Technical Report 403

Use of Analogy to Achieve New Expertise

Richard Brown

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

USE OF ANALOGY

TO ACHIEVE NEW EXPERTISE

by

Richard Brown

April 1977

Revised version of a dissertation submitted to the Department of Electrical Engineering in February, 1977 in partial fulfillment of the requirements for the degree of Master of Science.

ABSTRACT

We will take the view that the end result of problem solving in some world should be increased expertness. In the context of computers, *increasing expertness* means *writing programs*. This thesis is about a process, reasoning by analogy, that writes programs.

Analogy relates one problem world to another. We will call the world in which we have an expert problem solver the IMAGE world, and the other world the DOMAIN world. Analogy will construct an expert problem solver in the domain world, using the image world expert for inspiration.

Analogy uses a map (the analogy map) from the expertise of the domain world to the expertise of the image world. Expertise in a world may be divided into components corresponding to (1) declarative description (in the predicate calculus), (2) code for computing the values of predicates and functions, and (3) plans, which give the overall goal of the code and a method for achieving that goal; it documents (or, if you prefer, explains or describes) *what* the code does without describing *how* to do it.

A crude view of analogy is

> Map domain problem to image problem. Solve image problem. Lift image solution to get domain solution. Lift image theorems to get corresponding domain theorems. Lift image plans to get corresponding domain plans. Lift image code to get corresponding domain code. Now solve domain problem using new domain expertise.

The focus of this research was to develop algorithms to form analogy maps, and to lift solutions, justifications of solutions (how else can we believe they are correct?), plans, and their justifications. This process thus writes new expert problem solvers, hence achieving new expertise.

Our theory of analogy is built around the notions of an object, its type, and its representation. Objects (in our sense) are the subject of the theory of a world (that is, they are subjects in the sentences which describe the world). An intrinsic quality of an object is its type. We consider type to be meaningful in the description of a world, to the extent that we will present a technique to derive type (and type hierarchies) from world description. Thus, in geometry, one type would be "line"; the type "algebraic variety" (i.e., point, line, plane, hyperplane) would not be used because it typically does not appear in descriptions of geometry world. Finally, an object may have several representations. In geometry, we might represent a line as a list of points which are on it.

A reason for the relative success of expert problem solvers over uniform proof procedures is their ability to use special representations to conveniently encode knowledge about the world. In mathematics, the notion of representation is of extreme importance. These and other uses of the notion of a representation led to a realization that perhaps the single most important thing to be learned from reasoning by analogy was the "proper" way to represent objects in a world.

Since the expertise of a world has three components (code, plan, and description) we need to specify which component has the notion of representation. The descriptive component (predicate calculus) does not have representations, but does have the notion of object and type (in that type can be derived syntactically). The plan component has all three notions. In this component representations are manipulated only by pattern matching. Finally, in the code component, we have the notion of representation, but not (necessarily) the notions of object and

type. Representations may be implemented as property list lists, but this need not necessarily be the implementation.

A fairly complex picture of analogy emerges. The analogy map goes between world expertises, preserving components. Other processes (proving code and plans correct, and automatic programming processes) go between the components of world expertise. Analogy and these other processes are *not* independent. We will be able to detect "bugs" in processes which go between components. These "bugs" will correspond to "bugs" in the analogy map. Similarly, patches to the analogy map will induce patches in the various components of expertise.

Thesis Supervisor: Marvin Minsky

Title: Professor of Electrical Engineering

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N000-75-C-0643.

ACKNOWLEDGEMENTS

I would like to thank Kent Stevens for reading several early drafts of this thesis. Many of his fine suggestions contribute to its readability.

Charles Rich's many questions about the content of this research led to clarification (and, occasionally rethinking) of the issues involved.

Marilyn Matz' careful reading of an earlier draft led to many typographical and grammatical improvements.

Karen Prendergast prepared the diagrams and illustrations, for which I thank her.

Suzin Jabari produced the cover.

Finally, I would like to thank Patrick H. Winston and Gerald J. Sussman for their willingness to listen to my problems and frustrations.

OUTLINE

Introduction to Analogy 8
This chapter introduces our approach to reasoning by analogy.

9 Types of Analogy. We explore three dimensions in the various approaches towards an analogy process.

12 Analogy and Maps. This introduces our way of thinking about analogy as a map between the theory of two worlds.

19 Structure of Theory. The Theory of a world consists of three components: descriptions, plans, and code.

22 Comparison to Evans. We compare and contrast our sense of analogy to Evans's.

25 Focus of Reasearch. While this research touches many topics, our primary focus is on algorithms for understanding the use of representations, and on the theoretical limitations of these algorithms.

Overview of the Analogy Process. 28
This chapter outlines our analogy process. The algorithms will be explicitly given in a later chapter. Here we give detailed examples of the operation of three main analogy processes.

30 First vignette. We see how the analogy map is constructed between TIC-TAC-TOE and the game JAM.

43 Second Vignette. We see how the parts of a theory of a world are connected and used. The illustration is in rhetoric world and the blocks world.

52 Third Vignette. Using the descriptions in the second vignette, we see how new expertise is obtained in rhetoric world.

Geometry World 55
Before analogy can be used, we must describe a world, and give plans for solving problems in that world.

55 Justification of this World choice. We develop this world in order to present the analogy algorithms and in-depth examples.

56 Hilbert's Axioms. We give an axiomatic description of geometry world as a first step in developing the theory of this world.

60 A Language for Plans. We take the second step towards a theory of geometry world by presenting its plans.

73 Representation Claim Proofs. We develop a technique for proving facts about representations.

76 Constructions. Three types of construction problems are presented: trivial constructions, locus constructions, and missing-point constructions.

Analogy Algorithms 82
We give detailed algorithms used for reasoning by analogy.

82 Map formation and Extension. Summarization and map formation algorithms are used to map problems in one world to analogous problems in another.

85 Deriving Result justifications from Plan justifications. Problem solving is not theorem proving. We present an algorithm that justifies problem solving results from plans.

90 Debugging Algorithms. Bugs in result justifications are detected and classified. According to the type of bug, patches are inserted into plans.

Analogy Theory and Examples 95
Using the plans and algorithms presented in the two preceding chapters, we will work four examples.

95 Copying (Geometry). A trivial application of analogy can copy isomorphic aspects of a world, including special representations.

99 Adaptation (Geometry). A non-trivial application of analogy can adapt expertise to slightly different ends.

107 Innovation (Geometry). A non-obvious application of analogy creates new expertise in surprising ways.

113 Failures (Tower of Hanoi). We present one way the analogy process can fail.

Logics of Experts 115
This chapter is very theoretical. The conclusion we reach is that we have solved the analogy problem, subject, of course, to certain restrictions on the nature of the problem worlds.

117 Problem solving formalisms correspond to logics. We develop a connection between expert problem solving formalisms and non-classical systems of logic.

120 Logic of Worlds. Proof-theoretical limitations of any analogy process are presented. We develop a hierarchy of worlds, showing that geometry world was a good initial choice for studying analogy.

Conclusion and Future Work 130
This chapter may be read out of sequence.

130 Psychological validation. We examine some psychological data on transfer phenomenon as a first step towards validation of our theory of analogy.

135 Improving Analogy. Directions for future research are indicated in the areas of modal logic and more general representations.

Notes 138

Bibliography 142

This empty page was substituted for a blank page in the original document.

for inspiration.

TAXONOMY OF ANALOGY *This section may be skipped.*

Normally when one thinks of analogy, one remembers problems of the form

Angels are to men as *x* are to animals

where one skillful in verbal analogy would fill in the *x* with the word *men*. Evans's clasic AI program [E1] solved geometric analogy problems which have the form

Figure A is to figure B as
figure D is to figure 1, 2, or 3.

The word "analogy" also brings to mind that endangered species, the slide rule. The basis of the slide rule is, of course, the notion that numbers are analogous to lengths, and adding numbers is analogous to concatenating lengths.

Since the common examples of reasoning by analogy given above differ, and are different from our use of the term, we will now explore a taxonomy of analogy. Once we have this taxonomy we will be able to specify what kind of analogy process we wish to investigate, and to relate our investigations to other investigations of analogy. In developing a framework in which to express these differences, we must distinguish three dimensions: *setting*, *usage*, and *mechanism*.

Setting: Evans's geometric analogy program [E1] worked in only one world, having relations ABOVE, LEFT-OF, and INSIDE. We can thus claim that the setting of Evans's kind of analogy is *INTRA-WORLD*, as opposed to our *INTER-WORLD* analogies between different worlds (see [NOTE 2] for further discussion of this dimension).

Usage: We can use analogy in at least three different ways: as a kind of mnemonic device, as a reduction device, or as a speculation device. To make this distinction clear, consider the following examples:

1. In chemistry we might claim that organic acids are like inorganic acids, alcohols like bases, and esters like salts. From this and a knowledge of inorganic chemistry we can conclude that an organic acid and an alcohol react to form an ester and some water.

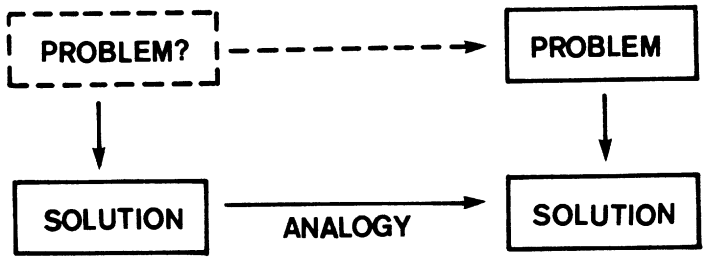
This is an example of mnemonic usage. While we might hazard some guesses about organic reactions on the basis of the analogy above, the only conclusion we feel confident in making is that both "sides" of the analogy are special cases of something more abstract.

2. In algebra, when we wish to study some complex structure (like a noncommutative group) it is often useful to look at homomorphic images. By "throwing away" part of the problem structure, we reduce hard questions to similar, hopefully easier questions.

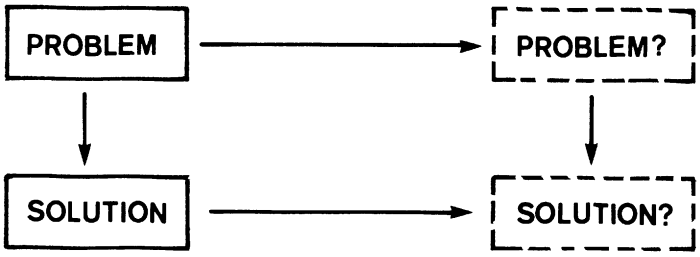
This is an example of reduction. Analogy can be used to reduce a problem to a simpler, easier peoblem in the same or a different problem world. Our ability to deal with an object (in some world) is closely related to our ability to represent that object in terms of other objects in its world. This observation links two amazingly successful problem solving techniques: "Divide and Conquer" (superposition) and "Change Representations" (linear transforms). One kind of result we would expect from a reductional use of reasoning by analogy is a way to represent objects in the problem world (which we term the domain world).

3. We might notice that relatively few chemical reactions (outside of breaking substances down by heating) occur during cooking. Pursuing this we might examine various organic reactions, such as ester formation (esters smell nice) with an eye towards being able to use them for cooking.

This use of analogy, speculation, seems at first to be not really analogy at all. However, as one begins to consider the processes involved (that is, mapping a problem and solution in one problem world to a problem and solution in another), one is forced to conclude that the term "analogy" could be used to describe this kind of reasoning. We can characterize the speculative use of analogy by noting that embarrassingly often we find ourselves with a very good solution, but with no problem that calls for it.



At other times we have a problem and solution, but unfortunately either the problem isn't interesting or the solution isn't profitable.



In both cases, analogy might be useful, and when used, it is used for speculation.

The third dimension in our taxonomy concerns the mechanism the analogy process uses to move between the image world and the domain world (or between image and domain problems in an intra-world setting). The mechanism used for moving between worlds is strongly dependent on the way worlds are described. That is, analogy mechanism is (for the most part) determined by descriptive mechanism. Thus we can make distinctions in this dimension by refering to descriptive mechanisms. There are at least three popular descriptive mechanisms:

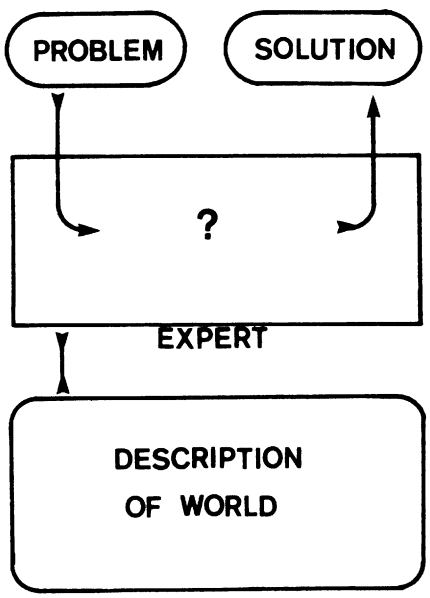
> *Analytic.* Objects in some world might be described (or , if your prefer, characterized) by a set of coordinates in an appropriate feature space. The analogy mechanism for this descriptive mechanism is then a map on the coordinates.
> *Example*: Evans [E1], MERLIN [M9].
>
> *Network.* Objects are described (characterized) as nodes in a network with distinguished links. If two nodes (or, more generally, sub-networks) are analogous, a network mechanism identifies links from analogous nodes as analogous, and terminals of analogous links as analogous, etc.
> This mechanism can be distinguished from the analytic mechanism by noting that with network analogies, the links names do not need to be isomorphic.
> *Examples*: Winston [W6].
>
> *Axiomatic* The emphasis in this mechanism is not on what objects are so much as on the way they behave, i.e., what various predicates and functions return when applied to them. We do not need to confine ourselves to the predicate calculus to use this descriptive mechanism.
> *Examples*: Sussman, Stallman [S7], Doyle [D1]

We will be concerned exclusively with an inter-world setting, a reductional usage, and an axiomatic mechanism.

MAPPING WORLD KNOWLEDGE

Our goal is to use reasoning by analogy to write programs. As a first approximation, the analogy first constructs a map from the domain (or problem) world to the image (or solution) world. It then must lift portions of the image expertise to create new domain expertness. For the remainder of this chapter we will assume that we *already have a map from the domain world to the image world.* After obtaining some understanding of how such a map is to be used, we will show in chapter [OVERVIEW OF ANALOGY, FIRST VIGNETTE--TIC-TAC-TOE] how these analogy maps are obtained.

Our goal in using analogy is to write programs. Before I write a program, I need a description of the world the program is to work in. If I am to write a program for dealing with, say, plane geometry, I first need a description of the interaction of points and lines. Similarly, if I am writing a program to play tic-tac-toe, I need to know the rules of the game.



DOMAIN WORLD

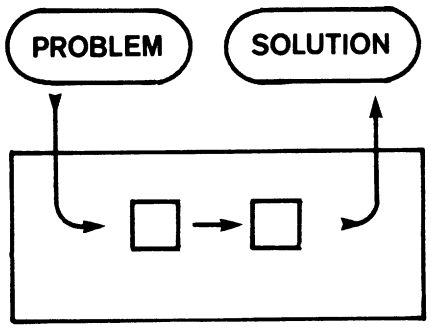


IMAGE WORLD

The observant reader may notice that we already have a description of the image world: the programs that deal with it. Why do we need another description? Unfortunately, this imperative description is not suitable for our purposes; its utility is both theoretically and practically limited. We are limited in practice by our inability to decide what a program is doing without being given further information. That is, we cannot determine the *plan* (i.e., what is being accomplished by the computation) of an algorithm given only the *code* (i.e., a sequence of operations which specify how the computation is accomplished). Moreover, we cannot, even in theory, deduce that a program is correct only on the basis of the program. For example

```
(DEFUN SOLVE-SECOND-DEGREE (A B C)
 (QUOTIENT (DIFFERENCE (SQRT (DIFFERENCE (TIMES B B)
                                         (TIMES 4 A C)))
                       B)
           (TIMES 2 A))).
```

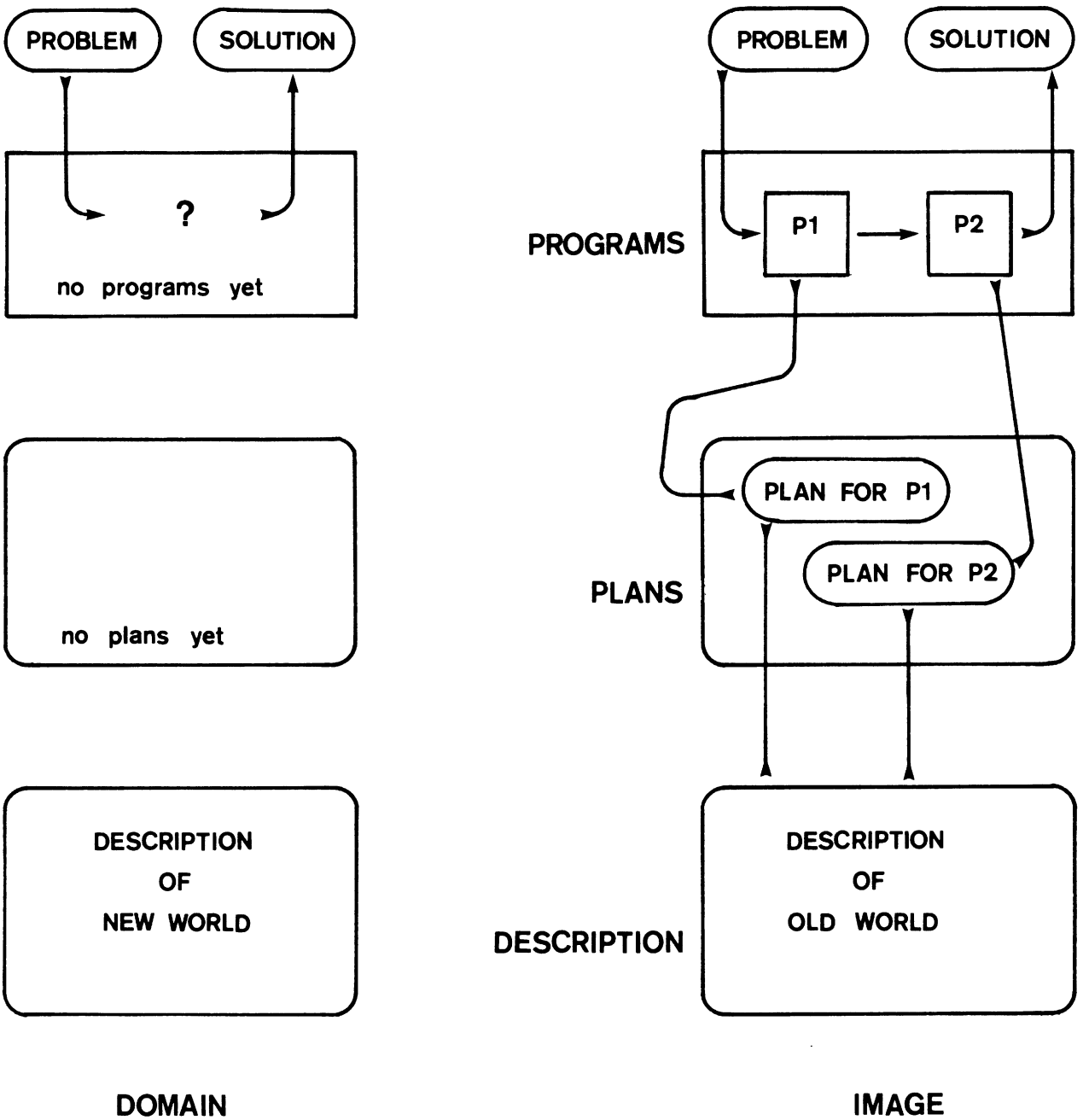
Is this program correct? We might say to ourselves "It is limited; it finds only one real root of a second degree polynomial. Within that restriction it works." But that raises the real concern: "Who said anything about polynomials?" Assuming arithmetic works (usually not a valid assumption due to truncation errors and round-off errors), the strongest claim we can make of this program is (1) it halts, and (2) it returns a number (equal to the number it computes). Nor is code easier to understand than the description of the world it works in, (see [NOTE 3]).

As a working definition, a plan is composed of two parts: an intention and a collection of subgoals and constraints. A plan is *consistent* if it can be justified (using the description of the problem world) by showing that accomplishing the subgoals subject to the constraints implies that the intention of the plan has also been accomplished (we will give an exact definition of plans in [GEOMETRY WORLD, LANGUAGE FOR PLANS]). In order to prove that code is correct, we first prove that its plan is consistent, and then prove that the code does what its plan specifies.

One might wonder why we wish to prove programs are correct. We don't. However, we cannot hope to correctly apply a program from one world to problems in another if we are unable

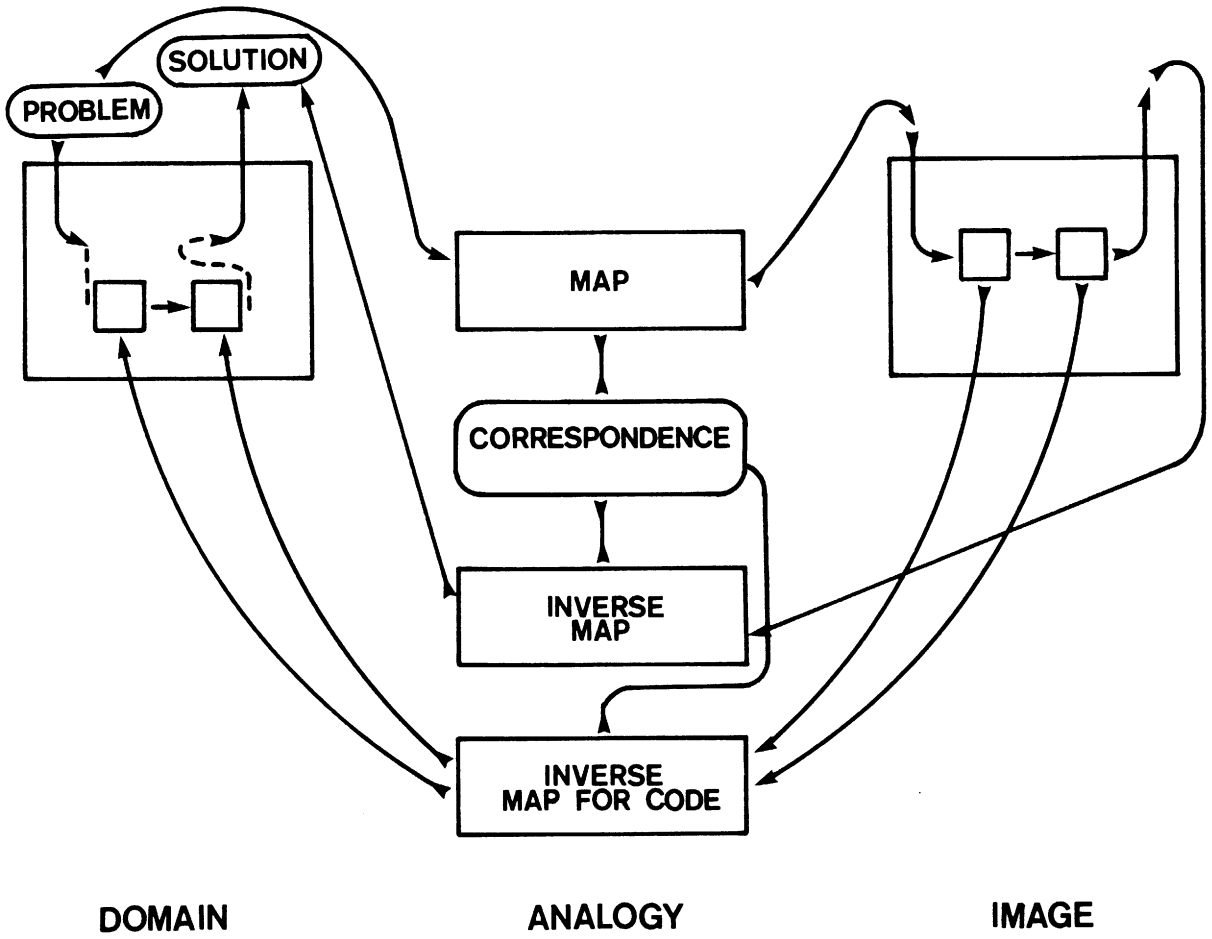
to determine if that program is correct even in its own world. We wish to be able to prove that a lifted program (that is, a domain program created by the inverse analogy map) is as correct as the original image program in analogous situations. To emphasize our point by exaggerating it, if the image code is incorrect, we wish to insure that the new domain code is incorrect in an analogous way, reasoning "we must have had good reasons to make the program incorrect in that particular way."

To overcome the difficulties in proving raw code correct, we insist that plans be given for the programs in an expert problem solver, and that those plans be justified by references to the description of the world. We will therefore have


PROBLEM
SOLUTION
?
no programs yet
no plans yet
DESCRIPTION
OF
NEW WORLD
DOMAIN
PROBLEM
SOLUTION
PROGRAMS
P1
P2
PLAN FOR P1
PLAN FOR P2
PLANS
DESCRIPTION
OF
OLD WORLD
DESCRIPTION
IMAGE

In the image world we have a *problem*, stated in terms of the world description (e.g., in the predicate calculus). In the image expert problem solver's set of programs, executing P1 and then P2 will produce a *solution*, either a truth value or some object of the image world. The code for P1 is attached by commentary to its plan, which is in turn attached to the image world description (e.g., axioms). In the domain world we have a problem and an anticipated solution, but lacking expertise (i.e., having no code and thus no plans) we are unable to forge a link between problem and suspected solution.

The analogy process uses a map from the domain world to the image world. A simple model of analogy would be



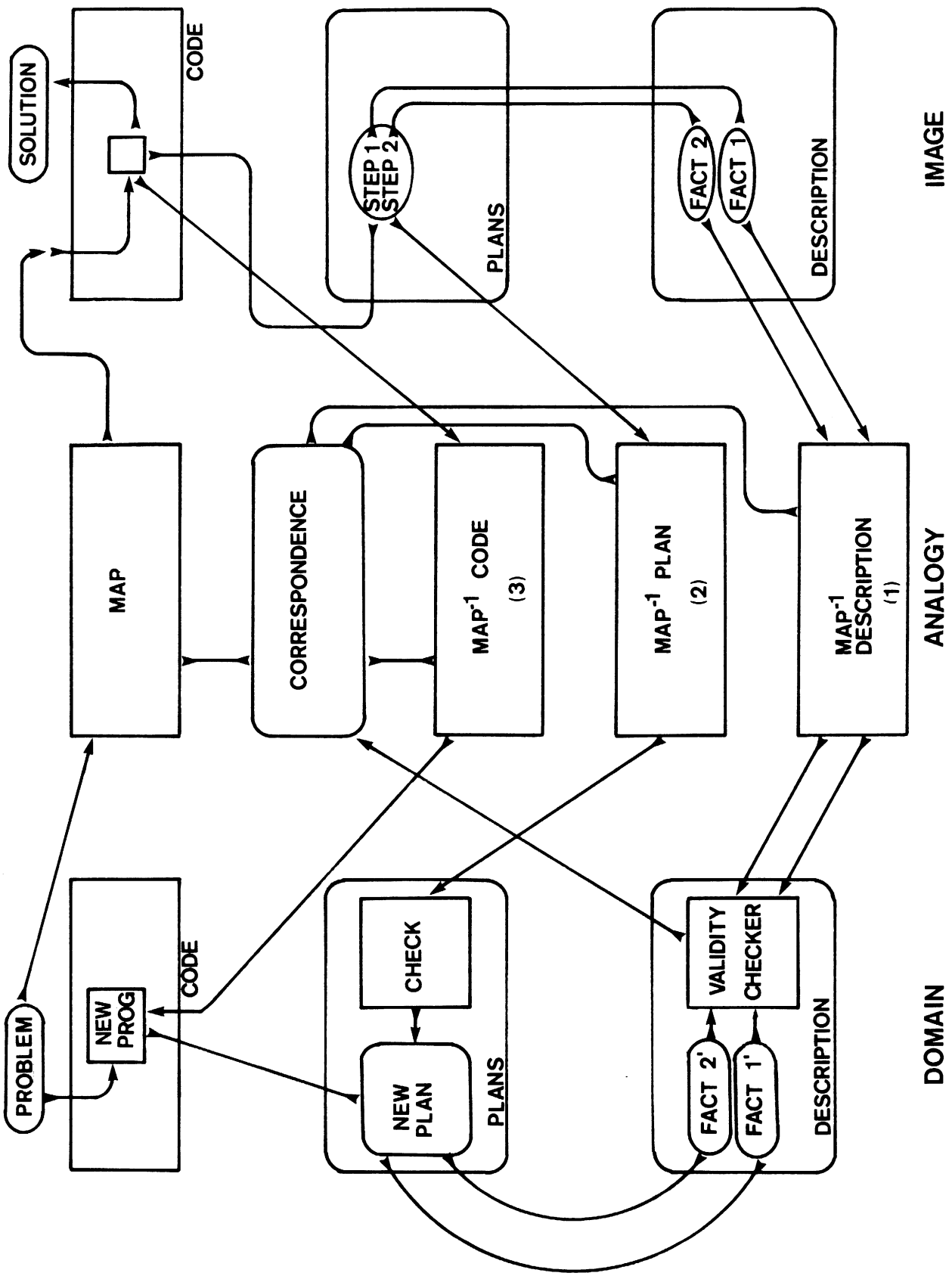
The meaning of this diagram is: take the domain problem, apply the map to get an image problem. Solve the image problem, and apply the inverse map to the image solution to get the solution in the domain world. We can go one step further (so that something will be learned from solving the problem). Apply the inverse map to the image program to get a program in the domain world.

