substitute A for G1, and try to prove S as well. But paramodulation allows us to perform the substitution over and over for every subgoal of (P G1), producing repeated instances of the same goals. This can be cut down by use of the deletion strategy, but these represent inferences that never needed to be done. This is yet another example where traditional theorem provers perform redundant computations. The simplest cure for this problem is to restrict application of paramodulation to the point where an expression is introduced into the goal tree. Using this restriction we get a much simpler search space for our example:

$$\begin{tikzcd}
& \text{(P G1)} \arrow[dl, dash, "\text{E1}"'] \arrow[dr, dash, "\text{E2}"] & \\
\text{(Q G1)} \arrow[d, dash, "\text{E2}"'] & & \text{(AND (P A) S)} \arrow[d, dash, "\text{E1}"] \\
\text{(R G1)} & & \text{(AND (Q A) S)} \arrow[d, dash, "\text{E2}"] \\
& & \text{(AND (R A) S)}
\end{tikzcd}$$

The method for reasoning with equality which we will use is a variation of paramodulation. In particular, we will take advantage of the special case where we have just (G1 = A) rather than (OR (G1 = A) EXP). If we have just (G1 = A), we can <u>replace</u> G1 with A everywhere and eliminate G1. With (OR (G1 = A) EXP) we have to keep around the old assertions about

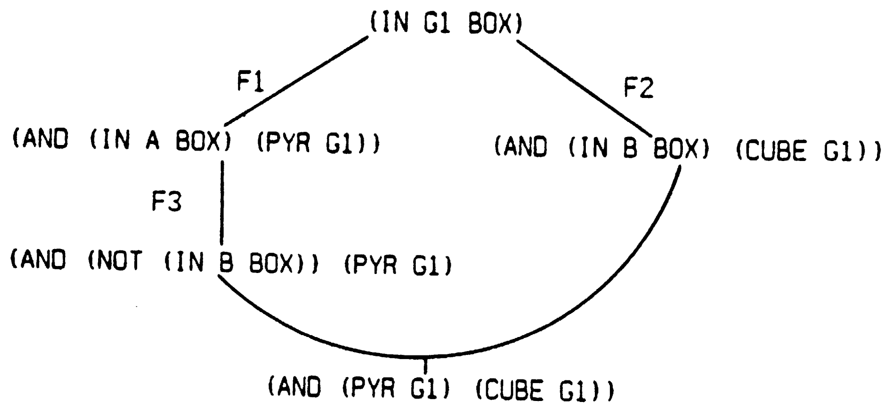
G1, since it might turn out that EXP was true and (G1 = A) was false.

What do we then do about cases like (OR (G1 = A) EXP)? Here the best solution may be to use a real data base split, like Nevins's. That way, in one data base we will have (G1 = A) by itself, and we can do the replacement. This replacement stategy fits especially well with our idea of identified constants. If we can replace an unidentified constant G1 by an identified constant A, we have completely gotten rid of an object whose identity was in question. There is a serious problem in resolution-like systems, in that two incompatible substitutions might be tried in different parts of the goal tree, which later combine. If the original substitutions have been lost along the way, the program cannot tell that it is losing.

To illustrate this, suppose that A and B are identified constants and we know in a particular situation that A is the only pyramid and B is the only cube, and that either A or B is in the box. We could represent these facts as:

```
F1. ((?X = A) <- (PYR ?X))
F2. ((?X = B) <- (CUBE ?X))
F3. ((OR (IN A BOX) (IN B BOX))
```

If we have some unidentified object named by G1, and we want to find out if it is in the box, we might get this goal tree using paramodulation:


(IN G1 BOX)
F1
F2
(AND (IN A BOX) (PYR G1))
(AND (IN B BOX) (CUBE G1))
F3
(AND (NOT (IN B BOX)) (PYR G1)
(AND (PYR G1) (CUBE G1))

In trying to prove (IN G1 BOX), we first try substituting A for G1 and proving that G1 is a pyramid. We could prove A is in the box, if we could prove B is not in the box. We cannot do this, so we back up and try substituting B for G1. We can now have one goal containing (IN B BOX) and one with (NOT (IN B BOX)), so we can apply RGR. This leaves us with the impossible goal of showing G1 is both a cube and a pyramid. What happened was that we combined a goal derived from substituting A for G1 with a goal which derived from substituting B for G1. Since G1 cannot be equal to both A and B, the resulting goal will be impossible to solve.

There are other ways to state these particular facts to get around this difficulty, but the problem is endemic to resolution-type approaches. With the replacement/splitting approach this sort of thing cannot happen. In the example above, suppose we restate the facts as:

```
(OR (?X = A) (NOT (PYR ?X)))
(OR (?X = B) (NOT (CUBE ?X)))
(OR (IN A BOX) (IN B BOX))
```

Using replacement/splitting, we start with the goal (IN G1 BOX). We cannot get anywhere without splitting, so we split (OR (G1 = A) (NOT (PYR G1))). In the first case, we assert (G1 = A) and attempt to show (IN A BOX). We could show (IN A BOX) if we could show (NOT (IN B BOX)). At this point we fail. As in paramodulation we backtrack to the top level goal. In this data base, the top level goal is (IN A BOX) rather than (IN G1 BOX), so no further substitutions can be tried and the entire proof fails.

Notice that the concept of an identified constant is crucial in making this work. If A were not an identified constant, we would try to substitute B for A, and end up with the same set of impossible goals as in the previous example.

Another important point is that relevancy is much easier to determine for splitting equalities than for splitting in general. An equality split of (OR (G1 = A) EXP) is relevant in two cases. (Recall that (G1 = A) means replace G1 by A, so G1 must be an unidentified expression). The two cases are:

Case I: G1 is a subexpression of some goal.
case II: G1 is a subexpression of some assertion which would match a goal if G1 had the appropriate value.

In contrast to a general splitting procedure, these two criteria can be easily checked without doing any forward inferences from the assertion being split. The basic idea behind this technique is simple enough, but getting an efficient control structure for it is tricky. We propose the following control structure:

If (G1 = A) is an existing atomic assertion or (NOT (G1 = A)) is an existing atomic goal, and we generate a new goal or assertion containing G1, replace G1 by A before processing. E.g., if we know (G1 = A) and we generate the goal (P G1), replace this goal with (P A).

If G1 is an unidentified expression, and we generate a (G1 = A) as an atomic assertion or (NOT (G1 = A)) as an atomic goal, replace G1 by A in all existing assertions and goals. Then perform any inferences which were blocked before by a failure of a match on G1. E.g., Suppose that (P G1) is a goal and (P A) and ((P ?X) <- (Q ?X)) are assertions. If we generate (NOT (G1 = A)) as a simple goal, the goal (P G1) would be replaced by (P A). We would match this against the assertion (P A), because this match was not possible before. We would not match (P A) against ((P ?X) <- (Q ?X)), however, since we would have already generated (Q G1), and (Q A) will be generated by replacing G1 by A here.

The only other rules we need for equality are a few basic simplification procedures:

1. If (AND (A = A) EXP) is a goal, replace this goal with EXP.
2. If either (AND (?X = A) (EXP ?X)) or (AND (A = ?X) (EXP ?X)) is a goal, replace the goal with (EXP A).
3. If (AND (NOT (A = B)) EXP) is a goal, and A and B are distinct identified expressions, replace this goal with EXP.

The degenerate cases of these rules and the dual rules for assertions should be obvious.

We turn now to the question of splitting. Suppose we have a goal (P G1) where G1 is an unidentified expression for which there is no immediate substitution. We will first try to prove (P G1) without substitutions created by splits. If this fails we will try those substitutions. Since subgoals of (P G1) will presumably also contain G1, we have the potential for invoking substitutions at that level as well. This is the same problem we saw with unrestricted paramodulation. To prevent this we will invoke substitution only at the point where a goal with G1 in its active part first appears. In subgoals of this goal we will mark G1 with an * to indicate that substitutions are not to be invoked. Ex: If we start out with the goal (AND (P A) (P G1)), G1 is not in the active part of the goal. If we solve (P A) then (P G1) becomes active. Subgoals of this would be of the form (Q *G1).

If we do need to try split-based substitutions for G1 we will collect all the assertions of the form (OR EXP_1...(G1 = A)...EXP_n) and add them to a list of assertions to be split later. We delay splitting until all other deductions have been tried, so that the sub-data bases will inherit as much as possible from the original data base. Since the same assertion may be relevant to more than one goal, care should be taken not to add the same assertion to the list more than once.