> This view of analogy has two defects:
> 1. We have no assurance that the solution is correct.
> 2. Even if the solution is correct in this particular case, we have no reason to expect that the new domain program is generally correct. Of course, if the two worlds are isomorphic (as is usually the case in the literature on "reasoning by analogy"), solution correctness will generally imply program correctness. For this reason, *one should not study analogy between isomorphic worlds.*

Although we are ignoring the problem of getting an analogy map for the moment, we must point out that it is meaningless to discuss map correctness independently of the results obtained by using it. An analogy map is correct if and only if the lifted solution, lifted plans, etc., are correct. The two defects noted above might be summarized as "we don't know that the analogy map is correct" by the definition of an analogy map being correct.

We can remedy the two defects outlined above by making use of plans and world descriptions. The map from domain problems to image problems can sometimes be extended to partial maps of domain descriptions to image descriptions. In these cases (and we will not consider any others) we can derive a description of why a particular solution is correct in terms of the image world description. This can be (inverse) mapped back into the domain world. See (1) in the diagram below.

If the facts which justify the image solution remain valid after being (inverse) mapped into the domain world, then we know that the solution is correct. More importantly, we can now apply the inverse map to image plans (see (2) in the diagram). If this plan is compatible with domain plans, we can apply the inverse map to programs (see (3) in the diagram). We will often abbreviate the expression "apply the inverse analogy map to" with the verb "lift."


SOLUTION
CODE
STEP 1
STEP 2
PLANS
FACT 2
FACT 1
DESCRIPTION
IMAGE
MAP
CORRESPONDENCE
MAP-1 CODE (3)
MAP-1 PLAN (2)
MAP-1 DESCRIPTION (1)
ANALOGY
PROBLEM
NEW PROG
CODE
CHECK
NEW PLAN
PLANS
VALIDITY CHECKER
FACT 2'
FACT 1'
DESCRIPTION
DOMAIN

At this point, the reader may have formulated a list of questions:

1. What if the map is many to one, or even many to many? Generally this will not happen with objects. On the other hand, it probably will occur with predicates and functions.

2. What if a fact (after being *lifted*) is not correct? Then the validity checker will object. We must try to salvage the plan.

3. Is it possible for a *lifted* plan to be invalid, even if all of its facts are valid? Yes, since part of the theory justifying the plan may not be valid.

4. Is it possible for a plan to be *lifted* if some of its facts are not valid? Yes.

5. Why can't we just translate programs and ignore *plans*? We cannot figure out what programs do. See the above discussion.

6. Why couldn't we just produce code directly from plans (in the domain world)? Perhaps, eventually, we will have the technology to do this. But we certainly don't have it now.

7. The idea of mapping is good. Where does the map come from? There are two sources: either analogy can be *told*, or it can figure one out for itself.

We will try to answer these questions in what follows.

STRUCTURE OF A THEORY

We have introduced three components of expertise because the analogy process requires them. We would now like to argue that this division of knowledge is appropriate on other grounds, first by connecting the two notions of *expertise* and *theory*, and second by showing the dangers involved in not making this division.

Marr and Poggio [M4] suggest that the theory of a world can be divided into four levels. Using their example of the Fourier transform, these levels are:

I. DESCRIPTION. Mathematically, the Fourier transform obeys various axioms. These axioms, and relations between the Fourier transform and other mathematical objects,

can be given independently from any discussion of how Fourier transforms can be "computed."

II. PLAN. Given that one wishes to take a Fourier transform of something, one can proceed in various ways: numerically by way of the old "slow" Fourier transform, or by the FFT, or the FFFT. One can take them "by analog means" as is done in the ear (one-dimensional Fourier transform) or by using lasers (two-dimensional Fourier transform). One can also take symbolic Fourier transforms by doing integration. Plans specify goals, intentions, and constraints.

III. CODE. Depending on the plan chosen, and the mechanism available, one can encode in, say, your favorite computer language, a program for actually computing the transform. Code specifies control flow and data flow (see [R1]).

IV. MECHANISM. Suppose that one has an FFT algorithm in FORTRAN. The efficiency of running this algorithm will depend on the particular computer having, for example, hardware multiplication, and bit-reversal instructions.

Normal usage of the term *theory* refers to a list of definitions and axioms, i.e., the description level above. Suppose I had a computer program that, given a periodic function, returned the Fourier transform of that function. Then it would be proper to claim that the computer program embodied, or had, or was a theory of Fourier transforms. This program would be a predictive theory of Fourier transforms at the code level. Now suppose I had two different programs which took symbolic Fourier transforms equally well (i.e., same answers on all problems), but one ran twice as fast as the other. It is conceivable that in some sense both programs have the same theory of Fourier transforms, but in another sense the faster one has a better theory.

Assuming that the observed difference is due to coding inefficiency, the two theories are different at the code level, but the same at the plan level.

With this extended notion of *theory*, it is easy to see that the components of expertise correspond to levels of a theory. This suggests that we should claim that analogy operates on theories of worlds. We will use "expertise" and "theory" interchangably from now on when refering to knowledge about a world.

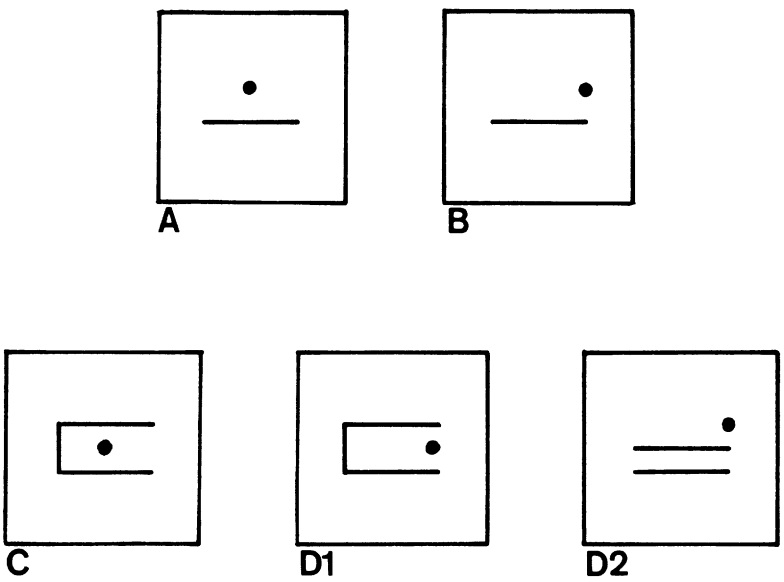
Goldstein [G3] used a similar division: at level I he had models; at level II he had plans for constructing pictures which would satisfy the models; at level III he had LOGO programs which were supposed to actually draw the pictures.

Thinking in these terms gives us insight into an important difference between Sussman's [S8] and Goldstein's approach to debugging. The real world does not present us with level I theories on a silver platter. Sussman therefore did not give his program debugging system direct access to the description of the blocks world. Rather, he arranged for his primitives (i.e., PUTON) to simply enforce the hidden blocks world description. Similarly, the level II theory was not explicitly given to his program debugger; it was merely hinted at in a number of places, leaving the task of synthesizing the level II theory to HACKER.

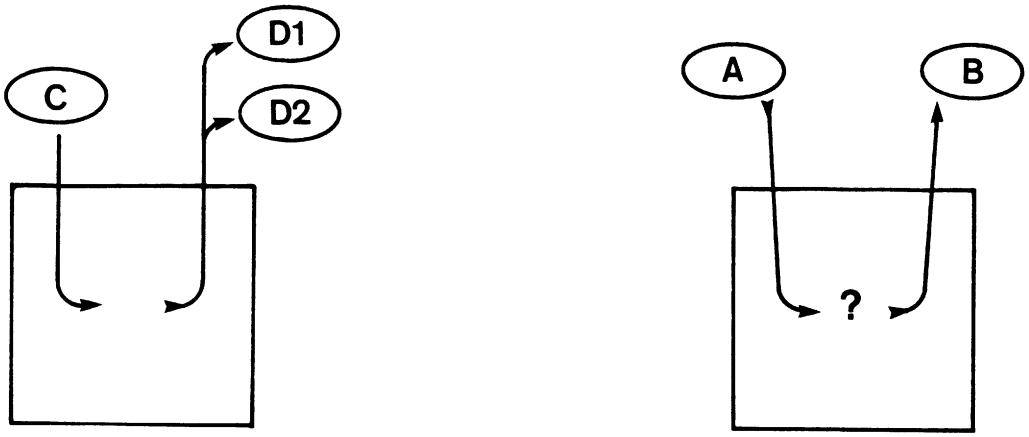
In attempting to reflect his view of the real world, Sussman paid a heavy price: most of the complexity of his thesis was devoted to untangling the theory of the blocks world, not to debugging programs (i.e., making level III actually do what level II said it should). We are not willing to pay this price; we insist that the three top levels be explicitly provided. We also insist that the code of our experts do what the plans claim.

EVANS'S GEOMETRIC ANALOGY PROGRAM

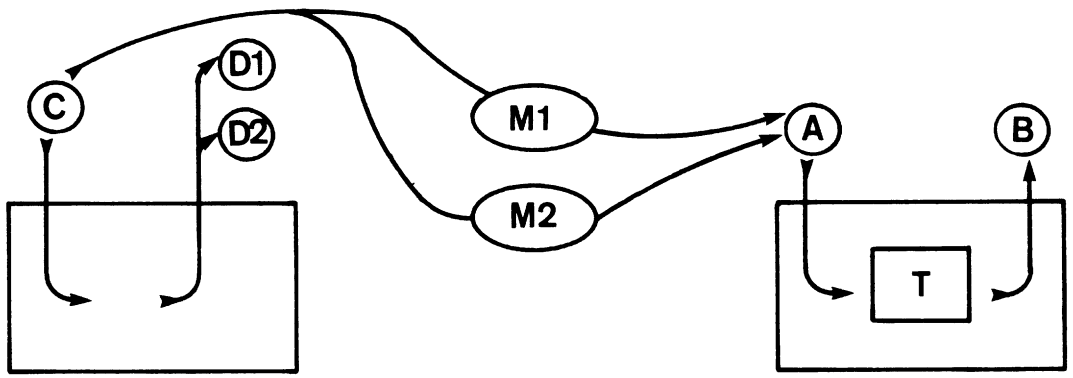
We can contrast our theory of analogy with that presented in Evans's landmark paper [E1].



Evans's program solved problems of the form "A is to B as C is to ..." where some list of pictures D1, D2 are provided to fill in the blanks. If we cast his problem form into ours, we will have



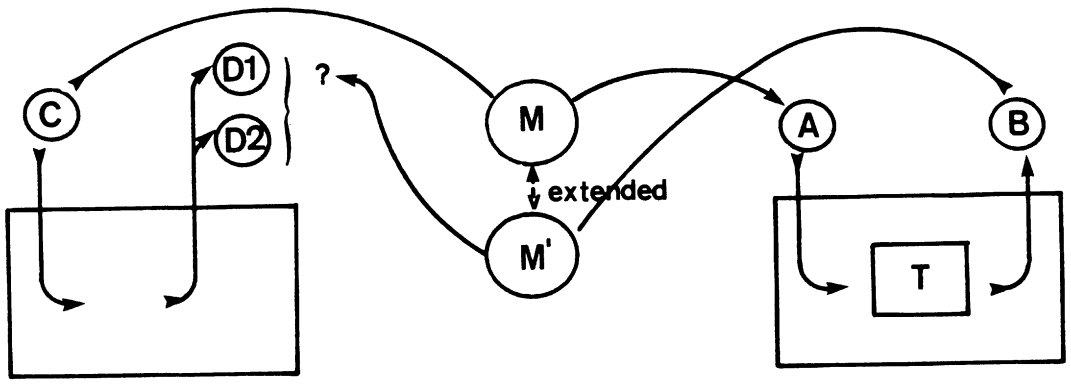
Evans first constructs a description of the operation of the image expert in order to obtain a correspondence between objects in A and in B. He then constructs a number of maps from C to A (maps M1 and M2, for example).



ANALOGY MAPS

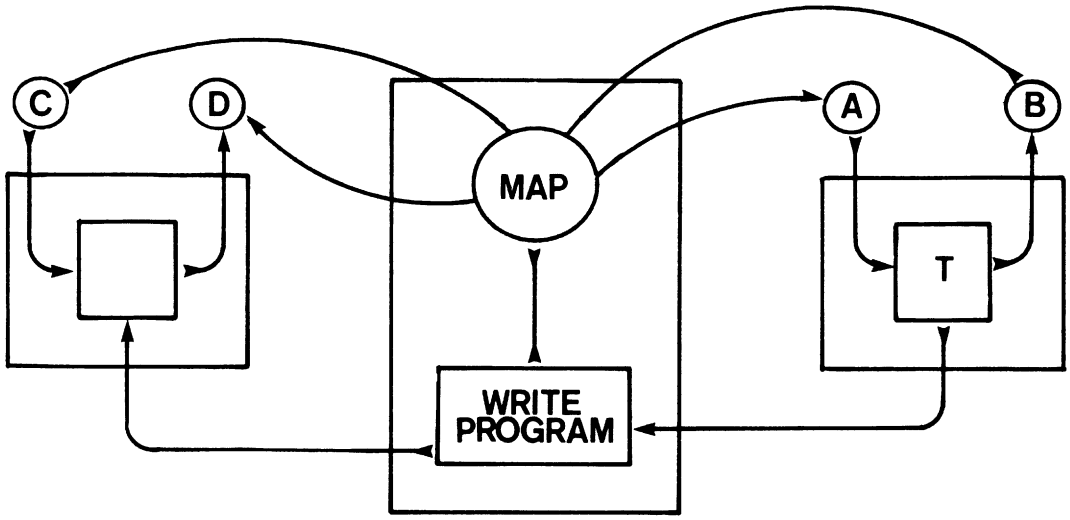
Evans then extends each map so that D1, D2, etc., can be mapped to B. Finally, Evans selects the "best" of these extended maps. B's inverse under this map is the answer. The result of Evans's analogy program is a "best fit." Winston [W5] suggests an improvement on Evans's scheme: instead of constructing all those maps and using the inverse at the last possible moment, why not use the inverse early in the effort. The modified procedure maps C to A, applies the transform T, then takes the inverse of the result B. This type of procedure is generally termed "analysis by

synthesis."



If $M^{-1}TM(C)=D1$, then D1 is the solution. Similarly for D2. Otherwise a new M^{1}, T, or M is tried. We can be clever, using failure analysis to guide the next selection.

If we could apply our theory of analogy to this kind of problem, we would take Winston's suggestion one step further. We would insist that the result of reasoning by analogy should be a program. After constructing and debugging the map between the two sets of diagrams, we would use the "winning" analogy map to write a program in the domain world so that the domain expertise may be extended. This last step, the core of our research, is the fundamental distinction between our approach to analogy and Evans's. (It might be argued that Evans also produces a program. For a refutation, see [NOTE 4].)



ANALOGY PROCESS

FOCUS OF THIS RESEARCH

The focus of this research was to develop algorithms to form analogy maps, and to lift solutions, justifications of solutions (how else can we believe they are correct?), plans, and their justifications. We are not setting out to make a theory of human reasoning by analogy; our interest is computational rather than psychological.

Our theory of analogy is built around the notions of an object, its type, and its representation. Objects (in our sense) are the subject of the theory of a world. That is, if the first order predicate calculus is the descriptive language, then the variales are quantified over the collection of all objects (and therefore the subjects of predicate calculus sentences). An intrinsic quality of an object is its type. The type of an object is unique (that is, an object cannot have two types), pre-specified, and immutable. We consider type to be meaningful in the description of a world, to the extent that we will present a technique to derive type (and type hierarchies) from world description. Thus, in geometry, one type would be "line"; the type "algebraic variety" (i.e., point, line, plane, hyperplane) would not be used because it typically does not appear in descriptions of geometry world. Finally, an object may have several representations.

REPRESENTATIONS

A reason for the relative success of expert problem solvers over uniform proof procedures is their ability to use special representations to conveniently encode knowledge about the world. In mathematics, the notion of representation is of extreme importance.

We will now give several examples of representations. The fundamental result of the theory of finite abelian groups is that they can be canonically represented as products of prime power groups. Another major result of algebra is that a polynomial can be represented either by a sequence of coefficients or by a sequence of values. The importance of the fact that a signal can be represented as the sum of sine and cosine signals is well known. A major insight results from the observation that if a signal is represented by the coefficients of a polynomial, then the Fourier transform of that signal can be represented by a sequence of values of that polynomial. Note that all of these representations are relatively unstructured; they are simply lists (sometimes ordered) of other objects in the same world. That is, groups are represented in terms of other groups, signal functions are represented in terms of other functions, and polynomials as a list of elements from the underlying field (which is part of polynomial world).

These and other uses of the notion of a representation led to a realization that perhaps the single most important thing to be learned from reasoning by analogy was the "proper" way to represent objects in a world.

Since our theory of a world is on three levels (code, plan, and description) we need to specify which levels have the notion of representation. The descriptive level (predicate calculus) does not have representations, but does have the notion of object and type (in that type can be derived syntactically). The plan level has the notions of object, type, and representation. At this level representations are lists of plan level objects and are manipulated only by pattern matching. Finally, at the code level, we have the notion of representation, but not (necessarily) the notions of object and type. Representations at the plan level may be implemented at the code level by

property list lists, but this need not necessarily be the implementation. Arrays, tables, and lambda expressions may also implement (plan level) representations. Typically, at the code level representations may be manipulated by the LISP functions CAR, CDR, and CONS.

ESTABLISHING CLOSURE

In Artificial Intelligence it is highly desirable to be able to solve all problems of a specified nature. Generally it has been fairly simple to state the constraints under which a problem space is closed (in the sense that all problems in the closed space can be solved). However, since analogy operates on theories of worlds, our problem space is the space of all worlds, and even stating a closure condition becomes a major project.

We claim (leaving the explanation and demonstration for chapter [LOGIC OF EXPERTS]) to be able to solve analogy problems at least between worlds whose underlying logics are negationless intuitionistic.

OVERVIEW OF ANALOGY

VIGNETTES -- TOWARDS THE ANALOGY PROCESS

Analogy is a process which operates on "expert problem solvers." Simply stated, the analogy process consists of three phases: *map*, *solve*, and *lift*.

Normally the analogy process will use the map from the previous problem. By using this "old" map we preserve the context of discussion of the previous problem. This means that some importance is attached to the order in which problems are presented to the domain expert, and thus to the analogy process, much as in the Winston learning program [W6].

To try a different analogy, we can either try forming a completely new analogy by starting with the empty map, or we can use some other previously constructed map as a starting place. We do not suggest any sort of backtracking. When we say (in what follows) "try another analogy" we have in mind *abandoning* the current map and starting afresh. A refinement of this idea would be to guarantee that when we try to form a "new" map, we will have at least one difference in the way object types are mapped (between the new map and current maps).

Finally, we assume that the image world is given.

We will now give an outline of the analogy process. We will give explicit algorithms in the chapter [ANALOGY ALGORITHMS], and examples of analogy operation later in this chapter and in chapter [ANALOGY EXAMPLES].

1. MAP. When some expert encounters a problem which cannot be solved (due to incompleteness on the part of the expert), analogy can map this problem into an analogous problem in some other world.

1.1. Summarize the domain problem. It is necessary to summarize the domain problem, since it may be impossible to find an analogy map capable of mapping the entire domain problem to an image problem.

1.2. Extend the current analogy map to include any new operations. Note that either the current map or the extension may be empty.

1.3. Apply the analogy map to the summarized problem. The result of this step is an analogous image problem.

2. SOLVE. The appropriate expert problem solver for the new world solves the analogous image problem. If the solution attempt fails, then we either change the analogy map, or change the analogous problem by including further details.

2.1. Obtain a solution to the analogous problem.

2.2. If no solution can be obtained, enlarge the summary from part 1.1 and continue the solution attempt.

2.3. Apply the inverse map (extending if necessary) to the solution.

2.4. In cases where the domain solution is anticipated (see [NOTE 5] for an explanation of "anticipation"), if it and the (inverse of) newly obtained solution disagree, form a different analogy map.

3. LIFT. If the solution attempt succeeds, then we need to "lift" the solution back into the domain world.

3.1. Obtain (from the expert problem solver) the reason why the solution is thought to be correct.

3.2. Apply the inverse map to the reasoning (extending the map if necessary).

3.3. If the reasoning is based on *false assumptions* (detected by the justification checker in [ANALOGY ALGORITHMS,DEBUGGING ALGORITHM]), form a different analogy.

3.4. If the reasoning is *incomplete*, note the presence of a bug and obtain a patch.

3.5. If we are *unable* to lift all of the reasoning behind the solution, and cannot find a replacement in the domain, try another analogy.

3.6. If we *are* able to lift all of the reasoning, then lift the plans and the code which

generated the solution, making use of previously detected bugs and their patches.

We will present three vignettes to illustrate: how analogy maps are developed, how process description can be used to obtain the reasoning behind a solution, and finally how programs can be lifted. Two world pairs are used in the vignettes. The first pair consists of two simple games: "Tic-tac-toe" and "Jam." The other pair of worlds consists of a version of the blocks world (as investigated by Fahlman [F1]), and the "rhetoric" world. These vignettes are very much "toy" problems; there is no guarantee that the analogy process (as presented in later chapters) can correctly deal with more complex problems in these worlds.

TIC-TAC-TOE WORLD

Surprisingly, the game of Tic-Tac-Toe (abbreviated TTT from now on) is interesting in its own right. The grade school 3X3 version of TTT is, of course, a draw if both players move optimally. The interest in TTT arises from higher dimensions and/or larger boards. The 3X3X3 game is a win for the first player (take the center). The popular 4X4X4 version is still unsolved (see Sheppard [S3] for strategic considerations). In this TTT, after three moves (6 ply) there are 1,499,409,707 positions, not accounting for symmetry.

TTT is isomorphic to so-called magic squares. In particular, one has an isomorph of 3X3 TTT called "number scrabble":

8	3	4	Players alternately select integers between 1 and 9.
1	5	9	The first player to total 15 wins.
6	7	2	

In number scrabble, of course, the player does not see the board.

One might naturally ask "Are there magic cubes corresponding to either the 3X3X3 or

4X4X4 TTT games?" The answer is no (see M. Gardner [G1]). Well, then, are there any magic cubes at all? The answer here is, surprisingly, "yes." Gardner gives an 8X8X8 magic cube, while Wynne [W9] gives a 7X7X7 cube. "Are there magic cubes of other orders, specifically 5X5X5 and 6X6X6?" The answers are, respectively, "yes, unknown, and unknown."

Banerji and Ernst [B1] investigated the use of analogy to transfer strategies from one form of TTT to another. Although, as we mentioned before, one cannot hope to gain much insight into the analogy process through considerations of isomorphic worlds, we will use TTT and another of its isomorphs (JAM, described below) to illustrate how initial analogy maps are formed.

Assume that we have an expert TTT playing program. We will describe JAM (i.e., give the rules and starting configuration of the game) and then use "reasoning by analogy" to obtain a JAM expert. Since the two worlds are isomorphic, we will be essentially finished after we develop the analogy map (although we will have no way of knowing this until we actually lift the various components of expertise). In non-isomorphic worlds (like the rest of the examples in this paper) the "develop map" step is only the beginning of the analogy process.

RULES FOR TIC-TAC-TOE AND JAM

TTT is played on a grid with squares labeled as above (except numbers are preceeded with the letter "S"). There is a type hierarchy imposed on the squares:

the CENTER square is S5
the CORNER squares include S8, S2, S4, S6
the SIDE squares include S1, S3, S9, S7

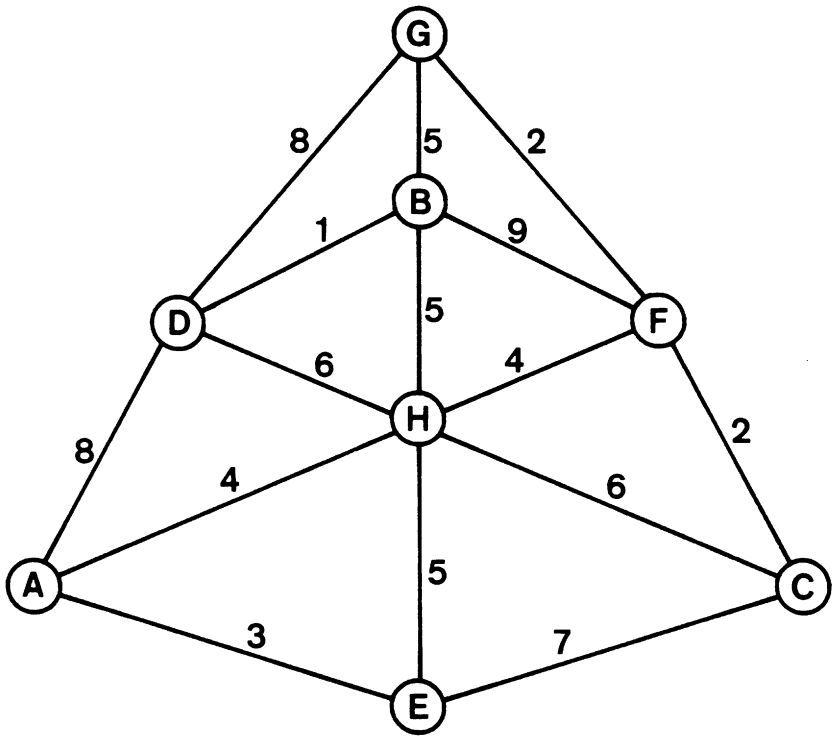
There are eight rows; each row is a triplet of squares. The game is won when all squares in a row are XED; it is lost when they are all ZEROED. A draw is likely. We list all the squares in the various rows:

ROW-A contains squares S8 S3 S4
ROW-B contains squares S1 S5 S9
ROW-C contains squares S6 S7 S2
ROW-D contains squares S8 S1 S6
ROW-E contains squares S3 S5 S7
ROW-F contains squares S4 S9 S2
ROW-G contains squares S8 S5 S2
ROW-H contains squares S4 S5 S6

Rows are also broken into types:

the DIAGONALS are ROW-G, ROW-H
the NORMALS are ROW-A, ROW-B, ROW-C, ROW-D, ROW-E, ROW-F.

The game of JAM is played on the following network:



JAM DIAGRAM

The circles are "towns" and the lines are "roads." All sections of a road are blocked when either the "red" player or the "blue" player blocks the road. A town is isolated when all roads leading to that town are blocked by one color. The first player to isolate a town wins. Thus if the red player

blocked roads 2, 6, and 7, then town C would be isolated, and red would win. The names of the roads and towns make the isomorphism to TTT clear.

WORLD DESCRIPTIONS

Now we will try to give the TTT and JAM world descriptions more formally in the predicate calculus. I would like to think that the world description below is natural, and merely a rewrite of the rules given above. In TTT we need to say:

```
(DECLARATION (CENTER S5)
             (CORNER S8 S2 S4 S6)
             (SIDE S1 S3 S9 S7))
```

This type of statement simply says that the "declared" assertions are always to be true. In order to say that the center, corners, and sides are all squares, we give the following facts:

```
(FORALL (X) (IMPLIES (SIDE X) (SQUARE X)))
(FORALL (X) (IMPLIES (CORNER X) (SQUARE X)))
(FORALL (X) (IMPLIES (CENTER X) (SQUARE X)))
```

One might wonder why we don't give the type/sub-type relation (e.g., SIDE is-a SQUARE) explicitly. We need to be able to discover relations like this anyway, so we will take this opportunity to show that the system can deduce them. The reader should observe that we have not said what the predicate SQUARE tests. None of this description is actually used explicitly by the TTT expert code; it is present because it is needed to justify the TTT expert and for use by the analogy process.

Continuing, we name the rows:

```
(DECLARATION (ROW ROW-A ROW-B ROW-C ROW-D ROW-E ROW-F)
             (DIAGONAL ROW-G ROW-H))
(FORALL (X) (IMPLIES (DIAGONAL X) (ROW X)))
```

We have rows, a special kind of row we call a diagonal, and three kinds of squares. The final bit of description says how the squares and rows relate:

```
(DECLARATION
        (IN-ROW ROW-A S8)
        (IN-ROW ROW-A S3)
        (IN-ROW ROW-A S4)
        (IN-ROW ROW-B S1)
        etc.)
```

As soon as we give the rules of the game, we will be done.

1. A square may be XED or ZEROED, but not both.

```
(FORALL (X) (IFF (XED X) (NOT (ZEROED X))))
```

This rule excludes the possibility of a square being blank for technical reasons (see [NOTE 6]).

2. The machine always plays X, so a win condition occurs as follows:

```
(IMPLIES (EXISTS (X) (FORALL (Y) (IMPLIES (AND (ROW X) (SQUARE Y) (IN-ROW X Y))
                                          (XED Y))))
        (WIN))
```

We do not know that (IN-ROW X Y) implies (ROW Y) yet, so the rule must be stated as above.

3. Similarly, we describe a lose condition

```
(IMPLIES (EXISTS (X) (FORALL (Y) (IMPLIES (AND (ROW X) (SQUARE Y) (IN-ROW X Y))
                                          (ZEROED Y))))
        (LOSE))
```

The list continues. There are some rules which cannot be stated because we are missing a notion of "change" (for example, the rule that play alternates).

I will admit that this isn't very pretty, but then again there probably isn't a very elegant way to say the above (other than putting it in English and pictures). In any case, we need to do the same analysis for JAM.

```
(DECLARATION (ROAD R1 R2 R3 R4 R5 R6 R7 R8 R9)
        (TOWN A B C D E F G H)
        (PLAYERS RED BLUE)
        (ON-ROAD R8 A)
        (ON-ROAD R8 D)
        (ON-ROAD R8 G)
        (ON-ROAD R5 G)
        (ON-ROAD R5 B)
        (ON-ROAD R5 H)
        (ON-ROAD R5 E)
        etc. )

(FORALL (X) (IFF (BLOCKED-BY X RED) (NOT (BLOCKED-BY X BLUE))))

(FORALL (Z) (IMPLIES (EXISTS (X)
                             (FORALL (Y)
                                     (IMPLIES (AND (TOWN X) (ROAD Y)
                                                   (PLAYER Z) (ON-ROAD Y X))
                                              (BLOCKED-BY Y Z))))
                     (WIN Z)))
```

We should make several comments about the differences in the two world descriptions:

1. JAM has players, while TTT does not (or so it seems)

2. TTT's expert plays "X", JAM is ambivalent

3. TTT has a type hierarchy, while JAM does not (or so it seems).

These differences were introduced in an attempt to follow the English description, but the real reason for them is to let us show that none of these differences confuse our analogy process.

WORLD DESCRIPTIONS GENERATE SEMANTIC TEMPLATES

The first step in the analogy map generation process is to produce a set of semantic templates. A semantic template is a specification of which object types can be valid arguments of a form. Thus, for each predicate and function, semantic templates give the possible argument types. For example, in JAM the predicate ON-ROAD takes two arguments: the first is a ROAD; the second is a TOWN. The semantic template for this predicate looks like

(ON-ROAD ROAD TOWN)

We will generate semantic templates in a purely syntactic way from the world descriptions given above.

The first step in forming semantic templates is to decide what types of objects are present in a world. We make the observation that unary predicates typically are type-checkers. Thus, by simply listing all the unary predicates we can get a list of potential object types:

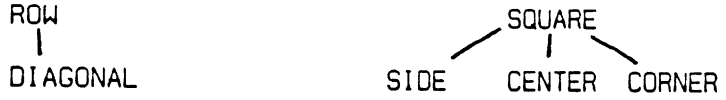
TTT	JAM
SIDE	TOWN
CENTER	ROAD
SQUARE	PLAYER
ROW	WIN *
CORNER	
DIAGONAL	
XED *	
ZEROED *	

We can reject some of these by noting that a type-checking predicate must be used as such, i.e., it must be applied to a quantified variable on the left hand side of an implication (for this purpose only, IFF is not decomposed into two implications). This eliminates XED and ZEROED in TTT, and WIN in JAM.

We can make use of a further observation: if P and Q are type checking predicates, then facts of the form

(FORALL (X) (IMPLIES (P X) (Q X)))

establish a type hierarchy. In the above, P is a kind of Q. Searching the description of TTT for this pattern yields the two hierarchies:



Since TTT is the image world, it would not be unreasonable to insist that these hierarchies be given

explicity. Alternatively, we could perform an exhaustive search of all objects in the world, since TTT world is finite. However, these other techniques are not generally available to us for use on the domain world (JAM) where we lack the requisite expertise.

Finally, we examine the facts about the games to determine the type of their arguments, generalizing upward in a type hierarchy if necessary. This gives us the semantic templates

```
(IN-ROW row square)
(XED square)
(ZEROED square)
```

in image world TTT, and for domain world JAM we get

```
(WIN player)
(ON-ROAD road town)
(BLOCKED-BY road player)
```

The interested reader should see [NOTE 7] for an early form of semantic template. Armed with the list of object types, the type hierarchy, and the semantic templates we have automatically derived, we are ready to form the analogy map.

SYNTACTIC GENERATION AND SEMANTIC REJECTION

We are about to use analogy to create enough expertise in a JAM expert to allow it to make a move. To do this we must first give the JAM expert a problem to solve, then form an analogy map from JAM to TTT.

We will generate in a very syntactic way possible maps from JAM to TTT, and use our TTT expertise to reject most of these proposals on semantic grounds. To start, we present the JAM expert with a problem to solve: our first move

```
(BLOCKED-BY R8 BLUE).
```

Since there is no JAM expert yet, we immediately resort to analogy. We want to map this assertion to the TTT expert in the hopes of gaining enough JAM expertise (by analogy, of course) to proceed.

We must first summarize the current situation in JAM and then try to map this summary to TTT. We discuss the summarization process in chapter [ANALOGY ALGORITHMS,MAP FORMATION AND EXTENSION]. For now, we accept the necessity of first mapping all the declarations in JAM, and then mapping the first JAM move. Thus the first order of business is to map the assertion (found in the first DECLARATION)

(ON-ROAD R8 A).

In developing maps, we start as high as possible in the image type-hierarchy. The domain types are ROAD, TOWN, and PLAYER, while possible image object types are ROW and SQUARE. Our choices for analogy maps from JAM to TTT are

map ALPHA	map BETA	map GAMMA
TOWN -> SQUARE	ROAD -> SQUARE	ROAD, TOWN -> SQUARE
ROAD -> ROW	TOWN -> ROW	PLAYER -> ROW
PLAYER -> ?	PLAYER -> ?	or vice versa

While we only have six possibilities now (two maps each for ALPHA and BETA, and the map GAMMA and its reverse), we will need to contain the possible combinatorial explosion somehow. One technique is to immediately prune the possiblities tree. Using GAMMA to map the predicate ON-ROAD, we get a partial image semantic template

(? SQUARE SQUARE)

with "?" indicating that the image predicate is unknown. This doesn't match any template in TTT, so we tentatively reject this map (the rejection isn't complete since we have more tricks up our sleeve in chapter [ANALOGY ALGORITHMS,MAP FORMATION AND EXTENSION]) to use if we must). Similar reasoning rejects GAMMA's reverse.

In maps ALPHA and BETA, we need to decide what to do with PLAYER. We always prefer NOT to map an object type at all over making DOUBLE maps. So for our first choice we leave PLAYER unmapped. This gives BLOCKED-BY a partial image semantic template

```
(? ROW) for ALPHA
(? SQUARE) for BETA
```

The image of BLOCKED-BY's semantic template under map ALPHA doesn't match anything except the type checking predicate ROW. Since this would give us a double map (e.g., both BLOCKED-BY and ROAD would go to ROW), we will tentatively reject it. BETA will map BLOCKED-BY to either XED or ZEROED by ignoring BLOCKED-BY's second argument, which is of type PLAYER (see chapter [ANALOGY ALGORITHMS,MAP FORMATION AND EXTENSION], map extension rule 4). Similarly, applying map BETA to the domain semantic template for ON-ROAD results in a partial image semantic template with an unknown image predicate:

```
(? SQUARE ROAD)
```

There is no direct match here, but there is a predicate with the same type inventory (i.e., the same number of each argument type in the semantic template): IN-ROW. We will tentatively use this map (by map extension rule 3). Our analogy map is now

```
ROAD -> SQUARE
TOWN -> ROW
ON-ROAD -> IN-ROW (switch arguments around)
BLOCKED-BY -> XED or ZEROED (igrore second arg)
```

Note that roads and towns are being mapped into type hierarchies. The inverse map will then impose a type hierarchy on the domain (JAM), which is exactly what ought to happen.

USING CONSTRAINT PROPAGATION

Now that we know the functional form of the map, we must determine the details of the correspondence between objects in the two worlds. For example, we have 9 roads and 9 squares. One way to proceed is simply to try all possible maps, and rely on the TTT expert to complain about the maps that are not cricket. Assuming that the map is to be one-to-one, there are 9! or 362880 possible maps. This is too many maps to consider, so we will make use of the type

hierarchy in TTT (the analogy map must be consistent with this hierarchy). The idea is to generate partial maps, and then use the TTT expert to force unique extensions. That is, suppose we pick a road to map to the center square (S5), and then pick 4 other roads to map to the corner squares. Then either the configuration is illegal as it stands, or all subsequent choices are forced. There are 9*C(8,4) or 630 of these maps. We can do even better, since if we pick a road as center and two towns as diagonals, then everything else is determined. There are only

$$1*C(4,2)+4*C(3,2)+4*C(2,2) \text{ or } 22$$

such maps. Our strategy is going to be to map roads to the center (and failing that to corners and sides), and map towns to diagonals (and failing that, to other rows). Expertise in TTT will provide constraints on possible images for roads and towns, and once a particular image is selected, TTT expertise will propagate further constraints. The reader should notice that our constraint propagation scheme does not make use of any expertise in JAM. For a discussion of an alternative "counting" scheme which violates the expertise restriction, see [NOTE 8].

IMAGE SEMANTICS AND DEPTH-FIRST SEARCH

To implement our scheme, we need the notion of "most restrictive type." I don't know a general way to determine when one sub-type is more restrictive than another. However, in finite worlds the most restrictive type is the one with the fewest members. We can use our expertise in TTT to determine that CENTER is more restrictive than the other two types of SQUARE, and similarly that DIAGONAL is more restrictive than ROW (we don't need semantics for the latter since any sub-type is more (or at least as) restrictive than its super-type).

Following this observation, we will guess that

R8 (which should be a corner) -> S5 (the center)

A (which should be a non-diagonal row) -> ROW-G (a diagonal)

This lets us map the first of the JAM declarations.

(ON-ROAD R8 A)->(IN-ROW ROW-G S5)

where the latter is true in TTT. Continuing, we map

```
(ON-ROAD R8 D)->(IN-ROW ROW-H S5)
(ON-ROAD R8 G)->(IN-ROW ROW-B S5)
:
:
(ON-ROAD R5 E) cannot be mapped
```

The semantics of the image rejects this particular map. We eventually go back to R8->S5 and decide that this must not be true, since it has not been possible to assume this and find an analogy map not rejected by the semantics of TTT. We will therefore change our mind, and map R8 to S8 (or, equivalently, some other corner). Proceeding we map A->ROW-G (again assuming diagonal) and

```
(ON-ROAD R8 A)->(IN-ROW ROW-G S8)
    D->ROW-A
    G->ROW-D
(ON-ROAD R5 G)->(IN-ROW ROW-D S5) false!
```

We go back to the last assumption, and try again. This time we map A->ROW-A, and D to a diagonal. This fails, so we finally try mapping G to a diagonal (say ROW-G in TTT world), which succeeds. In this way the map is completed.

The next problem is to map (BLOCKED-BY R8 BLUE)

(BLOCKED-BY R8 BLUE)->(XED R8).

This is rejected by the TTT program as an illegal move because *it* wants to make XED assertions. So we add

BLOCKED-BY -> ZEROED (when second arg = BLUE)

to the analogy map. We will get the other one when TTT gives back proper response, completing the map formation process.

It is worth pointing out once again that all the semantic knowledge resided in the image world. This is entirely fitting, since that is where the supposed expertise is. The result of playing JAM "by analogy" will be the construction of a JAM expert (which will be almost a carbon-copy of

the TTT expert). Since the two worlds are in fact isomorphic, the lifting process is straightforward. We will give an explanation of this lifting process in chapter [ANALOGY ALGORITHMS,DEBUGGING ALGORITHM], and an example in the third vignette (below) operating between the blocks world and rhetoric world.

SUMMARY -- COMBINATORIAL EXPLOSIONS

This TTT example raises the question of combinatorial explosion in the search for an analogy map. In the examples we will examine, this feared combinatorial explosion does not occur. Indeed, we have introduced several techniques to prevent it. However, our success on this example should not be taken as a guarantee that this syntactic search for an analogy map will always work.

For the purpose of making an analogy map, we have assumed that we have no expertise in the domain world, thus restricting ourselves to using only syntactic clues in the statement of the axioms of the domain world. We have also avoided using so-called high-level characterizations of the domain world predicates and functions. It is not hard to imagine that, for example, the hypothesis that some domain predicate is an equivalence relation (if true) could be quite useful in the map formation process.

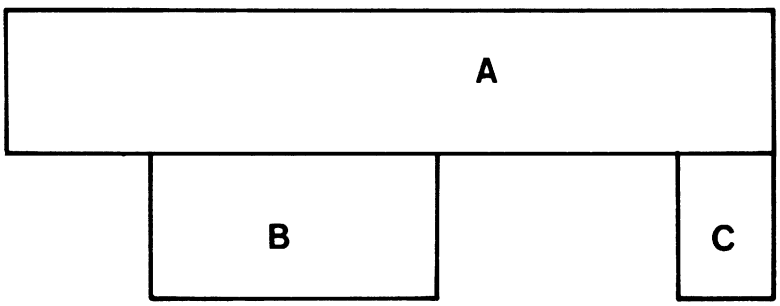
To fully state the rules of TTT, we would need to introduce the notion of change, either by using situation tags ([M5]) or by using some form of modal logic. While I have not fully investigated the impact of the presense of modal operators on the analogy process, they are particularly interesting since the use of modals (syntactically predicates of several variables and one predicate, like NOT, FORALL, and EXISTS) introduces another kind of semantic template, and also allows us to divide predicates into classes depending on which are influenced by which modals. For example, in TTT we might use the AFTER-MOVE-X modal operator to indicate that some experssion becomes true after some square has been XED. Surprisingly, introducing change by introducing modals actually reduces the combinatorial explosion. On the other hand, if situation tags are used in a way that is equivalent to using modals, then there is a slight increase in the

combinatorics of mapping object types due to the introduction of a new type: the situation-tag. Fortunately, by the time the images of the domain predicates are determined, all incorrect maps of object types have been rejected, so the analysis of this case proceeds as above.

INTRODUCTION TO BLOCKS AND RHETORIC WORLD

Our second example is more interesting and less finite. We will make use of an analogy between the notion of physical support in the blocks world and logical support in rhetoric world. The example is particularly interesting due to a demonstration that analogy is able to operate in the presence of logically inconsistent world descriptions.

In the blocks world, if block A's center of gravity is over block B, then (modulo friction and stability considerations) it is safe to remove block C. In this situation, C is said to be scaffolding.



In a discussion or debate, consider some conclusion A in a situation whose essential feature is B. Suppose some inessential feature of the situation C makes the conclusion more palatable. Then one could (successfully) argue that A would still be the appropriate conclusion on the strength of B alone. In this situation, C might be called "window dressing" for conclusion A, provided that C is a relatively minor argument. We take this as the definition of the window dressing predicate.

Suppose we wish to develop a "rhetoric world" expert along the lines of a highly successful blocks world expert. We will do this by means of "reasoning by analogy." In the above, we might discover that the situations are analogous, and thereby become interested in rewriting a program SCAFFOLDP of one argument in the blocks world to become a program WINDOW-DRESSINGP of one argument (in rhetoric world).

SCAFFOLDP -- THE EXAMPLE PROGRAM

Before we can begin to apply the analogy process, we must fully describe the blocks world expert, which is written in LISP. Our attention will be focused on the following program:

BLOCK is the block we suspect is scaffolding.
SP is used for several purposes, but it is the block BLOCK supports inside the loop.
SPSPL is a list of blocks which support the block supported by the block BLOCK.
It is not necessary to understand this program in detail at this time.

```
(DEFUN SCAFFOLDP (BLOCK)
  (PROG (SP SPSPL)
        (SETQ SP (GET BLOCK 'SUPPORTS))
        (COND ((CDR SP) (RETURN NIL)))
        (SETQ SP (CAR SP))
        (SETQ SPSPL (GET SP 'SUPPORTED-BY))
  LOOP
        (COND ((NOT SPSPL) (RETURN NIL))
              ((EQUAL (CAR SPSPL) BLOCK))
              ((STABLE (CAR SPSPL) SP) (RETURN 'TRUE)))
        (SETQ SPSPL (CDR SPSPL))
        (GO LOOP)))
```

This program determines whether or not some block is scaffolding, that is, it determines the truth value of the predicate SCAFFOLD. The program does not return FALSE, so it cannot say definitely that a block is not scaffolding; it is a fairly quick test that can be used when we don't wish to pay the price of a full analysis of the situation.

This program should be viewed as an imperative description of an aspect of behavior in the blocks world. It is a description of how to determine if a block is scaffolding; it does not

describe what the program does, only how to do it in terms of LISP functions. Indeed, complaints that this program is not transparent are quite justified. As we will need this "what" information, we will attach commentary to our program which will describe the plan of the program, after Goldstein [G3].

We need to say that the program first finds a block X such that BLOCK supports it. We will write this using the SUPPORTS predicate:

(SUPPORTS BLOCK X)

The program furthermore determines that X is unique. Then the program tries to find a Y such that Y supports block X:

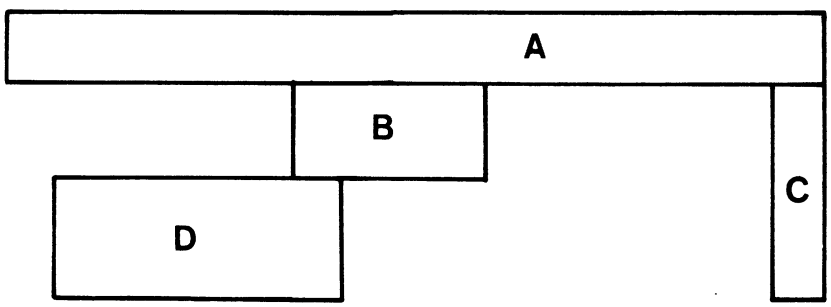
(SUPPORTS Y X)

subject to the restriction that the two blocks X and y be stable in isolation, for which we write:

(STABLE Y X).

INCONSISTENT DESCRIPTIONS ARE ALLOWED

It is worth pointing out here that the program has a "bug" in it. In the following situation, block C is clearly not scaffolding, yet the program will return "true."



While A is "stable" on B considered in isolation, B is not stable on D without the support C. We might insist (say, in AXIOM12), in the description of the blocks world, that if some block X is SCAFFOLDing, then the scene is "globally" stable if X is "disappeared."

```
:AXIOM12
(FORALL (X) (IMPLIES (SCAFFOLD X)
                     (GSTABLE (SCENE-WITHOUT X))))
```

Then the blocks world in inconsistent, since removing C (claimed by our program to be scaffolding) from the above scene results in an unstable block configuration. We will see that this program can nonetheless be proven correct by using a theorem of the blocks world. Indeed, we will see the theorem responsible for this (named FACT29) shortly. Thus it must be the case that the theory of the blocks world that we are using is inconsistent. See [NOTE 9] for a demonstration of the inconsistency.

Minsky closed his paper "A Framework for Representing Knowledge" [M7] with the following paragraph:

> I cannot state strongly enough my conviction that the preoccupation with Consistency, so valuable for Mathematical Logic, has been incredibly destructive to those working on models of mind. At the popular level ... At the "logical" level it has blocked efforts to represent ordinary knowledge, by presenting an unreachable image of a corpus of context-free "truths" that can stand almost by themselves. And at the intellect-modelling level it has blocked the fundamental realization that thinking begins first with suggestive but defective plans and images, that are slowly (if ever) refined and replaced by better ones.

The relevancy of this comment is that, despite what appears to be a predicate-calculus proof approach, we neither insist nor suggest that world theories be either consistent or complete.

FROM CODE TO PLANS

We will find it convenient to describe programs by stating their plans in terms of pattern matching. Thus, we will say that the way SCAFFOLDP determines that (SUPPORTS BLOCK X) is by doing a pattern match on part of BLOCK's representation:

(PATTERN BLOCK SUPPORTS (?X)).

That is, if the list found under the property SUPPORTS has one element, the list matches the pattern, and the pattern variable X is bound to that element. Similarly, the program finds Y by matching

(PATTERN X SUPPORTED-BY (* ?Y *))

where * will match any list of elements. Y supports X, which is also supported by BLOCK, the scaffold candidate. As soon as we make sure that the configuration of X and Y is (locally) stable in STEP3 below, and that BLOCK and Y are not equal (in STEP4), we will be willing to conclude that (SCAFFOLD BLOCK) is true. We can combine these into a description of a plan, specifically a plan TO-DETERMINE the truth value of a predicate SCAFFOLD in four easy steps.

```
(TO-DETERMINE SCAFFOLDP (SCAFFOLD BLOCK)
        (BIND X Y)
:STEP1  (PATTERN BLOCK SUPPORTS (?X))
:STEP2  (PATTERN X SUPPORTED-BY (* ?Y *))
:STEP3  (RESTRICT (Y) (STABLE Y X))
:STEP4  (RESTRICT (Y) (NOT (EQUAL BLOCK Y)))
        (RETURN TRUE))
```

This is the plan of the program. We have labeled the four steps using the colon convention (that is, :LABEL labels the LISP s-expression which follows). We now need to explicitly match up the steps in the plan with the program:

```
(DEFUN SCAFFOLDP (BLOCK)
  (PROG (SP SPSPL)
        (SETQ SP (GET BLOCK 'SUPPORTS))
        (COND ((CDR SP) (RETURN NIL)))
        (SETQ SP (CAR SP))
        ;<- STEP1 (X SP)
        (SETQ SPSPL (GET SP 'SUPPORTED-BY))
  LOOP
        (COND ((NOT SPSPL) (RETURN NIL))
              ;<-STEP2 (Y (CAR SPSPL))
              ((EQUAL (CAR SPSPL) BLOCK))
              ;<-STEP4
              ((STABLE (CAR SPSPL) SP)
              ;<-STEP3
                        (RETURN 'TRUE)))
        (SETQ SPSPL (CDR SPSPL))
        (GO LOOP)))
```

The semi-colen introduces commentary which (in this case) indicates that various plan steps have been tentatively completed, and gives a correspondence between plan variables and code variables.

Having said this, we have two concerns:

(1).Why is it that finding such a block Y allows us to conclude that (SCAFFOLDP BLOCK) should return TRUE?

(2) Why do we believe that the plan actually finds such a Y?

JUSTIFICATION OF PLANS

Suppose that in the description of the blocks world we declare "If some block supports another block, and the scene is stable without the first block, then the first block can be considered scaffolding (for the second)."

```
:FACT29
(FORALL (B1 B2)
        (IMPLIES (EXISTS (B3) (AND (DISTINCT B3 B2)
                                   (SUPPORTS B3 B1)
                                   (STABLE B3 B1)))
                 (SCAFFOLD B2)))
```

Having labeled this declaration FACT29, we can use it as the answer to question (1). This is the fact responsible for the inconsistency noted above.

Turning to question (2), we need to provide justification for the program's commentary (contained in the "plan"). We need to acknowledge the fact that if a block A appears in another block B's representation (under the indicator SUPPORTS), then we know (SUPPORTS B A). The same conclusion can also be drawn if B appears in A's representation under the SUPPORTED-BY indicator.

```
:RT10
(REPRESENTATION-CLAIM
   (X SUPPORTS (* Y *) (SUPPORTS X Y)) <justification>)

:RT11
(REPRESENTATION-CLAIM
   (Y SUPPORTED-BY (* X *) (SUPPORTS X Y)) <justification>)
```

We are using notation explained in chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS]. Labeling these two facts as RT10 and RT11 respectively, we can explain why we believe that the plan will accomplish its aims.

The plan justification has the form of a sequence of named statements relating steps in the plan to facts about the world. Each statement in a plan justification must give a rule by which the predicate is "deduced". The rule %RESTRICT refers to the semantics of that kind of plan step. EQTHM1 and DISTINCT-DEFINITION refer to equality and dictinctness definitions in the blocks world description.

```
(PLAN-JUSTIFICATION SCAFFOLDP
  (L1 (SUPPORTS BLOCK X) RT10 STEP1)
  (L2 (SUPPORTS Y X) RT11 STEP2)
  (L3 (STABLE Y X) %RESTRICT STEP3)
  (L4 (NOT (EQUAL BLOCK Y)) %RESTRICT STEP4)
  (L5 (NOT (EQUAL Y BLOCK)) EQTHM1 L4)
  (L6 (DISTINCT Y BLOCK) DISTINCT-DEFINITION L5)
  (L7 (SCAFFOLD BLOCK) FACT29 L6 L2 L3))
```

The plan justification is interesting in that use is not made of the fact that X is the only block

supported by BLOCK. Just as we will allow world descriptions to be inconsistent, so will we allow plan justifications to be incomplete (one might call this kind of incompleteness a *superstition*).

We now have three layers of knowledge about SCAFFOLDP: we have the actual code (i.e., how to do the computation), we have the plan (i.e., what the computation does), and finally we have the description (i.e., why the computation works). We have commentary linking the code to its plan, and we have the plan justification linking the plan to the description of the blocks world.

FROM PLAN JUSTIFICATION TO RESULT JUSTIFICATION

Having completed the description of the blocks world and its plans, we can finally start to apply the analogy process. In order to make use of code, plan, and justification we need to pose the expert problem solving system a problem. Suppose that in rhetoric world we want to show

```
(WINDOW-DRESSING C)
```

and we know that

```
(SUPPORTS B A)
(SUPPORTS C A)
(DEFENSIBLE A B)
(RELATIVELY-MINOR C A)
```

Assume that by using the analogy map generation process discussed earlier (and perhaps having solved previous problems), we have obtained the following analogy map:

```
WINDOW-DRESSING -> SCAFFOLD
SUPPORTS -> SUPPORTS
DEFENSIBLE -> STABLE
A -> A          RT512 -> RT10
B -> B          RT513 -> RT11
C -> C
```

Applying this map to the summarized rhetoric world problem (see chapter [ANALOGY ALGORITHMS,MAP FORMATION AND EXTENSION]), we get the assertions

```
:GIVEN1 (SUPPORTS B A)
:GIVEN2 (SUPPORTS C A)
:GIVEN3 (STABLE A B)
```

in the blocks world. By mechanisms explained elsewhere (chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS]), these assertions generate objects and representations (i.e., property lists):

```
A       SUPPORTS (), SUPPORTED-BY (C B)
B       SUPPORTS (A), SUPPORTED-BY ()
C       SUPPORTS (A), SUPPORTED-BY ()
```

If in the blocks world we now evaluate the predicate (SCAFFOLDP 'C) it will return TRUE. Interpreting the plan while running the code gives

```
call BLOCK = C
step1 pattern = (A), X = A
step2 pattern = (C B), Y = B
step4 ---true---
step3 ---true--- by GIVEN3
```

Note that Y is bound to B. That is because the first time around the loop, (CAR SPSPL) was C, and of course C equals C.