Another case to consider is the goal (P A) and an assertion (P G1), where G1 is an unidentified expression. Here we add to the list all the substitutions for G1 which are <u>compatible</u> with A. Two expressions are compatible unless they are distinct identified expressions. Finally, if a goal of the form (AND (NOT (G = A)) EXP) is generated, this will also be added to the list of formulas to be split (taking into account the duality between goals and assertions).

Two additional points need to be made. First, if the goal is (P G1) and the relevant assertion is (OR (EXP$_1$?X)...(?X = A)...(EXP$_n$?X)), we will add the specialized assertion (OR (EXP$_1$ G1)...(G1 = A)...(EXP$_n$ G1) to our list. Second, in order to mesh nicely with the other deductive procedures we have advocated, it is convenient to split only assertions where all of the disjuncts are equalities. (Note that this is not a problem for goals.) This can be easily accomplished simply by always putting equalities on the right when an assertion is generated. (OR P$_1$...P$_n$ (A$_1$ = B$_1$)...(A$_m$ = B$_m$)) can be used in our normal inference procedures until it is whittled down to (OR (A$_1$ = B$_1$)...(A$_m$ = B$_m$)). Furthermore, this makes good heuristic sense. If we express the fact that A is the only cube by ((?X = A) <- (CUBE ?X)), this assertion could be invoked every time we wanted to prove something about an unidentified expression. If, on the other hand, we write ((CUBE ?X) -> (?X = A)), this assertion will be invoked only when we assert that something is a cube, or we want to prove something is not a cube.

Once we have constructed our list of possible splits, we can use exactly the same splitting procedure Nevins uses. Since we used great care to select only equalities which are relevant to our problem (helped by the identified/unidentified distinction), in many cases we should have a far smaller set of facts to split.

There are special cases where this replacement strategy will not work. The most obvious case is a commutative function (F ?X ?Y) = (F ?Y ?X). Applying this rule to (F A B) would yield (F B A), but the rule applies to (F B A) as well, yielding (F A B) again. The best way to handle commutativity and associativity would probably be to build it into the pattern matcher. At any rate, sophisticated mathematical manipulation is not our goal.

We can hardly explore all of the consequences of our equality techniques, but the first question in the scenario provides one simple example. To recall the situation, A and B are known to be green, and A, B, and C are the only things in the scene. We can represent this situation by the following set of facts:

```
H1. (GREEN A)
H2. (GREEN B)
H3. (LOC A BOX)
H4. (LOC B BOX)
H5. (LOC C TABLE)
H6. ((PRESENT ?X) -> (OR (?X = A) (?X = B) (?X = C))
```

We also need some procedures for reasoning about where things are.

```
H7. (INCOMP (LOC ?X TABLE) (LOC ?X BOX))
H8. ((LOC ?X BOX) -> (PRESENT ?X))
H9. ((LOC ?X TABLE) -> (PRESENT ?X))
```

These procedures interact with the previous assertions to produce new assertions.

```
H10. (PRESENT A)
```

```
H11. (PRESENT B)
H12. (PRESENT C)
```

We would also get three assertions like (OR (A = A) (A = B) (A = C)), but these can be recognized as tautologies and discarded. They do serve as a check on the consistency of the data base, however.

What we want to show is that everything in the box is green. Formally, this is:

```
((LOC G1 BOX) -> (GREEN G1))
```

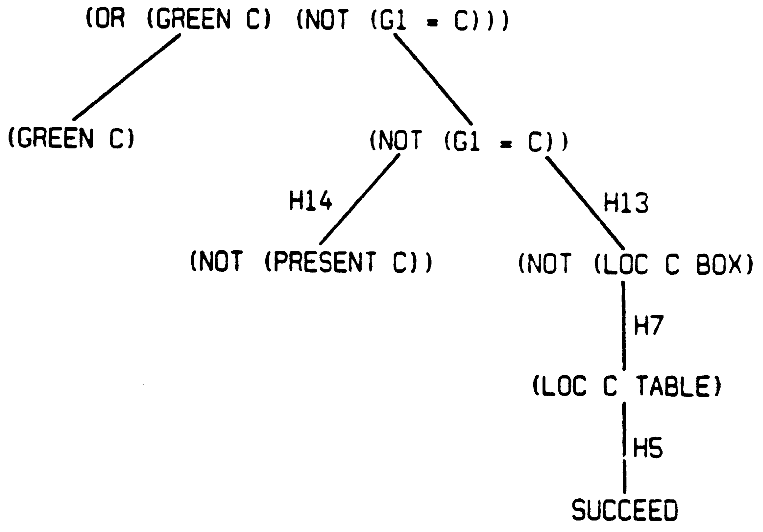
G1 is an unidentified Skolem constant, so anything we can prove about it is true in general. We are assuming that A, B, and C are identified constants. We will prove implications by natural deduction, asserting the antecedent and then deducing the consequent. So we assert H13. (LOC G1 BOX), and from this we get the following additional assertions:

```
H14. (PRESENT G1)
H15. (OR (G1 = A) (G1 = B) (G1 = C))
```

We now try to prove (GREEN G1). We have no assertions which match this goal, so we have to go directly to splitting. The only assertion our rules will pick to split is (OR (G1 = A) (G1 = B) (G1 = C)). We set up the following three goals in three separate data bases.

```
(OR (GREEN A) (NOT (G1 = A))
(OR (GREEN B) (NOT (G1 = B))
(OR (GREEN C) (NOT (G1 = C))
```

In the first two cases, the first goal is immediately satisfied. In the third case we do the following deduction:



The worst-case ordering of subgoals was intentionally chosen to show the entire search space. We start out by trying to prove that C is green but have no way to do that. So we turn to trying to show (NOT (G1 = C)). All the substitutions of C for G1 become subgoals of this goal. We know (PRESENT G1), so we try showing (NOT (PRESENT C)). But we know C is present, so this goal is abandoned. We also know (LOC G1 BOX), so we try to show (NOT (LOC C BOX)). We can attack this goal using (INCOMP (LOC ?X TABLE) (LOC ?X BOX)), generating the goal (LOC C TABLE). We know that this is true, so the proof is done.

If this example seems too simple to be interesting, consider that its simplicity is a virtue and not a fault. Using our deductive strategies, we took a straight-forward representation of the problem and generated an intuitively relevant set of goals. Contrast this with Nevins's procedure, for instance. If there had been other disjunctive assertions in the data base, his theorem prover might have split any number of them, before getting around to the one which was actually relevant to the goal. Our procedure, on the other hand, would ignore all but those which suggested substitutions for G1.

There is one final refinement we can use to make our equality stategy more efficient. In our example, we assumed that blocks were the only thing in our domain. We didn't bother asserting that A, B, C, and G1 were blocks, or make the distinction that any <u>block</u> in the scene is either A, B, or C. If there is more than one kind of object around, we should take note of this fact. Suppose we had wanted to prove that all <u>cubes</u> in the box were green. Then we would have asserted that G1 was a cube. If we knew that A was a pyramid, we would not want to consider the case (G1 = A). Similarly, if we wanted to know whether C was green, and we knew that C was a cube, we would not want to bother with the green pyramid G23.

We can handle these situations with a system of typed constants and variables. If we know that A is of type T1 (indicated by writing A as A/T1) and G1 is of type T2, where T1 and T2 are incompatible, the pattern matcher would reject any possible match between A/T1 and G1/T2. We can make this system of types hierarchial so that, while A/CUBE will not match G1/PYR, it will match G2/BLOCK. This idea of a hierarchy of types was first worked out as a solution to a related problem, the "symbol-mapping problem" described by Fahlman (1975), but [illegible] as well, to cut down possible equality substitutions. A [illegible] this idea is presented in Appendix B.

5. Control Structure Issues

Deductive techniques have been applied to a wide variety of problems, including mathematical theorem proving, automatic programming, and robot problem solving, as well as the sort of world modelling or common sense reasoning which we have discussed. We would argue that each of these problem domains is sufficiently different from the others so that each requires its own control structure. Consequently there is no such thing as the "right" control structure for reasoning.

Take the domain of robot problem solving for instance. In this domain, the question of whether a solution to a problem exists is usually not too difficult to answer. If there is no solution, either the goal itself will be physically impossible, e.g. (AND (ON A B) (ON B A); or each branch of the search tree will lead to an impossible subgoal. And physical impossibility in these simple domains is usually easy to check. The hard problem is that in a solvable problem the space of possible solutions is extremely large, or even infinite. The nature of the problem is to find the solution which is assumed to exist as quicky as possible. Sophisticated control strategies, such as debugging <Sussman 1973>, are needed to improve efficiency of the search.