We use this information in conjunction with the plan justification to generate the following proof that C is SCAFFOLD.

```
1. (SUPPORTS C A)  RT10
2. (SUPPORTS B A)  RT11
3. (STABLE B A) (GIVEN3)
.
.
6. (DISTINCT B C)
7. (SCAFFOLD C) FACT29 applied to 7, 2, 3
```

LIFTING JUSTIFICATIONS OF RESULTS

We can now obtain (via the inverse analogy map) the reason why C's analog (in rhetoric world) is window dressing. While the official justification for the conclusion (SUPPORTS C A) in step 1 of the proof is RT10, we can "unwind" this proof based on representations so that we know step 1 is "given" by GIVEN2 (see [ANALOGY ALGORITHMS,RESULT JUSTIFICATION]).

The only interesting part of this "lifting" process occurs when we try to lift the last step. We have (from the introduction to blocks world and rhetoric world) the following fact:

```
:FACT550
 (FORALL (A1 A2)
         (IMPLIES (EXISTS (A3)
                   (AND (DISTINCT A3 A2) (SUPPORTS A3 A1)
                        (DEFENSIBLE A3 A1) (RELATIVELY-MINOR A2 A3)))
                  (WINDOW-DRESSING A2)))
```

When we apply the inverse map to FACT29 in the blocks world, we discover (using the one-step deduction algorithm in [ANALOGY ALGORITHMS,DEBUGGING ALGORITHM]) that in rhetoric world LIFTED-FACT29 is not provable.

```
:LIFTED-FACT29
(FORALL (B1 B2)
         (IMPLIES (EXISTS (B3) (AND (DISTINCT B3 B2)
                                    (SUPPORTS B3 B1)
                                    (DEFENSIBLE B3 B1)))
                  (WINDOW-DRESSING B2)))
```

We do, however, discover that the consequent of the domain world version of FACT29 matches the consequent of FACT550 and that

1. All antecedents in LIFTED-FACT29 are true

2. LIFTED-FACT29's antecedents are a subset of FACT550's antecedents

3. The rest of LIFTED-FACT550's antecedents are true in rhetoric world.

This configuration indicates that there is a MISSING-PREREQUISITE bug in the analogy

FACT550 -> FACT29

in that the restriction (RELATIVELY-MINOR A2 A3) in rhetoric world's FACT550 was omited. With this noted, the conclusion that C is window dressing is justified. For a full discussing of bugs, see chapter [ANALOGY ALGORITHMS,DEBUGGING ALGORITHMS].

PATCHES IN RESULT JUSTIFICATIONS GIVE PATCHES TO PLANS

We now need to apply the inverse map to the plan. The (RETURN TRUE) step of the plan is justified by step L7 in the plan justification. This step in turn relies on FACT29. When the plan is lifted, we naturally also lift its justification. The bug noted above generates a patch to the plan and plan justification: a further restriction is applied to Y. This gives the domain plan for WINDOW-DRESSINGP:

```
(TO-DETERMINE WINDOW-DRESSINGP (WINDOW-DRESSING ARGUMENT)
        (BIND X Y)
:STEP1  (PATTERN ARGUMENT SUPPORTS (?X))
:STEP2  (PATTERN X SUPPORTED-BY (* ?Y *))
:STEP3  (RESTRICT (Y) (DEFENSIBLE Y X))
:STEP4  (RESTRICT (Y) (NOT (EQUAL ARGUMENT Y)))
:PATCH1 (RESTRICT (Y) (RELATIVELY-MINOR ARGUMENT Y))
        (RETURN TRUE))
```

and the appropriately patched plan justification (not shown).

We now lift the code. Most of the work is simply replacing function names. The only

problem is where to insert the patch, and what the modification looks like. In this case we order the difficulties associated with routines which compute the restriction and the difficulty associated with the routine to compute the patch. Then we insert code for the patch immediately before the code computing the next most difficult plan step (or, rather, before the code that completes the next most difficult plan step). In this case, the only step more difficult to compute than the patch is STEP4. Thus in the code, we will place code for PATCH1 immediately in front of code for STEP4. This gives

```
(DEFUN WINDOW-DRESSINGP (ARGUMENT)
  (PROG (SP SPSPL)
        (SETQ SP (GET ARGUMENT 'SUPPORTS))
        (COND ((CDR SP) (RETURN NIL)))
        (SETQ SP (CAR SP))
        ; <- STEP1 (X SP)
        (SETQ SPSPL (GET SP 'SUPPORTED-BY))
LOOP
        (COND ((NOT SPSPL) (RETURN NIL))
                    ; <- STEP2 (Y (CAR SPSPL))
              ((EQUAL (CAR SPSPL) ARGUMENT))
                    ;<- STEP4
              ((AND (RELATIVELY-MINOR ARGUMENT (CAR SPSPL))
                          ;<- PATCH1
                    (DEFENSIBLE (CAR SPSPL SP))
                          ;<- STEP3
               (RETURN 'TRUE))))
        (SETQ SPSPL (CDR SPSPL))
        (GO LOOP)))
```

Actually writing the patch requires some sophistication in programming.

Note that this program has the same "bug" (confusion between local and global defendability) that SCAFFOLDP has. Furthermore, like SCAFFOLDP, it superstitiously insists that the WINDOW-DRESSING support only one argument -- even though there is no apparent good reason for doing so.

In place of "point", "line", and "plane" we must at all times be able to say "beer mug", "table", and "chair."

--- David Hilbert

GEOMETRY WORLD

We saw in chapter [OVERVIEW OF ANALOGY,FIRST VIGNETTE -- TIC-TAC-TOE] the dangers of trying to study reasoning by analogy between isomorphic worlds. This observation gives us the first of the following criteria for the subject of our next, more advanced and interesting example.

Non-isomorphism. The pair must not be isomorphic.

Richness. The worlds should be rich in analogies.

Non-trivial. The worlds must be non-trivial, since almost any scheme will work on toy problems. Ideally we would choose a world in which there are still unsolved problems.

Well-defined. The worlds should be well-defined and understood. Preferably the description of the worlds should be obtained independently of our investigation.

Existing Expertise. There should already be an expert problem solver for these worlds. Again, we prefer that the expert be developed independently. Ideally, we want several expert problem solvers for the worlds

On the basis of these criteria we chose plane geometry and solid geometry as the pair of worlds in which to study analogy. We then restricted ourselves to a small portion of geometry, called *incidence geometry*, which concerns problems of points, lines, and planes intersecting and being determined (as in "two points determine a line"). Even these portions of geometry satisfy the criteria given above.

Recall from [INTRODUCTION TO ANALOGY, ANALOGY AND MAPS] that we plan to reason about the domain world, solid geometry, by solving analogous problems in the image world, plane geometry. However, before we can use analogy on these two worlds we must first:

1. Write the *code* for a plane geometry world expert problem solver.

2. Give the *plans* for the *code*.

3. Give the *justification* for believing the *plans* are correct.

4. In order to give the justification of the *plans*, we will need a *description* of plane geometry world.

There are a variety of ways to describe geometry world: set theoretic [F3] and analytic [W8] come to mind. We will use Hilbert's axioms [H2] to describe these two worlds, both because his axioms are closest in spirit to Euclid, and because in the literature they are the most widely used.

PLANE Geometry Axioms

We will have two groups of axioms, concerned with *incidence* and with *order*. The axioms are numbered so as to correspond with Hilbert's axioms in [H2].

> Plane geometry axioms are preceded with a P, while solid geometry axioms are preceded with S. We will give the predicate calculus versions of the axioms used in the problems. The remainder of the axioms are given for completeness.

P-I1. *Given two points, there is a line that contains them.*

```
(FORALL (A B) (IMPLIES (AND (PT A) (PT B))
                       (AND (LN (LINE A B))
                            (IN-LN (LINE A B) A)
                            (IN-LN (LINE A B) B))))
```

Note that we don't insist that the two points be distinct. This claims that two points determine at least one line.

P-I2. *For every two distinct points, no more than one line contains them.*

```
(FORALL (A B)
     (IMPLIES (AND (DISTINCT A B) (PT A) (PT B))
              (NOT (EXISTS (X Y) (AND (DISTINCT X Y) (LN X)
                                      (LN Y) (IN-LN X A)
                                      (IN-LN Y A) (IN-LN X B)
                                      (IN-LN Y B))))))
```

P-I3a. *Each line contains at least two points.*

The predicate calculus statement of this axiom has an occurance of the LN predicate on the left of an implication. This tells us that LN, a unary predicate, is a type checker (see [OVERVIEW OF

ANALOGY,FIRST VIGNETTE--TIC-TAC-TOE]).

P-I3b. *There are at least three non-collinear points.*

We also have a set of axioms dealing with the concept of "order." These are included, even though our examples are in incidence geometry, because of the important role they play in the representation of a line.

P-III1a. *The BTWN relation implies that the points are co-linear.*

```
(FORALL (A B C)
     (IMPLIES (AND (PT A) (PT B) (PT C) (BTWN A B C))
              (EXISTS (L) (AND (LN L) (IN-LN L A)
                               (IN-LN L B) (IN-LN L C))))))
```

P-III1b. *The order of the BTWN arguments may be reversed.*

```
(FORALL (A B C)
        (IMPLIES (AND (PT A) (PT B) (PT C) (BTWN A B C))
                 (BTWN C B A)))
```

P-II2. *For two points A and C there always exists at least one point B on the line containing A and C such that C lies between A and B.*

P-II3. *Of any three points on a line there exists no more than one that lies between the other two.*

Using the axioms above, we can prove important theorems, like

```
P-BTWN-THEOREM1:
        (FORALL (A B C D)
                (IMPLIES (AND (PT A) (PT B) (PT C) (BTWN A B C) (BTWN B C D))
                         (BTWN A C D)))

P-BTWN-THEOREM2:
        (FORALL (A B C D)
                (IMPLIES (AND (PT A) (PT B) (PT C) (BTWN A B C) (BTWN A C D))
                         (BTWN B C D)))
```

In addition to axioms, we also include definitions. For various reasons, we disallow definitions which introduce any new knowledge; definitions are strictly and purely notational. Enforcing this edict is harder than one might think. For example, we have the definition of INTERSECT:

P-DEF1. *Define a function INTERSECT. Insist that it take two distinct lines as arguments. If this condition is met, then the result is in both of the given lines.*

```
(DETERMINES (INTERSECT A B)
            ((LN A) (LN B) (DISTINCT A B))
            ((IN-LN A (INTERSECT A B))
             (IN-LN B (INTERSECT A B))))
```

The form for the function call is followed by a list of restrictions on the arguments (i.e., the function is only defined if the arguments meet these restrictions), and then a list of claims about the value returned by the function.

This definition requires some explanation. We have not declared the type of the result of applying the INTERSECT function to two distinct lines. We have only given the minimal properties we wish this returned object to have. The meaning of a DETERMINES definition is "If it can be proven for some type Q (Q may be, for example, PT, LN, or PL) that two objects of type Q are equal if they fulfill the claims of the definition, then the type of the value returned by the function being defined is Q." We can then prove that if two distinct lines have two points X and Y in common, then X = Y. Hence the intersection of these two lines is the point X (or the same point under the name Y), and thus we have proven that INTERSECT returns objects of type PT.

We are forced to define INTERSECT this way by purely logical considerations. (For a full discussion, see [NOTE 10]). However, this definition also makes reasoning about intersection by analogy easier. If we uniformly replace IN-LN by IN-PL, we will be able to prove in solid geometry that the intersection of two planes is not a point, and that it is a line. A very syntactic and natural transformation is all that is required to "lift" the definition.

SOLID Geometry Axioms

S-I1,S-I2,S-I3a,S-I3b,S-IIIa,S-IIIb,S-II2,S-II3 are all identical to P-etc.

S-DEF1. *We define a predicate of three arguments to be true if and only if the three arguments are points and there is no line containing all of them.*

```
(DEF-PRED (NON-LN A B C)
          (AND (PT A)
               (PT B)
               (PT C)
               (NOT (IN-LN (LINE A B) C))))
```

S-I4a. *For three non-collinear points there is always a plane containing them.*

```
(FORALL (A B C)
        (IMPLIES (NON-LN A B C)
                 (AND (PL (PLANE A B C))
                      (IN-PL (PLANE A B C) A)
                      (IN-PL (PLANE A B C) B)
                      (IN-PL (PLANE A B C) C)))))
```

S-I4b. *Every plane contains at least one point.*

```
(FORALL (P) (IMPLIES (PL P)
                     (EXISTS (A) (AND (PT A) (IN-PL P A))))))
```

S-I5. *For three non-collinear points, no more than one plane contains them.*

S-I6. *If two points of a line are in a plane, then all points in that line are in the plane.*

S-I7. *If there is one point in two distinct planes, then there is a second distinct point also in both planes.*

Note that INTERSECT is not defined in solid geometry. Although the above five axioms completely describe what a plane is, they don't give even a hint about how planes should be represented or manipulated by a program -- analogy must figure that out for itself.

Books on solid geometry are relatively rare. Lines [L2] is a good source of easy problems (and hard ones).

SEMANTIC TEMPLATES

The semantic templates for the two geometry worlds are derived according to the scheme outlined in chapter [OVERVIEW OF ANALOGY,FIRST VIGNETTE -- TIC-TAC-TOE].

```
SOLID                                PLANE
(IN-LN LN PT) predicate              (IN-LN LN PT) predicate
(LINE PT PT) function, val=LN        (LINE PT PT) function, val=LN
(BTWN PT PT PT) predicate            (BTWN PT PT PT) predicate
                                     (INTERSECT LN LN) function, val=PT
(PLANE PT PT PT) function, val=PL
(IN-PL PL PT) predicate
```

LANGUAGE FOR PLANS

Having described geometry world, we must now develop plans for dealing with it. A plan is something more abstract than a program, but still has an algorithmic feel to it. Goldstein [G3] suggests that a plan consists of PURPOSE statements attached to lines of code:

```
(PURPOSE code-reference explanation).
```

Goldstein gives examples of explanations:

(INSERT TRUNK TOP) -- Accomplish the TRUNK model part by a "state transparent sub-procedure" inside the code for TOP.
(ACCOMPLISH (PIECE i (SIDE 1 TRIANGLE))) -- model parts can be divided into pieces, each of which can be accomplished independently.
(SETUP HEADING FOR TRUNK) -- before a model part can be accomplished, a setup step may be necessary.

To Goldstein a plan is a sequence of purpose statements corresponding to an execution sequence (Manna and Wadlinger [M2]). Thus, if the program has a loop, we will have ROUND PLANS. If it uses subroutines, it might require an INSERT PLAN. This notion of plan seems quite popular. We find it again in Waters [W2] and Rich and Schrobe [R1]. Clearly, something describing control structure must be included in commentary somewhere, but if we can, we would like to use a more

abstract kind of plan.

Pratt [P3] suggests that programs be composed of two distinct components: the *competence* component which is evidently close to an axiomatization of the world, and a *performance* component, which consists of various heuristics offering advice about computations discussed in the competence component. From this I gather that the performance component contains plans, although Pratt's examples do not always support this conclusion.

PLANS ARE PROGRAMS SANS CONTROL STRUCTURE

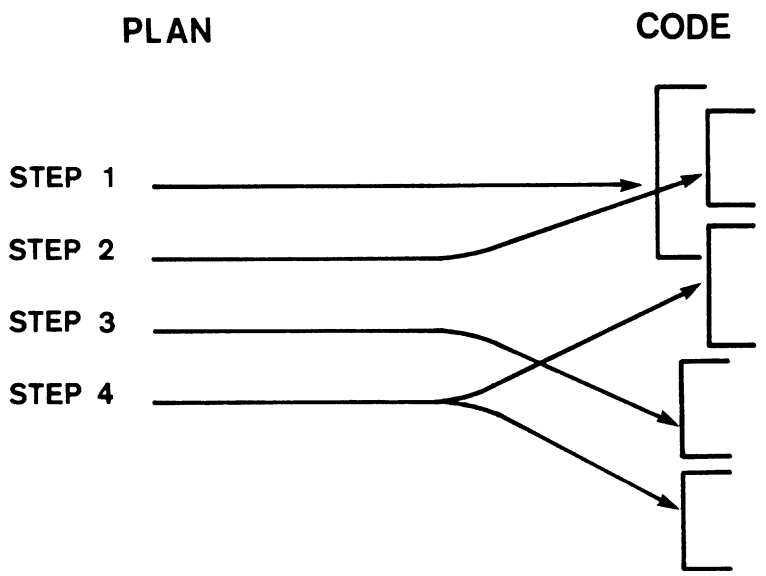
Good methodology demands that we give a definition of what a plans *are* which is independent of the way we represent them. Doing this requires two steps. The first, a procedure for generating programs from plans, gives a lower bound to the amount of knowledge in a plan. The second step derives a plan from a program by factoring out *control structure*. This sets an upper bound on the amount of knowledge in a plan. Anything between these two bounds will serve as a plan. In particular, it appears that Rich and Schrobe [R1] have a structure they call a *deep plan* which possibly can be abstracted into our notation for plans.

Suppose one is presented with a plan written in some suitable language (we will give such a language shortly). Since we claim plans are closely related to programs, we would like to believe we can more or less mechanically translate a plan into a program. A plan is a collection of steps which, when all simultaneously satisfied, insure that the goal (or intention) of the plan is satisfied. We demand that steps which give values to variables (the *set-point* for the variable) in the plan occur before the steps which use those variables (of course). Within this restriction, the algorithm randomly orders the plan steps. This insures that no control structure can be "hidden" in the original sequence of plan steps. The algorithm then takes the plan steps and macro-expands each one into a conditional (the plan step provided the predicate for the conditional) whose success branch goes to the next step, and whose failure branch (if present) goes to the most recent set-point

for a variable used in the expression. The control structure imposed by this algorithm is reminiscent of CONNIVER possibilities lists.

We describe this algorithm to set a lower bound on the amount of knowledge present in plans. Since we plan to use plans as a link between CODE and world DESCRIPTION, it is natural for us to ask that plans be written in terms of the world description instead of (say) procedure calls.

The algorithm just sketched generates a program, but generally not the program that is actually used by the expert. In going from code to plans, which are more abstract, we factor out *control structure*. In going from plan to program, plan steps can be reordered, overlapped, and split up. This observation sets an upper limit on the amount of knowledge in a plan.



Our view, then, is that plans are a list of goal descriptions with sufficient detail provided so that the plan may be algorithmically transformed into a program (and this algorithm should employ no additional information other than perhaps extrinsic descriptions of other plans). Plans are linked to programs by commentary describing the way plan variables are related to program variables, and the control structure imposed on the plan steps. It may be the case that code is linked by this commentary to several plans, as would be the case with some COND constructs.

PLANS, OBJECTS, AND REPRESENTATIONS

In order to write plans, one must adopt a way of looking at computations. Our view of computations, the *geometry formalism*, uses the notions of objects and their types and representations. It also uses the notion of a data base. This plan-level structure roughly corresponds to a CONNIVER context, with certain (severe) restrictions on the type of assertions that can be made. The *implementation* of a data base might be a CONNIVER-like associative data base.

In addition to the object types implied by the world description (lines, points, and planes), we will make use of equality buckets and "non-equality" (distinctness) buckets. Each object has a name. Two objects are EQ if they have the same name. We will see that two objects with different names may be EQUAL (and that this situation is *unavoidable* in geometry world).

In addition to names, an object may have one or more *representations*. For example, the representation of a line is an ordered list of points in that line. We keep representations under an *indicator* (in geometry the indicator happens to be the name of the type of the object). Two objects may have identical representations and be unequal, or they may be equal with differing representations.

As we mentioned earlier, one way to implement representations is to use property lists. Although the property list is a good model of representations, one should remember that representations are part of the plan level (not the code level) of the theory of geometry world.

WRITING PLANS

We will describe one way to write plans. The reader should be warned that the plan language we will present is somewhat limited in its expressive power. However, this limitation does not limit our ability to write code, only our ability to say what that code does. As to the degree of limitation, we will provide a surprisingly precise answer in the chapter [LOGIC OF EXPERTS].

In looking at various worlds, we find that plans deal with two basic operations: finding the name of an object from its description, and maintaining representations. For example, looking back to chapter [OVERVIEW OF ANALOGY,SECOND VIGNETTE -- BLOCKS] at the SCAFFOLDP plan, we can see that it searches for two blocks X and Y which meet some set of conditions. If these can be found, then a conclusion can be drawn.

The plans which find names of objects can be further subdivided into plans which correspond to predicates and those which correspond to functions. These considerations will lead to the following types of plans:

MAINTAINING REPRESENTATIONS
- TO-REPRESENT -- creates a representation
- TO-INCORPORATE -- adds an item to a representation
- TO-COMBINE -- combines two representations

BACKWARD-CHAINING COMPUTATIONS
- TO-FIND -- a function plan
- TO-DETERMINE -- a predicate plan

FORWARD-CHAINING COMPUTATIONS
- TO-CORRELATE -- generates trivial corollaries to assertions

SEARCH COMPUTATIONS
- TO-GENERATE -- a function plan which, given a predicate with one of its arguments unknown, returns possible objects for that unknown.

A plan has the form

(plan-type procedure-name world-description-form <step>$_j$)

Within a plan, several types of steps can appear. In the SCAFFOLDP plan the step types BIND, PATTERN, RESTRICT, and RETURN occur. We now will give the complete list.

(BIND <variable-spec>$_j$) Each *variable-spec* is either the name of a variable, or of the form (variable (function <arg>$_i$)). This kind of plan step is used to establish variable bindings.

(MAKE variable type) Create a new object of specified *type*, and give *variable* this object as its value.

(CONDITION predicate) Insist that the *predicate* is true. Typically a CONDITION failure indicates that the plan is not applicable.

(PREDICATE variable-list predicate) Assign values to variables in the *variable-list* such that *predicate* is true. Typically each variable will be given a list (or *segment*) of possible values by a TO-GENERATE plan or a data base search.

(PATTERN object indicator match-pattern) Note: the *object* or variables in the *match-pattern* can be unknown. If multiple pattern steps with the same *object* and *indicator* appear, then the *exclusive-or* of those steps is intended. This is the only implicit disjunction allowed in plans.

(RESTRICT variable-list predicate) The *predicate* must be true. The *variable-list* contains the variables whose values are being restricted. If this step fails, control (in the code corresponding to the plan) goes to the most recent set-point for a variable in the variable-list to find another value (if possible).

(ABSORB predicate) *Predicate* has been discoverd to be true and incorporated into a representation, so it should no longer be in the data base. Information can be found either in the data base or encoded in a representation. This plan step *conceptually* removes information from the data base that has been encoded in a representation. If the code does not maintain a data base, then there probably will not be code to implement this plan step.

(FREEZE object indicator) The representation under *indicator* on *object* should no be further modified. This is used when an equality is discovered to prevent redundancy. It will not be used in any examples, and is included for completeness.

(ASSERT predicate) *Predicate* has been discovered to be true, and should be remembered in the data base.

(RETURN value) *Value* will be TRUE or FALSE for predicates, and some object for functions and generators.

MATCHING

Matching, used in PATTERN plan steps, is the only mechanism by which representations of objects can be modified at the plan level of a theory (or, if you wish, the plan component of expertise). A pattern has the form

$$(var_1\ var_2 \ldots var_n)$$

where each var (a variable, with value either a single object, or a *segment*, which is a list of objects) is in one of the following forms:

name -- name must be bound before a pattern match is attempted. Only the object "name" or another object EQUAL to "name" can be matched.

=name -- name must be bound before pattern match is attempted. Only the object "name" can be matched. This is only used for dealing with equality buckets, and does not occur in any examples.

?name -- name is unbound. This will match any element, with "name" being bound to the object matched.

? -- as above, but no variable binding results

*name -- name is unbound. This will match a segment (of any length, including zero length). Name will be bound to the segment.

* -- as above, but no variable binding results.

^name -- name must be bound. During matching "name" is ignored. If the match is successful, the structure matched will be modified so that the object (or objects if "name" is a segment) will appear in this location. Duplications (i.e., EQ objects) will be disallowed. If "name" is bound to a sequence, the elements of the sequence will be inserted in order, with duplications eliminated. If "name" is bound to a single object, that object will be inserted if no duplications will result.

PATTERN steps include a hidden use of EQUAL so that usually one need not worry explicitly about equality in plans. We will see examples of cases where we *do* become concerned explicitly with equality relations in [GEOMETRY WORLD,CONSTRUCTIONS].

GLOBAL PLANS

We have built the notion of EQUAL into the plan language. To support this, we need to be able to give two other kinds of information. The first links type checking predicates at the descriptive level to the notion of type at the plan level:

(TYPE-CHECKER unary-predicate type-name)

This gives the notion of *type* special significance in plans.

Since equality is checked differently according to type (for example, the rational numbers A/B and C/D are equal as rational numbers if A*D is equal as an interger to C*B) we need to specify how two objects of the same type can be determined to be equal. (We ignore the case of objects of different type being equal). For this we use the form

(DEFINE-EQUAL type binary-predicate-name)

where the default binary predicate name is EQUAL. It goes without saying that the binary predicate must be an equivalence relation.

Thus in plane geometry

```
(TYPE-CHECKER PT PT)
(DEFINE-EQUAL PT EQUAL)
```

would say that PT is a type checking predicate at the descriptive level for objects of type PT at the plan level, and that to determine if two objects of type PT are equal, the predicate EQUAL should be used.

GEOMETRY FORMALISM

We have not given much attention to actual code, preferring rather to restrict ourselves to plans and descriptions. We instead assume that there exists an algorithm which produces implementation code from plans. Indeed, part of our definition of plan implied the existence of such an algorithm.

We base our formalism on four rather similar geometry theorem provers written by Gelernter [G2], Goldstein [G4], Nevins [N2], and Ullman [U1]. These all employ a single data base into which only true facts are asserted. We will also allow the assertion of negated predicates, since if the predicate is known to be false, its negation is a true fact. Disjunctions *cannot* be asserted into the data base. Since implications are logically equivalent to disjunctions, they also cannot be asserted.

The control structure used by these geometry theorem provers was basically a pattern-directed multi-processing AND/OR tree (Nilsson [N3]). Each node of AND/OR tree is a possibly "hung" process which does not communicate with any other node except for returning results. Associated with each node of the tree is a *priority*. These priorities (which are generated in some automatic way) are used to determine which node to make active next. This control structure is an early AI result [S5]. Since we have both a data base and the special representations, we can make deductions based on either. We will always prefer representation-based deductions. Finally, we note that when attempting to prove some proposition, we must simultaneously attempt to disprove it, since failing to prove a proposition does not mean the proposition is false.

EXAMPLE PLANS

We will make certain claims about the plan level representations of lines in plane geometry. According to representation claim RC1, if an object Y (Y should be a point) is in the representation of a line X, then we can conclude that Y is in the line X. According to RC2, we claim that objects in a line's representation are ordered by the between predicate BTWN.

```
:RC1
(REPRESENTATION-CLAIM (X LN (* Y *) (IN-LN X Y)) ...)

:RC2
(REPRESENTATION-CLAIM (X LN (* U * V * W *) (BTWN U V W)) ...)
```

We maintain these representations of lines with programs whose plans are as follows:

```
(TO-REPRESENT IL1 (IN-LN L A)
:IL1-1   (PATTERN L LN (^A))
:IL1-2   (ABSORB (IN-LN L A)))
```

This plan says that if L is a line and A is asserted to be in line L, and if L's representation under the indicator LN is currently empty, then add A to it. Having done this, there is no need to keep the assertion (IN-LN L A) in the data base, as it has been absorbed into the representation of line L. A TO-REPRESENT plan creates new representations. We use TO-INCLUDE plans to add to representations.

```
(TO-INCLUDE IL2 (IN-LN L A)
:IL2-1   (PATTERN L LN (? ^A))
:IL2-2   (ABSORB (IN-LN L A)))
```

This plan says that if A is asserted to be in line L, and L's representation under the indicator LN already has one point in it, then A should be added to the representation. We have been negligent in not making sure that the new point is not EQUAL to the point already present (see chapter [GEOMETRY WORLD, CONSTRUCTIONS]).

The plan below handles the case when we know

(BTWN X Y Z)

and a line containing the point X has a representation (under the LN indicator)

(Y Z W).

The plan then inserts X at the beginning of the representation, g ving a new representation

(X Y Z W).

```
(TO-INCLUDE IL3 (IN-LN L A)
:IL3-1   (BIND B C D)
:IL3-2   (PATTERN L LN (^A B C *D))
:IL3-3   (PATTERN L LN (*D C B ^A))
:IL3-4   (RESTRICT (B C) (BTWN A B C))
:IL3-5   (ABSORB (BTWN A [C B] [C D]))
:IL3-6   (ABSORB (BTWN [C D] [B C] A))
:IL3-7   (ABSORB (IN-LN L A)))
```

The square brackets are "combinatorial" so that

[C B]

is either C or B. Since the two PATTERN steps IL3-s and IL3-3 have the same object and indicator, either pattern match will do; the two pattern steps are logically disjoint.