Our domain is quite different. We have described what we are doing as "world modelling" and "commom sense reasoning", but perhaps a better name would be deductive information retrieval. In this domain we have no reason to expect success when we ask a system to try to deduce some goal. In fact, since we are particularly concerned with incomplete knowledge, we must be prepared for the answer "I don't know". So in many cases the entire problem space must be searched. This has two important consequences. First, the problem space must be finite. In true information retrieval situations this is not a serious limitation. In border-line cases, an arbitrary cut-off may have to be set. More significantly, the order in which the problem space is searched is relatively unimportant; what is important is restricting the size of the space. This is virtually the opposite of the robot problem solving situation.

In searching a problem space we have a number of choices to make which will affect how the space is searched. We will assume that at any point we have a set of conjunctive goals, any one of which is sufficient to satisfy our top level request. The choices we have to make are:

1. Which goal to attempt first.
2. Which component of the selected goal to attempt first.
3. What assertions are relevant to solving the selected component.
4. Which of the relevant assertions to try first.

Choices 1. and 4. affect only the order in which the space is searched, and so are relatively unimportant. Choices 2. and 3., on the other hand, affect the _size_ of the search space as well. So far in this paper we have concentrated on 3. The distinctions between antecedent and consequent processes, facts and procedures, and identified and unidentified expressions, all serve to organize a data base so that we will not be faced with overwhelming numbers of assertions being generated or selected as being relevant to a goal. Beyond that, we have confined ourselves to eliminating redundancies in the search procedure. We now turn our attention to 2.

Choice 2. affects the size of an exhaustive search in two ways. If the goal is (AND (P ?X) (Q ?Y)), and the goal can be proved, it does not matter whether we solve (P ?X) first or (Q ?Y) first. Since these subproblems are independent, the solution of one will not affect the solution of the other. If, on the other hand, one of these subproblems fails we would not have to look at the other one at all. This would reduce the search space, of course. Suppose (AND A B) is a goal where A and B are independent, and we have estimates s and p for the size of the search space and probability of success for each component. Then the expected cost of trying A first is s(A)+p(A)*s(B), and the expected cost of trying B first is s(B)+p(B)*s(A). Naturally, we want to choose the order which minimizes this cost. Some special cases are of interest:

1. As pointed out above, if p(A) = p(B) = 1, it does not matter which order we try.

2. If p(A) << 1, p(B) << 1, then pick whichever subproblem has the smallest search space.

3. If we have no estimate for p, then assume p(A) = p(B) and pick whichever has the smallest search space.

4. If we have no estimate of s, then assume s(A) = s(B) and pick whichever has the least probability of success.

This analysis generalizes easily to goals with more than one component, or independent sets of dependent components.

There is one other point we need to make about independent subproblems. If A and B are independent, and we are solving (AND A B) left to right, once we have solved A, no subsequent failure should be allowed to backtrack into A. Contrast this with the following Micro-Planner program.

```
(THPROG (X Y)
     (THGOAL (P $?X))
     (THGOAL (Q $?Y)))
```

Suppose we execute this program and find a value for $?X that makes (P $?X) succeed. Now suppose we cannot satisfy (Q $?Y). We will then backtrack and try to find another value of $?X. If we are using this program as a way of proving ($\exists xy$ ((P x) $\wedge$ (Q y))), this cannot possibly help. Micro-Planner cannot know this, however, since solving the goal (P $?X) might have side affects which are useful in solving (Q $?Y). This is the result of THGOAL being ambiguous between "achieve" and "prove".

Since our goals are purely deductive, we have no side effects to worry about. So we will not allow backtracking between independent parts of a goal. To be more precise, construct a graph in which each conjunct is a node and two nodes are connected if they have a variable in common. The maximal connected subgraphs are independent sets of dependent conjuncts. This definition covers cases like (AND (P ?X) (Q ?Y) (R ?X ?Y)), where the first two conjuncts would be independent unless they were conected by the third conjunct. To see that they are dependent, consider that the failure of (Q ?Y) might be due to a failure of (R ?X ?Y) for some particular value of ?X and all values of ?Y. In that case we would want to backtrack from (Q ?Y) to (P ?X).

It may seem that all this emphasis on independent goals is unnecessary since most top level requests would probably be dependent, like "Is there a red block in the box?" This could be represented by (AND (BLOCK ?X) (RED ?X) (IN ?X BOX)). Notice, however, that once we have picked a particular block to try, say A, the goal we have left is (AND (RED A) (IN A BOX)). If A is red but not in the box, we would not want the failure of (IN A BOX) to send us searching for a different method of proving (RED A).

It should be noted that resolution systems are capable of solving this problem by use of subsumption. In resolution, the goal $(A \wedge B)$ is represented by the formula $((\neg A) \vee (\neg B))$. If A and B are independent and A is solved, we will generate the formula $(\neg B)$. $(\neg B)$ subsumes $((\neg A) \vee (\neg B))$; so the latter may be deleted, and no more attempts, to prove A will be made. The advantage of partitioning a goal into independent parts is that we know in advance that any solution to one part will generate a goal which subsumes its ancestor, so the subsumption check can be skipped and the deletion made automatically.

We turn now to the question of goals with dependent parts. We mentioned before that the order of solving a conjunctive goal affects the size of the search space in two ways. The second and most important way is in the ordering of the dependent parts of a goal. Suppose we want to find a red block, (AND (RED ?X) (BLOCK ?X)). If we know of n red things and m blocks, then looking for red things and checking if they are blocks will yield a search space proportional to n; looking for blocks and checking whether they are red yields a search space proportional to m. This suggests the following rule for solving conjunctive goals which Kowalski <1974, p. 50> calls "the principle of procrastination":

> To solve (AND $A_1 ... A_n$) where $A_1 ... A_n$ are dependent, first solve the A_i which has the smallest number of solutions.

This strategy for ordering goals is optimal so long as one condition can be met - the cost of generating a solution to A_i must be approximately the same as checking whether something is a solution to A_i. We will postpone examining what happens when this condition fails, and consider another question first. When we talk about the principle of procrastination being optimal, that is in the context of a backtracking control structure. Is there some other type of control structure which allows even more efficient searching? In a modification to his theorem prover, Nevins suggests the following procedure for solving conjunctive goals <Nevins 1974b>: If (AND $A_1 ... A_n$) is a goal, independently generate all the solutions to $A_1 ... A_n$, and intersect these solutions to find the solution to the entire goal. Nevins is aware that this could be inefficient if the number of solutions to one of the goals is very much greater than some of the others, and he has ways of handling this problem. But what about the more reasonable cases where the number of solutions of each goal is about the same order of magnitude?

The answer to this question depends on how efficiently we can compute set intersections. One very efficient way to compute the intersection of two sets, if they are hash-coded, is to enumerate the smaller one and look each member up in the hash-table of the other set. In fact, if we do not have to include the cost of building the hash-tables, there could not be a significantly more efficient method. The cost of this method is linear in the size of the smaller set, and to intersect two sets we certainly have to look at all the members of at least one of them. If we have a data base indexed by hash-coding, this procedure is essentially the same as the principle of procrastination, i.e. generate the smallest set, and test each element to see if it is in all the others.

This discussion points out one case where the optimality condition on the principle of procrastination holds, and the principle is easy to apply as well. That case is where the goal matches only atomic assertions and no procedures. The optimality condition is satisfied because generating a solution and checking a solution each require only one data-base probe. The principle is easy to apply because the number of solutions to each component is simply the number of assertions which match it.

When a goal matches a procedure, things become more complicated. First, we cannot always apply the principle of procrastination, since in general the only way to tell how many solutions a procedure will produce is to run it. We would have to have auxiliary knowledge giving us an estimate of the number of solutions. Second, there is no reason to expect the optimality condition to hold. An example where the optimality condition does not hold and the principle of procrastination does not work is the following: Suppose we have an n^{th} degree polynomial equation and a list of m numbers. We want to know whether any element of the list is a solution to the equation. If $m > n$, the principle of procrastination would have us solve the equation and then check the solutions against out list. But unless $m >> n$, that is clearly absurd. In general, it is far easier to check a possible solution to a polynomial equation than to solve the equation (especially for degree > 4!).