The following plan handles the more complex case of putting the point A into the middle of a representation.

```
(TO-INCLUDE IL4 (IN-LN L A)
:IL4-1   (BIND B C D E)
:IL4-2   (PATTERN L LN (*B C ^A D *E))
:IL4-3   (RESTRICT (C D) (BTWN C A D))
:IL4-4   (ABSORB (BTWN B C A))
:IL4-5   (ABSORB (BTWN A C B))
:IL4-6   (ABSORB (BTWN A D E))
:IL4-7   (ABSORB (BTWN E D A))
:IL4-8   (ABSORB (BTWN [B C] A [D E]))
:IL4-9   (ABSORB (BTWN [D E] A [B C]))
:IL4-10  (ABSORB (IN-LN L A)))
```

The code corresponding to the above plans maintains representations for use by other code. The plans for two predicates which use these representations are given below.

```
(TO-DETERMINE IN-LNP (IN-LN X Y)
:IN-LNP-1        (PATTERN X LN (* Y *))
:IN-LNP-2        (RETURN TRUE))


(TO-DETERMINE BTWNP (BTWN X Y Z)
:BTWNP-1         (BIND L)
:BTWNP-2         (PATTERN ?L LN (* X * Y * Z *))
:BTWNP-3         (PATTERN ?L LN (* Z * Y * X *))
:BTWNP-4         (RETURN TRUE))
```

Notice that the BTWNP plan searches for a line by having "?L" in a PATTERN plan step. The plan need not specify how this search is to be accomplished because that is code-level knowledge.

The above provide a fairly complete picture of the plans concerning the representation of lines. It is sometimes necessary to duplicate information. For example, we also need the following plan.

```
(TO-CORRELATE INTERP-BTWN1 (BTWN X Y Z)
:IBTWN1 (BIND W)
:IBTWN2 (PREDICATE W (BTWN X Z W))
:IBTWN3 (ASSERT (BTWN Y Z W)))
```

This plan duplicates a fact used in the justification of BTWNP. Given an assertion (BTWN A B C), if we can find that (BTWN A C D), then we can conclude that (BTWN B C D). This plan is necessary because we occasionally need to draw inferences about BTWN when not all points have been inserted into a representation. We will later see an example of this (chapter [ANALOGY ALGORITHMS,RESULT JUSTIFICATION]). At the code level we know we would not want to "run" this TO-CORRELATE plan if we can absorb the BTWN assertion into a representation. The plane geometry expert has many more plans than those above We will have an opportunity to examine more of them in chapter [ANALOGY EXAMPLES, NON-TRIVIAL ANALOGY].

PLAN JUSTIFICATIONS

Plan justifications, which must be supplied to the analogy process, have the form

(PLAN-JUSTIFICATION plan-name <step>$_j$)

where each step has the form

(step-label predicate rule <argument>$_i$).

For example, we repeat the plan justification given earlier for SCAFFOLDP (chapter [OVERVIEW OF ANALOGY,SECOND VIGNETTE -- BLOCKS]):

```
(PLAN-JUSTIFICATION SCAFFOLDP
  (L1 (SUPPORTS BLOCK X) RT10 STEP1)
  (L2 (SUPPORTS Y X) RT11 STEP2)
  (L3 (STABLE Y X) %RESTRICT STEP3)
  (L4 (NOT (EQUAL BLOCK Y)) %RESTRICT STEP4)
  (L5 (NOT (EQUAL Y BLOCK)) EQTHM1 L4)
  (L6 (DISTINCT Y BLOCK) DISTINCT-DEFINITION L5)
  (L7 (SCAFFOLD BLOCK) FACT29 L6 L2 L3))
```

Three kinds of rules are used in plan justifications:

1. Theorem of world description. The arguments to this theorem will be other plan justification steps corresponding to the antecedents of the theorem (for example, step L7 above). We suppressed the correspondence between plan variables and theorem variables.

2. Semantics of a plan step (which is given as the argument). For example step L3 is true because of the result returned by STEP3 of the plan, which evidently was a RESTRICT plan step.

3. Representation claim. The argument will be a PATTERN plan step. For example, justification step L2 above. Again, the correspondence between plan variables and claim variables has been supressed.

PROVING REPRESENTATION CLAIMS

We are now in a position to prove that the two representation claims, RC1 and RC2, are true. In the literature on proving programs correct, one typically finds that proving properties about some data structure is very difficult. It is an interesting feature that in our plan language the proof of a representation claim always has a very simple and regular form. We will make use of this simplicity in the result justification algorithm (see [ANALOGY ALGORITHMS,RESULT JUSTIFICATION]).

The proof of a representation claim is always by induction on the size (length) of the representation. Each TO-REPRESENT plan contributes a *base step*, each TO-INCORPORATE plan contributes a *weak* induction step, and each TO-COMBINE plan contributes a *strong* (or *course-of-values*) induction step, required, for example, to combine congruence classes.

Let's start with RC1, since that is the easiest.

```
(X LN (* Y *) (IN-LN X Y))
```

As promised, the base step is given by the plan for IL1, which should be viewed as a kind of IF-ADDED demon. Matching up the variables in the claim with those in the code (i.e., (X L) (Y A)), we see that if the code actually does what the plan says (as we shall always assume), then the representation claim will be true after the code runs, since the call pattern (IN-LN L A) had to be true in order for the program IL1 to run. We write this as

```
(IL1-1 (((X L) (Y A)) (L1 (IN-LN X Y) %CALL)))
```

where %CALL is the reason we just gave that (IN-LN X Y) is thought true.

Similar reasoning holds for the plans IL2, IL3, and IL4, giving us the complete representation claim.

```
:RC1
(REPRESENTATION-CLAIM (X LINE (* Y *) (IN-LN X Y))
        (IL1-1 (((X L) (Y A)) (L1 (IN-LN X Y) %CALL)))
        (IL2-1 (((X L) (Y A)) (L1 (IN-LN X Y) %CALL)))
        (IL3-2 (((X L) (Y A)) (L1 (IN-LN X Y) %CALL)))
        (IL3-3 (((X L) (Y A)) (L1 (IN-LN X Y) %CALL)))
```

(IL4-2 (((X L) (Y A)) (L1 (IN-LN X Y) %CALL))))

With this, we can produce a PLAN-JUSTIFICATION for IN-LNP.

It is more difficult to prove representation claim RC2:

(X LINE (* U * V * W *) (BTWN U V W))

This claim is vacuously true after IL1 and IL2 have run, so we don't have an explicit base step! So much for rigor. Looking at IL3, we need to separate the insertions by IL3-2 and IL3-3 as we did above. Concentrating on line IL3-2, if we match up the variables

(U A) (V B) (W C)

then the claim is true due to the restriction in line IL3-4. If we match up the variables with

(U A) (V C) (W D)

then we know (BTWN %B V W), where in the proof of a representation claim we will write %B to reference the variable B in the plan being discussed. From the restriction we can conclude (BTWN U %B V). Finally we have a geometry theorem P-BTWN-THEOREM1 (in chapter [GEOMETRY WORLD,AXIOMS]) which lets us use these two conclusions to deduce (BTWN U V W).

The third way to match up the variables is

(U A) (V D) (W D).

We conclude, as above, (BTWN %C V W) and (BTWN %B %C V) by induction. The restriction insures (BTWN U %B %C). Two applications of P-BTWN-THEOREM1 give us our result. Putting all of this together we have part of the justification of representation claim RC2:

```
(REPRESENTATION-CLAIM (X LN (* Y *) (IN-LN X Y))
(IL3-2
   (((U A) (V B) (W C))
        (L1 (BTWN U V W) %RESTRICT IL3-4))
   (((U A) (V C) (W D))
        (L1 (BTWN %B V W) RC2 IL3-2)
        (L2 (BTWN U %B V) %RESTRICT IL3-4)
        (L3 (BTWN U V W) P-BTWN-THEOREM1 L1 L2))
   (((U A) (V D) (W D))
        (L1 (BTWN %C V W) RC2 IL3-2)
        (L2 (BTWN %B %C V) RC2 IL3-2)
        (L3 (BTWN U %B %C) %RESTRICT IL3-4)
        (L4 (BTWN U %C V) P-BTWN-THEOREM1 L2 L3)
        (L5 (BTWN U V W) P-BTWN-THEOREM1 L1 L4))) . . .)
```

Analysis for IL3-3 and IL4-2 is similar. With RC2 proven we can justify the plan for BTWNP.

In chapter [ANALOGY ALGORITHMS,RESULT JUSTIFICATION] we will show how these representation claims and their proofs are used by the analogy process.

EXTENDED PREDICATE FORMS

Another important role which representation claims play concerns *extended predicate forms*. We find it convenient to write (IN-LN L A B) instead of saying both (IN-LN L A) and (IN-LN L B). The representation claim RC1 above allows us to convert the claim (IN-LN L A B) directly to giving L a representation under the LN indicator of the list (A B), provided L does not already have a representation there. Similarly, if we have the two claims

```
(IN-LN L A B C E)
(BTWN A B C E)
```

we can give line L the represention (A B C E) without going through an expansion and contraction. Without the IN-LN claim, we would use the representation claim RC2

```
(* U * V * W *) --> (BTWN U V W)
```

to transform

```
(BTWN A B C E) -->
                (BTWN A B C)
                (BTWN A B E)
                (BTWN A C E)
                (BTWN B C E)
```

We will be able to use these representation claims to quickly translate information from one world to another. Of course, this use of representation claims is only incidental.

If the predicate is a type-checker, we make use of a standard object of type DBUCKET for which we have a representation claim

```
(* X * Y *) -> (NOT (EQUAL X Y))
```

This agrees with conventional usage, so that declaring (LN L1 L2 L3) will cause us to create a distinctness bucket (say) G015 with representation (L1 L2 L3) under the DBUCKET indicator.

CONSTRUCTIONS

Writing free verse is like playing tennis with the net down.
-- Robert Frost

As with certain kinds of modern poetry, writing a geometry expert that doesn't deal (at some level) with constructions is like playing tennis without a net. The construction problem is very hard; we should distinguish between different kinds of constructions in the hope of finding some kind that we can completely handle.

Consider the following problem:

```
TRIVIAL PROBLEM
Given:  (PT C)
        (LN L1)
        (LN L2)
        (NOT (EQUAL L1 L2))
        (IN-LN L1 C)
        (IN-LN L2 C)
        (IN-LN L1 (INTERSECT L2 L3))*
Prove:  (IN-LN L3 C)
```

Geometrically the situation is very simple. The starred assertion claims that a single point (say) D is in all three lines. Since C is in L1 and L2, which are known to be distinct, we can use the contrapositive of axiom P-I2 to show that C = D. Then since D is in line L3 "by construction", C must be also "by equality." This is an example of a *trivial* construction.

Intuitively, a trivial construction is one that could be accomplished by either searching and matching representations, or by creating an object and assigning it a representation, where the arguments to the construction function are all that are required to build the pattern. Thus, with our representation of lines, the functions INTERSECT and LINE would be trivial constructions. Similarly, if we represented triangles by listing the vertices, then it would be trivial to construct a triangle given its vertices. We (mathematically) close this definition by declaring that a construction accomplished by a *standard* sequence of trivial constructions is also a trivial construction. Since any construction can be accomplished by some sequence of trivial constructions, the restriction to a *standard* sequence above is required.

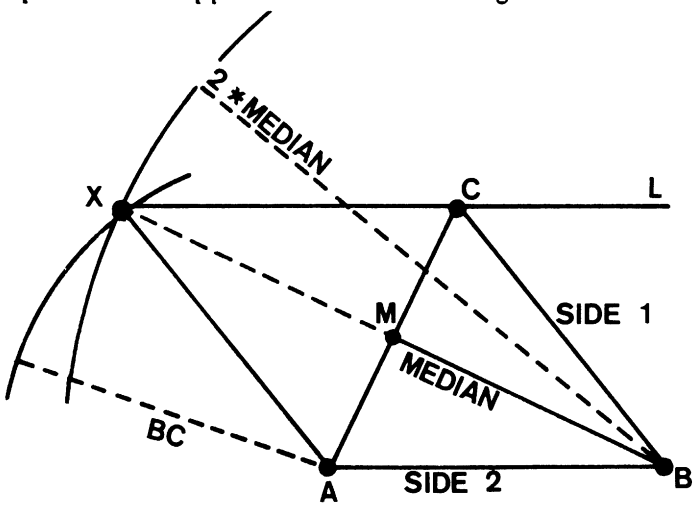
No prior geometry expert problem solving system was able to solve trivial construction problems; our technique represents an advance in this area. The reason others could not solve trivial construction problems is their insistence that there must always be a one-to-one correspondence between object names and their representations, i.e., using (and using the fact that one is using) canonical representations prevents one from being able to solve trivial construction problems. Consider: a corollary of a representation being canonical is that all objects must be distinct. Furthermore, the use of anonymous objects will not solve the problem, since at the time of creation we not only don't know the name of the object (already existing), but we don't know that we don't know its name. In the problem above, one must *first* create object D, and *only then* prove it is equal to an object already known.

We can solve trivial construction problems because we relax the constraints on representations: we insist only that representations fulfill their representation claims.

MISSING POINT CONSTRUCTIONS

Given two sides of a triangle and the median to the third, construct a triangle. As usual, only a compass and ruler (straight edge) may be used.

The reader may recall that the median of a triangle extends from the vertex to the midpoint of the opposite side (of the triangle).



MISSING POINT CONSTRUCTION DIAGRAM

This very hard problem implicitly names four points: A, B, C (the triangle vertices), and the midpoint M. Four lines are also implicitly named (three sides and the median). Unfortunately, we need a fifth point, not given by the statement of the problem. The reader might try solving this problem before reading further.

Wong [W7] has a heuristic that will generate this missing point which he calls the *midpoint reflection heuristic.* Assume that we have the diagram above. Reflecting B through M (the midpoint of AC) gives us a point X. X is at a distance 2*MEDIAN from B. XCBA is a parallelogram, so AX is congruent to BC (side 1) The solution, then, is to lay out line segment AB (given). Then construct point X at distance BC (given) from point A and 2*MEDIAN from B. Through X construct a line L parallel to AB Finally construct point C on L so that XC is congruent to AB. Triangle ABC is then the solution.

The key point is to construct the poin: X. The problem statement does not give even a hint that X exists or is needed. This is an example of a *missing point* construction; they are always hard.

A third type of construction, also used in the problem above, is the *locus* construction. Funt [F5] has written a program which can solve geometry problems using Polya's "pattern of two loci" [P1] provided that the names of all points are implicit in the statement of the problem, and that these points are distinct. Surprisingly, although Funt's program can solve rather involved locus constructions, it cannot perform trivial constructions! In fact, none of the expert geometry problem solvers currently available are able to solve the problem "trivial problem" given above.

EXPLANATION OF THE TRIVIAL CONSTRUCTION PROBLEM

Let's see how our formalism does on the "trivial construction problem". We have two kinds of plans for INTERSECT applied to lines: cne checks to see if we already have a point known to be in both lines (see chapter [ANALOGY EXAMPLES, TRIVIAL ANALOGY PROBLEM]), while the other creates a new object and asserts that it is in both lines.

```
(TO-FIND MAKE-INTERSECT (INTERSECT X Y)
        (BIND P)
:MI-1   (CONDITION (DISTINCT X Y))
:MI-2   (CONDITION (LN X))
:MI-3   (CONDITION (LN Y))
:MI-4   (MAKE P PT)
:MI-5   (ASSERT (IN-LN X P))
:MI-6   (ASSERT (IN-LN Y P))
        (RETURN P))
```

The plan does not make sure that the two lines are not parallel because the concept of parallel is missing in incidence geometry. This plan cannot be justified using the axioms in [GEOMETRY WORLD,AXIOMS]; again we see that theories need not be either consistent or complete. Suppose this plan produces an object D. Then after processing the assertions in the "given" portion of the trivial construction problem, we will have

OBJECT NAME	INDICATOR	REPRESENTATION
L1	LN	(C D)
L2	LN	(C D)
L3	LN	(D)
D1	DBUCKET	(L1 L2)
C	POINT	--
D	POINT	--

Now we will invoke the IN-LNP plan to evaluate

(IN-LN L3 C)

This will cause us to perform a pattern match

(* C *) against (D)

and as it stands this match fails, causing the program to hang. Later we will re-examine the hung process, and reconsider the pattern match using the implicit EQUAL test described in chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS]. This generates two goals

(EQUAL C D) (DISTINCT C D)

Hopefully we will be able to prove one of these.

We have a plan

```
(TO-DETERMINE PT-EQUAL1 (EQUAL X Y)
        (CONDITION (PT X))
        (CONDITION (PT Y))
        (BIND U V)
        (PREDICATE U (IN-LN U X))
        (PREDICATE V (IN-LN V X))
        (RESTRICT (U) (IN-LN U Y))
        (RESTRICT (V) (IN-LN V Y))
        (RESTRICT (U V) (DISTINCT U V))
        (ASSERT (EQUAL X Y))
        (RETURN TRUE))
```

which asks us to find two distinct lines that contain the two points. If we can do this, we assert that the two points are equal, and return true. This plan finds the lines L1 and L2, hence it returns TRUE, so the pattern match in IN-LNP succeeds, and in time returns TRUE. We will show how a proof of (IN-LN L3 C) may be obtained from this true result in chapter [ANALOGY ALGORITHMS,RESULT JUSTIFICATION].

In the situation above, we will never be able to put another point into L1's or L2's representation, because we will never be able to prove a BTWN assertion about the two points (and another point) already there. Alternatively, if we had already known of two points in L1 and L2, we would not have been able to put D in. This is caused by the "oversight" mentioned in plan IL2.

ANALOGY ALGORITHMS

Now that we know the structure of plans and domain descriptions, we can present three important algorithms used in the analogy process. I will try to present these algorithms with sufficient detail to convince the reader that I could write a program that performs as claimed, but yet not so much detail as to be tedious. Should I fall too far on one side or the other, I beg the reader's patience.

MAP FORMATION AND EXTENSION

We have described the process by which semantic templates are generated elsewhere (chapter [OVERVIEW OF ANALOGY,FIRST VIGNETTE -- TIC-TAC-TOE]). These templates form the input to the map formation algorithm. However, before this algorithm can be used we must decide which aspects of the domain problem should be mapped. We will see in chapter [ANALOGY EXAMPLES, NON-TRIVIAL ANALOGY] that the preliminary *summarization* step is absolutely necessary.

SUMMARIZATION

Since we are using a direct deduction system (as opposed to, say, resolution theorem proving [C1]) we can always add more assertions if a proof does not develop initially. We also have access to the current deduction AND/OR tree, so that we can see if any interesting outstanding questions need answers. These two considerations tell us that postponing the transfer of assertions to the image world won't cost us anything, and may be beneficial.

Suppose that in the domain world we are stuck (i.e., the current highest priority domain subgoal cannot be proven or disproven, and no other activity can take place) on the assertion (or function)

$$(P\ OB_1 \ldots OB_j).$$

At a minimum, then, we will need to map this to the image world. Predicate P and the objects OB_i are then needed in the summary. The summary will take the form of a number of "given" assertions, with the predicate we are stuck on as the "to prove" portion. We now need to specify what will be generated as the "given."

> The 0-order summary consists of the objects OB_i and the predicate P.
>
> The (n+1)-order summary includes all objects, predicates, and assertions in the n-order summary, plus
>
> (1) All the representations of the n-order summary objects, encoded by the relevant representation claims.
>
> (2) The subsets of any equivalence and distinctness buckets containing n-order summary objects, encoded in type-checking predicates.
>
> (3) Any assertion in the data base whose form does not use more than one object or predicate not in the n-order summary.
>
> (4) The entire equivalence and distinctness buckets containing objects in the (n-1)-order summary.

It is worth noting that the summarization algorithm may fail to include some assertions in the summary. These "widow" assertions may be included in the summary if called for by outstanding questions in the image problem solving effort. Since this algorithm is somewhat conservative, and dependent on having domain world representation claims, an implementation of our theory of analogy would probably require some heuristics concerning when this algorithm should be used.

MAP EXTENSIONS

The basic idea behind map extensions is that if domain type T maps to image type t and if P is some domain predicate with semantic template (P T), then P's image Q must have semantic template (Q t). For functions, there is an extra constraint that if the template (F T) has result of type R, then F's image G must have template (G t) with result of type r. In what follows, we will only discuss predicates, since the only difference between predicates and functions is this extra constraint.

The idea above can be used in several ways: if we know the mapping of a predicate, we can fill in the mappings of its argument type. Similarly, we can figure out where a predicate can go if we know the mappings of its arguments (in the semantic template, of course). It is important to keep in mind that we map *types* and *semantic templates*, not (except consequentially) objects and predicates.

The situation above is an *exact* map. We are willing to do a certain amount of violence to semantic templates, as outlined in the following rules, given in order of decreasing preference:

1. P->Q, (P S T)->(Q s t). This is the exact map described above.

2. P->Q, (P S T ... T)->(Q s t) where there is a representation claim for Q which would treat (Q s t ... t) as merely an extended form.

3. P->Q, (P S T)->(Q t s). This is the reordering map. We are willing to try mappings which preserve type inventory, but not type order.

4. P->Q, (P T T ... T)->(Q t t). We arbitrarily drop some of P's arguments, with the later ones preferred. This is the normal homogeneous argument case.

5. P->Q, (P S T T ... T)->(Q s t t). Again, we drop excess arguments of type T. This is the mixed homogeneous argument case.

6. P->Q, S->s, T->s, U->s, (P S T U)->(Q s s) where the argument U is arbitrarily dropped. Alternatively, S or T may be dropped. This is a weak homogeneous map.

The map formation algorithm also uses the following, more global rules.

A. Avoid many-to-one maps of object types by not mapping types. This means that exact maps may drop some arguments, because an argument's type is not being mapped.

B. Avoid many-to-one maps of predicates and functions by using the map which uses the fewest predicate multiple maps.

C. Map extended forms (see chapter [GEOMETRY WORLD,REPRESENTATION CLAIMS]) intact.

D. Map as high as possible into type hierarchies.

E. Map into the most restrictive sub-type first (see example in chapter [OVERVIEW OF ANALOGY,FIRST VIGNETTE -- TIC-TAC-TOE]).

As yet, we have not developed any heuristics for *adding* arguments. In all cases, what cannot be mapped is ignored.

RESULT JUSTIFICATION ALGORITHM

We need a way to transform a result and the justification of the plans used to generate that result into a justification of the result. The distinction between proof of program correctness (i.e., a proof that the program only produces correct results) and proof of the result generated by that program (i.e., a proof that the result is correct) is a subtle, but very important one. One proves that in general the result is correct, while the other proves that a particular result is correct.

In order to derive a justification of a result we employ two different techniques. The first is based on interpreting plans while running code, while the second is based on representation claims. The important features of our result justification algorithm are:

1. We can postpone justifying a result until *after* it has been generated.

2. We can produce a justification in a controlled manner, so that we will not look deeper into a justification than necessary.

PLAN BASED RESULT JUSTIFICATION

Commentary attached to the code of an expert problem solver signals when plan steps are completed. At that time we generate an instantiation of the plan step, using the explicit variable correspondence given in the commentary. However, since code can have awful things like loops and GOTOs, a plan step can be completed any number of times. We will only be interested in the most recent completion of a plan step.

When a RESULT plan step is completed, we record the most recent step instantiations, and associate them with the result. When the final result is obtained (i.e., the top level goal is achieved), it is a simple matter to trace through the plan steps and, using the justifications of those steps, obtain a justification of that result.

We will see that we also must record the plan step instantiation whenever an ASSERT plan step is completed, attaching the instantiation to the assertion. Similarly, when a PATTERN step inserts an object into a representation, the plan instantiation is attached to the inserted object, as well as the current "time" in order to unwind any subsequent representation-based deductions.

The completed plan instantiation will then be a list of CONDITION, PREDICATE, PATTERN, and RESTRICT plan steps, and of course the call pattern and any BIND plan steps which use a function call. Each instantiation will record the variable bindings currently in effect, and reference any subsidiary plan instantiations.

For example, if a PATTERN plan step's code made use of a hidden EQUAL test, then the instantiated PATTERN plan step would reference the plan instantiation associated with the result returned by the EQUAL test.

Returning to the trivial construction problem's solution, while processing the "given", we evaluated the form

```
(INTERSECT L2 L3)
```

by using the plan MAKE-INTERSECT. The result of this evaluation was a new object D (a point on

both lines). The plan instantiation for MAKE-INTERSECT making object D would look like

```
:MI#20
(MAKE-INTERSECT
        (MI-1 (X L2) (Y L3) LN-DISTINCT?#20)
        (MI-2 (X L2) LN?#22)
        (MI-3 (Y L3) LN?#23))
```

The important part of this plan instantiation is the variable correspondences for each step, and the plan instantiations for the subsidiary deductions (e.g. LN-DISTINCT?#20). Suppose we require a proof that

(IN-LN L3 D)

Whether this result is in the data base, or has been derived from a representation, we will know that plan instantiation MI#20 is responsible. We look at the instantiation to find the name of the plan. The plan justification (which, you will recall, isn't even true!) is

```
(PLAN-JUSTIFICATION MAKE-INTERSECT
        (S1 (DISTINCT X Y) %CONDITION MI-1)
        (S2 (LN X) %CONDITION MI-2)
        (S3 (LN Y) %CONDITION MI-3)
        (S4 (EQUAL P (INTERSECT X Y)) P-DEF1 S1 S2 S3)
        (S5 (IN-LN X P) P-DEF1 S4)
        (S6 (IN-LN Y P) P-DEF1 S4))
```

Once we have all of this, it is a fairly simple matter to use the plan justifications to generate a justification of the result. Some of the steps will be justified by referring to some theorem or axiom of the world description, others by reference to some plan, and finally some will be justified by representation claims. The first two present no difficulty -- we just do more of the same, being ever so careful to keep variable names straight. Unfortunately representation claims form a very effective road block.

Continuing the example, we may now derive a proof by first instantiating the justification, and then tracking back from the goal assertion to get

```
(S1 (DISTINCT L2 L3) LN-DISTINCT?#20)
(S2 (LN L2) LN?#22)
(S3 (LN L3) LN?#23)
(S4 (EQUAL D (INTERSECT L2 L3)) P-DEF1 S1 S2 S3)
(S6 (IN-LN L3 D) P-DEF1 S4)
```

REPRESENTATION CLAIM BASED RESULT JUSTIFICATION

Suppose the plane geometry expert outlined in chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS] were given the following assertions, where A, B, C, D, E, F, and G are known to be distinct points in line L, and initially L's representation is (A C).

ASSERTION	CODE	RESULT
1. (BTWN A C E)	IL3-3	rep=(A C E)
2. (BTWN B C D)		
3. (BTWN A B C)	IL4-2	rep=(A B C E)
4. (BTWN A G B)	IL4-2	rep=(A G B C E)
5. (BTWN B D E)	INTERP-BTW1(2)	assert (BTWN C D E)
6. (BTWN C D E)	IL4-2	rep=(A G B C D E)
7. (BTWN F A G)	IL3-2	rep=(F A G B C D E)

Now suppose we use BTWNP to deduce (BTWN F B E). Using the algorithm above, the "proof" we get of the fact is very short: it is true because representation claim RC2 says it is.

To drive the justification through deductions based on representations we will make use of:

1. The time each object in a representation was placed there.
2. The plan line that inserted the object into the representation.
3. The plan instantiation generated when the insertion was performed.

Step 1. Find the most recently inserted object of those referenced by the deduction. In the example this object is F.

Step 2. Find the plan line that inserted that object. In the example, that is line IL3-2.

Step 3. Reconstruct the correspondence between representation claim variables and

plan variables. This can always be done since we have the values of the plan variables in the instantiation attached to the inserted object. In the example, the correspondence is (U A) (V D) (W D).