Of course, solving arbitrary polynomial equations hardly falls into the category of information retrieval problems. In fact, most of the other domains for deductive reasoning which we mentioned have the feature that solutions are much easier to check than to find. We might want to take this criterion as being what distinguishes "problem solving" domains from "information retrieval" domains.

The optimality condition can fail in the other direction as well. Perhaps surprisingly, there are some goals for which it is easier to generate solutions than to check solutions. These are goals for which there is a "generator", a procedure which generates solutions in a well-defined sequence, and for which the only way of checking solutions is to see if they are in the sequence produced by the generator. There are cases of this which clearly <u>do</u> fall under the category of deductive information retrieval, so this is a problem we have to deal with. Suppose we define ABOVE recursively in terms of ON:

```
((ABOVE ?X ?Y) <- (OR (ON ?X ?Y)
                      (AND (ON ?X ?Z)
                           (ABOVE ?Z ?Y))))
```

This can be interpreted as "?X is above ?Y if ?X is on ?Y or ?X is on something which is above ?Y".

Suppose we want to test whether A is above something red - (AND (ABOVE A ?X) (RED ?X)). If we attack this goal in the order given, we will simply enumerate all the things A is above and check to see if it is red. If all color assertions are stored directly, checking for redness will take only one step. Now suppose we try the other order, generating red things and testing to see whether A is above them. To check (ABOVE A B) we ask "Is A on B? If not, is whatever A <u>is</u> on, on B? If not, is whatever <u>that</u> is on, on B?" etc. If there are n things which A is above and m red things, the cost of attacking the goal the first way will be proportional to n; the cost of the second way will be proportional to m*n. The principle of procrastination is completely irrelevant here. The first method will be more efficient no matter what the values of m and n are.

The reason that generators behave this way is that since they produce their solutions one at a time, they prevent us from taking advantage of the parallelism of our hash-coded data base. As a result, if we want to find the intersection of two generators, say (AND (P ?X) (Q ?X)), using our control structure, the best we can do is a search proportional to m*n.

In this situation Nevins's idea is worth reconsidering. Nevins, himself, suggests nothing more sophisticated than a simple intersection technique which also requires m*n steps. If m and n are large enough, however, the extra expense of building and using a hash-table is warranted. To be specific, suppose the cost of generating a P is a_1 and there are n P's, the cost of generating a Q is a_2 and there are m Q's, the cost of storing an item in the hash table is h_1, and the cost of looking up an item in the hash table is h_2. Using backtracking, the cost of finding all the solutions to (AND (P ?X) (Q ?X)) would be a_1n*a_2m, whichever order we use. If we build a hash table of P's and look up the Q's in it, the cost will be $(h_1*a_1n)+(h_2*a_2m)$. Since a_1, a_2, h_1, and h_2 are constant, the second method will be more efficient for sufficiently large values of m and n.

We have two other points to make about control structure. So far we have assumed that once we select a component of a goal, we will obtain one complete solution to that component before going on to the next component. In practice, it is sometimes useful to interleave subgoals of one component with subgoals of another. Suppose we want to find a red cube, where a cube is defined to be an equidimensional brick. If there are more red things than bricks, we will start by looking for cubes:

```
                (AND (CUBE ?X) (RED ?X))
                          |
                          |
      (AND (BRICK ?X) (EQUIDIM ?X) (RED ?X))
```

Once we have selected a particular brick, A, we will be left with the goal (AND (EQUIDIM A) (RED A)). But (RED A) can be checked simply by look-up, where (EQUIDIM A) requires expanding a definition and some computation, so we want to reverse the order of these two goals. Effectively, we want to reduce (AND (CUBE ?X) (RED ?X)) to (AND (BRICK ?X) (RED ?X) (EQUIDIM ?X)), where (RED ?X) is interleaved between the subgoals of (CUBE ?X). Notice that our representation for conjunctive goals makes this interleaving particularly easy. All we have to do is reorder our list of conjuncts. Nevins and Reiter, on the other hand, use splitting for conjunctive goals (as well as disjunctive assertions). This puts the different conjuncts into different data bases, which would make interleaving subgoals extremely difficult.

Our final point about control structure has to do with distinquishing input and output variables in procedures. In its simplest form, a procedure in a PLANNER-like language makes no distinction between input and output variables. ((P ?X ?Y) <- (EXP ?X ?Y)) can be invoked either by the goal (P A ?Z) or the goal (P ?Z A). Kowalski makes the same point about the procedural intepretation of predicate calculus <Kowalski 1974, p. 61>. Often this works very well and leads to a very compact representation of what would be several different procedures in an ordinary programming language, but there are cases where it leads to trouble.

Recall the definition of ABOVE which we used previously. Notice that we could have equally well written the definition this way:

```
((ABOVE ?X ?Y) <- (OR (ON ?X ?Y)
                      (AND (ON ?Z ?Y)
                           (ABOVE ?X ?Z)))
```

Both procedures define ABOVE recursively in terms of ON. The difference is that the first definition follows the ON-chain down from the (presumably) higher block, and the second definition follows the ON-chain up from the (presumably) lower block. Either of these procedures would work for checking (ABOVE A B), but only the first is suitable for enumerating the things A is above, and only the second is suitable for enumerating things above A.

To see what happens when we match a goal with the wrong procedure, consider the goal (ABOVE A ?X) and the second procedure. We start off enumeration with the object A is on, which is fine. Then we are left with the goal (AND (ON ?Z ?Y) (ABOVE A ?Z)). If we attack the goal in the order given, we will start finding random instances of (ON ?Z ?Y) totally unrelated to A. If we reverse the order, we start with (ABOVE A ?Z), but this is essentially our original goal, so we are in danger of infinite recursion.

Kowalski <1974, p. 73> presents an even more striking example. Consider the relation (SORT ?X ?Y) to mean ?X is a sorted version of ?Y. The simplest way to define SORT is as an ordered permutation, ((SORT ?X ?Y) <- (AND (PERM ?X ?Y) (ORD ?X)). This procedure would be fine as a method of checking (SORT [134] [341]), but it would be a terrible sorting procedure. Not even interleaving subgoals of PERM and ORD would help. It seems that what we need for SORT, ABOVE, and similar procedures are restrictions on variables forcing them to be input variables or forcing them to be output variables. Restrictions similar to these have been implemented in CONNIVER <McDermott and Sussman 1974>.

The moral of this chapter is that it is essential to have the semantics of the domain influence the control of the deductive process, but we do not as yet have a language for exercising such control. So, what is the point? Many other authors have said much the same thing. The point is that, at least for one type of problem domain, we have specified where we need control and what kind of control we need. Also, we have specified where we do _not_ need domain dependent knowledge for control. We do not need to worry about which of a group of alternative goals to pick first, or which of a group of alternative methods to try first. And for conjunctive goals we have a general ordering procedure (the principle of procrastination) which works whenever only atomic assertions are applicable. Things only become complicated when we have procedures matching a conjunctive goal.

Languages which do have control primitives have sometimes controled the wrong things. Micro-Planner had recommendation lists which allowed ordering alternative procedures, which does not matter very much. It did not have dynamic ordering of conjunctive goals, however, which matters a great deal. Kowalski <1974, p. 50> gives the impression that he believes the principle of procrastination always gives the optimal ordering of conjunctive goals, and that the only problem is that it can only be estimated rather than computed. Finally, Kowalski and Hayes both argue for a completely autonomous control language. We have tried to show, on the other hand, that in distinguishing between antecedent and consequent

procedures, and between input and output variables, the control should be imbedded in the assertions themselves.

6. Conclusions - Extensions and Limitations

This thesis has been about the application of procedural deduction systems to a limited class of problems, what we called in the preceding chapter "deductive information retrieval". We believe that no system will ever be suitable for expressing and solving every type of problem, and we have made no attempt to design a system that will. Nevertheless, as we pointed out in the first chapter, it is a truism that "programming languages are universal, so they can express anything". So it is difficult to say what the limitations of the current ideas are. For example, we pointed out that the control structure we envision is specifically not designed to cope with robot problem solving. The most natural way to describe plans for achieving (as opposed to proving) conjunctive goals would be:

```
((PLAN-FOR ?X [AND ?Y ?Z]) <- (AND (PLAN-FOR ?X ?Y)
                                   (PLAN-FOR ?X ?Z)))
```

That is, ?X is a plan for ?Y and ?Z if ?X is plan for ?Y and ?X is a plan for ?Z. This is a perfectly good way for checking plans, but a terrible way to generate them. The problem seems to be that deductive languages are purely applicative; they cannot make assignments to variables, they can only bind them. Effective problem solving requires the ability to take an almost correct plan and modify it to be a correct one (see <Sussman 1973>). This idea seems to require assignment.