Step 4. In the representation claim proof, find the clause for the plan line from step 2 and the correspondence from Step 3. From this clause read off the justification. This gives

```
(L1 (BTWN G B E) RC2 IL3-2)
(L2 (BTWN A G B) RC2 IL3-2)
(L3 (BTWN F A G) ASSERTION7)
(L4 (BTWN F G B) P-BTWN-THEOREM1 L2 L3)
(L5 (BTWN F B E) P-BTWN-THEOREM1 L1 L4)
```

Since the representation claim proof is inductive, the justification generation algorithm will be recursive. Continuing, in line L2 G was most recently inserted, by IL4-2 with correspondence (U C) (V A) (W D). Thus we find that L2 should expand to

```
(L2-1 (BTWN A G B) ASSERTION4)
```

As for line L1, again G was most recent. However, we get a different variable correspondence (U A) (V D) (W E) which directs us to a different clause in the proof of RC2. Here we find that P-BTWN-THEOREM2 justifies line L1 provided that we can prove (BTWN A B E). B was inserted by IL4-2 in response to assertion 3. Again, P-BTWN-THEOREM2 justifies our conclusion, provided (BTWN A C E) can be proven. Finally, C was inserted in response to assertion 1.

The proof of (BTWN F B E) references assertions 1, 3, 4, and 7, but *not* assertions 2, 5, and 6. This is as it should be, since those three assertions had nothing to do with our result. On the other hand, the justification of (BTWN A D E) would include references to assertions 1, 2, 3, 5, and 6, but *not* assertions 4 and 7.

DEBUGGING ALGORITHM

Using the result justification algorithm we can reduce a justification in a controlled fashion to any desired detail. The result and its justification will be in terms of the world description (and possibly plan and representation claim references). Using the map extension algorithm and by paying careful attention to where objects and assertions originally came from, we can lift the result and (to some extent) its justification back to the domain world.

If the lifted assertion is true in the domain world (i.e., the lifted assertion is in the data base, or is provable by the domain expert), then its justification need not concern us. Similarly, if the assertion is demonstrably false we are in trouble, and either must go back to the image world for a different solution, or (as is more likely) select a different analogy map.

Hence we are only interested in assertions that *might* be true. To determine if an assertion is in fact true, we need to check its justification.

1. If the inverse map of the justification is known, then expand the justification step in the *domain* world.

2. If the justification is part of the world description, see if the lifted version of the world description is valid in the domain world.

3. If the justification is either a representation claim or a plan, make a record that the claim or plan might be a good thing to lift, and then expand that justification step in the *image* world.

4. If the justification could not be lifted, try to continue, noting the possible presence of a MISSING-POSTCONDITION bug.

The above reduces the problem of justification checking to checking the validity of some part of a world description. Since world descriptions are theorems in the first order predicate calculus (with a few exceptions made for definitions), it is tempting to suggest that we simply drag out our favorite resolution theorem prover and let 'er rip.

Certainly that will prove the theorem true if it is. Since our world description may be

inconsistent and since we do not insist that our plans be consistent with the world description, we might very well succeed in proving the theorem is true even if it isn't! Besides, if the theorem is not provable, we want to know why. Bugs in theorems correspond to bugs in plans.

ONE-STEP DEDUCTIONS

We employ a very limited form of theorem proving: the one step deduction. We will attempt to prove a theorem as follows:

> 1. For all domain world axioms, lemmas, and theorems whose consequent has a non-null intersection with the consequent of the theorem we are trying to prove, see if its antecedent clauses are included in the antecedent of the theorem we are proving.
>
> 2. Similarly, try the contrapositives of all lemmas, axioms, and theorems.
>
> 3. For each axiom, lemma, and theorem in the world description, generate (individually, of course) its disjunctive normal form. Compare (using the unification algorithm [C1]) this with the theorem we are trying to prove, also in disjunctive normal form.

The reason for the first two attempts is primarily aesthetic, although some increase in efficiency should also result. Steps 1 and 2 appear to work if they prove only *some* of the consequences. Indeed, a particular application of a theorem may require only some of the consequents. We know what consequent we want (the result we are trying to justify), so these steps are correct as stated.

If none of these work, we can try expanding the justification in the image world, and lifting that. However, more often failure indicates a bug in the analogy map.

BUG DETECTION

We are primarily interested in three kinds of bugs: MISSING-PREREQUISITE, UNNECESSARY-PREREQUISITE, and MISSING-POSTCONDITION. The reader might imagine that there is a fourth, which might be called UNNECESSARY-POSTCONDITION. I have never seen an example of this, although it might occur. Both MISSING-PREREQUISITE and MISSING-POSTCONDITION bugs manifest themselves in the justification checker. Suppose we have the following situation

```
have:   (IMPLIES (AND A1 A2) (AND C1 C2))
want:   (IMPLIES (AND A2 A3) (AND C1 C3))
result: C1
```

Then step 1 will fail to confirm that what we have corresponds to what we want. The nature of the mismatch is that A1 is an excess antecedent in what we have, and A3 is an excess antecedent in what we want. If A1 is valid in the domain world, it corresponds to a potential MISSING-PREREQUISITE, and A3 to a potential UNNECESSARY-PREREQUISITE.

For the MISSING-PREREQUISITE bug detected above, we

1. Note the bug that A1 is a MISSING-PREREQUISITE.
2. Note the correspondence with a warning about the potential UNNECESSARY-PREREQUISITE A3.
3. Assert C2 into the domain data base.
4. Justify C1 with the domain fact we have.

When we lift the plan which gave rise to this bug, we will generate either a CONDITION or RESTRICT clause for A1 as a patch.

MISSING-PREREQUISITE bugs can be detected in another way. Any plan that adds objects to a representation must be documented in all relevant representation claim proofs. Inability to provide this new proof clause may indicate that either the representation claim must be

dropped or weakened, or that a MISSING-PREREQUISITE bug has been detected.

A MISSING-PREREQUISITE bug is fairly simple to detect, since it is caused by a (potential) domain expert failing to do something it *must* do. On the other hand, MISSING-POSTCONDITION and UNNECESSARY-PREREQUISITE bugs are caused by (potential) domain experts failing to do something they *should* have done, and trying to do something they *shouldn't* have. We should not be surprised that the conditions for confirming these two bugs appear byzantine.

A MISSING-POSTCONDITION possibility is signaled in step 4 of the justification checker. If the justification succeeds, and if the call pattern of a plan that was used to supply justification to fill in the gap is the same as the (lifted) call pattern of the plan that generated the MISSING-POSTCONDITION, then it is possible that the domain plan should have asserted the MISSING-POSTCONDITION. Now we see if we have a one step proof of the MISSING-POSTCONDITION based on domain plan justification and the conclusion the MISSING-POSTCONDITION was supposed to have justified. If we can do all this, then generate a new plan step (generally an ASSERT plan step) for the domain plan.

UNNECESSARY-PREREQUISITES are problematic in that usually their presence is not signaled by anything in the final result justification. True, we do generate a warning above, but this warning is only used if the antecedent is false in the domain world. Generally we must resort to a different technique.

We examine the AND/OR tree in the image expert. We find an OR node, all of whose subgoals have hung. If the disjunction of the subgoals is *false* or *meaningless* in the domain world, then that node may represent an unnecessary prerequisite. To test this, we must re-solve the original problem, generating a TRUE out of one of the subgoal nodes. If the new result justification is valid (when lifted) and made use of the node result, but not of the TRUE we generated in the subnode, then the UNNECESSARY-PREREQUISITE bug is confirmed. When we lift the plan corresponding to the OR node, we ignore the plan step that generated the subgoal.

The reason for solving the problem again is to insure the relevance of the plan hung up by the unnecessary prerequisite. Clearly we don't want to lift irrelevant plans. Similarly, if the plan is used and produces an invalid justification, then we were in error to generate the TRUE. We will see an example of this in chapter [ANALOGY EXAMPLES, NON-TRIVIAL ANALOGY].

ANALOGY EXAMPLES

The importance of special representations (for example, listing points in order for the representation of a line) cannot be overstated. Waldinger and Levitt [W1] point out "To as great an extent as possible, we have chosen representations that model the semantics of the concepts we use so as to make our deductions shorter and easier." A major thrust of this research has been to arrange for analogy to be able to lift these special representations to new worlds. The first three problems in this chapter illustrate lifting the representation of lines in two ways.

PROBLEM 1 -- a trivial analogy

While plane and solid geometry are not isomorphic, there is a sub-world of solid geometry that *is* isomorphic to plane geometry. The purpose of our first problem is to show how analogy functions in this simple case. Just to keep things from being devoid of complications, we will have the analogy process teach the solid geometry expert a new concept: lines can intersect.

PROBLEM 1. *Show that the intersection of distinct lines containing a point C is that point C.*

```
(FORALL (K L C) (IMPLIES (AND (LN L K)
                              (PT C)
                              (IN-LN K C)
                              (IN-LN L C))
                         (EQUAL C (INTERSECT K L))))
```

Several observations should be made. First, note that the LN predicate has two arguments. Input to the problem solver is processed by special procedures. Since we know that LN is a type checking predicate from the semantic template generation process, we expand this form into a *distinctness* bucket containing the two lines as objects as described in chapter [GEOMETRY WORLD,REPRESENTATION CLAIMS]. The second observation is that INTERSECT has never been defined in solid geometry.

The bare-bones solid geometry expert knows how to deal with problems of the above form: having created objects named K, L, and C, it then asserts (LN L) ... (IN-LN L C) into the data base. Finally, it tries to evaluate

(EQUAL C (INTERSECT K L))

by first evaluating the function call

(INTERSECT K L).

Since there is no procedure willing to perform this computation, the solid geometry expert reports a failure. At this time the solid geometry data base contains two assertions:

(IN-LN K C)
(IN-LN L C)

The most we can reasonably expect from analogy is:
1. Learn what INTERSECT means.
2. Learn how to implement a function that computes INTERSECT.
3. Learn how to react to assertions of the form
(IN-LN line point).
4. Learn how to represent lines.

Although this is the most we can expect, analogy gives us a little bit more!

STEP 1: MAP

The current outstanding problem is (EQUAL C (INTERSECT K L)) so the 0-order summary has EQUAL, INTERSECT, C, K, and L. The 1-order summary includes the two IN-LN assertions. Thus we will map everything. The map formation algorithm suggests two possibilities:

LN->LN	LN->PT
PT->PT	PT->LN
INTERSECT->INTERSECT	INTERSECT->LINE
IN-LN->IN-LN	IN-LN->IN-LN order of arguments reversed

We prefer exact maps, so we use the one on the left. The map on the right is interesting in that analogy just tried to invent projective geometry.

STEP 2: SOLVE THE IMAGE PROBLEM

Plane geometry expertise is fully developed. When each assertion is entered in the data base, it is examined by a procedure (invoked by the pattern of the assertion). This has occurred, and the data base is now empty. The assertions have been used to set up objects and their representations. We have

```
K       type=LN, representation=(C)
L       type=LN, representation=(C)
C       type=PT, no representation
OB1     type=OBUCKET, representation=(K L)
```

where the last object is a "distinctness bucket."

We now evaluate (INTERSECT K L) (using a procedure FIND-INTERSECT whose plan is given below), and get a return value C. We apply the inverse analogy map to the image object C, and get the domain object C. We evaluate (EQUAL C C), getting TRUE as a result. This indicates it would be worth while "lifting" the definition of INTERSECT and the proof that "C" is the correct value.

STEP 3, PART 1: OBTAIN IMAGE JUSTIFICATION

We used the plan FIND-INTERSECT in the solution.

```
(TO-FIND FIND-INTERSECT (INTERSECT L1 L2)
        (BIND P)
:FI1    (CONDITION (LN L1))
:FI2    (CONDITION (LN L2))
:FI3    (CONDITION (DISTINCT L1 L2))
:FI4    (PATTERN L1 LN (* ?P *))
:FI5    (PATTERN L2 LN (* P *))
        (RETURN P))
```

```
(PLAN-JUSTIFICATION FIND-INTERSECT
        (L1 (LN L1) %CONDITION FI1)
        (L2 (LN L2) %CONDITION FI2)
        (L3 (DISTINCT L1 L2) %CONDITION FI3)
        (L4 (IN-LN L1 P) RC1 FI4)
        (L5 (IN-LN L2 P) RC1 FI5)
        (L6 (IN-LN L1 (INTERSECT L1 L2)) P-DEF1 L1 L2 L3)
        (L7 (IN-LN L2 (INTERSECT L1 L2)) P-DEF1 L1 L2 L3)
        (L8 (EQUAL P (INTERSECT L1 L2)) P-THM22 L1 L2 L3 L4 L5 L6 L7))
```

where P-THM22 is the contrapositive of axiom P-I2. We get a proof of (EQUAL C (INTERSECT L1 L2)) from the plan justificaion of FIND-INTERSECT by the result justification algorithm in chapter [ANALOGY ALGORITHMS,RESULT JUSTIFICATION]. This proof uses RC1, P-DEF1 (the definition of INTERSECT), and P-THM22.

STEP 3, PART 2: LIFT IMAGE JUSTIFICATION

We lift P-DEF1, getting S-DEF1. We then apply the inverse analogy map to P-THM22 to get (in solid geometry)

```
:S-THM22
(FORALL (A B X Y)
        (IMPLIES (AND (LN X) (LN Y) (DISTINCT X Y) (PT A) (PT B)
                      (IN-LN X A) (IN-LN X B) (IN-LN Y A) (IN-LN Y B))
                 (EQUAL A B)))
```

We use the second kind of one step justification (chapter [ANALOGY ALGORITHMS,RESULT JUSTIFICATION]) to get a proof of S-THM22. Knowing S-THM22 to be true, we can complete the lifted proof and add the following to the analogy map for use in the next step.

```
S-THM22 -> P-THM22
S-I2 -> P-I2
```

STEP 3, PART 3: LIFTING PLANS

We noted that FIND-INTERSECT, IL1, and RC1 were used in the course of solving this problem. Analogy now rewrites these into solid geometry plans. No bugs were detected in the result justification, so this is a straightforward copy operation. We add the procedural correspondences

```
S-FIND-INTERSECT -> FIND-INTERSECT
S-IL1 -> IL1
S-RC1 -> RC1
```

to the analogy map.

LESSONS FROM PROBLEM 1

What have we learned as a result of solving this problem? As much as could be expected. We have both a definition and code for intersections of lines (under some circumstances), We have the beginnings of code dealing with representations of lines. We also know what to do with IN-LN assertions of *multiple* point arguments by virtue of the solid geometry representation claim S-RC1.

PROBLEM 2 -- Non-trivial analogy problem

We now turn to our second problem. Let us assume that, by processes similar to that given above, all the necessary expertise in dealing with points and lines has been learned by the solid geometry expert.

We now want to increase our solid geometry expertise to include the representation of planes. We will start by posing a problem which asks solid geometry to prove that a plane is

determined by two intersecting lines. Granted, this is an obvious solid geometry fact. However, the rigorous proof of this fact is far from obvious. The skeptical reader might wish to try his hand. We wish to prove that

PROBLEM 2. *Three distinct points which determine two distinct lines also determine a plane.*

```
(FORALL (A B C P)
        (IMPLIES (AND (PT A B C)
                      (NOT (EQUAL (LINE A B) (LINE B C)))
                      (PL P)
                      (IN-PL P A B C))
                 (EQUAL (PLANE A B C) P)))
```

We used an extended format for PT (which the solid geometry expert knows how to deal with) and for IN-PL (which it doesn't). There are trivial construction of both lines (understood) and planes (not understood). Finally, the problem mentions points, lines, and planes, so the analogy map must be non-trivial.

Processing is forced to halt when we try to assert

```
(IN-PL P A B C)
```

Recall that IN-PL is expected to have two arguments, the first of type PL and the second of type PT. This form does not match its semantic template.

STEP 1: MAP

We must eventually use analogy to find what this assertion means. Having no good reason to abandon the analogy map used in problem 1, we continue with

```
PT -> PT
LN -> LN
etc.
```

To find a mapping of IN-PL, we ask "What predicate do we have in plane geometry that takes two arguments, one of them of type PT and the other one of another type?" One answer is IN-LN. But this has an object of type LN as its other argument, so on the basis of matching semantic templates

(rule 1 of the map extension algorithm in chapter [ANALOGY ALGORITHMS,MAP FORMATION AND EXTENSION]) we conclude

```
PL -> LN
IN-PL -> IN-LN
```

and add these to form a many-to-one analogy map.

Construction of Analogous Problems Involves SUMMARIZING

We will eventually be forced to solve a problem in plane geometry to know how to treat the expression (IN-PL P A B C) in solid geometry. If we simply map *everything* we know down to plane geometry, we will end up with a contradiction: (IN-LN P A B C) and the claim (DISTINCT (LINE A B) (LINE B C)) conflict with axiom P-I2. This means we must summarize the current situation into a sub-problem. The next question is "What sub-problem?" We have domain objects:

```
A       type=PT
B       type=PT
C       type=PT
OB1     type=DBUCKET, representation=(A B C)
OB2     type=LN, representation=(A B)
OB3     type=LN, representation=(B C)
OB4     type=DBUCKET, representation=(OB2 OB3)
```

We are hung up by the IN-PL assertion and therefore by our goal (EQUAL (PLANE A B C) P). We will first generate the 1-order summary and then use the map extension algorithm to map the summary to plane geometry. We start off with the objects P, A, B, and C, their representations, and type declarations resulting from relevant distinctness buckets. These assertions constitute the "given" portion of the sub-problem constructed by the summarization process in chapter [ANALOGY ALGORITHMS,MAP FORMATION AND EXTENSION]. The "to prove" portion is supplied by the assertion solid geometry could not deal with. Continuing, we assert in plane geometry

```
(PT A B C)
(LN P)
(IN-LN P A B C)
```

The last assertion would give P the representation (A B C) if the assertion (BTWN A B C) were also present. Since it isn't present, we expand the IN-LN assertion to

(IN-LN P A) (IN-LN P B) (IN-LN P C)

The first two are then removed from the data base by the TO-REPRESENT plan IL1 and the TO-INCLUDE plan IL2, while the object P is given the representation (A B). The third assertion causes an attempt to prove or disprove the three statements

(BTWN A B C) (BTWN A C B) (BTWN C A B)

by the TO-INCLUDE plans IL3 and IL4. (These plans may be found in chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS] in the section giving EXAMPLE PLANS). Naturally no progress is made on any of these, so we continue to summarize.

We need to map the current domain goal

(EQUAL (PLANE A B C) P)

to an appropriate image goal in plane geometry. Finding no exact match for the mapped semantic template, we note that the arguments to PLANE are of homogeneous type, and that the semantic template for LINE in the image is also of homogeneous argument type, with the appropriate (under the map) value type. By rule 4 of the map extension algorithm we add

PLANE -> LINE

to the analogy map, with a note to *arbitrarily* drop the last argument.

STEP 2: SOLVE IMAGE PROBLEM

Then, noting that (PLANE A B C) -> (LINE A B), it is easy to evaluate the latter expression in plane geometry to get the object P, and by golly

(EQUAL P P)

STEP 3, PART 1: OBTAIN IMAGE JUSTIFICATION

We used the following plane geometry plan to get this result:

```
(TO-FIND FIND-LINE (LINE P1 P2)
        (CONDITION (PT P1))
        (CONDITION (PT P2))
        (CONDITION (DISTINCT P1 P2))
        (BIND L)
        (PATTERN ?L LN (* P1 * P2 *))
        (PATTERN ?L LN (* P2 * P1 *))
        (RESULT L))
```

The proof rests on LINELEMMA1 (proven by using P-I2):

```
:LINELEMMA1
(FORALL (P1 P2 L1 L2)
        (IMPLIES (AND (PT P1)
                      (PT P2)
                      (DISTINCT P1 P2)
                      (LN L1)
                      (LN L2)
                      (IN-LN L1 P1 P2)
                      (IN-LN L2 P1 P2))
                 (EQUAL L1 L2)))
```

The image proof generated by the result justification algorithm applied to the plan justification for FIND-LINE above reads

1.	(PT A B)	given
2.	(LN P)	given
3.	(IN-LN P A B)	given
4.	(LN (LINE A B))	P-I1 applied to 1
5.	(IN-LN (LINE A B) A B)	P-I1 applied to 1
6.	(EQUAL P (LINE A B))	LINELEMMA1 applied to above

We have now solved the summarized problem. Unfortunately, when we try to "lift" the justification, we find it is not correct!

STEP 3, PART 2: LIFT IMAGE JUSTIFICATION -- DEBUGGING the Analogy

The first hint of trouble occurs when we try to justify step

4* (PL (PLANE A B C)).

To do this, we need to lift P-I1 as it was used in this step:

```
I1* (FORALL (A B C)
        (IMPLIES (AND (PT A) (PT B)) (PL (PLANE A B C))))
```

This theorem is not true, but no matter, because we try using brute force (our one step deduction algorithm), and fail to prove I1*. Good -- we have detected a bug! We do find, however, that we can prove (PL (PLANE A B C)) in one step using S-I4a. The "lifted" portion of plane geometry allows us to prove in solid geometry that (NON-LN A B C). (The appropriate portion of plane geometry would be lifted by this exercise in any case by a recursive application of the analogy process).

We can now classify the "bug" in the analogy to be a MISSING-PREREQUISITE. With this in mind, we add

S-I4a -> P-I1

to the analogy map, and a line providing non-collinearity to the proof in solid geometry.

Having lifted step 4 of the image result justification, we proceed to lift step 5 as

5* (IN-PL (PLANE A B C) A B) *S-14a*

after checking that S-I4a does indeed prove this. This step will also need modification. We now need to lift LINELEMMA1 (above) in preparation for lifting step 6 of the image justification. This lemma in turn depends on axiom P-I2.

Following the proof checking algorithm, we translate the lifted version of LINELEMMA1 to disjunctive normal form, then compare this to all axioms (also in their disjunctive normal form). We discover that axiom S-I5 gives us the desired equality provided that the points are not collinear and that all three points are in both planes. We just proved the former, and know the latter is true by S-I4a (appropriate assertions were made when it was applied).

We make note of a second MISSING-PREREQUISITE bug on our bug list, patch line 5* to show that point C is also in (PLANE A B C), and give the proof for

```
PLANELEMMA1:
(FORALL (P1 P2 P3 PL1 PL2)
        (IMPLIES (AND (PT P1 P2 P3)
                      (NON-LN P1 P2 P3)
                      (PL PL1) (PL PL2)
                      (IN-PL PL1 P1 P2 P3)
                      (IN-PL PL2 P1 P2 P3))
                 (EQUAL PL1 PL2)))
```

Having done all that, we update the analogy map by adding

PLANELEMMA1 -> LINELEMMA1
S-I5 -> P-I2

to the analogy map. Note we cleverly got back the proper number of points: LINELEMMA1 quantified two points, while PLANELEMMA1 quantifies three!

STEP 3, PART 3: CHECKING FOR UNNECESSARY PREREQUISITES

This completes the proof. We are still not ready to lift plans. We have one anomaly remaining: in plane geometry there is an outstanding question concerning the order of points A, B, and C in the "line" P. We can easily prove in solid geometry that

```
(NOT (OR (BTWN A B C) (BTWN A C B) (BTWN C A B)))
```

We can conclude that there is not an "obvious" candidate in solid geometry for the BTWN relation with this analogy. Having thus applied the UNNECESSARY-PREREQUISITE bug detection algorithm, we also add this bug to our bug list.

STEP 3, PART 4: Lifting Plans for Problem 2

We can now lift the plans. Although the bugs were detected by logical means, they are noted with respect to the plan lines which gave rise to them. When we lift these questionable lines, the bug type tells us what actions need to be taken to repair the plan. Thus we get

```
(TO-FIND FIND-PLANE (PLANE P1 P2 P3)
        (CONDITION (PT P1))
        (CONDITION (PT P2))
        (CONDITION (PT P3))      ;due to LINELEMMA1 bug
        (CONDITION (DISTINCT P1 P2 P3)) ;LINELEMMA1 bug
        (CONDITION (NON-LN P1 P2 P3))   ;S-I4 bug
        (BIND L)
        (PATTERN ?L PL (* P1 * P2 *))
        (PATTERN ?L PL (* P2 * P1 *))
        (RESTRICT (L) (IN-PL L P3))     ;LINELEMMA1 bug
        (RESULT L))

(TO-INCLUDE S-IL3 (IN-PL P A)
        (PATTERN P PL (^A *))
        (ABSORB (IN-PL P A)))
```

where S-IL3 came from plane geometry IL3, with several plan steps deleted due to the UNNECESSARY-PREREQUISITE bug. We also add

```
S-IL1 -> IL1
S-IL3 -> IL3
FIND-PLANE -> FIND-LINE
```

to the analogy map. We also lift a representation claim about the representation of planes:

```
:SP-RC1
(REPRESENTATION-CLAIM (X PL (* Y *) (IN-PL X Y)) ...)
```

We learned from this example a little about representing planes, and how to construct a plane from given points. We also learned that there is no concept corresponding to BTWN which applies to points in a plane. It is important to remember in all this that by "learn" we always mean "write code for."

PROBLEM 3 -- Non-obvious analogy

So far, the problems have been interesting, but not spectacular in that the analogies were fairly obvious. The problem we will now solve has no obvious solution.

The problem involves the notion of a line being in a plane. We recognize that "IN" is a transitive, non-symmetric binary relation in solid geometry: if A is IN B, and B is IN C, then A is IN C, but if A is in B, then B is not necessarily in A (almost certainly not). The crux of the problem is that there are *no* transitive, non-symmetric binary relations in plane geometry (as we have described it). That we call "IN" by different names (IN-LN, IN-PL) according to argument types just makes things worse. In this example we will find an analogy where none can reasonably exist.

The above example indicates we could replace both IN-LN and IN-PL in solid geometry with a single predicate IN without affecting the analogy process operation.

It is also worth pointing out that we don't need higher level descriptions like "transitive, non-symmetric binary relation." In fact, if high-level descriptions were to form the basis of an analogy process, then that process would evidently fail on this problem.

A SECOND-ORDER DEFINITION

Suppose we wish to introduce the notion of "a line being in a plane" to our budding solid geometry expert. We cannot say merely "if a point is in a line and that line is in a plane, then the point is in the plane" because that only tells us how to use the fact that a line is in a plane, not how to deduce it. We might wish to add "a line is in a plane if all points on that line are in the plane", which is correct, but testing for this condition involves a proof by contradiction. We dislike proofs by contradiction for reasons detailed in chapter [LOGICS OF EXPERTS,FORMALISMS AND LOGICS]. Therefore we might try to add "a line is in a plane if two points of that line are in the plane." This, of course, duplicates axiom S-16 in a definition, and thus cannot be allowed.

What we will do is similar to the device used with INTERSECT: we will claim that for line L and plane P, (IN-PL P L) is a predicate such that

```
(FORALL (A L P)
        (IMPLIES (AND (PT A) (LN L) (PL P)
                      (IN-LN L A) (IN-PL P L))
                 (IN-PL P A)))
```

In other words, the above is true "by definition." This is a "second-order" definition, because it really means

```
(FORALL (A L P ALPHA)
        (IMPLIES (IMPLIES (AND (PT A) (LN L) (PL P)
                               (IN-LN L A) (ALPHA P L))
                          (IN-PL P A))
                 (IMPLIES (ALPHA P L) (IN-PL P L))))
```

where ALPHA is a universally quantified *predicate*.

STEP 1: MAP

We have our definition. To find how the defined predicate (IN-PL plane line) is to be implemented, we use a clever trick: we pose the definition as a problem! We continue using the same analogy developed by problems 1 and 2, so that

```
PT -> PT
LN -> LN
PL -> LN
IN-PL -> IN-LN when applied to plane and point
```

The image semantic template of IN-PL applied to plane and line under the current analogy is a predicate applied to two lines, i.e., the same semantic template as EQUAL (applied to lines). With the axioms given, it is also the only semantic template match. In fact, there are four reasons why EQUAL is a good choice for the analogy:

(1) Pragmatic. This choice works.
(2) Tradition. In algebra one investigates the structure of groups and rings by way of maps that send problematic substructures to identity elements, i.e., one imposes equivalence relations on the structure.
(3) Conjectural. Suppose we wanted to choose P to maximize the size (cardinality) of the set "Q such that for all X,Y (P X Y) implies (Q X Y)." I suggest that EQUAL would be one of the best choices. In other words, EQUAL "does more" than any other predicate.
(4). Philosophical. We really want to write a program. A common joke is that writing programs is the same as debugging a blank sheet of paper. We are essentially using EQUAL as a blank stimulus in the hopes of debugging the response.

So we add as an exact analogy map

IN-PL -> EQUAL when applied to planes and lines.

With this map, the plane geometry problem we pose is

```
(FORALL (A L P)
        (IMPLIES (AND (PT A) (LN L) (LN P)
                      (IN-LN L A) (EQUAL L P))
                 (IN-LN P A)))
```

STEP 2: SOLVE THE IMAGE PROBLEM

To make a long story short, the image problem is solved with the proof being "by definition of equality." When this justification (a reference to a second-order equality axiom) is lifted, we get a proof "by definition of IN-PL" which is indeed correct!