Notice, however, that we can use the following hack:

```
((PLAN-FOR ?X [AND ?Y ?Z]) <- (AND (PLAN-FOR ?U ?Y)
                                   (PLAN-FOR ?V ?Z)
                                   (COMBINE ?X ?U ?V [AND ?Y ?Z])))
```

This would mean to find a plan for ?Y and ?Z, find a plan for ?Y and a plan for ?Z, and combine them to be a plan for ?Y and ?Z. Warren <1974> uses a similar technique in a planning system written in predicate calculus. The trouble with this is that it has lost all naturalness of the previous assertion. What we have done is to abandon using the language as a straight-forward representation system, and we have begun to use it just as a general-purpose programming language. Whether having a clean interface to the real deductive system outweighs its limitations as a programming language is largely a matter of taste. At any rate, the question is not so much what is possible, but rather what is appropriate.

With this in mind we will end with a list of undeveloped ideas which seem to be useful additions to the procedural deduction paradigm as we have described it. Whether to regard these ideas as limitations or extensions of what has gone before is an open question, but we will list them in order of how easily they appear to fit into the current paradigm.

1. Returning answers

The techniques we have used in this thesis fall short of most procedural deduction systems in at least one respect. They are not very good about keeping track of the correspondence between variables in the top-level goal and those in its subgoals. This makes returning answers a problem. Even if this were cleaned up, the reasoning by cases mechanisms make the final value of these variables of questionable utility. One easy solution to this problem would be to use the ANSWER predicate of Green <1969>. Whether this approach is ultimately satisfactory requires further study.

We can make one improvement on both PLANNER and Green's system, however, thanks to out notion of identified expressions. Suppose we know that John has a telephone number, but we do not know what it is. We might represent this as (PHONE-NUM JOHN G6907). If we ask the system what John's phone number is, G6907 would be a most un-helpful answer. We really want answer variables to be restricted to matching identified expressions, with an attempt made to evaluate any unidentified expression that would otherwise match.

2. Modelling change

One of the most pressing problems is the further development of ideas about modelling change. The most serious limitation of the ideas in Appendix A is that they do not handle facts which depend on other facts. We currently assume that an action knows about all the relations which it might affect. But we might want to define a new concept in terms of old ones, without changing the specification of the actions which would affect it. We need to have something like PLANNER erasing theorems extended to our more complex domain.

3. Control language

In the preceding chapter, we pointed out where auxiliary control information is needed to order conjunctive subgoals. We have no language for expressing this information, however. McDermott <1974b> is currently working on this problem, and we hope to make use of his results.

4. Goal filtering

Another valuable feature would be the ability to reject goals which are self-contradictory, or more generally, goals which are easily shown to be false. Resolution systems have some limited capabilities in this regard. A self-contradictory goal is represented in a resolution system by tautologous assertion, which is easy to test for. Also, if a goal is contradicted by a single assertion, the subsumption procedure will delete it. In other cases, however, some deduction may be required. Suppose we could estimate the size of the space to be searched in trying to prove a goal, and also the size of the search space for its negation. If the latter were significantly smaller, it would be worth trying to disprove the goal first, before attempting to prove it.

The use of models as goal filters in geometry theorem provers

<Gelernter 1963> <Reiter 1973> represents a kind of computational filtering, as opposed to deductive filtering. This type of filtering is more efficient, but not universally applicable. It requires the ability to easily construct a model, and most important to be able to easily distinguish the accidental vs. essential features of that model. In geometry problems this is easy. The numerical values for line lengths and angle sizes (except for 90 and 180 degrees) are almost always accidental. Similarly, in electronic circuit models, the exact values for components could be accidental. But in most common sense domains, like the blocks world, we know and ask about a lot of specific information of this kind, so constructing filtering models would be much more difficult.

5. Repeated goals and shared partial goals

One ability any deductive system needs is to be able to recognize a goal that has been generated before. This always saves computational effort, and when a goal is generated as a subgoal of itself, it prevents the simplest kinds of infinite recursion. In resolution systems, subsumption serves this function also.

There is a related problem, however, which subsumption does not handle. Suppose we have (AND (P ?X) (Q ?X)) as a goal and later we generate (AND (P ?X) (R ?X)). Neither of these goals subsumes the other, but they do have the component (P ?X) in common. If we try to solve the second goal in the usual way, we will be solving (P ?X) twice. We can improve on this by remembering all the solutions generated by (P ?X) the first time (i.e. assert them in the data base), and on subsequent goals containing (P ?X) refrain from using procedures or complex assertions.

There are two complications to this procedure. First, suppose another goal like (AND (NOT (P A)) EXP) intervened between the first and the second goals containing (P ?X). The second (P ?X) will have a possible solution via Restricted Goal Resolution, which was not available to the first (P ?X). Second if we interleave subgoals of (P ?X) and (Q ?X), we may not have generated all the solutions to (P ?X). Some possible solutions to (P ?X) may have been rejected because they failed to satisfy some subgoal of (Q ?X). A system that attempts to do goal-sharing would have to take these problems into account.

6. Recognizing goals

All of the deductions we have done, have consisted of breaking down goals into (hopefully) easier subgoals. Sometimes it would be helpful to be able to reverse the process. In the preceding chapter we pointed out that (AND (PERM ?X [3 1 2]) (ORD ?X)) is a very poor way to sort the list [3 1 2]. But suppose these two goals came together by accident, perhaps by being generated as subgoals of two other goals, or by an application of RGR. We would want the system to say, in effect, "Aha! This is a sorting problem," and transform this goal into (SORT ?X [3 1 2]). A HACKER critic does much the same thing when it sees (AND (ON A B) (ON B C)), and recognizes it as a stacking problem.

7. Plausible inference

One common criticism of deductive systems is that they can do only logically rigorous inferences, not the heuristic, plausible inferences of common sense reasoning. As Collins <1975> says:

> There are negative tricks, functional tricks, visual imagery tricks, inductive tricks, and undoubtedly many more that people use to circumvent the holes and uncertainties in their knowledge. These all lie outside the deductive logic of which the advocates of theorem-proving and the predicate calculus are so fond.

This criticism really misses the point, however. What is the difference between the rigorous implication $(A \supset B)$ and the plausible implication $(A \supset (\text{PROBABLY } B))$? Very little, really. The only significant difference that I can see, is that with the plausible inference if we run into a contradiction we know where to look for trouble. If no contradictions are encountered the two types of inferences behave essentially the same. So, a plausible inference is simply one that we know how to debug.

Debugging inferences is by no means an easy problem. The only significant work on the question seems to be that of McDermott <1974a>. Collins has reported no such mechanism in his SCHOLAR system. So for all his complaints about purely deductive logic, he might as well have used a purely deductive system himself.

8. Reasoning about knowledge and belief

Another criticism of traditional deductive systems, and one which is justified, is Minsky's <1975> observation that they are monotonic. This means that if A can be deduced, there is no further knowledge which can be added that will make A not deducible. This is quite unlike human reasoning. If we decide that it is safe to cross the street, and it is pointed out to us that a car is coming, we are perfectly capable of retracting our previous inference. Having a mechanism for debugging plausible inferences would help, but I believe that an all out attack on this problem requires powerful methods for reasoning about one's own knowledge and beliefs. E.g., it is not adequate simply to always believe that it is probably safe to cross the street. We have to examine our knowledge to see if we believe anything that would be evidence against the safety of crossing the street. We might express this formally as:

```
((SAFE (CROSS I STREET)) <-
        (NOT (DEDUCIBLE (PLAUSIBLE (NOT (SAFE (CROSS I STREET)))))))
```

PLAUSIBLE looks around for reasons why it might not be safe to cross the street. DEDUCIBLE says whether any were found, so if none were found it is safe to cross the street.

If the semantics of PLAUSIBLE are unclear, the semantics of DEDUCIBLE are even less clear. DEDUCIBLE cannot be simply a recursive call to the deductive system. In one way or another, we have hidden in there all the traditional philosophical problems of epistemic logic <Linsky 1971>. For example, suppose we want to say something as simple as "I know everything that is on the table." We could represent this as ((ON ?X TABLE) -> (DEDUCIBLE (ON ?X TABLE)). Suppose that the objects we know are on the table are A, B, and C. If someone asserts that some unidentified object G7302 is on the table we will infer (DEDUCIBLE (ON G7302 TABLE)). If we interpret DEDUCIBLE too literally, we will produce a contradiction since (ON G7302 TABLE) was not previously in the data base. What we want to infer is (OR (G7302 = A) (G7302 = B) (G7302 = C)). This is just the traditional problem of "quantifying in" in computational form.