STEP 3: LIFT THE PLANS

We now lift the plans. The image plan IN-LNP first searched P's representation for A, and failed. It then used the implicit EQUAL test to search for any "line" EQUAL to P, found L, and proceeded to search L for A.

```
(TO-DETERMINE IN-LNP (IN-LN X Y)
:IN-LNP-1       (PATTERN X LN (* Y *))
:IN-LNP-2       (RETURN TRUE))
```

Thus the "lifted" plan should first search P's representation for the object A, and failing that, search for any line (because L's inverse is a line) satisfying the relation (IN-PL P x) where x is a line, and then search there for A.

The image plan is IN-LNP. The implicit EQUAL test shows up right before the pattern step. We used a representation claim about equality buckets to do this test. To lift this claim (under the current analogy map) we search for a representation claim with a consequence (IN-PL X Y). We find and verify SP-RC1 (written in the course of solving problem 2). This gives us a lifted plan

```
(TO-DETERMINE IN-PLP2 (IN-PL X Y)
        (BIND Z)
        (PATTERN X PL (* ?Z *)) ;SP-RC1
        (RESTRICT (Z) (LN Z))   ;map restriction
        (PATTERN Z LN (* Y *))
        (RESULT TRUE))
```

So we learn three things by solving this problem: first that the use of the "definition" should be in backward chaining rather than forward chaining, and second the point and details of the patch required to implement the new interpretation of IN-PL, and third that lines can appear in the representation of planes. Now if

```
(IN-PL plane line)
```

is ever asserted, we know what to do.

Problem 3, Continued

Thus armed, suppose we are given the problem:

```
(FORALL (A B C P)
        (IMPLIES (AND (PT A B C)
                      (PL P)
                      (IN-PL P A B)
                      (IN-LN (LINE A B) C))
                 (IN-PL P C)))
```

The current analogy map transforms this to a plane geometry problem

```
  Given:  (PT A B C)
          (LN P)
          (IN-LN P A B)
          (IN-LN (LINE A B) C)
  Prove:  (IN-LN P C)
```

The justification for the TRUE result obtained reads

1. (IN-LN P A) given
2. (IN-LN P B) given
3. (EQUAL P (LINE A B)) FIND-LINE evaluating (LINE A B)
4. (IN-LN (LINE A B) C) given
5. (IN-LN P C) equality, 3, 4

When we lift this reasoning, we find that step 4 is no longer valid. We have

S1. (IN-PL P A) given
S2. (IN-PL P B) given
S3. (IN-PL P (LINE A B)) generated by evaluating (LINE A B)

The lifted step of the proof S3 is from image step 3 by the correspondence IN-PL -> EQUAL when IN-PL is applied to a plane and a line. The image proof step 3 was produced as a result of evaluating (LINE A B), so one suspects that lifted step S3 should be generated under similar circumstances.

We use our solid geometry expert S-MAKE-LINE to construct (LINE A B) noting the possibility of a MISSING-POSTCONDITION bug, and use the one step deduction algorithm to supply the rest of the proof

S4. (EQUAL L (LINE A B)) S-MAKE-LINE
S5. (IN-LN L A)
S6. (IN-LN L B)
S7. (IN-LN L C) given (image assertion 4)
S8. (IN-PL P C) S-I6 S5,S6,S1,S2,S7

Since S3 was generated by a plan with (after inverse mapping) a call pattern (LINE X Y), and the call pattern for S-MAKE-LINE is the same, the conditions for MISSING-POSTCONDITION are satisfied. The second-order definition of IN-PL (for this particular set of arguments) gives us the one step deduction that (IN-PL P L) is true. Thus we patch the S-MAKE-LINE, and of course fix up the plan justification.

```
(TO-FIND S-MAKE-LINE (LINE P1 P2)
        (CONDITION (PT P1))
        (CONDITION (PT P2))
        (CONDITION (DISTINCT P1 P2))
        (BIND L)
        (BIND P)
        (MAKE L LN)
        (PATTERN L LN (^P1 ^P2))
        (PATTERN *P PL (* P1 * P2 *))    ;PATCH
        (PATTERN *P PL (* P2 * P1 *))    ;PATCH
        (ASSERT (IN-PL P L))             ;PATCH
        (RESULT L))
```

Ah, what a crooked path we have walked. If no natural analogy is available, the analogy process blithely uses an unnatural one to achieve new expertise.

PROBLEM 4 -- A failure of analogy

We will briefly sketch one way that the analogy process can fail. We will try to reason about "Farmer and River" world on the basis of "Tower of Hanoi" world

> *Farmer and River world*: A farmer takes a fox, a chicken, and a bag of grain on a journey. They come to a river, and find a small boat. The farmer can only fit one of the three objects in the boat at one time. The fox cannot be left alone with the chicken, nor the chicken with the grain. What sequence of movements will result in the farmer and his three objects getting to the other side of the river?
> *Formally* Permitted two-object groups are (FOX, GRAIN). Sides of the river are SIDE1 and SIDE2. Initially FOX, GRAIN, and CHICKEN are ON SIDE1. We want them ON SIDE2.
>
> *Tower of Hanoi world*: this is the 3-ring version. There are three pegs (named PEG1, PEG2, and PEG3). PEG1 has three rings on it (BIG, MEDIUM, and SMALL). With the restriction that a ring may never be placed on a smaller ring, how can all three rings be moved to PEG2?
> *Formally* Permitted two-object stacks are (in order from top to bottom) (BIG, SMALL), (BIG, MEDIUM), (MEDIUM, SMALL).

The map formation algorithm maps the ON predicate to ON, objects of type SIDE to objects of type PEG, and objects of type COMPANION to objects of type RING (this is all obvious from the semantic template for ON).

From the initial configuration, it is clear that

SIDE1 -> PEG1

From the goal configuration, we can easily decide that

SIDE2 -> PEG2

We now have three possible maps:

MAP1	MAP2	MAP3
FOX -> BIG	FOX -> BIG	FOX -> MEDIUM
GRAIN -> SMALL	GRAIN -> MEDIUM	GRAIN -> SMALL
CHICKEN -> MEDIUM	CHICKEN -> SMALL	CHICKEN -> BIG

If we use either MAP1 or MAP3, then the (inverse map) situation after the first Tower of Hanoi expert move leaves the chicken with the fox. This is good for the fox, but not for the chicken. Thus we use MAP2. Now look at the solution to the Tower of Hanoi generated by our expert, and the inverse map of those movements, recalling that we prefer *ignoring object maps* to double maps.

SMALL->PEG2	CHICKEN->SIDE2
MEDIUM->PEG3	IGNORE
SMALL->PEG3	IGNORE
BIG->PEG2	FOX->SIDE2
SMALL->PEG1	CHICKEN->SIDE1
MEDIUM->PEG2	GRAIN->SIDE2
SMALL->PEG2	CHICKN->SIDE2

Sure enough, analogy gave us the solution! Then why do we claim this example is a failure of the analogy process? Simply this: after getting the solution we try lifting the plans which generated it. This effort is a disaster!

Since we insist that analogy actually increase expertise, we have no choice; we must consider this application a failure.

LOGICS OF EXPERTS

The purpose of this chapter is to develop a way to express the limitations of our theory of analogy. We have seen examples of the analogy process operating in four different pairs of worlds. We now ask "Is there something special about these worlds that might allow this analogy process to work here but not in other worlds?" To answer this question we will present a closure claim: given a description of a world this claim will tell whether or not that world can be either the domain or image world in the analogy process. That is, we will express the limitations of our analogy process in terms of a limitation on the kinds of worlds it can be used on.

LIMITATIONS OF ANALOGY

The analogy theory has two components: the horizontal, concerned with extending and applying the analogy map and its inverse, and the vertical, concerned with transforming and relating programs to proofs and proofs to programs. The horizontal component has been discussed in preceding chapters, so we will say nothing more about it here. However, the vertical component is fairly algorithmic, and fortuitously lends itself to a rather interesting kind of analysis. The results of this analysis will include

1. A limitation on our ability to go from program to proof.
2. A limitation on our ability to go from proof to program.
3. A hierarchy of world complexity.

The third result is the most important. On the way to this, we obtain results indicating that various current research efforts could, in theory, be successful, while other efforts would (in theory) be unsuccessful. We can also show that the world of geometry was a very good world in which to initially study analogy. The second limitation gives us an indication that using resolution

theorem proving to do expert problem solving is a very bad idea.

SUMMARY

We will develop a technique by which we can characterize the power of an expert problem solving system such as the geometry formalism introduced in chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS]. We will also present a characterization of the power required to *handle* a world. These two characterizations are in the same terms, so we can compare them to tell whether the formalism *in principle* can handle the world, and similarly whether a world needs *in principle* a less powerful formalism than the one proposed.

The point of all this is twofold. First, as outlined above, we can derive limitations to any analogy process using proof-theoretic techniques. Second, we can make a philosophical statement concerning the relative merits of problem solving theories (in a particular world). Given two equally strong theories (i.e., both "do" as much), we can state: *The better theory of problem solving uses the weaker formalism.* To believe otherwise is bad science. If some expert problem solver does not handle aspects of a world that could in principle be handled, then we can similarly criticize that effort for being too timid.

DEFINITION OF FORMALISM

Often we would like to say that some programming language is "more powerful" than another. This claim is usually not true due to annoying universal turing machine arguments. For example, we would like to claim that a theorem prover along the lines of (say) Ullman's [U1] geometry theorem prover is not as powerful (in an absolute sense) as a resolution theorem prover. By this we mean that from the same "given", the resolution theorem prover would obtain a proof while the other would not. But since both are written in a language powerful enough to produce a

universal turing machine, they can simulate each other, hence our claim must be false.

A *formalism* might be thought of as a way to enforce a gentleman's agreement not to make universal turing machines. A programming language, in addition to providing programming constructs, supplies a philosophy of techniques. If we can extract a promise to "play the game", we can make interesting (and valuable) claims about the limitations of the language which would be falsified if use were made of that language's universal capabilities.

Since we wish to restrict *what can be done* without (significantly) affecting *how* it is done, the proper place to begin is with *plans* and not *code*. By placing rather severe restrictions on the nature of plans and their justification, we can limit the power of a formalism to the point where important observations cannot be obfuscated by sophomoric universal turing machine arguments.

That is, by considering a formalism to encompass a programming language, a plan language, and a plan justification language, we will be able to show that, for instance, the geometry formalism introduced in chapter [GEOMETRY WORLD,LANGUAGE FOR PLANS] is strictly weaker than a resolution theorem prover, *not* because you couldn't write *code* for the resolution theorem prover, but because you couldn't write a *plan* for that code.

PROBLEM SOLVING FORMALISMS CORRESPOND TO LOGICS

Suppose we have a formalism like the one used earlier for geometry. It had a single data base into which only true assertions were entered. These assertions had only two possible forms:

$$(\text{predicate } obj_1\ obj_2\ \ldots\ obj_n)$$

or

$$(\text{NOT } (\text{predicate } obj_1\ obj_2\ \ldots\ obj_n))$$

We remarked that this formalism had some limitations in that it could not prove some theorems in the classical propositional calculus. The "reasoning by cases" example (using single letters to represent assertions of the form above)

REASONING BY CASES

```
GIVEN:
        (IMPLIES A (AND B C))
        (IMPLIES B (OR E F))
        (IMPLIES C (OR E G))
        (IFF F (NOT G))

PROVE:
        (IMPLIES A E)
```

cannot be proven, since we need at some time to assert F and at some other time assert (NOT F). The restriction to a single data base implies that this course of events will result in both F and (NOT F) being in the data base, hence the data base will be inconsistent. The alternative, asserting (NOT E) and deriving a contradiction, is a clear violation of the edict "keep the data base consistent."

Coupled with an observation that the formalism will apply modus ponens automatically, the above suggests that the formalism implements some kind of logic, but not the classical propositional calculus. One might now naturally ask "What other kind is there?" We can axiomatize the classical propositional calculus as follows (Kleene [K1], page 82):

CLASSICAL PREDICATE CALCULUS

```
Inference Rule:  A, (IMPLIES A B) |- B

1a. (IMPLIES A (IMPLIES B A))
1b. (IMPLIES (IMPLIES A B)
             (IMPLIES (IMPLIES A (IMPLIES B C)) (IMPLIES A C)))
2.  (IMPLIES A (IMPLIES B (AND A B))
3a. (IMPLIES (AND A B) A)
3b. (IMPLIES (AND A B) B)
4a. (IMPLIES A (OR A B))
4b. (IMPLIES B (OR A B))
5.  (IMPLIES (IMPLIES A C)
             (IMPLIES (IMPLIES B C) (IMPLIES (OR A B) C)))
6.  (IMPLIES (IMPLIES A B) (IMPLIES (IMPLIES A (NOT B)) (NOT A)))
7.  (IMPLIES (NOT (NOT A)) A)
```

According to Kleene, replacement of axiom 7 with

```
7* (IMPLIES (NOT A) (IMPLIES A B))
```

results in a new logic: intuitionistic propositional calculus.

Intuitionistic logic insists that proofs be constructive. One way in which to enforce this

edict is to disallow "proofs by contradiction." Another is to disavow the principle of the excluded middle

(OR (NOT A) A)

This can be seen to be equivalent to axiom 7 above. We can now ask "Does the geometry formalism implement intuitionistic logic?" We will see that the answer is "No."

It is evident that axiom 7 is not implemented, simply because double negatives cannot be asserted. Axiom 7*, on the other hand, is supported (implemented) because we insist that the data base be kept consistent. Clearly 1a, 1b, 2, 3a, 3b, and 5 are implemented. However, 4a and 4b are not, since disjunction cannot be asserted. Neither is axiom 6 implemented. Apparently, the logic which the geometry formalism implements is not even as strong as intuitionistic logic.

One might ask about other problem solving systems, and the logic that corresponds to them. Suppose we employ a resolution theorem prover as the basis for an expert problem solver, as was done, for example, in QA3 and STRIPS. Then the logic associated with these formalisms can easily be seen to be classical logic.

Suppose we have the geometry formalism, but allow facts to be deleted from the data base. Then the formalism corresponds to some kind of modal logic. It might seem that in this particular case we would gain some ability: the reasoning by cases problem above which could not be solved by the geometry formalism could be solved if we allow assertions to be deleted from the data base. The reasoning, which is incorrect, goes something like this: "First we assert F, then determine that this proves E, then delete the assertion that F is true." The reasoning is incorrect because after asserting F we make subsidiary deductions based on F being true, and although F is subsequently deleted, these subsidiary deductions are not, (since deleting F does not delete those facts asserted after F was asserted) leaving the potential of an inconsistent data base. We will return to this point shortly.

NEGATIONLESS INTUITIONISTIC LOGIC

We can easily verify that the following will be supported by the geometry formalism:

NEGATIONLESS INTUITIONISTIC LOGIC

```
Inference rules:
P, Q |- (AND P Q)
P, (IMPLIES P Q) |- Q
R, (IMPLIES P Q) |- (IMPLIES P (AND Q R))

Axioms:
1. (IMPLIES P (AND P P))
2. (IMPLIES (AND P Q) (AND Q P))
3. (IMPLIES (IMPLIES P Q) (IMPLIES (AND P R) (AND Q R)))
4. (IMPLIES (AND (IMPLIES P Q) (IMPLIES Q R)) (IMPLIES P R))
5. (IMPLIES (AND P Q) P)
```

This set of inference rules and axioms form a logic extensively studied by Griss [G6] called *negationless intuitionistic logic.*

FURTHER ABSTRACTION -- LOGIC OF WORLDS

By going from a formalism to the logic corresponding to that formalism, we have obtained a certain degree of abstraction. We shall now take one more step. We want to develop the notion of "logic needed by a world." To do this, we show (below) that the "contents" of a world are somewhat independent of the axiomatization of that world (for the relation of models to worlds, see [NOTE 11]).

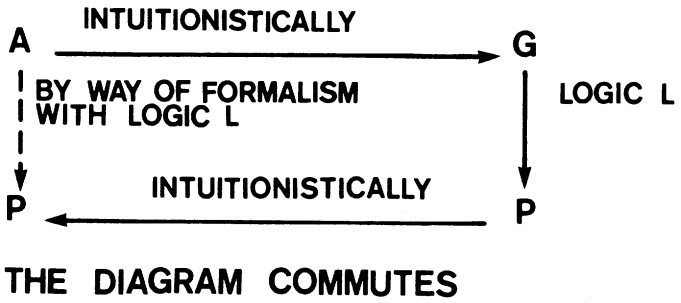
We need to clarify the sense in which a world is handled by a set of axioms and a logic. We can characterize a world by listing the styles of questions we typically ask of the world. That is, one geometry world is characterized by questions of congruence and incidence. Another geometry world (which we have not discussed before) deals with questions concerning compass and straight-edge constructions. In the blocks world we want to answer questions of planning, position, and support. In analysis we are concerned with convergence (in various ways). The question "can

an angle be trisected with compass and straight-edge?" is not part of geometry world (as taught in high school). The usual proof that the answer is "no" makes extensive use of group theory (see Fraleigh [F4]).

The notion of a world being handled is admittedly fuzzy. We can pin the notion down somewhat by associating some number of *milestone problems* with a world. Then a world can be handled if its milestone problems can be solved.

Griss demonstrated that with respect to his axiomatization of Euclidian geometry, negationless intuitionistic logic was sufficiently powerful to derive the milestones associated with classical axiomatic geometry [G7]. We must now ask if we can make this claim with respect to our (or, rather, Hilbert's) axiomatization. For an answer we have

> EQUIPOTENCY THEOREM. Suppose a formalism with corresponding logic L implements a world with axiomatization A, and we have a proof in logic L that (IMPLIES G P), and that intuitionistically (IMPLIES A G). We can then prove P in the formalism with an implementation axiomatized by A.



For example, suppose we want to know if the geometry formalism is expressive enough to allow us to write an expert geometry problem solver. How could it not be? Suppose there were a result P that could be proven from Hilbert's axioms (call these axioms A). If we could not get our expert to prove P, then we would be in trouble.

Suppose we know Griss has an axiomization G (which happens to be almost identical to Hilbert's) for which there is a negationless intuitionistic proof of A. That is, (IMPLIES G P) by negationless intuitionistic logic. Finally suppose that Hilbert's axioms A imply Griss's axioms G intuitionistically (they do). Then we can use the equipotency theorem to guarantee that we can construct programs in our formalism that will let us prove P from A. Furthermore, a corollary of the equipotency theorem tells us that the programs can be constructed algorithmically (for instance, by an analogy process along the lines we have suggested).

What this equipotency theorem says is that we can discuss the logic required to "handle" a world independently of both the formalism to which the logic corresponds and the axiomatization of the world (up to intuitionistic equipotency). If we select some minimal axiomatization of a world, then the *logic required by that world* is the weakest logic required to solve the appropriate milestone problems.

The proof of the theorem is trivial: simply prove G from A by intuitionistic means. Then write the implementation of G in the formal system, using the proof obtained as justificaion. [There is a bug in this proof which we will correct shortly]. Since L corresponds to the formalism, the formalism is now capable of proving P.

TWO APPLICATIONS OF THE EQUIPOTENCY THEOREM

Sussman suggests that early ARSE may implement negationless intuitionistic logic. ARSE is certainly that powerful. Jon Doyle [D1] investigated the possibility of writing a geometry theorem prover in early ARSE. We know by the equipotency theorem that this effort could have been completed successfully. Doyle was, of course, more interested in controlling the deduction process than proving theorems per se.

While on the subject of efficiency, we must make a confession: we cannot, of course, make any claims about the efficiency of various logics, nor compare the efficiency of one formalism to another on the basis of their logics. Nonetheless, I conjecture that the weaker the logic, the more efficient the implementation.

If Doyle had been investigating an analysis theorem prover, we know that the effort would have been unsuccessful since we know that intuitionistic logic is not sufficiently powerful for this task (see Heyting [H1]).

A FALSE VERSION OF THE EQUIPOTENCY THEOREM

Suppose, in the statement of the equipotency theorem, we were to claim that classically (IMPLIES A G). Surprisingly, this falsifies the theorem! The reason is explained by Kleene [K1, chapter XV, section 82, particularly page 509], which we restate:

> A formula is said to be *recursively realizable* if it is effectively computable. If A is recursively realizable and G can be deduced from A intuitionistically, then G is also recursively realizable. Furthermore (and this is important in what follows) the realization of G can be obtained from the intuitionistic proof.

If the proof of (IMPLIES A G) is non-intuitionistic, then all bets are off. The reason has to do with the proof being constructive. If an existence proof is not constructive, then one cannot derive a function which will compute the object claimed to exist from the proof. (Indeed, such a function may not even be recursively realizable, but this is not the point.) With respect to the equipotency theorem, we can no longer guarantee that we can implement the "axioms" G in terms of functions from A. Thus, we will be bitten by the indicated bug. Similarly, if the proof is intuitionistic, we are guaranteed of a way of realizing G. This note fixes the proof above.

Corollary: G can be realized algorithmically.

A NEGATIVE RESULT

Analogy requires an ability to transform proofs to plans to code. This is used to associate "bugs" in proofs to "bugs" in code, and to associate proof "patches" to code "patches." One might ask if this is, in theory, always possible. The answer is "No."

The reason is given above -- in order to produce an effective computation from an existence proof, that proof must be intuitionistic. Therefore, if some expert produced a result for which we could only obtain a non-constructive proof (because the underlying logic of the expert

was not intuitionistic), then any analogy process along the lines we have suggested would be unable to use the lifted non-constructive proof to aid in plan or code generation.

> Although very unlikely, there may be a logic between intuitionistic and classical logic such that proofs in this logic can "generate" realizations. That is, Kleene proved
>
> intuitionistic -> realizable
>
> and further showed that
>
> not (classical -> realizable)
>
> We are now claiming the converse:
>
> realizable -> intuitionistic
>
> but this might not be true.

The analogy process cannot be guaranteed to work if the problem solving formalism has classical predicate calculus as its corresponding logic. In particular, analogy cannot in principal always succeed if the problem solving formalism is based on a resolution theorem prover.

I believe the above is the first argument against resolution which involves neither efficiency considerations nor psychological speculations! If a problem solving formalism has classical predicate calculus as its corresponding logic, then under the inverse analogy map we may find ourselves with a non-intuitionistic proof, and thus not be able to translate the proof (even if it is still correct) into procedures.

A FORMALISM'S CORRESPONDING LOGIC MAY BE MODAL.

As we remarked earlier, if a formalism allows facts to be deleted from a data base, then we will require some kind of modal logic as the logic corresponding to the formalism. It is not strictly the case that the world must be described by a modal logic. McCarthy and Hayes[M5] suggested that the "situational calculus", based on the classical predicate calculus, be used to describe worlds in which things "change."

Like all good ideas, the idea behind the situational calculus is fairly simple. To each predicate and function we attach a "state variable" (also called a situation tag). For example, in the Towers of Hanoi one might make the claim

```
(FORALL (X S) (IMPLIES (ON-PEG P1 R2 S) (ON-PEG P1 R2 (MOVE-RING X P1 S))))
```

which is supposed to indicate that moving a ring onto some peg does not result in any rings already on that peg being removed. The "S" is a state variable.

This approach is less than elegant. The difficulties are well-known, and many were pointed out in the McCarthy and Hayes article introducing the approach. The first difficulty is known as the "Frame problem." In worlds with several functions on states (or even one state change function), one is forced to deal explicitly with what does not change, as well as what does. Another difficulty arises in describing frame shift information. For example, suppose Robbie (a well-dressed robot) in his top-hat, spats, and vest goes outside. It is clear in this circumstance that Robbie's top-hat, spats, and vest are also outside, since any state change involving Robbie's location also affects his attire. The situational calculus is not well suited to describing this type of world. For a further discussion of this problem, (and a solution) see Minsky [M7]. The two difficulties are sides of the same coin. The *Frame problem* concerns getting rid of facts no longer valid, and the *Shift problem* concerns determining what new facts are valid.

Rather than remaining in the classical predicate calculus, we could introduce modalities. McCarthy and Hayes discuss this possibility. Instead of the above statement in the situational calculus, one would have the following statement in some sort of temporal modal logic:

```
(FORALL (X)
  (IMPLIES (ON-PEG P1 R2)
           (AFTER-MOVE-RING X P1 (ON-PEG P1 R2))))
```

If the formalism supports the notion of multiple-CONTEXTs (as above), then the logic corresponding to the formalism will necessarily be a modal logic (or, equivalently, the formalism

will use situation tags). The important question (for us) is "Does the analogy process break down on formalisms whose corresponding logic is modal?" The answer is not clear at this time.

Stepankova and Havel[S6] investigated the relation between a formalism implementing "deletion" and the situational calculus. Their terminology is "image space" and "situation space." Their result (their second correspondence theorem) indicated that a "solution" to a problem in image space (i.e., a sequence of operations) exists iff there exists a (classical predicate calculus) proof that the solution exists in situation space. That is, the proof that a solution exists can always be converted to a sequence of operations which solve the problem, and vice versa. This result is not at odds with our discussion above, due to a restriction on the axiomatization of the image space (essentially eliminating quantified situations). As they remark, this axiomatization is (almost) always infinite.

ASSOCIATING LOGICS TO WORLDS

We pointed out above that there is a sense in which a world "requires" a logic, specifically the minimal logic that can "handle" the domain. We also saw that, while one can remain in predicate calculus, it is often convenient to use modal logic to describe a world.

We would like to claim that some world "requires" a modal logic. Such a claim would not be at odds with McCarthy and Hayes. Yes, the situational calculus is sufficient. But it may not be necessary. Specifically, there is a rather natural hierarchy of modal logics (discussed below). Then it is reasonable to claim that a domain "requires' some modal logic (but not a stronger one) in the sense that the logic can handle the domain.

HEIRARCHY OF WORLDS

We have seen that we can associate a formalism with a logic. We have also seen that we can associate a logic with a world (via the equipotency theorem). Finally there is a natural hierarchy (induced by potency) associated with logics.

This gives us a hierarchy of problem worlds. Near the bottom we have worlds which can be "handled" using negationless intuitionistic logic. Geometry is such a world, which indicates that from the standpoint of minimizing complexity the choice of geometry as an initial world in which to explore analogy was correct. If we introduce *operators* (like AFTER-MOVE-RING in a tower of Hanoi expert) that in some sense "change the state of the world", then we will require some kind of modal logic formalism.

Chained modal logics have a sequence of CONTEXTs, with only one "valid" at any time. It seems to be true that chained modal logics are no more powerful than their non-modal "base" logic in the sense that if one could prove (IMPLIES P Q) in the chained modal logic, then one could prove it (with the same proof) in the "base" logic. Tower of Hanoi only requires a chained modal logic, as does (apparently) the blocks world in its simple version [W3][F1].

Within chained modal logics (which are usually thought of as temporal modal logics), we can distinguish two varieties: "memoryless" and those with PLANNER-style backtracking. Tower of Hanoi clearly only needs the "memoryless" variety, which allows the state to change, but then prevents return to the old state (except by doing an operation which turns out to be the inverse of the operation just performed). The blocks world, on the other hand, was originally thought to require backtracking.

The difference between memoryless chained modal logics and chained modal logics with backtracking is this: if we can delete an assertion and all assertions made since then according to computational history (i.e., chronological backtracking), then we have implemented a kind of backtracking. There is another, which might be termed *dependency backtracking* and is used by

Sussman, Doyle, *et. al.* [S7][D1].

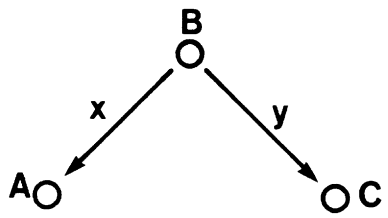
As we pointed out above, simply having the ability to delete assertions does not (in general) increase the power of a formalism to the extent that the "reasoning by cases" example can be solved. On the other hand, we can solve this problem with backtracking, provided disjunctive assertions are supported by the formalism.

```
                1.ASSERT B
                2. ASSERT C
                3. ASSERT (OR E F) *
split
        4. ASSERT F              9. ASSERT E
        5. ASSERT (NOT G)
        6. ASSERT (OR E G)*
        7. ASSERT E
        8. DELETE 4
```

Of course, at step 4 we would have needed to try proving (NOT F) to insure that the data base would not be inconsistent when we made the assertion. Nevins [N1] describes this approach in more detail.

A more complex modal logic, "tree modal logic", allow exploration of multiple CONTEXTs. Fahlman's [F1] blocks world program used CONNIVER's multiple-CONTEXT capabilities. Another use of more complex modalities occurs in describing "belief" systems. These modal logics seem to be more powerful than their "base" logics.

Another way to make the logic more complex is to introduce the BEFORE modality



In CONTEXT A we might ask "what would have happened if BEFORE X (i.e., in CONTEXT B) we had done Y (i.e., what is true in C)?" One suspects that "common sense" reasoning may demand logics with BEFORE modalities -- so a "common sense" world would have been a very bad choice as an initial world in which to study analogy.

CLAIM OF CLOSURE

We claim to have achieved a solution to analogy problems between worlds that require no logic stronger than negationless intuitionistic logic. The reason for believing this claim is that such a world can be handled in the geometry formalism (by the equipotency theorem). This claim does not imply that all the algorithms described in this thesis are free from combinatorial explosions; the map formation algorithm is particularly sensitive to the number of types and the number of "distinguished" objects of each type.

FUTURE WORK

PSYCHOLOGICAL VALIDATION

We have not concerned ourselves with the psychological validity of our analogy theory. We feel that first we must answer the question 'How can we get a machine to learn by analogy?" Only then are we in a position to ask "Is it possible that people learn by analogy in a similar way?"

George F. Luger [L3] reported some preliminary results of a study to determine what (if any) effect the solution of a problem had on the subsequent solution of a similar (in this case, isomorphic) problem. We will look at these results rather carefully.

The two tasks Luger used were the four ring Tower of Hanoi peoblem, and the Tea Ceremony. The Tea Ceremony playing mat is shown below. Four tasks:

FF: feeding fire (least noble)
SC: serving cakes
ST: serving tea
RP: reading poetry (most noble)

are initially performed by the Host. The Ceremony is a ritualistic dance in which tasks are transferred from one person to another according to two rules:

1. A person may only give up the least noble task they are performing at the time.
2. A person may take on a new task providing it is not more noble that the least noble task he is already performing.

The Ceremony is completed when the Youth is performing all four tasks.

	HOST	YOUTH	ELDER
FF	SC		
ST	RP		

This problem is isomorphic to the Tower of Hanoi (see chapter [ANALOGY EXAMPLES,AN ANALOGY FAILURE]). Luger reported the results of an experiment imvolving two groups of subjects. Group I was first given the Tea Ceremony, then Tower of Hanoi. Group II was first given Tower of Hanoi, then Tea Ceremony. Both groups were given *toys* (like the above mat) to work with. They were told that they could, if they wished, reset the *toy* to start over.

Timings, and number of moves were recorded, along with the standard deviation (s).

GROUP I 19 subjects	TC time 518 s=366	TOH time 149 s=125	TOTAL time 667 s=387
	moves 101 s=60	moves 40 s=29	moves 141
GROUP II 23 subjects	TOH time 399 s=210	TC time 306 s=223	TOTAL time 705 s=306
	moves 75 s=33	moves 66 s=33	moves 141

Luger then reported that times for Tea Ceremony were significantly different for both groups (at .03 confidence level) and the times for Tower of Hanoi were also significantly different for both groups (at .01 confidence level). Similarly both groups used significantly different numbers of moves. Luger then concluded that something was learned by doing the first problem that could be "transferred" to the second. By definition, transfer is accomplished by "reasoning by analogy."

OUR ANALYSIS OF LUGER'S DATA

In what follows, we will be using a confidence level of .05 (which is admittedly liberal) giving a critical value for Student's T test of about 1.68 (it is 1.689 for 36 degrees of freedom, 1.684 for 40 degrees of freedom, and 1.681 for 44 degrees of freedom). Note: we are asking if one score

is better than another, not whether they are merely different (which would give a lower t-test critical value).

We ask the following statistical questions:

1. Did Group I do better on TOH? v=40, t=4.559. Yes.

2. Did Group 2 do better on TC? v=40, t=2.31. Yes

These are the two answers which led to Lugar's conclusion. However, it is easy to see that these are not necessarily the proper questions! Consider: if *learning by analogy* did occur, then we expect to see some improvement within each group. This means we want to ask

3. Did Group I improve on its second problem? v=36, t=4.16. Yes.

4. Did Group II improve on its second problem? v=44, t=1.46. NO!

That's very surprising. Evidently there was *reasoning by analogy* in only one direction. Wait. Maybe (for some obscure reason), TC was just harder than TOH.

5. Did Group I do worse on their first problem (TC) than Group II on theirs (TOH)? v=40, t=1.255. No.

6. Did Group II take longer to do both problems? v=40, t=0.347. No.

Having thus performed a two-way analysis of variation, we observe that the following conjecture is supported by the evidence:

> The solution mechanism generated by the TC problem was immediately applicable to TOH. The solution mechanism generated by TOH was only marginally applicable to the TC problem. Obtaining the solution mechanism for TC is marginally more difficult than obtaining the mechanism for TOH.

EXPLANATION OF RESULTS

Simon [S4] gives several different algorithms for solving Tower of Hanoi problems. We will look at two of them (the target peg for these problems is peg 2):

1. Goal Recursion Strategy. This employs the concept of a pyramid. Originally a 4-pyramid is on peg 1. To move an n-pyramid from peg A to peg B, first move an (n-1)-pyramid to peg C, then move a ring from A to B, then move an (n-1)-pyramid from C to B.

2. Move-pattern strategy. On odd-numbered moves, move the smallest disk (which is necessarily movable). On even-numbered moves, move the next-smallest disk that can be moved (it can only go one place). The smallest disk moves in a constant direction.

1 --> 2 --> 3 --> 1	if the number of disks is odd
1 <-- 2 <-- 3 <-- 1	if the number of disks is even.

Failure to note the "kicker" at the end of the move-pattern strategy will result in successfully moving the rings, but to the wrong peg. That the target peg was selected ahead of time is an extremely bad feature of the experiment as it served only to add noise to the result. One suspects that the large standard deviation reported is due more to this feature than to differences in individual abilities.

With the goal recursion strategy, demands on memory increase linearly with the number of rings, whereas the move-pattern strategy makes a constant memory demand. On the other hand, the move-pattern strategy is perceptually driven, requiring a "toy" where goal recursion does not.

For our purposes, the important difference between the two strategies is that the move-pattern strategy requires a predicate FREE-TO-MOVEP which must be easy to compute.

Goal recursion requires the notion of "ordered subsets", or "pyramid."

We will assume that Group I (TC first) learned the goal recursion algorithm while Group II learned the move-pattern strategy (due to the ease of computing FREE-TO-MOVEP). This assumption is not supported by the experimental results. Indeed, Luger does not appear to realize that more than one strategy is available. Nonetheless, I do not find this assumption at variance with my intuition.

If this assumption is correct, then our theory of analogy predicts that there will be little observable improvement in Group II's performance, while we should observe much improvement in Group I. If analogy is to lift the move-pattern strategy to the Tea Ceremony world, it must first lift FREE-TO-MOVEP. That predicate is relatively expensive to compute in TC world. This will block the analogy process, at least temporarily. This blocking effect will naturally be reflected in the time taken to solve the problem.

No similar problems are encountered in lifting goal recursion to Tower of Hanoi world. Indeed, some bonus is given since "pyramids" can (presumably) be identified more readily.

Recall that our analogy process is not satisfied with a lifted solution until it has been justified, either by lifting justifications from the image world, or by generating them in the domain world using one step deductions. We might speculate that a subject will become aware of a solution (and the analogy which obtained it) only after that solution's justification and plans have been lifted. If this speculation is correct, our theory of analogy will makes two more predictions. First, the subjects will not realize the problems are identical until they are "within sight of the solution." That is, they will work on the problem, and then experience an "Aha!" towards the end. Furthermore, those subjects in Group II who do no better on TC than TOH will not notice the isomorphism at any point.

IMPROVING ANALOGY

In examining the analogy process, we can see at least four areas which require further development before analogy can become truly useful.

MAP EXTENSION and DEPTH-FIRST SEARCH

After applying constraint relaxation, the map formation and extension algorithm embarks on a depth-first search on the tree of partial maps. In general, when one uses a depth-first search, it is because one has nothing better to do.

Presumably, when a partial map is rejected by the image semantics, the *way* it is rejected indicates what our next guess should be. Indeed, in the TTT example we observed two modes of failure:

1. An assertion could not be mapped.

2. The image assertion was false.

We used this information to isolate the highest node on the partial map tree which caused the error.

In a similar manner, we suspect that careful failure analysis should indicate not only where the error occurred (in the partial map tree), but which would have been a better choice. Although we can see a glimmer of how this might work in the TTT example, we don't have enough clear examples to formulate an algorithm.

PATCHES INTO CODE

We have not really come to grips with the problem of inserting patches into code, although an obvious simple-minded trick was mentioned in the blocks world example.

The relation between plans and world descriptions is always made explicit. The same cannot be said for code and plans. Although we can tell when code has completed a plan step, we cannot tell (in general) when that plan step was started. We certainly have no explanation of why the particular ordering and interleaving of plan steps was chosen.

The commentary indicated above may not be needed to insert patches into code; we may be able to derive the required information from code and plan. On the other hand, even if we had all this commentary we wouldn't know what to do with it. Clearly more research into this aspect of the analogy process is required.

MODAL LOGIC

Operators might be thought of as predicates on predicates. We might write the statement "Adults don't cry over spilt milk" as

```
(FORALL (X Y) (IMPLIES (AND (ADULT X)
                            (MILK Y))
                       (AFTER-SPILLED Y
                                      (NOT (CRYING X)))))
```

where AFTER-SPILLED is an operator.

By design, none of the examples we have seen required the use of operators. In chapter [LOGICS OF EXPERTS,LOGIC OF WORLDS] we discussed modal logic and its relation to operators. It is clear that most interesting worlds use operators, so extending the analogy process in this direction is important.

The map formation and extension algorithm would deal with operators after a little

modification. Unfortunately, the plan language, the result-justification algorithm, the structure of representation claims, and the theorem matcher portion of the debug and patch algorithm all must be altered to handle operators.

A different problem, which might also be solved by the mechanisms needed to handle modalities, concerns the use of functionals. In statistics, it is very convenient to think in terms of a SIGMA functional, which takes a data set (a series of numbers), applies a function to each element of the data set, and then delivers the sum. Formally, SIGMA is a function of a function, and thus it cannot be directly expressed in the first-order predicate calculus.

MORE FLEXIBLE REPRESENTATIONS

We restricted representations to have the form of a list of objects so that REPRESENTATION-CLAIMs could have a simple form and proof, and so that they could be used in handling extended predicate forms. If representations are made more complex (for example, allowing arbitrary list structures, or arrays, or hash tables as representations), we will be forced to give up this simplicity. Since real programs make extensive use of these more complex representations, it is important that the analogy process be able to deal with them.

Another extension to the analogy theory concerns describing plans for doing things like sorting representations. The difficulty is that representation claims are considered to be true for all time. This means we cannot say, as we would want to in a sorting program, that some claim about the representation (list) is now true, where it might not have been true previously.

Although work has been done on this problem by Suzuki [S9], Manna and Waldinger [M1], and others in relation to proving programs correct, we have not yet been able to bring techniques from that school to bear on our problems.

NOTES

NOTE 1.

We will frequently speak of an analogy map, which goes from the domain to the image world just like a good map should.

Since our real interest is in the lifting operation (which is our shorthand for "applying the inverse analogy map"), the reader might wonder why we don't have our maps go the other way. The reason is that the map from domain to image world may be many to one. This means that the "inverse map" is not well defined, so it is strictly speaking incorrect to call it a map.

The notion of "lifting" unfortunately creates some confusion because all the diagrams have the domain on the left, while "lifting" requires one to think of it on the top. The term "lift" comes from considering topological covering spaces and their map to the space they cover.

NOTE 2.

It might be argued that the inter-world/intra-world distinction is an illusion, since if world A is analogous to world B (for example, p ane geometry world is analogous to solid geometry world), then they must be sub-worlds of some larger world. That is, we can transform inter-world analogy into intra-world analogy by finding a super-world.

While the observation above is correct, (indeed, we can go both ways since intra-world analogy might be considered a special case of inter-world) it misses the mark. The distinction centers on whether relations (or, if you prefer, predicate symbols) have different meanings on the two sides of the analogy. If the analogy map is always the identity map on predicate symbols (or at least the identity map on non-unary predicate symbols), then we may assume that we are working with an intra-world analogy.

NOTE 3.

Pratt [P2] gives a fundamental result that loopless programs are only as tractable as the theory (descriptive component of expertise) of the world in which the program works. That is, even if the code is very simple in form (loopless, in fact), in order to prove some property of it we may need to prove hard theorems about the world in which it works. If the code has loops, then we may need to prove strictly harder theorems than those for loopless programs.

NOTE 4.

Recall that Evans's problem statement had the form

A is to B as C is to ...

Part of Evans's analysis concerned the transformation which was applied to the A diagram to obtain the B. When Evans finally selects the best map, the transformation can be mapped, and thus it might be said that Evans also uses analogy to write programs. If this is a program, it is a rather peculiar one. Suppose I were to give Evans's "program" another picture (say A1) which was identical to diagram A. The "program" would not be able to run, except in the very weak sense of allowing Evans analogy algorithm match A1 and A, and hence deliver diagram B as the answer. That is, if one wishes to assert that Evans's analogy program writes other programs, then one must admit that the interpreter for these new programs is Evans's analogy program, and in that sense cannot be distinguished from input data.

NOTE 5.

In the back of a text book one might find answers to the problems in the text. These solutions constitute anticipated answers; they are used by the student to check results. For the same reasons the author provides these answers, we want to give analogy access to anticipated

solutions.

How can we give these anticipated solutions to analogy? One way is to state "problems" as assertions. The problem for the expert problem solver (and thus for analogy) is to convince itself that the assertion is true. We can also ask the expert to prove theorems. Similarly, in the midst of solving some domain problem, the expert may find itself blocked unless some predicate is true; anthropomorphically the expert wants the true result from that predicate and therefore has an anticipated solution.

Another form of solution anticipation is a check on the answer. For example, suppose the expert used analogy to find the square root of a number. Then the expert could check the solution by squaring it.

NOTE 6.

We wish to use non-modal logic in our TTT world description. If we had blank squares, then, for example, the center square, S5, would initially be blank.

(BLANK S5)

Now suppose the X player takes the center

(XED S5)

Is S5 blank? No. This means that something that used to be true has suddenly become false. Of course one solution to this problem is to use situation tags, which is equivalent to using modal logic in this example.

We are using a different technique. Initialy each square is either XED or ZEROED, *but we don't know which*. If we then learn that

(XED S5)

we might "prove" that the corner square S8 is zeroed.

(ZERCED S8)

In this way play can proceed.

Stating the rule

A square may be XED or ZEROED, but not both.

as we did forces us to use a non-modal formalism to write the TTT expert. Then it is a simple matter to show that TTT only needs negationless intuitionistic logic, so we can use the closure claim in chapter [LOGIC OF EXPERTS] to show that analogy can handle TTT world.

NOTE 7.

An early form of semantic template was used by Kling [K2], although he did not try to derive semantic templates from world descriptions. For comparison, his semantic types were STRUCTURE, SET, OPERATOR, RELATION, OBJECT, PROPERTY, RELSTRUCTURE. The obvious objection to this set is that it is useful only in an algebraic setting. Furthermore, in Kling's system semantic templates were necessarily supplied by the "user" of the system. Finally, the fact that he had so few types means that they were not very useful in restricting possible maps. These problems are eliminated by automatically deriving templates from the world description.

NOTE 8.

It is very tempting to say "in TTT there is one square on 4 rows, and in JAM there is one road containing 4 towns, so that must be the correct correspondence." This counting argument is very appealing, but must be rejected. Similarly, we avoided taking advantage of the fact that there are 9 roads (squares) and 8 towns (rows) earlier. The reason we reject this style of counting argument is that first, it makes use of domain world (JAM) knowledge to count the number of roads and towns (which we cannot assume to have yet), and second it assumes the two worlds are isomorphic by testing for equality in the numbers resulting from counting. It also assumes that the worlds are finite, but finiteness is not the problem here. While assuming that two worlds are isomorphic might be a good heuristic, we don't want to make the notion fundamental to the

analogy process.

NOTE 9. By using the axioms for global stability, we can determine that if block C is removed from the scene in the diagram in chapter [OVERVIEW OF ANALOGY,SECOND VIGNETTE -- BLOCKS], then the scene without C is not globally stable. However, using FACT29

```
:FACT29
(FORALL (B1 B2)
        (IMPLIES (EXISTS (B3) (AND (DISTINCT B3 B2)
                                   (SUPPORTS B3 B1)
                                   (STABLE B3 B1)))
                 (SCAFFOLD B2)))
```

we can prove that block C is scaffolding. Then using AXIOM12

```
:AXIOM12
(FORALL (X) (IMPLIES (SCAFFOLD X)
                     (GSTABLE (SCENE-WITHOUT X))))
```

we can prove that the scene is globally stable without block C. We have now proven both a predicate and the negation of that predicate. The blocks world description is inconsistent.

NOTE 10.

We define the intersection function as a function of two arguments X and Y. We claim that if X and Y are distinct lines, then the following assertions are true

```
(IN-LN X (INTERSECT X Y))
(IN-LN Y (INTERSECT X Y))
```

We carefully avoid claiming that the result of applying the intersect function to two lines is a point.

The problem is that since a definition cannot convey any new information, we first interpret the definitions in the Herbrand Universe, so the result of applying the INTERSECT function to two lines X and Y is a list of three elements:

```
(INTERSECT X Y)
```

for which the two assertions above are true in the Herbrand Universe. If we were to claim that the above list is a point, what we would really be saying is that there is a homomorphism of the Herbrand Universe onto plane geometry world such that (INTERSECT X Y) is mapped onto a point. But, that such a map exists is new information.

Suppose we were to say that it is a "point." Since we are discussing (classical) mathematics, functions are deterministic, so the point returned by INTERSECT must be unique. Further suppose (AND (DISTINCT A B) (PT A) (PT B)). Then by P-I1 we have a line X = (LINE A B). Suppose there were a distinct line Y such that (IN-LN Y A) and (IN-LN Y B). Then (INTERSECT X Y) returns points A and B and by the result being unique, A = B, contradiction! Thus such a line Y does not exist. Indeed, we know that it does not, but we have just proven this fact without using axiom P-I2. This should not be surprising because the contrapositive of P-I2 is the proof of uniqueness. In other words, this "definition" really contains an axiom hidden in the result-type declaration. We must disallow such definitions.

On the other hand, if we allow non-deterministic functions, we would be able to prove that the INTERSECT function, which we could declare returns objects of type PT, is determinsitic by the contrapositive of axiom P-I2. The reason this approach (which is the typical one in mathematics) was rejected is simply that in the predicate calculus all functions are deterministic.

NOTE 11.

Mathematically, a model for a set of formula F is an interpretation M for F such that every formula of F is true in M (see [M3], p. 145). Strictly speaking, then, a model for the description component of a theory is not independent of that axiomatization, although when considered as a map, the image of the interpretation may be independent.

Consider (say) finite abelian group world. There are many models of finite abelian groups (at least one for every integer). Some of these models may be excluded by some axiomatization but not by others (specifically, the abelian group with only one element). Furthermore, theories may have non-standard models (for example, non-standard analysis).

Based on these considerations we can say that the notion of a world includes a set of images of standard models of a theory.

BIBLIOGRAPHY

[B1]Banerji, R. B., and Ernst, G. W., "Strategy Construction Using Homomorphisms Between Games", 1972, *Artificial Intelligence Vol. 3*, p.223-249, North-Holland Publishing Company, Amsterdam.

[C1]Chang, Chin-Liang, and Lee, Richard Char-Tung, *Symbolic Logic and Mechanical Theorem Proving*, 1973, Academic Press, New York.

[D1]Doyle, Jon, *Analysis by Propagation of Constraints in Elementary Geometry Problem Solving*, 1976, MIT AI Working Paper 108.

[E1]Evans, Thomas G., *A Program for the Solution of Geometric Analogy Intelligence Test Questions*, in [M3].

[F1]Fahlman, Scott E., *A planning System for Robot Construction Tasks*, 1973, TR283.

[F2]Feigenbaum, Edward A., and Feldman, Julian eds., *Computers and Thought*, 1963, McGraw-Hill, Inc., New York.

[F3]Forder, Henry George, *The Foundations of Euclidean Geometry*, 1958, Dover, New York.

[F4]Fraleigh, John B., *A First Course in Abstract Algebra*, 1972, Addison-Wesley Publishing Company, Reading, Mass.

[F5]Funt, Brian V., *A Procedural Approach to Constructions in Euclidean Geometry*, 1973, MS thesis, University of British Columbia.

[G1]Gardner, M., Mathematical Games, *Scientific American* Jan. 1976, p. 118-123

[G2]Gelernter, H., *Realization of a Geometry-Theorem Proving Machine*, in [F2].

[G3]Goldstein, Ira P., *Understanding Simple Picture Programs*, 1974, TR294.

[G4]Goldstein, Ira P., *Elementry Geometry Theorem Proving*, 1973, AI Memo 280.

[G5]Greenberg, Marvin J., *Euclidean and non-Eudlidean Geometries*, 1974, W. H. Freeman and Company, San Francisco.

[G6]Griss, G. F. C., "Logic of Negationless Intuitionistic Mathematics", 1951, *Proceedings of the Section of Sciences, Series A*, North-Holland Publishing Company, Amsterdam.

[G7]Griss, G. F. C., "Negationless Intuitionistic Mathematics III to IVb", 1951, *Proceedings of the Section of Sciences, Series A*, North-Holland Publishing Company, Amsterdam.

[H1]Heyting, A., *Intuitionism: An Introduction*, 1971, North Holland Publishing Co., Amsterdam.

[H2]Hilbert, David (translated by P. Bernays), *Foundations of Geometry*, (1971), (The Open Court Publishing Company, La Salle, Illinois).

[K1]Kleene, Stephen Cole, *Introduction to Metamathematics*, 1952, D. Van Nostrand Company, Inc.,

New York.

[K2]Kling, Robert Elliot,*Reasoning by Analogy with Application to Heuristic Problem Solving: a Case Study*, 1971, Stanford AIM-147.

[L1]Lewis, C. I., and Langford, C. H., *Symbolic Logic*, 1959, Dover, New York.

[L2]Lines, L., *Solid Geometry*, 1965, Dover, New York.

[L3]Luger, George F., *Behavioral Effects of Problem Structure in Isomorphic Problem Solving Situations*, 1975, Edinburgh D.A.I. Research Report No. 4.

[M1]Manna, Zohar, and Waldinger, Richard, "Knowledge and Reasoning in Program Synthesis", 1975, *Artificial Intelligence*, Vol. 6, p. 175-208.

[M2]Manna, Z., and Waldinger, R., "Is "Sometime" Sometimes Better than "Always"? Intermittent Assertions in Proving Program Correctness", 1976, *Proc. of 2nd International Conf. on Software Engineering*, 32-39.

[M3]Margaris, Angelo, *First Order Mathematical Logic*, 1967, Blaisdell Publishing Company, Waltham, Mass.

[M4]Marr, D., and Poggio, T., *From Understanding Computation to Understanding Neural Circuitry*, 1976, MIT AI Memo 357.

[M5]McCarthy, J., and Hayes, P., "Some Philosophical Problems From the Standpoint of Artificial Intelligence", 1969, *Machine Intelligence 4*, Edinburgh U. Press.

[M6]McDermott, Drew Vincent, *Assimilation of New Information by a Natural Language-understanding System*, 1974,TR291.

[M7]Minsky, M., "A Framework for Representing Kinoledge",1974, MIT AI Memo 306.

[M8]Minsky, Marvin ed., *Semantic Information Processing*, 1968, MIT Press, Cambrdge.

[M9]Moore, J., and Newell, A., *How can MERLIN Understand?*, 1973, Carnegie-Mellon University memo.

[N1]Nevins, Arthur J., *A Relaxation Approach to Splitting in an Automatic Theorem Prover*, 1974, MIT AI Memo 302.

[N2]Nevins, Arthur J., *Plane Geometry Theorem Proving Using Forward Chaining*, 1974, AI Memo 303.

[N3]Nilsson, Nils J., *Problem-Solving Methods in Artificial Intelligence*, 1971, McGraw-Hill, New York.

[P1]Polya, George, *Mathematical Discovery: On Understanding, Learning, and Teaching Problem Solving*, 1962, John Wiley & Sons, New York.

[P2]Pratt, V. R., *Semantical Considerations on Floyd-Hoare Logic*, 1976, MIT Laboratory for

Computer Science TR-168.

[P3]Pratt, Vaughan R., *The Competence/Performance Dichotomy in Programming*, 1976, class notes.

[R1]Rich, C. and Shrobe, H. E., *Initial Report on a LISP Programmer's Apprentice*, 1976, TR-354.

[S1]Sacerdoti, Earl D., "The Nonlinear Nature of Plans", 1975, *Advance Papers of the Fourth IJCAI.*

[S2]Scandura, Joseph M., Wulfeck II, Wallace H., "Higher Order Rule Characterization of Heuristics for Compass and Straight Edge Constructions in Geometry", 1974, *Artificial Intelligence Vol. 5*, p. 149-183, North-Holland Publishing Company, Amsterdam.

[S3]Sheppard, David A., "A Plane Strategy for 3-D TTT", 1975, *J. Recreational Mathematics*, Vol. 8(3).

[S4]Simon, Herbert A., "The Functional Equivalence of Problem Solving Skills", 1975, *Cognitive Psychology* Vol 7, p. 268-288, Academic Press, Inc.

[S5]Slagel, James R., *A Heuristic Program that solves Symbolic Integration Problems in Freshman Calculus*, in [F2].

[S6]Stepankova, Olga and Havel, Ivan M., "A Logical Theory of Robot Problem Solving", 1976, *Artificial Intelligence*, Vol. 7, p. 129-161.

[S7]Sussman, Gerald Jay, and Stallman, Richard Matther, *Heuristic Techniques in Computer Aided Circuit Analysis*, 1975, MIT AI memo 328.

[S8]Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*, 1973, TR297.

[S9]Suzuki, Norihisa, *Automatic Verification of Programs with Complex Data Structures*, 1975, Stanford AIM-279.

[U1]Ullman, Shimon, *A Model-driven Geometry Theorem Prover*, 1975, AI Memo 321.

[W1]Waldinger, R. J., and Levitt, K. N., "Reasoning About Programs", 1974, *Artificial Intelligence*, Vol. 5, p. 235-316.

[W2]Waters, Richard C., *A System for Understanding Mathematical FORTRAN Programs*, 1976, MIT AI memo 368.

[W3]Winograd, Terry, *Procedures as a Representation for Data in a Computer Program for Understanding Natural Language*, 1971, TR17.

[W4]Winston, P. H., (Ed.) *The Psychology of Computer Vision*, 1975, McGraw-Hill, New York.

[W5]Winston, Patrick, personal communication 1976.

[W6]Winston, Patrick H., *Learning Structural Descriptons from Examples*, 1970, TR231.

[W7]Wong, Richard, *Construction Heuristics for Geometry and a Vector Algebra Representation of Geometry*, 1972, MAC TM28.

[W8]Woods, Frederick S., *Higher Geometry*, 1961, Dover, New York.

[W9]Wynne, Bayard E., "Perfect Magic Cubes of Order 7", 1975, *J. Recreational Mathematics*, Vol. 8(4).

TR -- MIT AI Lab Technical Report

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 3/21/96

Report # AI-TR-403

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: 145 (152-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): FOLLOWS PAGE NUMBER 7

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-145) UNM'ED TITLE PAGE, UNM'ED ABSTRACTS (2), UNM FUNDING AGENT, UNMARK, 6-7, UNMBLK, 9-145
(146-152) SCANCONTROL, COVER, SPINE, DOD, TRGT'S (3)

Scanning Agent Signoff:

Date Received: 3/21/96 Date Scanned: 4/17/96 Date Returned: 4/25/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-403	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Use of Analogy to Achieve New Expertise		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard Brown		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 April 1977 13. NUMBER OF PAGES 148
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
Analogy
Automatic Programming
Debugging
Planning
Reasoning
Expert Problem Solving

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

A new theory will be presented in which analogy is viewed as a process that writes expert problem-solving programs. Algorithms will be presented which are used in this theory and will illustrate the operation of these algorithms in six examples. In this theory, the analogy process reasons between different problem worlds. Four different pairs of problem worlds will be used in these examples with special attention given to geometry world (plane and solid geometry). Theoretical limitations and psychological validity are discussed.

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)*

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 04/17/1996

M.I.T. Libraries
Document Services