9. Non-inferential invocation of knowledge

The final limitation of traditional deductive systems which we will look at is the following: The only way one goal or assertion can invoke another is if there is an <u>inferential</u> connection between them. A goal A invokes another goal or assertion B only if $(B \supset A)$. An assertion A invokes an assertion B only if $(A \supset B)$. But there are other useful control paths as well. In an active (as opposed to a passive) recognition system, <u>assertion</u> of some key features would trigger the <u>goal</u> of recognizing a particular object. In a system for reasoning about actions, a goal involving a certain action could trigger <u>antecedent</u> deductions to deduce the state of the world if that action were performed. Then the goal would be solved in a consequent-directed manner in the new state.

Appendix A - Modelling Change

This section is an early discussion of an approach to reasoning about change, given incomplete knowledge. It is included as an appendix rather than as part of the main body of the thesis because these ideas still require much more development. There are no guarantees that these ideas are completely compatible with the rest of the thesis, but I believe the basic approach is sound.

As we mentioned before, one of the first problems that arises in modelling change, when incomplete knowledge is introduced, is expressing facts that involve more than one state of the world. Devising a notation is not hard. We will introduce an operator "S" which takes two arguments, the name of a situation, and an assertion or goal. For example, (S FOO (ON A B)) would mean that A is on B in situation FOO. We can then, of course, build up expressions that mention more than one situation, e.g., ((ON ?X ?Y) -> (S (MOVE ?Y ?Z ?W) (ON ?X ?W))). This could mean that if ?X is on ?Y in the current situation, then in the situation which results from moving ?Y from ?Z to ?W, ?X will be at ?W.

We will implement situations as contexts, so asserting (S FOO BAR) would assert BAR in situation context FOO, and the goal (S FOO BAR) would be solved by proving BAR in context FOO.

So far we have not said very much. We have introduced a notation which is essentially a variant of McCarthy's situation calculus <McCarthy 1968>. We differ from McCarthy, however, in assuming that the system will have a notion of "current situation", and that expressions not in the scope of a situation operator will be interpreted as applying to that situation. The importance of this will be seen later.

The serious problem that we have is to make use of this notation so that, given an incomplete description of a situation and a (possibly incomplete) description of the effects of an action, we can express what we know about the state which results from performing that action in that situation. This is the frame problem.

Our approach to the frame problem is an extension of the PLANNER approach. We start from the observation that when an action occurs, it is normally easier just to specify the changes, than it is to give a complete description of the new situation. So at the simplest level, to record the results of an action, we make a copy of the current situation context, delete the assertions that can no longer be relied upon, and add the assertions that we know the action makes true. The copy of the current situation context could either be a distinct data structure, or it could be a virtual copy, as is produced by the CONNIVER context mechanism. This is where the difference between our notation and McCarthy's is significant. If every statement had to have a situation operator, we would have to say something like (S S0 FOO) to express that FOO is currently true. But then we would have to have a different statement to express that FOO was true in the succeeding situation. This blocks one situation context from inheriting assertions from its predecessor.

The problem comes down to specifying the additions and deletions.

Additions are relatively straight-forward. A name is created for the new situation, and additions are made to that context. Conditional additions are made as follows: if A is to be added to the new situation S1 on condition B, then (B -> (S S1 A)) is added to the old context. This differs from the usual PLANNER method where the condition is evaluated at the time the new situation context is created. Since we are dealing with incomplete knowledge, we need to leave this fact as an implication. We also need to let these implications combine with more complicated expressions. For example, if we knew ((ON A B) <- (S S1 (FOO A B))) and we performed (MOVE B PLACE1 PLACE2), we would know ((LOC A PLACE1) <- (S S1 (FOO A B))) in the resulting situation.

Deleting assertions is more complicated. Conditional deletions will be expressed by something like (REMOVE FOO UNLESS BAR). The use of "UNLESS" is significant because, in order to keep the data base sound, an assertion must be deleted, or at least modified, whenever there is any doubt about its remaining true. To be more specific:

> If (REMOVE (P ?X) UNLESS (Q ?X)) is activated in situation S1, where (P A) is asserted, (P A) will be retained in the new situation context if (Q A) can be shown, (P A) will be deleted if (NOT (Q A)) can be shown, and (P A) will be replaced by ((P A) <- (S S1 (Q A))) otherwise.

As an example, take the MOVE operation we have been considering. The other side of the addition rule we looked at is the following deletion rule:

```
For (MOVE ?X ?Y ?Z), (REMOVE (LOC ?W ?Y) UNLESS (NOT (ON ?W ?X)))
```

This says that if we move ?X from ?Y to ?Z, then for anything else at ?Y, we have to remove that fact, unless we can show that it is not on ?X. So if we know (LOC A PLACE1) and (LOC B PLACE1), and we (MOVE A PLACE1 PLACE2) in situation S1, we would end up with:

```
((LOC B PLACE1) <- (S S1 (NOT (ON B A))))
((LOC B PLACE2) <- (S S1 (ON B A)))
```

In deleting facts other than simple assertions, there are two sources of complexity - first, quantification and identity, and second, logical connectives. Problems of quantification and identity arise when the pattern being deleted might match, but is not identical to the pattern of the assertion. The troublesome cases are when the removal pattern has a constant where the assertion has a variable; and when the removal pattern and the assertion have non-identical constants which might be equal. For the first case, suppose we know that everything is in the box. If we move A, we want to remove the fact that A is in the box. So we might have to match (REMOVE (LOC A BOX)) against (LOC ?X BOX). We cannot simply delete (LOC ?X BOX), since we still know that everything except A is in the box. So we have the following rule:

> If the removal pattern is of the form (P A1...An) and the assertion is of the form (P ?X1...?Xn), then the assertion is replaced by ((P ?X1...?Xn) <- (OR (NOT (?X1 = A1))...(NOT (?Xn = An)))).

In the example, this would leave us with ((LOC ?X BOX) <- (NOT (?X = A))).

The second case is similar. If we know (LOC A BOX) and we want to remove (LOC G1 BOX) where G1 is an unidentified constant, then unless we know that G1 is not the same as A, we can no longer rely on (LOC A BOX). Instead we have to replace it by ((LOC A BOX) <- (NOT (G1 = A))). So the rule in its most general form would be:

> If the removal pattern is of the form (P A1...An) and the assertion is of the form (P B1...Bn), where for all i, $1 \le i \le n$, (1) Ai and Bi are both constants and at least one of them is unidentified, or (2) Ai is a constant and Bi is a variable, the assertion is replaced by ((P B1...Bn) <- (OR (NOT (B1 = A1))...(NOT (Bn = An))))

The other source of complexity is logical connectives. If an action has the effect of removing A, how does it affect (A <- B), (A -> B), or (OR A B)? (It should be obvious that (AND A B) is not a problem.) The first observation is that removing A may have no effect on these other statements at all. If we inferred A from B and (B -> A), we do not know <u>a priori</u> whether to also remove B or (B -> A). It depends on whether the connection between A and B is causal - B causes A - or accidental - we just happen to know that in this situation A is true if B is true. If the connection is causal, then when A is removed, B must be removed. If the connection is accidental, then B is unaffected, but (B -> A) must be removed.

So, we will classify complex assertions as either causal or accidental, replacing ->, <-, and OR with $->_A$, $->_C$, etc. For instance, the fact that bachelors are unmarried might be expressed as ((BACHELOR ?X) $->_C$ (NOT (MARRIED ?X))), since if someone stops being unmarried, he stops being a bachelor. But the fact that A is the only widget in the world would have to be expressed as ((WIDGET ?X) $->_A$ (?X = A)), since this assertion would become false if we built another widget. In fact, if we require that a constant always refer to the same object, any assertion involving equality will be accidental, since no action can affect the fact that an object is identical only to itself. For this reason, the equality and inequality tests generated by pattern matching will always be accidentally related to the expressions that produced them. E.g., if we have the assertion (LOC ?X BOX) and we remove (LOC A BOX), we get ((LOC ?X BOX) $<-_A$ (NOT (?X = A))).

How, then, will we handle the removal of an accidental assertion? The two cases we have to consider are $<-_A$ and OR_A (since $->_A$ is reducible to $<-_A$). For <-, since (P $<-_A$ Q) will be invoked only by a goal of P or an assertion of (NOT P), (P $<-_A$ Q) will be indexed only by P. This means that if (NOT Q) is removed, the system will not notice that (P $<-_A$ Q) should be removed also. (Recall that removing (NOT Q) means that Q might have just become true. Hence, it might now be the case that P is false and Q is true.) For this reason, in accidental implications the right-hand side must always be bound to the situation in which it was asserted. So, asserting (P $<-_A$ Q) in situation S1 will actually result in adding (P $<-_A$ (S S1 Q)) to the data base. In this form, the assertion remains true as long as P is not removed. A similar argument would show that (P $->_A$ Q) should be asserted as (P $->_A$ (S S1 Q)). Equalities and inequalities are an exception to this rule, however. Since no action can affect their truth value, they do not need to be bound to a situation. So ((LOC ?X BOX) $<-_A$ (NOT (?X = A))) can stay in that form.

We can handle (OR_A P Q) similarly. Since OR creates multi-pattern consequent theorems, we will just treat them as such. (OR_A P Q) would be replaced by (P $<=_A$ (S S1 (NOT Q))), and (Q $<=_A$ (S S1 (NOT P))), where <= is like <-, except it is used only in a goal-directed way.

To illustrate these rules we use the following example: Suppose we know that A is either at PLACE1 or at PLACE2. If we move everything at PLACE1 to PLACE2, how do we show that A is at PLACE2? First we define the action MOVE-ALL. (MOVE-ALL ?X ?Y) adds (NOT (LOC ?Z ?X)), and adds (LOC ?Z ?Y) if we have (LOC ?Z ?X). That is, nothing is now at ?X, and anything that was at ?X is now at ?Y. It removes any assertion that anything is at ?X, (LOC ?Z ?X), and assertions that ?Z is not at ?Y, unless ?Z was not at ?X, ((NOT (LOC ?Z ?Y)) UNLESS (NOT (LOC ?Z ?X))).

Now suppose we perform (MOVE-ALL PLACE1 PLACE2) in situation S1 where ((LOC A PLACE2) $<-_A$ (S S1 (NOT (LOC A PLACE1)))) is asserted, producing situation S2. S1 and S2 then contain the following assertions:

S1: ((LOC A PLACE2) $<-_A$ (S S1 (NOT (LOC A PLACE1))))
    ((LOC A PLACE1) $->_A$ (S S2 (LOC A PLACE2)))

S2: ((LOC A PLACE2) $<-_A$ (S S1 (NOT (LOC A PLACE1))))
    ((LOC A PLACE1) $->_A$ (S S2 (LOC A PLACE2))))
    (NOT (LOC ?Z PLACE1))

We can prove (LOC A PLACE2) in S2 if we can prove (NOT (LOC A PLACE1)) in S1. Since the second assertion in S1 is equivalent to ((NOT (LOC A PLACE1)) $<-_A$ (S S2 (NOT (LOC A PLACE2)))), we can generate the subgoal (NOT (LOC A PLACE2)) in S2. But this is the negation of our original goal, so, by the rule for combining goals, [illegible]

Appendix B - Inheritance of Properties

In his current thesis progress report <Fahlman 1975>, Scott Fahlman focuses on one of the central questions in the representation of knowledge: How is it that when a person learns that an individual A is a member of a class C, all the properties of C's seem to be attached to A without effort? Fahlman's example is that when we learn that Clyde is an elephant, we can easily answer whether Clyde is gray, wrinkled, or fond of peanuts.

To see that this is a problem, consider how we might try to handle it in PLANNER. Specifically, suppose we have an assertion (ELEPHANT CLYDE) and a goal (COLOR CLYDE GRAY). PLANNER gives us two ways to express the fact that all elephants are gray. One way is to have an antecedent theorem which, whenever something is asserted to be an elephant, asserts that it is gray. We would presumably have to have similar theorems, however, for all other properties of elephants (wrinkles, liking peanuts, etc.). Worse yet, we have the problem of inheriting all the properties of superclasses of elephants (mammals, animals, physical objects, etc.). This suggests that asserting that Clyde is an elephant would cause hundreds or even thousands of additional facts to be asserted.

The other alternative is to have a consequent theorem which attempts to prove something is gray by proving that it is an elephant. But again we would have theorems saying sperm whales, battleships, etc. are gray. We would have to search through all the types of gray things until we stumbled upon elephants.

Fahlman's proposed solution to this problem is to represent these facts as a network stored in special purpose hardware. The hardware would allow markers to be propagated in parallel along superset links from the CLYDE node, while monitoring the GRAY node to see if a marker reaches it. Fahlman's system also facilitates finding the intersection of any group of sets, by propagating markers along subset links from the set nodes, and collecting the nodes which receive markers from all of the starting nodes.

We have a counter proposal to Fahlman's which, though less powerful (it cannot do arbitrary set intersections), still seems to solve the original problem, and is readily implemented on conventional hardware. In fact, our proposal only requires modifying a PLANNER-type pattern matcher.

The main idea is to use class markers on both constants and variables. The class markers on variables are somewhat like Micro-Planner's variable restrictions, but are more powerful. Then we will represent the fact that all elephants are gray by (COLOR ?x/ELEPHANT GRAY). This way we can make the pattern matcher do all the work, and fine tune the indexer to make it efficient.

The class markers will be chosen from a classification hierarchy. The idea of such a hierarchy is is familiar in psychology <Collins and Quillian 1969> and is receiving increasing attention in AI <Martin 1974>. Fahlman uses this idea as well, calling it the IS-A hierarchy.

An important point about the hierarchies we will want to use is that, while they may be very bushy, they are never very deep. Probably the most elaborate hierarchy in the real world is the taxonomy of animals. This hierarchy contains levels for species, genus, family, order, class, and phylum. Here and there, there are things like sub-species, sub-class, and sub-phylum. Above the phylum level, we might have markers such as ANIMAL, LIVING-THING, NATURAL-OBJECT, PHYSICAL-OBJECT, PHYSICAL-ENTITY, and finally just ENTITY. A classification hierarchy containing all these levels would still have a depth of only fourteen or fifteen. And surely common sense hierarchies are much shallower. COD IS-A FISH IS-A ANIMAL... is probably typical.

The pattern matcher we need is basically a PLANNER-type matcher with an additional type checker. A class marker will consist of a list of all the superclasses of the item to which the marker is attatched. So asserting (CLASS CLYDE ELEPHANT) will result in attatching to the constant CLYDE a list like (PHYSOB ALIVE ANIMAL MAMMAL ELEPHANT). Variables will have similar lists. The type checking will be done by comparing marker lists. Since the lists should be short, this will be easy to do. Two marker lists will match only if one list is an extension of the other. We will consider the cases that arise, assuming this condition has been met. We will use "variable" to mean unassigned variables; assigned variables act like constants.

1. Variable vs. variable - The variables are bound to each other as in PLANNER, and they both get the more restrictive marker list.

2. Constant vs. constant - As in PLANNER, the match succeeds only if the constants are identical.

3. Constant vs. variable

a. If the constant is more restricted than the variable, then the variable gets assigned the constant and takes on its additional restrictions.

b. If the variable is more restricted than the constant, the variable gets assigned the constant, but we generate an extra subgoal of showing that the constant meets the additional restrictions on the variable.

For some examples, consider the following: If we know (COLOR ?x/ELEPHANT GRAY) and we want to show (COLOR CLYDE/ELEPHANT GRAY) we can do this by straight-forward matching. (We are using ELEPHANT as shorthand for the list of markers mentioned above.) If we know (OCCUPY-SPACE ?x/PHYSOB) and we want to show (OCCUPY-SPACE CLYDE/ELEPHANT), we again succeed because comparing marker lists shows that ELEPHANT is a subclass of PHYSOB. Finally, suppose we know that Clyde is an animal, but not what kind. If we want to show (COLOR CLYDE/ANIMAL GRAY), we would match (COLOR ?x/SPERM-WHALE GRAY) and (COLOR ?x/ELEPHANT GRAY), generating subgoals (CLASS CLYDE/ANIMAL SPERM-WHALE) and (CLASS CLYDE/ANIMAL ELEPHANT). We would not match (COLOR ?x/BATTLESHIP GRAY), however, since BATTLESHIP is incompatible with ANIMAL. We would only match things that are compatible with what we already know about Clyde.

This idea can be extended to classification schemes which are not strict hierarchies. If we classify individuals along different dimensions, a class might have a tree or a partial order of superclass nodes. We could still do the same type of matching procedure, except we would be comparing these more complicated data structures rather than simple lists. This should work as long as the number of nodes above any given node remains small.

So far, it is not clear that we have gotten rid of our combinatorial problems, only that we have pushed them into the pattern matcher. But at this level, we can pretty much get rid of them by the simple expedient of indexing items by class markers, as well as by constants. There are probably many good ways of doing this, but we will sketch one simple scheme.

Suppose we add assertions to a bucket until it gets too large to search efficiently; then we subdivide it into smaller buckets. The trick is that whenever we subdivide a bucket, we do it in two ways, both by the next level of class specification of the current element of the item, and by the first level of class specification of the next element of the item. The following example should make things clear. Suppose we have a bucket with two items, (A/(BC) D/(E)) and (F/(BG) H/(I)). We would call this the 1-B bucket, because it is the one which results from sorting the first element by the classifier B. If we wanted to subdivide this bucket, we would do it by the next class level of the current element, producing the 1-BC and 1-BG buckets, and by the first class level of the next element, producing the 1-B,2-E and 1-B,2-I buckets.

There are two points of detail worth mentioning. First, if there were an item whose first element had only a B class marker, we would have also created a 1-B bucket which could not be broken down further by subclass. Second, when the class markers are exhausted, we can subdivide by constants as in PLANNER.

Using this scheme, if we have a goal we can go directly to a bucket containing all the items [illegible] markers are extensions of the ones in the goal. If we also pickup any buckets marked along the way, we will have looked at all the buckets which might contain an item matching the goal.

BIBLIOGRAPHY

Bledsoe, W. W. (1971) Splitting and Reduction Heuristics in Automatic Theorem Proving. Artificial Intelligence, 2: 55-77.

Bledsoe, W. W. and Bruell, P. (1973) A Man-Machine Theorem Proving System. Advance Papers of the Third International Joint Conference on Artificial Intelligence, 56-65.

Carbonell, J. R. and Collins, A. M. (1973) Natural Semantics in Artificial Intelligence. Advance Papers of the Third International Joint Conference on Artificial Intelligence, 344-351.

Chang, C. and Lee, R. C. (1973) Symbolic Logic and Mechanical Theorem Proving. New York: Academic Press, Inc.

Collins, A. M. and Quillian, M. R. (1969) Retrieval Time from Semantic Memory. Journal of Verbal Learning and Verbal Behavior, 9, 240-247.

Collins, A. M., Warnock, E. H., Aiello, N., and Miller, M. L. (1975) Reasoning from Incomplete Knowledge, in D. G. Bobrow and A. M. Collins (eds.) Representation and Understanding, 385-415. New York: Academic Press, Inc.

Davies, D. J. M. (1973) POPLER: A POP-2 PLANNER. School of Artificial Intelligence, University of Edinburgh, MIP-89.

Davies, D. J. M. (1974) Representing Negation in a PLANNER System. AISB Summer Conference, University of Sussex, 20-36.

Fahlman, S. E. (1975) Thesis Progress Report: A System for Representing and Using Real-World Knowledge. MIT Artificial Intelligence Laboratory, AIM-331.

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

Gelernter, H. (1963) Realization of a Geometry-Theorem Proving Machine, in E. A. Feigenbaum and J. Feldman (eds.) Computers and Thought, 134-152. New York: McGraw-Hill Book Company.

Green, C. (1969) Application of Theorem Proving to Problem Solving. Proceedings of the International Joint Conference on Artificial Intelligence, 219-239.

Grossman, R. W. (1975) Representing the Semantics of Natural Language as Constraint Expressions. MIT Artificial Intelligence Laboratory, WP-87.

Hayes, P. J. (1973) Computation and Deduction. Proceedings MFCS Symposium. Czech. Academy of Sciences.

Hewitt, C. (1972) Description and Theoretical Analysis (Using Schemata) of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot. MIT Artificial Intelligence Laboratory, AI-TR-258.

Hewitt, C. (1975) How to Use What You Know. Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, 189-198.

Kowalski, R. (1973) Predicate Logic as a Programming Language. Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Memo 70.

Kowalski, R. (1974) Logic for Problem Solving. Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Memo 75.

Linsky, L. (ed.) (1971) Reference and Modality. London: Oxford University Press.

Martin, W. A. (1974) OWL Notes. Unpublished MIT course notes.

McCarthy, J. (1968) Programs with Common Sense, in M. Minsky (ed.) Semantic Information Processing, 403-418. Cambridge, Mass.: The MIT Press.

McDermott D. V. (1974a) Assimilation of New Information by a Natural Language-Understanding System. MIT Artificial Intelligence Laboratory, AI-TR-291.

McDermott, D. V. (1974b) Advice on the Fast-Paced World of Electronics. MIT Artificial Intelligence Laboratory, WP-71.

McDermott, D. V. and Sussman, G. J. (1974) The CONNIVER Reference Manual. MIT Artificial Intelligence Laboratory, AIM-259a.

Minsky, M. (1975) A Framework for Representing Knowledge, in P. H. Winston (ed.) The Psychology of Computer Vision, 211-277. New York: McGraw-Hill Book Company.

Moore, R. C. (1973) D-SCRIPT: A Computational Theory of Descriptions. Advance Papers of the Third International Joint Conference on

Artificial Intelligence, 223-229.

Nevins, A. J. (1974a) A Human Oriented Logic for Automatic Theorem-Proving. Journal of the Association for Computing Machinery, 21: 606-621.

Nevins, A. J. (1974b) A Relaxation Approach to Splitting in an Automatic Theorem Prover. MIT Artificial Intelligence Laboratory, AIM-302.

Reiter, R. (1973) A Semantically Guided Deductive System for Automatic Theorem-Proving. Advance Papers of the Third International Joint Conference on Artificial Intelligence, 41-46.

Rich, C. and Shrobe, H. E. (1975) Understanding LISP Programs: Towards a Programmer's Apprentice. MIT Artificial Intelligence Laboratory, unpublished Master's thesis.

Rulifson, J. F., Derksen J. A., and Waldinger, R. J. (1972) QA4, a Procedural Calculus for Intuitive Reasoning. SRI Artificial Intelligence Center, Technical Note 73.

Sussman, G. J., Winograd, T., and Charniak, E. (1971) Micro-Planner Reference Manual. MIT Artificial Intelligence Laboratory, AIM-203a.

Sussman, G. J. and McDermott, D. V. (1972) From PLANNER to CONNIVER - a Genetic Approach. Proceedings of the Fall Joint Computer Conference, 1171-1179.

Sussman, G. J. (1973) A Computational Model of Skill Acquisition. MIT Artificial Intelligence Laboratory, AI-TR-297.

Warren, D. H. D. (1974) WARPLAN: A System for Generating Plans. Department of Computational Logic, School of Artificial Intelligence, University of Edinburgh, Memo 76.

Winograd, T. (1971) Proceedures as a Representation for Data in a Computer Program for Understanding Natural Language. MIT Artificial Intelligence Laboratory, AI-TR-235.

CS-TR Scanning Project
Document Control Form

Date : 4 / 4 / 96

Report # AI-TR-347

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 118 (125-images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☒ Other:

Check each if included with document:

☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other:

Page Data:

Blank Pages (by page number): FOLLOW TITLE PAGE & TABLE OF CONTENTS

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-118) UN#'ED TITLE PAGE, UN#BLANK, UN#ABSTRACT, UN#ACK, UN#TABLE OF CONT, UN#BLK, 1-112
(119-125) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 4 / 4 / 96 Date Scanned: 4 / 17 / 96 Date Returned: 4 / 18 / 96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-347	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Reasoning from Incomplete Knowledge in a Procedural Deduction System		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert Carter Moore		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0643
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE December 1975 13. NUMBER OF PAGES 120
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence	Procedural Deduction
Representation of Knowledge	Imcomplete Knowledge
Theorem Proving	Search Heuristics

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Procedural deduction languages, such as PLANNER, have been valuable tools for building models involving the notion of an explicit model of a problem situation. This report explores methods of increasing the ability of procedural deduction systems to deal with incomplete knowledge. The report examines in detail, problems involving negation, implication, disjunction, quantificiation, and equality. Control structure issues and the problem of modelling change under incomplete knowedge are also considered.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)